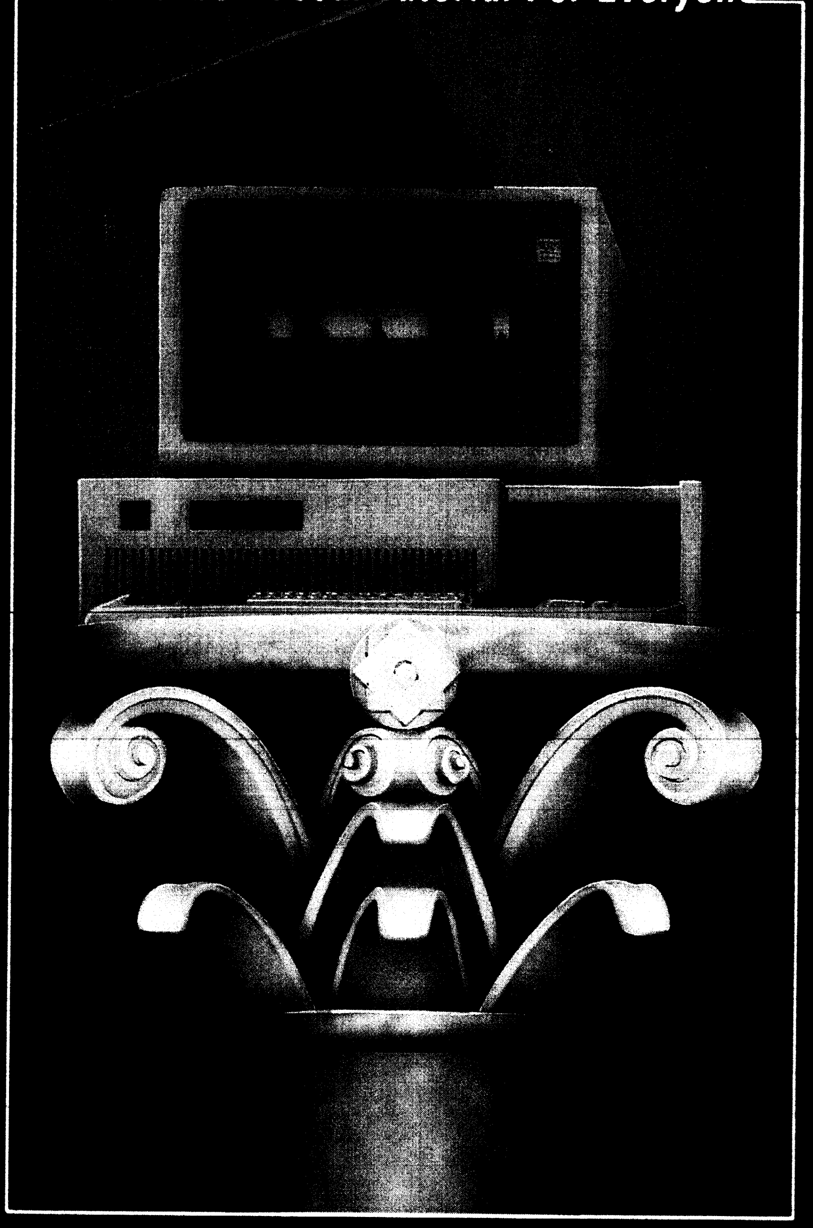
TUTOR

The Turbo Pascal Tutorial For Everyone



TURBO TUTOR

Borland's No-Nonsense License Statement!

This software is protected by both United States Copyright Law and International Treaty provisions. Therefore, you must treat this software *just like a book* with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing up your software and protecting your investment from loss.

By saying, "just like a book", Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another—so long as there is **No Possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's Copyright has been violated.)

WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, BORLAND INTERNATIONAL, INC. ("BORLAND"), warrants the same to be free of defects in materials and workmanship for period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, BORLAND will replace the defective diskette or documentation. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, special, incidental, consequential, or other similar claims.

BORLAND INTERNATIONAL, INC., SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO DEFECTS IN THE DISKETTE AND DOCUMENTATION, AND THE PROGRAM LICENSE GRANTED HEREIN, IN PARTICULAR, AND WITHOUT LIMITING OPERATION OF THE PROGRAM LICENSE WITH RESPECT TO ANY PARTICULAR APPLICATION, USE, OR PURPOSE. IN NO EVENT SHALL BORLAND BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL OR OTHER DAMAGES.

GOVERNING LAW

This Statement shall be construed, interpreted and governed by the laws of the state of California.

Third Edition
Printed in USA
9 8 7 6 5 4 3

Turbo Tutor

A Self-Study Guide to Turbo Pascal

Copyright 1986 by
Borland International, Inc.
4585 Scotts Valley Drive
Scotts Valley, CA 95066
USA

Table of Contents

Introduction xiii
About this Book xiii
What We'll Teach You xv

PART I TURBO PASCAL FOR THE ABSOLUTE NOVICE 1

Chapter 1 Getting Started with Turbo Pascal 3
Using the Disk with this Manual 4
TUTOR.PAS and the .EX Files 4
Main Menu 4
Subprograms 4
Modifying the Examples 5
Files on the Disk 5

Chapter 2 Computers: Myth Versus Reality 7
Beyond the Misconceptions 8
Computers are Fast 8
Computers are Stupid 8
Computers are Literal 8

Chapter 3 Computer Basics 11
Computer Hardware 11
Central Processing Unit 12
Memory 12
Mass Storage 12
Input Devices 13
Output Devices 13
Digital Data 13
Computer Software 14
Operating Systems 15
Application Software 16
Characters 16
Review 19

Chapter 4 A Brief History of Programming 21
In the Beginning... 21
A Programming Shorthand 22
High-Level Languages 23
Programming Languages and Microcomputers 24
And Finally...Pascal 24
Interpreters and Compilers 25
The Turbo Pascal Advantage 26

Chapter 5 Getting Ready to Use Turbo Pascal 29
Backing Up Your Original Turbo Disk 30
  Making Backups: MS-DOS and IBM PC-DOS 30
Getting Ready to Work 31
  Making a Turbo System Disk 31
  Installing Turbo Pascal 31
    Using TINST with Non-IBM PC Systems 33
    Using TINST on the IBM PC and Compatible Systems 35
    Additional Files 35
Another Backup 36
Using a Single Drive System 37
Using a Hard Disk 37

Chapter 6 Using Turbo Pascal 39
Starting Turbo Pascal 40
  Main Menu Overview 41
  Choosing a File Name 41
Using The Editor 42
Writing the Program 44
  Typing the Program 44
  Compiling and Running the Program 45
  Saving Your Source Program 45
  Saving Your Compiled Program 46
  Finishing Up 47
Review 48

PART II A PROGRAMMER'S GUIDE TO TURBO PASCAL 51

Chapter 7 The Basics of Pascal 53
Some Pascal Terms 53
Data Types 54
  Predefined Data Types 54
  User-Defined Data Types 55
Identifiers 55
    Exercises 58
Reserved Words 59

Constants 59
Constant Definitions 60
Variables 61
Variable Declarations 61
Operators 62
Expressions 63
The Order of Operations in Expressions 64
Exercises 65
Statements 66
Comments 66
A Program Example 67
The Program Heading 69
The Declaration Part 69
The Statement Part 69
Exercises 72
Review 73

Chapter 8 Program Structure 75
The Program Heading 75
The Declaration Part 76
Formatting Your Declarations 77
The Statement Part 79
Formatting Your Statements 79
Statement Types 80
Comments: The Rest of the Story 81
Review 84

Chapter 9 Predefined Data Types 87
Integers 87
Integer Operators 88
Integers and Arithmetic Overflow 89
Exercises 89
Byte 90
Real 90
Real Operators 91
Exponential Notation 91
Boolean 93
Char 94
Strings 96
String Constants 97
Putting Control Characters in a String 97
Declaring String Variables 97
Review 99

Chapter 10 Defined Scalar Types 101
Enumerated Scalar Types 102
Ordinal Values 103
Standard Functions for Scalar Types 104
Cyclical Enumerated Types—Avoiding Range Errors 106
Exercises 106
Range-Checking 106
Undefined Values in Enumerated Types 107
Subranges 108
Subranges as Anonymous Types 109
Input and Output 110
Memory Usage 110
Review 110

Chapter 11 Control Structures 111
Conditional Execution: The If Statement 111
Compound Statements and the If Statement 112
Boolean Expressions 113
More Boolean Operators 114
Exercises 115
Repetitive Tasks 116
Iteration 116
The While Statement 117
The Repeat...Until Statement 118
The For Statement 119
Simulating a Step Size Larger Than 1 121
Endless Loops 122
The Case Statement 123
The Constant List 125
Review 125

Chapter 12 Procedures and Functions 127
Subprograms 127
Scope 130
Exercises 133
The Lifetime of Local Variables 135
Parameters 135
Functions 138
Recursive Subprograms 141
Forward Declarations 142
Scope and Recursion 143
The Exit Procedure 144
Review 146

Chapter 13 Arrays 147
Array Assignments 151
Range-Checking and Arrays 152
Initializing an Array 152
Representing an Array in Memory 153
Packed Arrays 155
Review 155

Chapter 14 Strings 157
String Types 157
String Operators, Functions, and Procedures 158
  String Assignments 159
  The Length Function 159
  The Concat Function and the + Operator 160
  The Copy Function 161
  The Pos Function 161
  The Delete and Insert Procedures 162
  Miscellaneous Character Functions 164
Representing Strings in Memory: Strings as Arrays 164
String Comparisons 166
Numeric Conversions 167
Strings as Parameters 169
Review 170

Chapter 15 Records 173
The With Statement 176
Variant Records 179
Free Unions: Omitting the Tag Field 182
Review 183

Chapter 16 Sets 185
Building a Set: The Set Constructor 186
Defining a Set Type 186
Set Operations 188
  Set Membership: The In Operator 188
  Set Equality and Inequality 188
  Set Union, Intersection, and Difference 189
  Set Inclusion Operators 189
  Set Disjunction 189
Review 190

Chapter 17 Pointers and Dynamic Allocation 191
Pointers 192
Dynamic Allocation: The New Procedure 193
Dereferencing Pointers 194
The Nil Pointer 194

Linked Lists 195
The Heap 199
The MaxAvail Function 199
Deallocation of Dynamic Variables: Memory Management 200
Dispose 200
Mark and Release 202
Review 203

Chapter 18 Files 205
Turbo's Pascal I/O Procedures 205
Read and Readln 206
Write and Writeln 206
Write Parameters 207
Get and Put 210
File Types 210
Reading and Writing Text Files 211
The Assign Procedure 213
Reset and Rewrite 214
Read and Readln with Text Files 214
Eof, Eoln, SeekEof, and SeekEoln 215
The Close Procedure 215
Random Access Files 215
Creating a Random Access File 216
Properties of Random Access Files 219
The Seek Procedure 220
The FilePos and LongFilePos Functions 220
The FileSize and LongFileSize Functions 221
The Append Procedure 221
The Truncate Procedure 221
Other Built-in File Procedures and Functions 221
For MS-DOS and PC-DOS Only:
Directory Management Procedures 222
Talking to Your Computer's Peripherals: Device I/O 223
Logical Devices 223
Standard Files 224
Advanced Keyboard Handling:
KeyPressed and the Standard File Kbd 224
Untyped Files 225
Declaring Untyped Files 226
Using Untyped Files 226
MS-DOS/PC-DOS Only: Specifying a Block Size 227
I/O Error Handling 227
Review 230

Chapter 19 A Sample Program 231
Turbo Typist 231
Step 1: The Program Body 233
Step 2: File I/O 233
Step 3: The Main Control Loop 233
Animation 234
Dare to Experiment 237
Review 237

PART III ADVANCED TOPICS IN TURBO PASCAL 239

Chapter 20 Stacks, Queues, Deques, and Lists 241
Linked Lists 241
  Single vs. Double Links 241
Starting a Linked List 242
  Circular Linked Lists 243
  Insertion 244
  Deletion 245
Stacks 245
Queues 247
Deques 249
Lists 250
Review 251

Chapter 21 Trees, Graphs, and Other Linked Structures 253
Introduction to Trees 253
Binary Trees 254
  Searching Binary Trees 254
  Inserting into Binary Trees 255
  Traversing Binary Trees 256
  Deleting Nodes and Subtrees 257
  Non-Binary Trees 258
Graphs 259
Sparse Arrays 260
  Mixed Sparse Arrays 262
  When to Use Sparse Arrays 263
Review 263

Chapter 22 Sorting and Searching 265
Sorting 265
  Insertion Sort 265
  Shellsort 267
  Quicksort 268
Searching 270
  Sequential Search 270
  Binary Search 271

Hashing....272
External Search....273
Review....274

Chapter 23 Writing Large Programs....275
A Program's Memory Requirements....275
Too Much Data....276
Too Much Code....277
Too Much Text....278
The Stack....278
The Heap....280
Solutions....280
Include Files....280
Modular Programming....281
Libraries....282
Using Overlays to Save Code Space....282
Menu Program Example 1: Overlays....285
Location of Overlay Files....286
Overlay Restrictions....286
Chaining....287
Executing Files....289
Menu Program Example 2: Chain and Execute....290
Overlay vs. Chain/Execute....291
Review....292

Chapter 24 Typed Constants....293
Defining a Typed Constant....294
Array Constants....295
Record Constants....296
Set Constants....296
Special Properties of Typed Constants....297
Mutability....297
Lifetime....297
Scope....297
Typed Constants as Static Variables....298
Typed Constants as Initialized Variables....299
Manipulating Components of Constants....299
Saving Constant Space....300
Passing Constants as Var Parameters....301
How Typed Constants are Stored in Memory....301
A Final Caveat: Typed Constants and Execution from Memory....302
Review....302

Chapter 25 The Goto Statement 303
Syntax of the Goto Statement 303
How to Use the Goto Statement—And Why Not to 304
Review 306

Chapter 26 Absolute Variables and Untyped Parameters 307
Absolute Variables 307
Untyped Parameters 309
  Using Untyped Parameters 310
Review 312

Chapter 27 Computer Numbering Systems and Boolean Operations on Integers 313
Integers as Bits and Bytes:
  How Integers are Represented in Memory 313
  Place Value 313
The Special Relationship Between Binary and Hex 317
Two's Complement Notation: Representing Negative Numbers 317
How Byte Values are Stored in Memory 318
Boolean Operations on Integers and Bytes 319
The Shifting Operators: Shl and Shr 320
Review 321

Chapter 28 Using 8088/8086 Assembly Language with Turbo Pascal 323
External Subprograms 323
Accessing Parameters from External Subprograms 325
Allocating Local Variable Space 327
Allocating Static Storage 327
Libraries (PC-DOS/MS-DOS Only) 328
The Inline Statement 330
  Syntax and Semantics of the Inline Statement 330
Speeding Up Turbo Programs with Inline Statements 332
  Interrupt Handling 332
Review 334

PART IV APPENDICES 335

Appendix A Syntax Diagrams for Turbo Pascal 337

Appendix B Exercise Solutions 351
Chapter 7 351
  Answers to Sample Program Questions 351
Chapter 9 352
  First Set 352
  Second Set 352

Third Set 352
Chapter 10 352
Chapter 27 353
First Set 353
Second Set 353

Appendix C Summary of Standard Procedures and Functions 355
Input/Output Procedures and Functions 355
Arithmetic Functions 356
File Handling Routines 356
Heap Control Procedures and Functions 357
Miscellaneous Procedures and Functions 357
Scalar Functions 358
Directory-Related Procedures and Functions (PC/MS-DOS) 358
Screen-Related Procedures and Functions 359
String Procedures and Functions 359
Transfer Functions 359
IBM PC Procedures and Functions 360
Basic Graphics, Windows, and Sound 360
Extended Graphics 361
Turtlegraphics 361

Appendix D Using Turbo Pascal with Other Borland Products 363
SideKick 363
The Notepad 363
Reminders 363
Documentation 364
Include Files 364
Interruptions 365
The Calculator 365
The ASCII Table 365
SuperKey 365
Using the Predefined Macros in TURBO.MAC 366
Fast Entry to Turbo 366
Turbo Pascal Templates 366
Saving Your Work 367
Other Macros 367
Other SuperKey Functions 367
Turbo Lightning 368

Appendix E Programming Style 369
Consistency: A Matter of Style 369
Indentation 369
Spelling 370

Parameters 371
Structures 372
Constants 374
Comments 374
Summing it Up 375

Appendix F Using Turbo Pascal with CP/M-80 and CP/M-86 Systems 377
Making Backups 377
Installation and BDOS Errors 378
Compiling to Disk 379
The Execute Option 380

Appendix G Common Questions and Answers About Turbo Pascal 381
General 381
16-bit Machines Only (including IBM PC) 385
IBM PC Only 386
CP/M-80 Machines Only 389

Appendix H Glossary 391

List of Tables

3-1 Switch Usage 13
3-2 Partial ASCII Character Set 18
6-1 Turbo Pascal Editor 49
7-1 Pascal Reserved Words 59
9-1 Complete ASCII Table 95
14-1 String Procedures and Functions 159
18-1 Write Parameters and Output 208

Introduction

Congratulations on joining the ranks of the over 500,000 owners and users of Turbo Pascal! You probably bought this book to learn to use Turbo Pascal, or to overcome a problem you're trying to solve with Turbo Pascal. Well, that's exactly what this book is for, and we don't think you'll be disappointed.

Only after you begin writing Turbo Pascal programs will you have an appreciation for the particulars of the language. For this reason, it's important that you use this book as a tutorial, and actually try entering and running the programs that we show and explain.

About this book . . .

If you're a novice computer user or programmer, you'll want to begin with Part I, "Turbo Pascal for the Absolute Novice." This part of the book will give you basic information about computers and programming, as well as a simple example of a Pascal program.

If you're a more experienced programmer, you may want to skim through Part I and go right to Part II, "A Programmer's Guide to Turbo Pascal." This section provides you with the basic elements of Pascal programming, taking you step-by-step through the different aspects of Pascal in general, and Turbo Pascal in particular.

Initially, you may want to write only simple programs. However, when you learn what you can do, you will probably want to learn more complicated things. For this reason, we've included Part III, "Advanced Topics in Turbo Pascal." This part of the book gives a brief introduction to such topics as nonlinear data structures, linked lists, stacks, typed constants, writing large programs, and more. Each of these topics could fill an entire book, so our explanation is really just an introduction.

Appendices A through H provide information about using other Borland products with Turbo Pascal, a glossary of programming terms, a complete set of syntax diagrams showing how each component of Turbo Pascal is used, the standard procedures and functions of Turbo Pascal, common questions and answers about Turbo Pascal, general guidelines for programming style, basic Turbo Pascal operations for CP/M-80/86 users, and a solution guide to the exercises provided throughout the book.

This tutorial comes complete with a disk filled with programs to run, and a ready-made library of routines to copy into your own programs. The routines are both timesaving and educational, especially if you tailor them to your needs.

This book is not intended to replace the *Turbo Pascal Reference Manual.* Rather, its goal is to help you grasp basic Pascal principles; the reference manual can then be used to give exact definitions of the Turbo Pascal implementation.

This book assumes that you have version 3.0 (or later) of Turbo Pascal. If you have an earlier version, you'll find a few differences, but they are not really significant until you get into advanced programming. (If you are that knowledgeable, you may want to consider buying an upgrade so that you can take advantage of Turbo Pascal's latest features and functions.)

For the sake of clarity, various typefaces have been used in this manual. The body of this manual is set in a normal typeface, an alternate typeface is used in program examples, and italics are used to mark reserved words in the main text as well as in programming examples.

We also refer to several different products in this manual; the following is a list of them and their respective companies:

- SideKick, SuperKey, and Turbo Pascal are registered trademarks and Turbo Lightning is a trademark of Borland International, Inc.
- IBM is a registered trademark of International Business Machines, Inc.
- Xenix and MS-DOS are registered trademarks of Microsoft Corp.
- WordStar is a registered trademark of MicroPro International, Inc.
- Kaypro is a registered trademark of Kaypro Corp.
- DEC is a trademark of Digital Equipment Corp.
- 8088 is copyrighted by Intel Corp.
- Z80 is a registered trademark of Zilog Inc.
- Zenith Data Systems is a registered trademark of Zenith Corp.

What We'll Teach You

Before we teach you how to program in Pascal, we'll give you a little background on computers and computer programming. We'll also encourage you to develop some habits that will keep your programs simple and unconfused, no matter how complex the problem you are trying to solve.

Next, we'll look at each part of a Pascal program and explain its function. We'll also look at different kinds of numbers and what they are used for. We'll teach you about a logical algebra called Boolean.

From time to time we will refer to the *Turbo Pascal Reference Manual* that comes with your Turbo Pascal disk. While the reference manual may seem a bit confusing to you at this time, it will prove to be a real treasure chest of information once you know more about Pascal.

Besides this tutorial and the *Turbo Pascal Reference Manual*, there are a few other things you'll need. First, you'll need a computer. The best way to learn Pascal is to actually do the things we tell you to do as you go through the tutorial. You'll also need a supply of blank disks (unless your computer has a hard disk). In addition, you'll need your Turbo Pascal program disk and the Turbo Tutor disk (the one that came with this book).

Once you've obtained all of these items, find a comfortable and quiet place with a large work area so that you can spread out your work and keep everything handy.

Finally, it will be helpful if you have a printer connected to your computer. You see, although Turbo Pascal will find most of your syntactical errors for you—mistakes where you leave out a punctuation mark or misspell a variable name—it won't find *all* errors. Every now and then, you will find yourself stumped by a logical bug—something in your program that won't let it work the way you think it should. A printed copy of your program can be of great value at times like these, so you can see and compare different parts of your program at the same time, or so you can compare a new version with an older version.

We sincerely hope you enjoy exploring Turbo Pascal. Good luck and have fun!

PART I

Turbo Pascal for the Absolute Novice

SECTION ART BY GEORGE BUCK

1 Getting Started with Turbo Pascal

Welcome! You are about to begin an exciting and rewarding learning experience. This book will teach you how to write computer programs in Pascal, a very powerful and flexible programming language. To be specific, you'll be learning how to program in Turbo Pascal, which is Borland International's name for its version of Pascal.

There is no easy way to learn a "foreign" language—the best way is to use it. The rewards will be well worth the effort; when you finish (and practice what you've learned), you'll be able to program in one of the best-selling languages in the history of computers. Moreover, you'll gain control over your computer, making it do what you want, in just the way that you want.

We don't know how much you know about computers, but we'll assume that you're like most people when they get their first computer—a bit overwhelmed and perhaps intimidated by all the fancy words that describe how to use your computer. If it's any consolation, computers are much easier to use now than they were a few years ago, and this trend will continue. Early "personal" computers consisted of many individual components that were not really intended to work together, and were poorly documented to boot. They were generally usable only by dedicated hobbyists and computer professionals. Today, you can (and probably did) buy an integrated computer system with all of its components, and perhaps several programs, ready to plug in and use.

If you don't know much about your computer, your most important information resource may be the dealer who sold it to you. If you don't have a dealer who can help you, a friend who already knows how to use computers can be a lot of help. And if you don't know anyone with a personal computer (or even if you do), a users' group can be invaluable. (Users' groups usually focus on a particular computer system or software program.) You can find out about users' groups by

looking through computer magazines and other trade publications, or by asking a computer dealer.

Having contacts such as these may prove indispensable in a time of need One missing piece of information can prevent you from doing anything with your computer. It could take hours or days to find that piece of information in a manual or book (if you can find it at all), whereas someone who's been through it before could possibly fill in these gaps for you in a matter of minutes.

With that said, we'll continue with the assumption that you know how to do certain basic functions with your computer: how to turn it on and off, how to "boot" the computer (load its operating system so that you can run other programs), copy files and entire disks, and other basic operations. And now we'll tell you something about the disk and its files.

USING THE DISK WITH THIS MANUAL

TUTOR.PAS and the .EX Files

In addition to the examples described throughout this manual, we have provided an online tutorial that describes and demonstrates important Pascal and microcomputer topics. The main program, TUTOR.PAS, uses several separate subprograms also contained on your disk (in .EX files). You can use TUTOR to look at the source code for each procedure, run the procedure, and then take a quiz on the programming topic demonstrated.

Main Menu

When you load TUTOR.COM (or compile and run TUTOR.PAS), the screen will clear and split into two windows. A menu of procedures will be displayed in the lower window. Each procedure listed on the menu demonstrates one or more Turbo Pascal features. To select an example or to see the quiz questions associated with each topic, follow the menu instructions listed on the function bar at the bottom of your screen.

Subprograms

After you select one of the examples on the main menu, the lower window will clear and TUTOR will load the procedure's source code from disk (therefore, when running TUTOR, all .EX files must be present on the logged drive). You can scroll through the source code in the lower window, run the procedure in the upper window, or take a quiz on the selected topic using the keystrokes displayed on the function bar at the bottom of your screen.

Modifying the Examples

Once you have studied an example using the TUTOR "shell," you may wish to make changes to the example in order to master its material. Assuming the example we are studying is called ARRAY1. EX, here is an easy way to modify and test the code:

- Copy ARRAY1.EX to ARRAY1.PAS rather than modifying the .EX file itself (this way the source code displayed when running TUTOR.COM will not contain your modifications). Load the compiler, then press W and specify ARRAY1.PAS as your work file. Press E to enter the editor and Ctrl Q C to move to the end of the file, then follow the instructions contained in the comment at the end of the file. (It will instruct you to move a few lines of text.) Make sure there is no MAIN file specified on the Turbo main menu (press M and ↵ at the main menu to de-select a MAIN file). Compile and run the short program by exiting the editor (Ctrl K D) and pressing R (for **R**un).

Note: Always use a backup copy of the Tutor disk and put the distribution copy in a safe place. This way if you modify the .EX files, you can easily restore them from your original disk.

Files on the Disk

A complete list of the files included with Turbo Pascal Tutor is contained in the READ.ME file on your distribution disk. Here is a brief description of the disk:

README.COM	Run this program to view the READ.ME file and learn about last-minute changes to Tutor and information about obtaining technical support.
TUTOR.PAS	To use this tutorial program, load the Turbo Pascal compiler and specify TUTOR as your MAIN file. Compile TUTOR by typing the sequence OCQC. When the compilation is complete, quit Turbo and type TUTOR to run the tutorial program.
.EX files	Contains sample procedures that demonstrate Pascal topics. These procedures are included with the TUTOR.PAS program and can be compiled and run by following the instructions outlined in the previous paragraph. You can also compile each .EX file separately by following the instructions at the end of each file.

.PAS files contain several larger Turbo Pascal programs:

- MANUAL.PAS contains all source code examples printed in the Turbo Tutor manual. Run README.COM to look for a a list of the examples in MANUAL.PAS, and information on how to run them.
- TYPING.PAS is a typing game that uses many Pascal structures and data structures (sets, arrays, records, strings, enumerated scalar types, booleans, and so on).
- FILEMGR.PAS is a disk-file manager that demonstrates many DOS utilities (DIR, COPY, RENAME, CHDIR, TYPE, and so forth).
- ANIMALS.PAS uses binary trees and "artificial intelligence" to play a guessing game.
- LISTT.PAS lists your Turbo Pascal programs. It comes with several .LPT files, which contain printer codes for specific printer brands and models.
- .INC and .LIB files are include modules for several of the .PAS programs.

Don't forget to refer to the READ.ME file on your distribution disk for the latest information about Turbo Tutor's programs and files.

That's about it for introductions. Now let's begin our tutorial by briefly mentioning some things about computers in general: the myths surrounding them, their capabilities, and their limitations.

2 Computers: Myth versus Reality

Turbo Pascal is a complete and powerful programming language that can be used to write almost any kind of program you might want to use, including databases, text editors, games, and a host of other applications. But, for now, let's begin by exploring some of the most common myths about computers.

There are a lot of misconceptions about what computers are and what they can do. In fact, there are so many misconceptions, we're going to devote this entire chapter to dissolving myths and presenting some important points. We'll look at some basic properties of computers and programming languages.

If you've just bought your computer, or if you're thinking about buying one, you may have some unrealistic expectations. Many people form their ideas about computers based on what they've seen in science-fiction movies or read in science-fiction books. Many of these otherwise harmless presentations give computers powers beyond those of mere machines—such as true intelligence and human emotions. We've seen at least one movie about a computer that learns about life by watching television, composes and performs songs, and even falls in love!

It sure would be nice if computers could do all of this. Then there would be no need for you to read this book to learn how to make your computer do things. You'd be able to point it at whatever you were doing and it would learn the task instantly and perfectly. But for the time being, you'll have to be satisfied with programming your computer by using the keyboard and a programming language like Turbo Pascal.

BEYOND THE MISCONCEPTIONS

What are computers *really* like, then? Well, beyond all of the misconceptions lie three basic properties shared by all modern computers:

- Computers are fast.
- Computers are stupid.
- Computers are literal.

Computers Are Fast

Computers are much faster than human beings at doing certain simple, repetitive tasks, like counting or adding numbers together. For instance, if you were to write a simple program (perhaps in Turbo Pascal) that told an IBM PC® to count from 0 to 10,000 by 1s, it could finish the job in about 0.26 seconds. Knowing this, you could easily imagine it counting to 100,000 or even 100 million in just a few seconds. (For comparison purposes, if you were to count a number every second, it would take you about three hours to reach 10,000.)

Computers Are Stupid

Computers are stupid—that is, they have no "common sense" or intuition. They cannot do *anything* without being told first.

Every single action a computer takes is based on an instruction given to it. Thus, a computer—the actual, physical *hardware*—is useless without sufficient *software* (that is, without the necessary commands to make it do useful work). If you were to turn on your computer hardware, and it had no software installed, it would do absolutely nothing. It wouldn't even know how to display characters on the screen or accept characters from the keyboard. This is why all modern computers come with some software built into their electronic circuits—enough to run BASIC, or to read other programs (such as the operating system) from a disk.

Once you have completed this tutorial, you will be able to give your computer instructions to perform new tasks—you will be creating your own software.

Computers Are Literal

Computers take every instruction they are given literally. They do exactly what they are told—no more, no less. This means that if there are mistakes in your instructions, the computer will do its best to do exactly what you told it to do. For example, if you were to write a program that told the computer to count from 0 to 10,000 by zeros, it

would count "0 . . . 0 . . . 0 . . ." very quickly (and, potentially, forever) until you somehow stopped it. A human being would quickly see that counting by zeroes accomplishes nothing, and would give up; the computer has no such wisdom.

The paradox here is that computers are simultaneously more and less powerful than we generally believe. There are some tasks that computers do very well, in times too short for us to perceive. Because of that, we often fall into the trap of thinking that a computer can do anything very well and/or very fast.

The bottom line is that computers aren't magic. They don't do *everything* well, and there may be some things you're hoping to make your computer do that it won't be able to. Remember, though, that there are some things that computers do very well indeed, tasks that are useful and/or entertaining. Most of all, you may discover (as many of us have) that there is a certain fascination in making that mass of wire and silicon do just what you want it to. Be forewarned, though: Programming can be psychologically addictive, and you may find yourself spending countless hours hunched over your keyboard, adding just one more feature or removing one last bug.

Well, enough about computer myths and facts. Just keep in mind that you must tell your computer exactly what you want it to do, and when and how you want it to do it. Fortunately for all of us, this process is now much simpler than it used to be—as you shall see in the next chapter.

3 Computer Basics

"THINGS ARE ALWAYS AT THEIR BEST IN THEIR BEGINNING."

—Blaise Pascal, *Lettres Provinciales*, No. 4

Now that we have exposed some of the more common misconceptions about computers, we will teach you some basic computer concepts—what they are, how their components interact, and so forth. It is by no means the final word on computer operation; rather, it is meant to be a brief, concise, up-to-date discussion on the modern microcomputer. In this chapter, we'll cover the following concepts:

- Computer hardware
- Digital data
- Computer software
- Characters

So let's get started—we have much to learn.

COMPUTER HARDWARE

Hardware refers to the equipment comprising a computer system. For the purpose of this discussion, we'll assume that your computer is one of those capable of running Turbo Pascal. This type of computer would consist of the following hardware:

- Central Processing Unit (or CPU)
- Memory (Random-Access Memory (RAM) and Read-Only Memory (ROM))
- Mass storage device(s) (floppy disk drive or hard disk drive)
- Input devices (keyboard)
- Output devices (display screen, printer)

Your computer may have more components, but this is a typical minimum configuration for effective programming.

Central Processing Unit

The CPU is the "brain" of the computer. It's really not very smart—it is capable of executing *very* simple instructions. It can do such things as get a number stored in a memory location, get another number, add the two numbers, and put the result in yet another memory location. The type of CPU determines the types of programs your computer will run. Some popular CPUs are the 8088© and the Z80® (which are usually used in computers running the CP/M® operating system) and the 8088 (used in the IBM PC to run PC-DOS or MS-DOS). We'll talk more about operating systems in a moment.

Memory

Read/write, *random-access memory* (RAM) stores programs and data while the computer is turned on. This type of memory is changeable and very fast. It is called random-access memory because the computer can read or write any part of it at any time (rather than having to access things in a particular order).

Another type of memory in your computer is called *read-only memory* (or ROM). ROM is used to store the instructions (programs) the manufacturer built into the computer; these are not alterable. For example, the IBM PC has BASIC stored in ROM so that you need not insert a disk to write BASIC programs (although you would need a disk or a cassette recorder to store your own program permanently).

Mass Storage

Computers have certain limitations; for example, they have a limited amount of RAM. The RAM in any computer can hold only a certain number of programs and a certain amount of data, not all the programs and data you would want to have available. Another limitation is that, in most cases, turning off the power to the computer will cause everything stored in RAM to be erased.

The solution to both of these limitations is called *mass storage*. Also referred to as secondary storage (to differentiate from memory), these peripheral devices can store large volumes of data. Mass storage on a micro is available in the form of a *disk drive*—something your computer must have to store programs and data. Thus, you can have many *floppy disks* (software medium), each containing different programs and data files.

Your computer may even have two floppy disk drives, or it may have a hard disk drive in addition to a floppy disk drive. Multiple disk drives let you have more programs and data available at any given moment, and a hard disk drive gives you access to huge amounts of data and will load and store that data many times faster than a floppy disk drive. For

this book's purposes, we will refer to mass storage as a disk drive unless it is important to distinguish between a floppy disk and a hard disk.

Input Devices

Your primary input device is your *keyboard*. You can also have other input devices, such as a mouse, a digitizing tablet, a modem, or a touch screen. The distinguishing characteristic of an input device is that it *sends* data to the computer.

Output Devices

Your primary output device is probably your *display screen*. When you type, the characters usually appear on the screen; however, they don't go directly to the screen—first they go to the CPU, where they are evaluated and then sent to the screen.

You may have other output devices in your computer system, for example, a printer or a modem. The main feature of an output device is that it *receives* data from the computer.

DIGITAL DATA

To oversimplify a very complex piece of equipment, think of your computer as a huge number of switches. Each switch has only two positions, on and off. When a switch is on, it lets electricity flow through a wire to another part of the computer, and when it's off, no electricity can flow from that switch. Let's call the *on* state of the switch a "1" and the *off* state a "0."

A single switch can control (or represent) only a single *bit* of information. But by arranging two switches side by side, four different things can be controlled or represented, as shown in Table 3-1.

Table 3-1

Switch 1	Switch 2	# Represented
OFF	OFF	0
ON	OFF	1
OFF	ON	2
ON	ON	3

By using eight switches arranged next to one another, you can control or represent 256 different things. Each switch you add doubles the number of possible states. This system of representing items by either a

0 or a 1 is called the *binary* (or base 2) *system*. The system of numbers you're most familiar with is the *decimal* (base 10) *system*.

Now let's look at how the computer's memory stores instructions and data. For the moment, think of RAM as a group of switches arranged in a matrix of 8 columns of switches by 64,000 rows of switches. Each of the 64,000 rows can store an 8-bit code representing a number, a character, or an instruction that the CPU will understand. (The CPU is "told" what operation to perform via a coded instruction from memory.)

As it turns out, 8 bits is a handy number for computers to work with, and so a group of 8 bits has been given a special name: a *byte*. The memory we just described (8 bits by 64,000 bits) is said to be a 64,000-byte memory. And since engineers often use the abbreviation *kilo* to mean 1,000, a 64,000-byte memory is often referred to as 64 kilobytes, or simply a 64K memory. If your computer contains 512 Kbytes of RAM, it has the equivalent of 8 columns of switches by 512,000 rows of switches.

The actual "switches" are microscopic electronic circuits capable of controlling the flow of electricity. They are contained in integrated circuits (also called *chips*) that are installed in your computer.

The CPU retrieves instructions stored in RAM by requesting the contents of a particular address in memory. The address defines the intersection of a row and a column. The binary (1s and 0s) code contained in memory passes over wires (or printed circuits) to the CPU. CPU instructions are codes that instruct the CPU to move data from one location to another, add, subtract, multiply, and divide data, and perform other very basic operations on data. This binary code is called *machine language*. (Some people like to write programs in machine language and then enter their instructions by entering the actual 1s and 0s required by each instruction and data element.)

If the instruction retrieved by the CPU happens to require data, the instruction tells the CPU to go to a particular memory address and get that data. If the instruction creates new data (say, by adding two numbers), it will instruct the CPU to place the new data in an empty (or unneeded) memory location.

COMPUTER SOFTWARE

So far, we've been trying to restrict our discussion to computer hardware. This is quite difficult, however, because software is so closely related to hardware. One without the other is useless. *Computer software* refers to instructions that the computer can read to make it perform some function. Software that is encoded in ROMs is sometimes referred to as *firmware*—sort of a compromise between hardware and

software. In fact, some of the messages displayed by an IBM PC when you turn on the power are the result of quite complex firmware. So, our original comment is true—without some type of software, your computer will do absolutely nothing.

In general, computer software can be divided into two broad categories:

- Operating systems
- Application software

Operating Systems

As a computer user (or a programmer, for that matter), you don't want to have to tell the computer how to accept characters from the keyboard, display characters on the screen, send characters to the printer in the correct format, write a byte of data to the disk, read a byte of data from the disk, and so forth. Indeed, if you had to tell the computer how to do all of these things, you'd never get around to the original problem you set out to solve.

Such menial and routine tasks are instead handled by an *operating system*. The operating system is the program that displays a prompt and allows you to type on your keyboard. The operating system knows that when you type a program name, you want to load that program into memory from a disk drive and begin executing the program. The operating system also knows how to do routine tasks such as copy disks, erase files, rename files, display a list of files, and perhaps keep track of the date and time.

The primary purpose of the operating system, therefore, is to perform some of the most basic functions you would expect your computer to do, in response to commands that you (or a program) issue. This frees you to concentrate on what needs to get done, rather than the nitty-gritty details of day-to-day operations.

Another vitally important function of an operating system is to hide the differences between different types of hardware from you and the software you run. For instance, the CP/M operating system runs on hundreds of computers of all brands, shapes, and sizes. Each of these computers may have its own unique combination of terminals, printers, mass storage devices, and other hardware. Yet, if you know the command to show a file on the screen (which, incidentally, is *TYPE*, followed by the name of the file), you can walk up to *any* CP/M system and make it show you a file. Similarly, Turbo Pascal will run on nearly *all* of these machines (though you may have to "install" it first), because the way that a program asks CP/M to perform a task (such as reading a file) is the same on all CP/M systems.

The operating system you run on your computer is largely determined by the type of CPU in your computer. For example, if your computer has a Z80 CPU, you will most likely use the CP/M operating system. (This is just about your only choice.) On the other hand, if you have an IBM PC (which has an 8088 CPU), you can run PC-DOS or MS-DOS, CP/M-86, Concurrent DOS, XENIX, and several others. You may also choose a particular operating system because of the *application programs* you'd like to run, or your preference for a certain command style.

Application Software

Application software generally consists of specific task-oriented programs. Some of these programs may come with your computer, while still others are available for purchase. Application software may also be written by you, your friends, or your company to solve a particular problem. Examples of application software are word processors, spreadsheets, database managers, Borland's SideKick, and communications software.

Assemblers, compilers, and *interpreters* are special kinds of application software used to translate programs written in a computer language, such as Pascal or BASIC, into a form your computer can run. Virtually all modern application software is created using one of these tools (including Turbo Pascal, which was written using an assembler). We will talk more about computer languages, assemblers, interpreters, and compilers in the next chapter.

CHARACTERS

When you type on your keyboard, or when your program displays information on your screen or printer, the computer is sending binary codes (1s and 0s) to input/output devices. Circuits inside keyboards or other input devices convert meaningful characters into codes, and circuits inside output devices (such as display screens) convert codes into meaningful characters.

Characters are the letters, numbers, punctuation marks, symbols, and in some cases, graphics components displayed by your computer's screen and depicted on its keyboard. The computer's definition of a character also includes many "invisible," or nonprinting, characters that are used as signals to the computer hardware rather than to display or print things.

The printing characters are those that appear on your keyboard. They include the upper- and lower-case letters of the alphabet (*A* through *Z*

and *a* through *z*), the digits 0 through 9, and the punctuation symbols (,./?()#&, and so forth). And, believe it or not, spaces (the characters you get when you press the space bar) are considered to be characters, since you can display and print a space.

Some computers have an extended character set, which is often specific to a particular manufacturer. Extended character sets may have graphics-type symbols that can be used to construct boxes, charts, and other simple graphic images, or they may have foreign-language character sets. You'll need to refer to your computer or terminal's documentation to learn what, if any, extended characters it has, and the methods by which you can use them.

Nonprinting characters (those characters that don't display anything on the screen) are also called *control characters*, performing a particular function when they are entered from, or written to, the terminal. Control characters can be entered from the keyboard by holding down [Ctrl] and pressing a letter of the alphabet. For example, you can issue a [Ctrl][G] by holding down the [Ctrl] key and pressing [G].

Many control characters have their own keys. For example, the [↵] key actually issues a [Ctrl][M] character, and the [⇆] key actually issues a [Ctrl][I] character. The effect of issuing a control character depends on the program your computer is running. For example, when running WordStar®, pressing [Ctrl][G] deletes the character the cursor is on.

Thus, your computer has printing characters (alphabet, digits, symbols, space, and perhaps graphics characters) and nonprinting characters (control characters), but does not store characters directly in its memory or on its disk; instead, it stores codes that *represent* the characters.

Now it would be a real mess if all computer manufacturers used their own codes for representing characters. But fortunately, all computer manufacturers follow a standard established by the Electronic Industries Association (EIA), called the American Standard Code for Information Interchange (or ASCII for short). (However, IBM mainframes use EBCDIC.) The ASCII character set is nothing more than a unique numeric code for 128 frequently used characters. When you type a character on your keyboard, the keyboard sends an ASCII code to your computer. When a program displays a character on your screen, the program is sending an ASCII code to circuits that convert ASCII codes into images of characters. And when a program prints a character on your printer, it does so by sending an ASCII code over the wires to your printer.

So, now you know what people are talking about when they refer to ASCII characters. Table 3-2 shows the printing characters of the ASCII character set. In Chapter 9, we'll show the complete character set

together with codes in several different number bases (binary, decimal, octal, and hexadecimal).

Note that there is a fixed relationship between the upper- and lower-case letters of the alphabet. Each lower-case letter's code is exactly 32 greater than its upper-case equivalent. This means, for example, that you can convert characters entered from lower case to upper case by simply subtracting 32 from their ASCII code (or vice versa, by adding 32). This is all you really need to know about characters to get started.

Table 3-2 Partial ASCII Character Set

Code	Character	Code	Character	Code	Character
32	(space)	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	del

We'll talk more about the complete ASCII character set and its uses later in this tutorial.

REVIEW

This chapter briefly discussed the basics of modern microcomputers. A modern microcomputer consists of a central processing unit (CPU), random-access memory (RAM), read-only memory (ROM), a mass storage device (disk drive), and a keyboard and screen. An operating system provides you with some basic operations and allows you to run application programs. When you tell your computer to do things, you do so by sending it characters in the form of ASCII codes. Your display and printer also work by receiving ASCII codes and converting them into characters.

This should be enough information to get you started. We are now ready for a discussion of programming languages in general, and Pascal in particular.

4 A Brief History of Programming

While an understanding of the history of computer programming isn't a prerequisite for using Turbo Pascal, we think you'll find this information both interesting and informative. It will also give you an appreciation of just how far computer technology has come in a very short time. However, if you'd like to get started with Pascal programming right away, skip ahead to Chapter 5.

IN THE BEGINNING . . .

In the early days of computers, there was nothing resembling a modern programming language. Programs were created by connecting wires inside the computers. Then someone had the marvelous idea of installing switches in place of the wires. (Many computers still have switches that can be used for entering small programs, but the majority of modern computers use other methods.)

To program by flipping switches, you had to convert everything to a number. (Since the switches had only two positions—on and off—each switch could represent only one of two digits, a 1 or a 0. We referred to this in Chapter 3 as a binary, or base 2, system.) You would enter one instruction by flipping up to 64 switches (representing one number or code), press another switch to enter the instruction, then repeat this process to enter another instruction. A series of lights corresponding to each switch indicated the contents of each instruction entered. In many cases, the result of running a program was simply a display of lights that had to be converted into a meaningful number. And the programmer was in big trouble if one of those light bulbs burned out!

Let's say you wanted to make the computer do one simple task: calculate the result of 2 + 2. You would have to convert these numbers to binary (0000000000000010 + 0000000000000010), enter the numbers, then enter the binary code to add the contents of the two memory

locations holding those numbers (perhaps a code such as 0001100011010001). You would then press the Run button, the lights would blink, and you would have your answer displayed in a row of lights. The correct answer would be 0000000000000100, which is the binary representation of the decimal number "4." If instead you got the answer "1," you probably inadvertently set a switch that told the computer to divide rather than add. You can imagine the complications of dealing with negative numbers and fractions.

A PROGRAMMING SHORTHAND

The practice of entering instructions as binary numbers (also called machine language) was far too tedious, time-consuming, and error-prone to allow productive work. Frustrated programmers soon devised ways to make the computers themselves perform this chore, and began using a shorthand—an English-like method of representing instructions. This shorthand was called *assembly language*.

Using assembly language, the programmer could enter a line such as "MOVE (1),(2)" (which might move the contents of memory location 2 to memory location 1), and the computer would do the "dirty work" of converting the programmer's shorthand into machine codes. The program that performed the conversion was called an *assembler*.

Assemblers are still widely used today. To program in assembly language, you need an assembler designed specifically for the CPU in your computer (Z80, 8088, and so on), and a good understanding of both advanced programming techniques and the computer hardware you are using. It also helps to have an inexhaustible supply of patience and a knack for detailed planning and organization. Assembly language programming is not for everyone, but many people prefer to program in it above all other languages—even though it is sometimes much easier to perform the same task using Pascal. (See Chapter 28, "Using 8088/8086 Assembly Language with Turbo Pascal.")

Assembly language allows the programmer to create the smallest, fastest programs possible for a given machine (which is why Turbo Pascal is written in assembly language). However, both the assembler and its assembly language programs can only be used with one particular CPU type. It is possible to spend months (and many thousands of dollars) writing an assembly language program for a particular computer, but to use it on another type of computer one would have to learn another assembly language and then spend a lot of time converting the original program to a new one for the other machine. This is

one reason why "high-level" programming languages, such as Pascal, were invented.

So, for the vast majority of us, the best way to write application software is to use a language like Turbo Pascal.

HIGH-LEVEL LANGUAGES

A *high-level programming language* is a programming language in which a single line of text may do the work of many lines of assembly language, saving the programmer much of the tedium of assembly language programming (another reason why high-level languages were developed). A high-level language also eliminates the need for the programmer to know many of the technical details of the machine on which the program will run. Most high-level languages "look" the same to the programmer, regardless of what computer is being used.

In theory, if you had two very different computers and had the same programming language available for each (the programs that allow the machines to understand the language are machine-specific), you could write a program in the language and it would run on both computers. This is called *portability*, and is a very important consideration, especially when much effort goes into the writing of a single program.

When the United States Department of Defense (DOD) decided that it wanted to run its programs on many kinds of machines, it began to look for a portable programming language. The language they chose was the "**C**ommon **B**usiness **O**riented **L**anguage," or *COBOL* for short. By using COBOL they found that the same program could be run on different computers without modifications. (They have since switched to using Ada.)

However, the ability to run the same program on two computers—even if the two computers have the same language—is not universal. Certainly, if you have a computer that has color, graphics, and sound capabilities, and another computer that has none of these, you wouldn't expect a program that used the features of the first machine to run properly on the second. For this reason, most languages have a set of standard instructions that will run on any computer. When features are added to a language to take advantage of a particular computer's special features, they are called *extensions*. If you wish to make a program truly portable, you must avoid using extensions.

COBOL was the first of many high-level languages developed. When scientists learned that computers could be helpful in their calculations, they invented a language specifically designed to help them process scientific formulas. They called it, appropriately enough, FORmula

TRANslator (or *FORTRAN*). Still widely used today, much of FORTRAN's popularity is due to the fact that IBM adopted it as the "official" language for its mainframe computers.

Other languages, with such unusual names as ALGOL, APL, SNOBOL, and LISP, also appeared on the scene. Each of these languages has special features that make it useful for a particular kind of work—PROLOG, for example, is widely used in artificial intelligence (AI) research, and APL is well-suited to some types of scientific and statistical work.

PROGRAMMING LANGUAGES AND MICROCOMPUTERS

When microcomputers came along in the early 1970s, they had very little memory and ran much slower than their larger "siblings." While it was possible to use FORTRAN and COBOL on them, these languages stretched the microcomputer's resources to its limit (while also being too complex for the average microcomputer user to learn).

A simpler language was needed, and the language that most microcomputer manufacturers chose was called *BASIC* (an acronym for "**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode"). Originally developed at Dartmouth University with funds from General Electric Corp., BASIC is relatively easy to learn, and is an excellent choice for solving simple problems.

However, BASIC is quickly outgrown as your programming needs become more sophisticated. It is not a convenient language to use when writing a large, complex program, or when many programmers are working on the same program at once. It is difficult to divide a BASIC program into many small parts and construct each part separately. And because BASIC discourages this "divide-and-conquer" approach to programming, many teachers of computer science believe that BASIC promotes bad programming habits. Pascal, as we shall see, began as one educator's answer to this problem.

AND FINALLY . . . PASCAL

Pascal is one of the newer programming languages. It was developed by a distinguished computer scientist named Niklaus Wirth in Zurich, Switzerland, in 1970–1971. Wirth based some of Pascal's concepts on other computer languages he helped develop, PL/1 and ALGOL. He also designed Pascal to teach his students how to program a computer effectively.

Experience has shown that good programming starts with defining the problem, breaking it down into small parts, and then writing com-

mands that tell the computer how to solve each of these smaller problems, one at a time. Pascal is a "structured" language, which means that it is conducive to writing programs in small pieces by following predefined steps. (COBOL was the first structured language.) Certain parts must be placed in certain locations within a program and must follow certain rules. You'll learn all about structured programming in later chapters.

Structured languages were once quite awkward, but have become more and more human- and problem-oriented. It may not be apparent initially, but as you learn to read Pascal programs, you'll find that the statements read very much like English sentences. The result is that the definition of the problem, the smaller parts that comprise it, and the ways in which the smaller problems are solved, are all very easy to see.

INTERPRETERS AND COMPILERS

Regardless of what computer language you are using, the computer (which "thinks" only in terms of 1s and 0s) has to translate the English-like words, which people understand, into machine language, which the computer's CPU understands. The programs that perform this translation can be divided into two broad categories: interpreters and compilers.

What is the difference between an interpreter and a compiler? Both convert a high-level language to one the computer will understand—the difference has to do mostly with *when* that conversion takes place.

Turbo Pascal and most other high-level languages are compiled. Most versions of BASIC, however, are interpreted. The difference is in how your programs are executed by your computer.

An interpreter is a program that "translates" your instructions *during* execution. You run the interpreter program every time you want to execute your application program. The interpreter program then reads each line of your program, one at a time, and performs the necessary functions.

A compiler, unlike an interpreter, translates your entire program, from beginning to end, into machine language *before* the program begins to run. Once the translation process is finished, the compiler is no longer needed; the application program can run on its own, usually directly from the computer's operating system prompt. Because the translation is not being done while your program is being run, a compiled program almost always runs faster than one which must be interpreted.

Which of these two approaches is best for a given problem? The answer depends on your point of view. Large programs written with

interpreters tend to require less memory than an equivalent compiled program, but often execute as much as 10 to 50 times slower. Programs compiled with a powerful, efficient compiler can approach the size and speed of assembly language; however, because the program must be recompiled each time a change is made, the speed with which you can test your program, find errors, fix those errors, and retest the program is greatly reduced.

An interpreted language, such as BASIC, is most efficient if you are in the process of testing and modifying your program. There is no need to wait for the program to recompile—at any time, you can type RUN and try your program out.

Turbo Pascal is as convenient to use as an interpreted language, unlike most "traditional" compilers. With traditional compilers, you must write your application program using a stand-alone text editor, then start up the compiler and give it the name of the file you want to compile. And once you've done that, you can go have lunch, since most compilers take several minutes to compile a moderately complex program. Then, when you return to link and execute your program, you may find mistakes (commonly referred to as *bugs*). If so, you'll have to restart your editor, reload the original program, and try to figure out what went wrong (known as *debugging*). Then, you must recompile the program and test it again. You will usually have to repeat this editing compilation testing process many times to finally produce a program that works the way you want it to.

THE TURBO PASCAL ADVANTAGE

Turbo Pascal is a compiler, but unlike most compilers, it doesn't require you to endure the long, grueling cycle just described. First, you can perform all of the functions necessary to write a program without leaving the Turbo environment. When you start Turbo, a menu appears on your screen, and all of the necessary program development functions can be summoned by typing a single key. Second, Turbo has a built-in editor (with user-definable commands so you don't have to learn a new editor). Third, the Turbo compiler is fast. This means that you can compile most programs in a matter of *seconds* instead of minutes. Fourth, once compiled, you can run your program to test it without leaving the Turbo environment. And finally, when errors are found, Turbo automatically re-enters the editor and points to the line of the program that it "thinks" contains the error.

The editing/compilation/testing process is so fast in Turbo Pascal that there is virtually no distinction between it and an interpreter-type language.

Turbo Pascal is the state of the art in powerful, easy-to-use programming environments. In the next chapter, you will learn how to prepare your computer for Turbo Pascal.

5 Getting Ready to Use Turbo Pascal

Before you can use Turbo Pascal, there are a few things you must do: write-protect your disks, copy your disks to make backups and working copies. This chapter explains how to do all of these things, and also covers how to use Turbo Pascal on single floppy disk and hard disk systems.

First, and foremost, *we strongly recommend that you write-protect your original disks before beginning to copy*, to guard against accidental erasure. The write-protect procedure you use will depend on the size of your disk—on 8-inch disks, you *remove* an adhesive tab from a notch to write protect; on 5-1/4-inch disks, you *place* a tab over a notch; on 3-1/2-inch disks you *slide open* the tab. After you have write-protected your disks, make a *backup copy* of your original Turbo Pascal disk. Next make a *working copy* of Turbo Pascal that is configured specifically for your needs. And finally, make a backup copy of your working copy, so that if something goes wrong with your working copy you can easily make a new one.

To begin, we'll assume you know a little something about your computer and its operating system (CP/M, MS-DOS, or whatever). Here is what we figure you already know:

- How to turn your computer on and off.
- How to "boot" your computer (that is, bring up your operating system).
- How to perform a few simple commands (DIR, COPY [or PIP for CP/M], FORMAT, and so forth).
- How to format a blank disk.
- How to make a "system" disk (one that will start your computer).
- How to copy files from one disk to another.
- How to delete files on a disk.

- How to use a hard disk if you have one (to create directories, user areas, volumes, or whatever is necessary to separate certain files from others on your system disk).

You don't need to know much more than that to start using Turbo Pascal. If you're not sure you know how to do these things, or if you're not that familiar with your system, take a day or two to play around with it. Get to know it. *Read your manuals.* Best of all, have someone who *does* know these things take the time to teach you the basics. You'll find it a lot easier to learn Turbo Pascal if you're not nervous about, or scared by, your computer.

Now, let's get started.

BACKING UP YOUR ORIGINAL TURBO DISK

The first thing you *absolutely must* do (if you have not done so already) is to make backup copies of your original Turbo Pascal disk and Turbo Tutor disk. Disks are easily damaged by many things: coffee, sneezes, dust, smoke, folding or bending, magnetic fields, and heat—not to mention program bugs and accidental erasure due to typing errors. Despite our best intentions, most of us will, at one time or another, fall prey to one of these disk-destroying perils. The following instructions will help you make these vital backup copies. (CP/M-80/86 users should refer to Appendix F, "Using Turbo Pascal with CP/M-80 and CP/M-86," for additional system-specific operations.)

Making Backups: MS-DOS and IBM PC-DOS

If your computer is running PC-DOS or MS-DOS, you can use the DISKCOPY program (normally found on your DOS disk) to back up your originals onto two new disks. After using the appropriate write-protect procedure for your disk, put the original in drive A: and the blank disk in drive B:, then type the command:

```
DISKCOPY A: B: ↵
```

Note that this command assumes you have two disk drives, named A: and B:, and asks DOS to copy all the information from one to the other. If you don't actually have two disk drives, don't worry—DOS is usually smart enough to know this, and will ask you to swap the original disk and the copy until all of the information has been transferred. Remember that when DOS asks for the "disk for Drive A:," it is referring to the original. Likewise, the "disk for Drive B:" refers to the new copy. Also, if your new disk is not formatted, DISKCOPY will automatically format it to the same specifications as the original disk. If the new disk is already formatted, DISKCOPY will skip this step.

If you have problems using the DISKCOPY utility, or can't find it on any of your disks, you can use the FORMAT command to format a disk and the COPY command to copy all of the files from the original to the backup disk. To do this, type the commands:

```
FORMAT B: ↵
COPY A:*.* B: /V ↵
```

Again, if you have only one disk drive, the computer will usually ask you to swap disks to complete the copy. (In case you were wondering, the "/V" at the end of the previous command causes DOS to reread what it has just written, to make sure that the copy came out okay.) If you are not sure how to make a copy on your system, we recommend that you consult your manual, your computer dealer, or your "resource person" for help. Once you have created your backup copies of the Turbo Pascal and Turbo Tutor disks, we suggest that you place your originals in a safe place, to be used for emergencies only. You will use your backup copies of these disks to create your working disks and to work through this tutorial.

GETTING READY TO WORK

The next set of steps will result in making a system (bootable) disk with Turbo Pascal on it, installing Turbo Pascal on your system, and perhaps erasing files that you don't need so that you have maximum space for your programs. (If you have a hard disk, you will want to skip the information about formatting and making a system disk, and instead read the section "Using a Hard Disk.")

Making a Turbo System Disk

If you are using a system with one or two disk drives, the first thing you want to do is make a *bootable* copy of your Turbo Pascal backup disk. A bootable disk is one that has a copy of your operating system on it, so that it will "boot" the computer when you turn it on. This is much more convenient than having to insert your operating system disk to start the computer, then switching disks to use Turbo Pascal. Follow these steps to create your working disk:

1. Create a system (bootable) disk; depending on your operating system, this can be a one- or two-step process. If you're using PC-DOS or MS-DOS, the command to do this is

   ```
   FORMAT A:/S ↵
   ```

   This will format a disk and place the operating system on it.

2. Copy all the files on your Turbo Pascal backup disk (the Turbo Pascal disk, not the Turbo Tutor disk) to the new system disk.

You may not need all the files on the disk, but for now, it's easier to copy them all.

3. Check to make sure the disk boots properly.
4. Now put your Turbo Pascal backup disk away for future use. You will probably want to use it to make new copies of files you delete from your working disk (to make space for your own programs), or to make a new working disk.

Installing Turbo Pascal

Now that you have a system disk with Turbo Pascal on it, you are ready to install Turbo Pascal. By this we mean set up Turbo Pascal for your particular computer hardware so that it can work as efficiently as possible with your terminal or keyboard and display.

Turbo Pascal is designed to run on many different systems. Some of the files you copied to your working disk—TINST.COM, TINST.MSG, and (on some versions) TINST.DTA—can help you install your version of Turbo Pascal. Once you're done, you can delete these files, since you won't need them anymore (unless you change your hardware configuration).

If you're using an IBM PC or compatible (such as the Compaq®), chances are you can run Turbo Pascal without doing any installation at all. On the other hand, if you're running under CP/M, then you'll almost certainly have to run the installation program to tell Turbo Pascal how to display things on the screen.

Another reason you might want to run TINST is to change the editing commands used by Turbo Pascal. The built-in editor uses commands similar to those found in WordStar, a popular word-processing program. Chapter 6 covers the editor commands in detail; however, if you want to change any (or all) of the editing commands, TINST will help you. We've given you this option so that you can customize the editor commands to be the same as commands in other programs you frequently use.

One last reason to use TINST: If you're using PC-DOS or MS-DOS version 2.0 or later, and especially if you're using a hard disk, you can define a path name for TURBO.MSG, the error message file. Let's say you've stored both TURBO.COM and TURBO.MSG in the directory C:\UTIL\PASCAL, and you're currently logged into the directory A:\SOURCE\PASCAL. If you run Turbo Pascal (using the command C:\UTIL\PASCAL\TURBO), Turbo will think that the error message file (TURBO.MSG) is in A:\SOURCE\PASCAL (the currently logged drive and directory). However, if you've previously used TINST to define the "Msg Path"

as C:\UTIL\PASCAL\TURBO.MSG, then there's no problem. (If you don't understand any of this, you probably don't use directories and don't need to worry about this.)

Using TINST With Non-IBM PC Systems

To install Turbo Pascal on a CP/M, CP/M-86, or generic MS-DOS system, insert your Turbo Pascal work disk into a drive (you don't need to do this if you have a hard disk), make that drive the default drive, log the drive in (CP/M and CP/M-86 systems only; see Appendix F), and type:

```
TINST ↵
```

to start the Turbo Pascal installation program. You will see a menu that looks like this:

```
                 Turbo Pascal Installation Menu.
Choose installation item from the following:
[S]creen installation  !  [C]ommand  installation  !  [Q]uit

Enter S, C, or Q: _
```

While you may eventually want to perform the **Command** installation function (which changes the keys used to control the Turbo editor), for the moment you will only want to use the **Screen** installation function. Press S to invoke this option.

TINST will respond by giving you a list of terminals and personal computers to choose from. It should look something like this:

```
Choose one of the following terminals:
1) ADDS 20/25/30            17) Qume
2) ADDS 40/60               18) RC-855
3) ADDS Viewpoint-1A        19) Soroc 120/Apple CP/M
4) ADM 3A                   20) Soroc new models
5) Ampex D80                21) SSM-UB3
6) ANSI                     22) Tandberg TDV 2215
7) Hazeltine 1500           23) Teleray series 10
8) Hazeltine Esprit         24) Teletex 1000
9) Kaypro with hilite       25) Televideo 912/920
10)Kaypro, no hilite        26) Texas Instruments
11)Lear-Siegler ADM 20      27) Visual 200
12)Lear-Siegler ADM 31      28) Wyse WY-100/200/300
13)Liberty                  29) Zenith
14)Morrow MDT-20            30) None of the above
15)Osborne 1                31) Delete a definition
16)Otrona Attache
```

If you see the name of your computer or terminal on this list, your job is simple: type the number corresponding to your computer/terminal, then press ↵.

If you don't see the name of your terminal on this list, you must either find out which of the terminals listed is most similar to your terminal

or computer, or give TINST precise information about the special characters needed to control your particular screen.

Since it's much, much easier to find a terminal similar to yours on the list than to type in the required information, you may want to try the former approach first. Here are some guidelines to help:

- Frequently, the documentation that comes with a terminal will say that the terminal *emulates* another terminal (which may be on the list). If so, you can use the menu selection for the terminal your system emulates.
- If your terminal is a DEC™ (Digital Equipment Corp.) VT-100, or emulates one, you can use the ANSI menu item. This item will also work on terminals described as "ANSI-compatible."

  If your terminal is a DEC VT-52, or emulates one, use the Zenith menu selection. (The Zenith Data Systems® terminals use an enhanced version of the VT-52 command set.)
- If you see an item on the list that is not your terminal but is made by the same manufacturer, chances are the control characters will be the same. For instance, the DEC Rainbow computers use the same control characters as the DEC VT-100 terminal (the ANSI menu selection). Try this menu selection and see if it works.

If all else fails, it is probably best to consult your local computer "guru" for help. He or she may have to create a new item on the menu by selecting "None of the above" and entering the codes specific to your system, or she may be able to get information on which of the existing selections will work.

Appendix L, entitled "Installation," in the *Turbo Pascal Reference Manual* gives a complete description of how to respond to the questions asked by the "None of the above" selection. The "delete a definition" selection on the menu is used to remove an undesired terminal definition from TINST's file of terminal information. Since it never hurts to keep extra definitions around, you will rarely (if ever) need to use this option.

Once you have set the correct terminal type, the rest is easy. TINST will ask one more question:

```
Hardware dependent information
Operating frequency of your microprocessor in MHz (for
delays): 4
Change to: _
```

Most CP/M-80 systems run at 4 MHz, so you can just press [↵] to answer this question (indicating no change). If you have an especially old system, or if you have a high-speed system that uses the Z80B or Z80H microprocessor, you may need to specify a higher or lower number. CP/M-86 and MS-DOS systems run at various speeds. Refer

to the documentation that came with your computer if you are not sure (usually a section labeled "Specifications" will list the speed). (Note: It is *not* fatal, or even much of a problem, if this number is not specified correctly. It is used only to time delays in your programs.)

Now press Q to exit TINST, and you can begin to use Turbo.

Using TINST on the IBM PC and Compatible Systems

On the IBM PC and compatible systems, there is usually no reason to use TINST (at least at first), since Turbo is already configured to work on this system. However, you may want to use it if

- Your computer has a graphics display board that is subject to "snow." By using TINST, you may be able to improve your screen's performance.
- You want to change the editor commands. Since the Turbo editor uses the IBM PC's arrow and movement keys, it's unlikely that you will need to do this right away.
- You want to change the path Turbo uses to locate the file TURBO. MSG. This is generally useful only on hard disk systems where TURBO.MSG may reside in a different directory than the text of your program.

The procedures for doing each of these things are described in the *Turbo Pascal Reference Manual* (Appendix L, entitled "Installation").

Once you have installed Turbo, and are satisfied that it is working correctly (if you can follow the instructions in the next chapter, it is working correctly), feel free to delete all TINST files (TINST.*). This will free up space on your working disk. You can go back to your copy of the master disk if you need to use TINST again.

Additional Files

There are a number of other files that may also be on your working disk. You can keep them or delete them as you please. Here's a brief list of them, with an explanation of what they do.

TURBO.MSG. This file contains the compiler error messages. If you delete this file, any error found during compilation will be identified only by a number. You must then look up the specific error in the *Turbo Pascal Reference Manual* (Appendix E, entitled "Compiler Error Messages"). We recommend you keep this file on your working disk until you absolutely must use the disk space for something more important.

TURBO.OVR. You'll see this file only if you're running under the CP/M-80 operating system. This file lets you execute programs from

within the Turbo Pascal menu system. If you delete this file, you'll have to exit to your operating system prompt to run programs. We recommend you keep this file unless disk space is a problem.

GRAPH.P and **GRAPH.BIN**. These files only appear in Turbo Pascal version 3.0 or later for the IBM PC. They have the declarations and object code for special graphics routines. Again, if you have enough disk space, keep them.

.PAS files. Your disk will contain several Turbo Pascal sample programs. Some versions have more sample programs than others. At the very least, you will have the CALC files, which provide a complete source for a small spreadsheet program. These are good files to look at and play with. You might also want to print them for later study. You can delete them when you choose without affecting Turbo Pascal's operation.

READ.ME. This file has any updates, errata, notes, and so forth, that didn't get into the manual. This is another good file to print, after which you'll probably want to delete it.

Other files. There might be additional files on your disk. Some of these files are necessary for system operation (such as COMMAND. COM on an MS-DOS system disk). If there are other files you don't feel you'll need, you can go ahead and delete them. If it turns out that you need them, you can retrieve them from one of the copies you made.

ANOTHER BACKUP

What? Another backup? Yes, you need to make one more copy, and there's a good reason for it. So far, you have made a working system disk with Turbo Pascal and other assorted files on it. You've done any necessary installation, and you're all set to go to work. If your working disk gets destroyed, you'll have to reinstall Turbo Pascal by going through TINST again, once more deleting unnecessary files. On the other hand, if you take a moment now to make a copy of your working disk, you'll be able to restore it in a matter of minutes.

So, take another blank disk and make an identical copy of your working system disk using DISKCOPY for MS-DOS. When you're done copying, save that copy along with your original Turbo Pascal master disk.

Now you should have a master Turbo Pascal disk and a master Turbo Tutor disk filed away. You should also have a copy of the Turbo Pascal master disk and a copy of your Turbo Pascal work disk filed along with the first two. And you should now also have a ready-to-use, working, bootable copy of the Turbo Pascal disk and a copy of the Turbo Tutor disk.

USING A SINGLE DRIVE SYSTEM

Unlike almost any other high-level language compiler, Turbo Pascal can run very well on a computer with a single disk drive (such as a very basic model of the IBM PC or some of the Morrow Designs systems). The only file you absolutely must have on your single working disk is TURBO.COM, which is less than 40K in size. Assuming the disk drive holds 360 Kbytes and the operating system takes up about 40 Kbytes, then you have around 280 Kbytes for all your source and object (compiled) files, as well as for any of the auxiliary files mentioned earlier. As you go through this book, you can copy the .PAS files associated with the current chapter from the Turbo Tutor disk to your working system disk, then erase the file when you no longer need it.

USING A HARD DISK

If your computer has a hard disk, free space is probably not a concern for you. If you are using CP/M, you will probably want to copy both the Turbo Pascal and the Turbo Tutor disks directly into user area 0 on the hard disk.

If you are using MS-DOS, you can create one subdirectory with all the Turbo Pascal files in it, and another subdirectory with all the Turbo Tutor files in it. Then use TINST to set the Msg Path to the Turbo Pascal subdirectory. (You may also want to use the MS-DOS PATH command to make your computer look in the right directory whenever you type Turbo. See your *DOS Reference Manual* for information about creating subdirectories and using the PATH command.)

Well, you've done it! You're now ready to start using Turbo Pascal and the Turbo Tutor examples. Next, you'll learn how to start Turbo; use the editor to create, look at, and modify programs; and use the menu selections to compile and run a program.

6 Using Turbo Pascal

You should now have your original Turbo disks safely filed away and have your working copies ready to use. Now the excitement begins! You might want to get a sheet of paper and a pen or pencil so you can take notes as you go.

In this chapter, you will sit down at the computer and go through all the steps required to write, compile, and run a simple program. Are you ready? Okay, let's get started.

STARTING TURBO PASCAL

The first step is to get the Turbo Pascal main menu on your screen. All you have to do is follow these easy steps:

1. Take your Turbo Pascal disk (the working copy you made in the previous chapter) and put it into your disk drive. (If you have more than one disk drive, put it in the one designated "A:" by your computer; if you have a hard disk, skip this step entirely.)
2. Turn on the power (or, if it's already on, *reboot* your computer by whatever means you would normally use).
3. Wait for the operating system prompt. It will look like one of the following:

   ```
   A:_
   A>_
   A0>_
   A\>_
   A\:_
   C:\>_
   (or some other letter, if you have a hard disk)
   ```

   Your prompt may look somewhat different, depending on your computer and how it has been set up. The important things to look for are a letter indicating which is the logged disk drive and a cursor (an underline or a box, either solid or blinking) indicating that you can now type commands.

4. Type:

   turbo

   and press [↵]. The [↵] key may be marked Return on your keyboard, or it may look like this: [↵] or [CR]. (If you have a hard disk, switch to the correct volume, directory, or user area before typing turbo. For example, if you have an IBM PC with Turbo Pascal in a directory named TP, type CD\TP and press [↵], then type turbo and press [↵].)

   In a moment, the following will appear on your screen:

```
TURBO Pascal system Version 3.01A
PC-DOS
Copyright (C) 1983,84,85 BORLAND Inc.


Default display mode

Include error messages (Y/N)? _
```

   Your display may look a little different if you have a CP/M version of Turbo Pascal, but the important information will still be there. (See Appendix F, "Using Turbo Pascal with CP/M-80 and CP/M-86 Systems.")

5. Notice that Turbo is asking you if you want to include error messages. If you answer no, you will get an error code number when an error in your program is found, which means you will have to look it up in the *Turbo Pascal Reference Manual*. Since you are just starting out, we suggest that you answer yes. To do so, press [Y]. This will tell Turbo Pascal to display error messages that are self-explanatory.

   After you answer this question, the Turbo Pascal Main Menu will appear on your screen. It will look like this (this is a PC/MS-DOS screen):

```
Logged drive: A
Active directory: \
Work file:
Main file:

Edit Compile Run Save

Dir Quit compiler Options

Text: 0 bytes
Free: 62024 bytes

 >_
```

   Again, the exact appearance of this screen depends on which type of computer you are using to run Turbo Pascal (see Appendix F for CP/M-specific information).

Main Menu Overview

The main menu provides you with commands and information. Command names have their first letter capitalized and highlighted. To select a command, press the appropriate (capitalized) letter. Information that will be displayed includes the name of the currently logged disk drive, the active directory (in MS-DOS versions), the name of the work file, the name of the main file (if used), the number of bytes of text in your defined work file, and the number of bytes of memory available in the computer for the text editor. We'll cover each of these things as we need them.

Rather than go into how to use each command in detail (the *Turbo Pascal Reference Manual* does that), we're going to teach you how to use some of the commands by having you enter, compile, and run a program. To do these things, you'll need to know how to use the following main menu selections:

- Work file
- Edit
- Run

Let's begin.

Choosing a File Name

You must first name the file you are going to work with. There are two ways to specify a work-file name. We'll use one easy way, and then tell you an even easier method. Do these steps now:

1. Press W to select Work file from the selections on the main menu. The screen will display:

   ```
   Work file name:_
   ```

2. Type the name that you want to give your first program. For this exercise, type the name:

   ```
   first
   ```

   After you type the name and press ↵ the screen will display:

   ```
   Loading A:\FIRST.PAS
   New file

   >
   ```

Now, let's pause for a moment and examine a couple of things. First, notice that Turbo Pascal automatically added the file name extension .PAS onto the end of the file name you entered. This extension identi-

fies Turbo Pascal source programs on your disk. (You could have entered your own extension and it would have been used instead, but there is generally no reason to do this.)

Second, notice the words "New file" that appear on the screen. This means that Turbo Pascal did not find a file by the name of FIRST.PAS on the currently logged disk (and directory), so it is creating a new file by that name. If it *did* find a file by that name, it would load it and then wait for you to press E (for **E**dit) before displaying it on your screen.

Finally, notice that the main menu has not changed. One thing you need to know is that the main menu is not updated every time you type a command. In fact, if you continue to enter commands, the menu will scroll off the top of the screen, and you won't be able to see it. Fortunately, there is a simple way to bring back the main menu and/or to update it with new information. The way to do this is to press ↵. Do this now, and watch what happens.

Also note that when the menu is redisplayed, your work-file name is placed in its proper place next to the **W**ork file command. So remember, whenever you think the main menu should be on the screen and it isn't, or whenever the main menu does not contain the information you think it should, press ↵.

Now that you have specified a file name and redisplayed the main menu, let's talk about an easier way to specify a file name. The easier way is to select **E**dit when no work-file name is specified. Turbo Pascal is smart enough to know that there is no file specified and asks you to enter a file name. You can try that the next time you run Turbo.

USING THE EDITOR

You could write Turbo Pascal programs using any ASCII-type text editor, but there's really no reason to leave the Turbo environment to work on your programs. The editor that comes with Turbo Pascal is ideal for writing Pascal programs, and using it will make your work sessions much less tedious.

Now that you've specified a work-file name, you are ready to enter the editor. To do this, press E. Since this is a new file, the screen will be clear except for the following:

```
Line 1    Col 1   Insert    Indent  C:FIRST.PAS
```

(If you had supplied the name of an existing file, pressing E would have brought up the first few lines of that file on your screen.) This screen is where you will spend many, many hours exercising your creativity and expertise. Think of it as an artist's canvas—your creations are limited only by your skill, your knowledge, and your imagination. The line at the top of the screen is a status line. It tells you

some important things about what you are doing. The first two entries tell you the position of the cursor in your file. The word "Insert" tells you that you're in insert mode (which means what you type will push existing text to the right), and the word "Indent" tells you that indent mode is on (more about this later). The last entry on the status line shows the name of the file you are editing. The remainder of the screen is blank, and this is where your program will go.

Before you actually start typing your program, let's spend a little time learning how to move around the screen. First, press the ↵ key about ten times. This will put several blank lines in your file. Notice how the cursor moves down as the status line continually displays your current position.

Now, how do you go back to line 1? There are several ways, but the first one we're going to use works on all computers with any version of Turbo Pascal. To move up, hold down the Ctrl key and press the E key. Each time you press E, the cursor will move up one line.

To move down again (this time without inserting new lines), press Ctrl X (pressing Ctrl X means to hold down the Ctrl key and the X key simultaneously; we'll represent all control commands in this manner from now on). Each time you press Ctrl X, the cursor moves down one line. However, once you reach the spot where you stopped pressing ↵, the cursor will no longer move down. This is because the Ctrl X command does not insert lines—it only moves among the existing lines in the file.

Next, let's try moving the cursor to the right. To do this, press Ctrl D. Each time you press Ctrl D, the cursor moves one space (or one character) to the right.

After you've moved to the right a few spaces, try moving to the left. To do this, press Ctrl S until you reach the left margin. Each time you press Ctrl S the cursor moves one space (or one character) to the left.

None of these cursor movements will erase text on your screen. They serve only to move the cursor.

You may think that it will be difficult to remember these cursor movement commands, but take a look at the following diagram:

```
  E
S   D
  X
```

This diagram shows the approximate location of the keys with respect to one another. Notice that they form a diamond shape, with each point corresponding to the direction the cursor will move when used in conjunction with the Ctrl key: Ctrl E moves up, Ctrl X moves down, Ctrl S moves left, and Ctrl D moves right. Borland chose this command structure because WordStar by MicroPro, the most popular word-processing program today, uses these same keys for cursor

movement. If you are one of the many people who use WordStar regularly, Borland's command structure should be second nature to you.

WordStar developed this cursor-movement structure because at its (WordStar's) inception, the only thing keyboards had in common was the control key. Now that many keyboards have special keys, WordStar (and Turbo Pascal) can use these as well. If you have an IBM PC, try using the arrow keys on the right side of the keyboard to move the cursor around. You'll see that they perform the same control-key combinations indicated previously (use the keys that are most comfortable for you).

When you're done experimenting with moving around the screen, move the cursor back to the top, to Line 1, Column 1. You can do this by moving one space at a time, or by using the QUICK command: Ctrl Q R. For those of you who are interested, there is a complete list of editor commands at the end of this chapter. Now, you're ready to start typing.

WRITING THE PROGRAM

This section is not really about learning how to write programs, rather, it explains how to enter and run a program. Therefore, we don't expect you to completely understand what we are doing here. Concentrate instead on the methods; we will show you exactly what to type.

Typing the Program

Begin by typing the following program exactly as it is shown (punctuation is very important, but spacing is not). If you make a mistake, use the cursor movement commands, and/or the backspace and delete keys to make changes. If you get really confused, you can erase the entire line the cursor is on by pressing Ctrl Y. Now, type the following program, substituting your name for the blank line:

```
program MyName;

begin
  ClrScr;
  Writeln('Hello world, my name is ________________')
end.
```

This is a complete Turbo Pascal program. In a moment you'll compile and run it, but first let's take a look at what it does.

The first line of the program starts with the word **program** and identifies the program by name; this line ends with a semicolon. The next line starts with the word **begin**, which signals the beginning of

the program. Next, *ClrScr* clears the screen when the program runs; this line also ends with a semicolon. Then, the *Writeln* command displays on the screen the text enclosed in parentheses and single quotation marks. Finally, the word **end**. (with the period) signals the end of the program. (Chapter 11 and Appendix E discuss more about programming style and structure.)

Compiling and Running the Program

At this time, you should have your program typed and displayed on your screen. To compile and run the program, you must return to the main menu. To return to the main menu, press Ctrl K D, then ↵. Now the main menu will appear on your screen again. Notice that the display of text and free space has changed slightly.

There is a **C**ompile selection on the main menu, but we'll use a shortcut. Instead of selecting **C**ompile, select **R**un by pressing R. Do this now, and keep a close eye on the screen.

You'll see some messages flash briefly on the screen, and then your program will begin running. This is what has happened:

1. Turbo Pascal looked at your program and realized it wasn't compiled.
2. Turbo Pascal then compiled your program because it cannot run an uncompiled program (it is not an interpreter).
3. Turbo Pascal then executed your program as soon as it finished compiling it.
4. Turbo Pascal returned you to its prompt (⟩), awaiting further instructions.

If you did your job correctly, you will see the screen clear and your message appear at the top. If you have made a mistake that Turbo Pascal can find, it will display an error message and return you to the editor to make corrections. After correcting the file and pressing Ctrl K D to exit, you can then press R to compile and run the program again.

Saving Your Source Program

Your source program (the text version of your program that you just entered) exists only in your computer's memory. If you turn off the power, reset your computer, or exit from Turbo Pascal, your work will be lost. To save the source program, select the **S**ave option from the main menu by pressing S.

When you select **S**ave, Turbo creates a disk file containing your program text. You can save your file whether or not the main menu is on

your screen, as long as the Turbo prompt (›) is there. If you feel more comfortable seeing the main menu before performing an operation, press [↵] before you select **S**ave.

After you save the file, it remains in memory as well. You can continue making changes to it as you wish, but if you do, remember that you must save it again to avoid losing your changes.

Saving Your Compiled Program

After you select **R**un, Turbo compiles your program into your computer's memory and runs it. The compiled program is only in memory and will be lost when you turn off the power. This is really not a problem since you've saved your source program and can load, compile, and run it again in a matter of seconds. However, that's not convenient. It would be better if you could save your compiled, executable program and run it directly without using Turbo Pascal.

Well, you can! To do this, make sure the main menu is on your screen, and then select compiler **O**ptions (note that the letter that selects this command is *O*, not *C*). (Those of you with CP/M machines should refer to Appendix F for more information.) When you select compiler **O**ptions, the following will appear on your screen:

```
compile->Memory
         Com-file
         cHn-file
command line Parameters:

Find run-time error  Quit

>
```

While this may look complicated, there are really only two selections you should be concerned with right now. They are the **M**emory selection and the **C**om-file selection. To create an executable program on your disk (as opposed to compiling into memory), you must select **C**om-file by pressing [C]. When you do, the screen will change slightly to display additional information, as follows:

```
         Memory
compile->Com-file
         cHn-file
minimum cOde segment size:     0000 (max 0D28 paragraphs)
minimum Data segment size:     0000 (max 0FDC paragraphs)
mInimum free dynamic memory: 0400 paragraphs
mAximum free dynamic memory: A000 paragraphs

Find run-time error  Quit

>
```

Again, this may seem confusing right now, but all you're concerned with is the selection to quit. Do this now by pressing [Q]. After you press [Q], the main menu will return.

You have compiled your program in memory; here's how to compile it to a disk. Select Compile from the main menu by pressing [C]—do this now. Your screen should now look something like this:

```
Logged drive: A
Active directory: \

Work file: A:\FIRST.PAS
    Main file:

Edit     Compile  Run   Save

Dir      Quit  compiler Options

Text:   111 bytes
Free: 61913 bytes

>

Compiling--> A:\FIRST.COM
6 lines

Code:        0007 paragraphs (    112 bytes), 0D21 paragraphs
free
Data:        0002 paragraphs (     32 bytes), 0FDA paragraphs
free
Stack/Heap: 0400 paragraphs ( 16384 bytes) (minimum)
            A000 paragraphs (655360 bytes) (maximum)

>
```

Again, this display has more information than most of you need. The important thing is that the job is done. You now have a file on your disk called FIRST.COM. You can now exit from Turbo Pascal and run this program just like any other program you buy.

Finishing Up

You have started Turbo Pascal, selected the editor, typed in a program, compiled and run the program from memory, saved the source code, and compiled to a .COM file. You've learned a lot, but there's one more thing you need to know: how to quit Turbo Pascal.

When you're ready to quit, select Quit from the menu by pressing [Q]. Normally, you'll be returned immediately to your operating system prompt. However, if you've forgotten to save your source file, the following message will appear before you exit:

```
Workfile FIRST.PAS not saved.  Save (Y/N)? _
```

You can choose either to save the file or abandon it. After answering this question, the operating system prompt will appear.

REVIEW

There are many more selections, options, and commands available in Turbo Pascal for your use, but you can get by with what you've learned in this chapter for quite some time. With the information presented you should be able to do the following:

- Start Turbo Pascal from your bootable work disk or hard disk.
- Specify the name of a work file.
- Enter the editor and type in a program.
- Move the cursor to various parts of the screen to edit your program.
- Exit from the editor and return to the main menu.
- Compile your program into memory and run it.
- Save your source program on disk.
- Compile your program into a .COM file on disk.
- Exit from Turbo Pascal and return to your operating system prompt.

You've jumped the highest hurdle—you've actually used Turbo Pascal to write your first program. Now all you need to do is to learn how to design and implement your own Turbo Pascal programs. And that's the subject of the rest of this book.

Table 6-1 lists all of the editor commands we referred to earlier in the chapter.

Refer to Appendix L in the *Turbo Pascal Reference Manual* for instructions about how to customize editor commands to suit your needs. We recommend that you set up the editor to work the same as the spreadsheet or word processing program you use most often. If you have function keys available, you may want to install the editor to take advantage of these.

Table 6-1 Turbo Pascal Editor

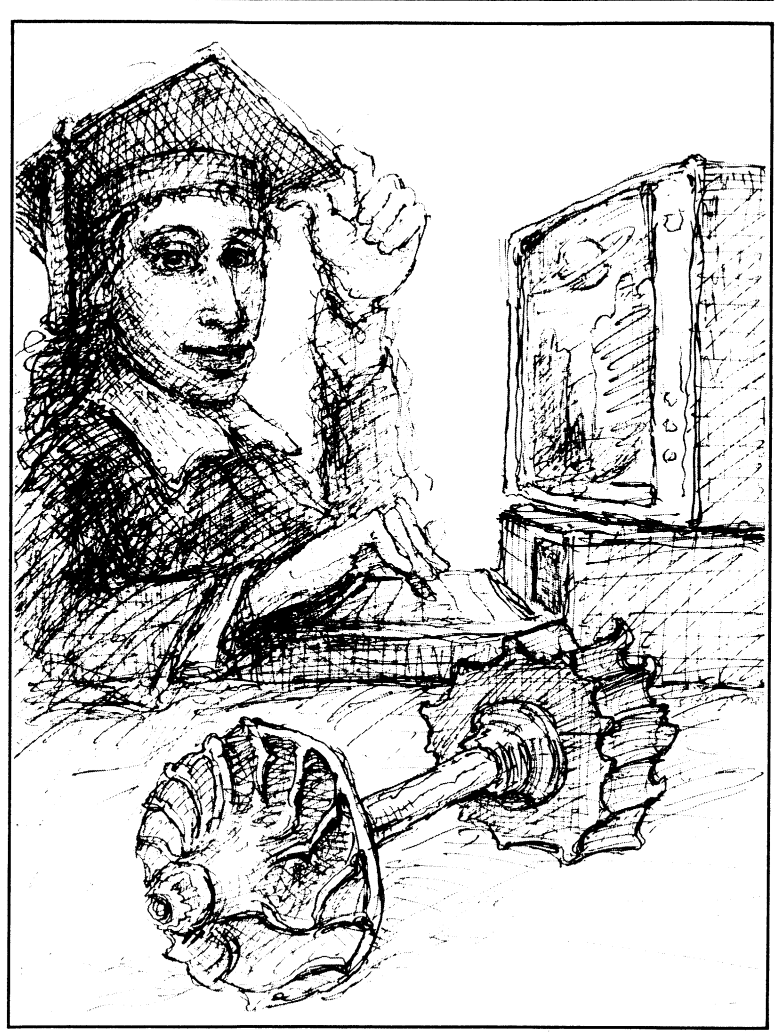
	Cursor Movements	
1	Character left	Ctrl S
2	Alternative	Ctrl H
3	Character right	Ctrl D
4	Word left	Ctrl A
5	Word right	Ctrl F
6	Line up	Ctrl E
7	Line down	Ctrl X
8	Scroll up	Ctrl W
9	Scroll down	Ctrl Z
10	Page up	Ctrl R
11	Page down	Ctrl C
12	To left on line	Ctrl Q S
13	To right on line	Ctrl Q D
14	To top of page	Ctrl Q E
15	To bottom of page	Ctrl Q X
16	To top of file	Ctrl Q R
17	To end of file	Ctrl Q C
18	To beginning of block	Ctrl Q B
19	To end of block	Ctrl Q B
20	To last cursor position	Ctrl Q P
	Insert and Delete	
21	Insert mode on/off	Ctrl V or Ins
22	Insert line	Ctrl N
23	Delete line	Ctrl Y
24	Delete to end of line	Ctrl Q Y
25	Delete right word	Ctrl T
26	Delete character under cursor	Ctrl G
27	Delete left character	Del
28	Alternative	—
	Block Commands	
29	Mark block begin	Ctrl K B
30	Mark block end	Ctrl K K
31	Mark single word	Ctrl K T
32	Hide/display block	Ctrl K H
33	Copy block	Ctrl K C
34	Move block	Ctrl K V
35	Delete block	Ctrl K Y
36	Read block from disk	Ctrl K R
37	Write block to disk	Ctrl K W

Table 6-1 Turbo Pascal Editor, Continued

Miscellaneous Editing Commands		
38	End edit	Ctrl K D
39	Tab	Ctrl I
40	Auto tab on/off	Ctrl Q I
41	Restore line	Ctrl Q L
42	Find	Ctrl Q F
43	Find & replace	Ctrl Q A
44	Repeat last find	Ctrl L
45	Control character prefix	Ctrl P

PART II

A Programmer's Guide to Turbo Pascal



7 The Basics of Pascal

In the first part of this book you learned how to enter, compile, and run a simple Turbo Pascal program to display some words on your screen. We taught you how to write a program (by copying it from a book), but we've yet to teach you how to design your own.

So before we move on, let's make sure you've acquired some necessary skills. You should be able to do the following:

- Start up your Turbo Pascal work disk.
- Load or create a work file.
- Enter and modify a program using the editor.

If you're still not comfortable using Turbo Pascal to perform these operations, go back to Chapter 6 for a quick review.

SOME PASCAL TERMS

Now you're ready to learn some of the basic concepts of the Pascal language. We'll begin by defining a few key Pascal terms, and the ideas behind them:

- Data type
- Identifier
- Reserved word
- Constant
- Variable
- Operator
- Expression
- Statement
- Comment
- Program Heading

- Declaration Part
- Statement Part

DATA TYPES

When you want Turbo Pascal to perform an operation (store, recall, manipulate, display) on a piece of data, you must tell Turbo Pascal what *type* of data it is. By specifying data types (which you'll learn how to do later), you can indicate to Turbo Pascal what kind(s) of operations it can perform on that data.

To illustrate, suppose that your data consists of the two numbers 3 and 4. Because 3 and 4 are numbers (in this case, integers), it makes sense for Turbo Pascal to be able to add them and return the sum 7.

Now suppose your data items consist of Tuesday and March. Clearly, it doesn't make much sense to add these two pieces of data together, and, in fact, Turbo Pascal will not let you do so. However, it is possible to perform a different operation on these items (for instance, finding the first Tuesday in March). If you invent such an operation—and explain to Turbo Pascal exactly how to go about doing it—Turbo will gladly perform it on any two data items of the proper types.

Predefined Data Types

What types of data can Turbo Pascal manipulate? The answer is almost any type—provided Turbo Pascal has some basic information about that particular type of data, such as the possible values of data items of that type, how big each item will be when it is stored in memory, and what operations can be performed on that data.

There are some types of data that are used in almost every computer program, and these would be hard to define if you had to write an explicit definition. (For instance, could you imagine trying to "explain" to the computer what a number is? Or listing all of its possible values?) For this reason, Turbo Pascal provides you with a number of *predefined data types*:

Type	Examples
integer	3, 0, −17382
byte	3, 0, 255
real	3.1415926, 0.00, −6.67E−22
char	'A', '$', '0'
boolean	TRUE, FALSE

We'll discuss the details of these data types and how to use them in later chapters. But for the moment, here's a synopsis of their properties.

An *integer* is a "counting," or whole number, that is, a number without a fraction. Integers can be negative. The numbers 5, −20, 0, and −32355 are all integers, but 3.5 is not (because it contains a fraction). Integers have a limited range of values. In Turbo Pascal (for the IBM PC, MS-DOS, and CP/M), numbers of type integer are not allowed to be bigger than 32767 or smaller than −32768.

A *byte*, like an integer, is also a "counting" number. Unlike an integer, however, a byte cannot be negative, and can only have a value from 0 to 255.

A *real* is a number that can (but doesn't have to) contain a fraction, like 1.5, −0.33335, or 24E15. (The *E* means that the number is interpreted to be in scientific notation; that is, it is to be read as 24×10^{15}) Data objects of type real usually have a much broader range of values than those of type integer.

A *char* is exactly what it sounds like: an ASCII character. It is useful for handling any kind of textual information.

A *boolean* is a data item that can have only two values: TRUE and FALSE. It is useful when your program needs to remember whether something is true or not.

User-Defined Data Types

Besides having predefined data types, Turbo Pascal lets you define your own. For instance, we might define the days of the week to Turbo Pascal as:

```
type
  day = (Sunday, Monday, Tuesday, Wednesday, Thursday,
         Friday, Saturday);
```

Turbo would then recognize all of the days listed as being of the new type *day*. Turbo Pascal allows you considerable flexibility in creating and manipulating your own data types. We'll cover the hows and whys of user-defined data types later in this tutorial.

IDENTIFIERS

Another term vital to your understanding of Pascal is *identifier*. An identifier is, very simply, a name for something—a piece of data, a piece of your program, a place in your program, or a data type. When you write a program, you name its parts by declaring identifiers to represent them.

In Turbo Pascal, some identifiers are declared for you without requiring any action on your part. For instance, the names of the predefined data types mentioned previously are predeclared identifiers.

An identifier must begin with a letter (numbers and punctuation characters won't do). This first letter may be followed by any combination of letters and digits, *but may not contain spaces!* An identifier may be as short as a single character or as long as 127 characters.

In standard Pascal, only letters and digits are allowed in an identifier. Turbo Pascal, however, allows you to use one special character: the underscore (_). An underscore may be used anywhere a letter would normally be used, and is handy for representing spaces (which are not allowed).

When Turbo Pascal reads your program, it ignores the case type of the letters within an identifier; therefore, the following identifiers all refer to the same value:

```
STARTINGLOCATION
Startinglocation
startinglocation
StartingLocation
```

Although all of these identifiers are equivalent, we feel the last one is easier to read than the others. Thus, this is the way you'll see identifiers depicted in this manual and other Borland books. Note that the following identifier is not equivalent to any of the previous ones:

```
Starting_Location
```

Adding an underscore is the same as adding another character—it completely changes the meaning of the identifier. Here are some examples of legal identifiers:

```
TURBO
square
persons_counted
BirthDate
DayOfTheWeek
AVeryLongIdentifierIndeed
The_2nd_Extremely_Long_Yet_Legal_And_Acceptable_Identifier
```

And here are some illegal identifiers:

`3rd_Root`	Starts with a digit instead of a letter
`Two Words`	Contains a space
`Two&Two`	Contains an illegal character (&)

To illustrate how the parts of the Pascal programming language can be used, we'll borrow a technique that has been used in many programming textbooks. Called *syntax diagrams*, this method illustrates the syntax, or grammar, of a language.

As an example, let's look at the syntax diagram for an identifier shown in Figure 7-1.

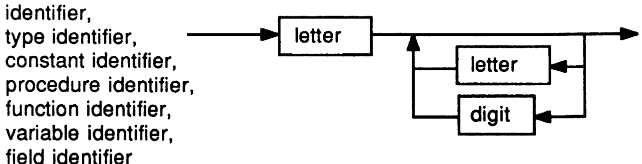


Figure 7-1 Syntax Diagram of Identifier

Like all syntax diagrams, this one defines the language elements (parts of speech, if you will) named at the upper left-hand corner of the diagram. In this case, the diagram shows the syntax of many (in fact, all) different kinds of identifiers in the Pascal language.

How do you read a syntax diagram? Well, you start at the arrow at the upper left-hand corner of the diagram, and follow the arrows through any boxes, ovals, or circles until you reach an arrow that leaves the diagram at the right. As you pass through each box, oval, or circle, whatever is specified inside them must also appear (in order) in the program element you are checking. If not, you must back up and try another path. If you are able to make it through the syntax diagram without breaking any of these rules, then the syntax of the program element is correct.

The boxes, ovals, and circles used within a syntax diagram have distinct meanings. A box contains a word that represents an object defined in another syntax diagram. An oval or a circle contains a symbol or a word that must be typed exactly as it is shown.

For example, in the top left-hand corner of Figure 7-1 is a box with the word "letter" in it. This means that the first thing that must exist in an identifier is a letter (we went over this a few paragraphs back). Following the arrow from the first box, you can either exit to the right (in which case your identifier would consist of a single letter) or you can follow the path down to one of two boxes (one contains the word "letter," the other contains the word "digit"). After leaving either of these boxes, you can follow the arrow up and exit, or you can go through the "loop" again, adding either another letter or another digit each time through.

Since each of the words in this diagram are enclosed in boxes, each of the words is defined by another syntax diagram. Figure 7-2 shows the syntax diagram for letter.

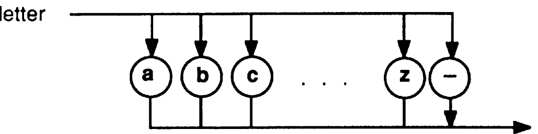


Figure 7-2 Syntax Diagram of Letter

Starting at the top left-hand corner of Figure 7-2, you are presented with a number of alternate paths, each to a single letter of the alphabet. Since each letter is in a circle, that letter must be entered exactly as shown. Finally, if you trace a path from any letter, you will find that the only path is one that exits the diagram. This means that each time you see the word "letter" in any syntax diagram, only one letter may be used.

The syntax diagram for the word "digit" is shown in Figure 7-3.

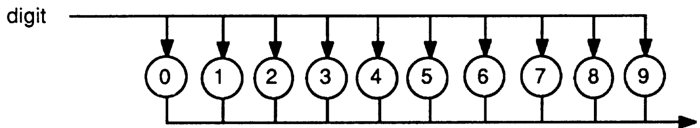


Figure 7-3 Syntax Diagram of Digit

You can read this diagram exactly as you did the letter diagram. Each pass through the diagram results in only one digit being selected.

Exercises For practice, try using the preceding syntax diagrams to check the syntax of the following "identifiers." Can you make it through the whole diagram with any of them? (The answers to these exercises can be found in Appendix B.)

```
Hen3ry _Turbo_Pascal
5th_Amendment
Three+Four
Good Work
```

Syntax diagrams are overkill when it comes to explaining identifiers, letters, and digits. But now that you know how to read them, these diagrams will prove quite valuable when we get to more complex constructs. We'll provide plenty of examples as well as syntax diagrams for each part of the Pascal language covered. And if you need to find the syntax of any part of the Turbo Pascal language quickly, you can refer to the complete set of diagrams in Appendix A.

RESERVED WORDS

When you begin writing programs, you will see that identifiers are used virtually everywhere. You will create your own identifiers for almost every data type, data object, and piece of code that you use. Aside from the syntax requirements we just covered, there are almost no limitations on the names you can give identifiers. There are, however, a few combinations of letters that cannot be used as identifiers because Pascal uses them as reserved words.

Reserved words (listed in Table 7-1) are words that have special meanings when used in your programs. They cannot be used for any other purpose (except in comments—we'll talk more about them later on). For example, you can't declare a variable named **program**, or name a program **begin**. You can use reserved words only in the way Pascal decrees.

To help you remember which are the reserved words as you look at sample programs, they are shown in **boldface type** throughout this and other Borland manuals. (Note, however, that reserved words do not appear bold when you type them into a program.)

Table 7-1 Pascal Reserved Words

Standard Pascal Reserved Words					
and	do	for	mod	procedure	to
array	downto	function	nil	program	type
begin	else	goto	not	record	until
case	end	if	of	repeat	var
const	file	in	or	set	while
div	forward	label	packed	then	with
Additional Turbo Pascal Reserved Words					
	absolute	inline	shl	string	
	external	overlay	shr	xor	

CONSTANTS

A *constant* is a piece of data (a number, perhaps, or some text) that remains the same while you're running your program, or even between runs. For instance, suppose you want to calculate a percentage based on a fraction, as follows:

100 * Numerator / Denominator

where * represents multiplication and / represents division. In this example, the number 100 is a constant (in fact, the number 100 is implicit in the very idea of a percentage). We therefore type it in as an explicit constant in the program.

Constants are not limited to numbers, as shown in the first sample program at the end of Part 1. When we write

```
Writeln('Hello world, my name is Joe')
```

the string of characters 'Hello world, my name is Joe' is a constant; more specifically, it is a *string constant*. The standard Pascal language allows you to use constants of types integer, real, char, and boolean, plus string constants.

Figure 7-4 shows the syntax diagram for a constant. Beginning in the upper left-hand corner (at the word "constant"), you can choose one of three possible paths: to a box, circle, or oval. If you take the upper path, you may select an optional sign ("+" or "−"), then either a constant identifier or an unsigned number (each of which is explained in another syntax diagram). Pursuing the lower path requires a *string* (a piece of text consisting of zero or more characters surrounded by single quotes). You can take either of these paths, but not both, when making a constant.

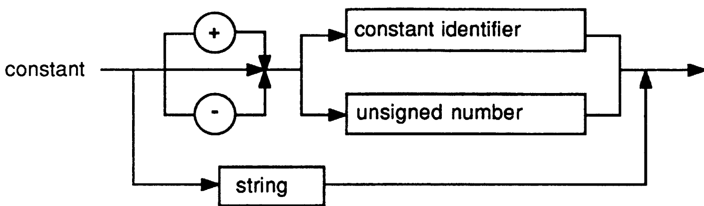


Figure 7-4 Syntax Diagram of Constant

Constant Definitions

To show how a constant definition might be useful, consider the following problem. Suppose a programmer had to write a program to compute compound interest on a bank account. Since the program would undoubtedly use the compound interest number several times, the programmer would have to type it in every time. The repetitiveness of such a task creates room for error—one that could cost the bank money. To avoid this problem a special numeric constant called *e* (a scientific and mathematic notation with the value of 2.718281828) could be declared as an identifer to represent the constant:

```
const
  e = 2.718281828;
```

This would replace the ponderous string of digits with a single character! After this definition, every mention of the identifier *e* would be equivalent to typing in the compound interest percentage. Constants are defined in a part of your program called a *constant definition part*. The syntax of this part of a program is shown in Figure 7-5 and Figure 7-6.

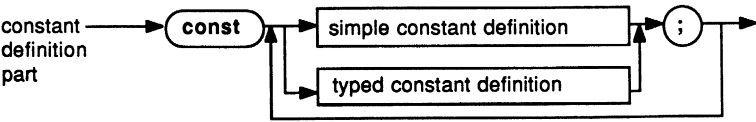


Figure 7-5 Syntax Diagram of Constant Definition Part

Figure 7-6 Syntax Diagram of Simple Constant Definition

Again, to practice your use of syntax diagrams, you may want to trace through the previous diagrams (as well as those in the back of the book) to verify that the definition of *e* really has the correct syntax.

Since this is only a brief introduction, the details of *typed constants*, and how to define them, will be given in future chapters, along with all of the rules for declaring constant identifiers.

VARIABLES

In almost any program, you'll need to use pieces of data whose values change during program execution. These data items are called *variables*. A variable is a place in your computer's memory where this data is kept.

All variables have names. Like all the other names that you declare when you write a program in Turbo Pascal, the name of a variable is an *identifier*, and should conform to the syntax for an identifier as shown previously. When you tell Turbo Pascal that you intend to use a variable in your program, you must give the compiler the variable's name and its data type. This is called a *variable declaration*, and causes Turbo Pascal to set aside a place in memory for the variable and to remember its name.

Variable Declarations

Variable declarations are made in a part of your program called (not surprisingly) the *variable declaration part*. Here are some examples of variable declarations:

```
var
  FirstInteger : Integer;
  SecondInteger, ThirdInteger : Integer;
  ASCII_Character : Char;
  RealNumber : Real;
```

The preceding declarations define three variables of type integer, plus one each of types char and real. Note that variables of the same type can be grouped into the same declaration, separated by commas. (*SecondInteger* and *ThirdInteger* are declared this way.) The syntax of this part of a program is shown in Figure 7-7.

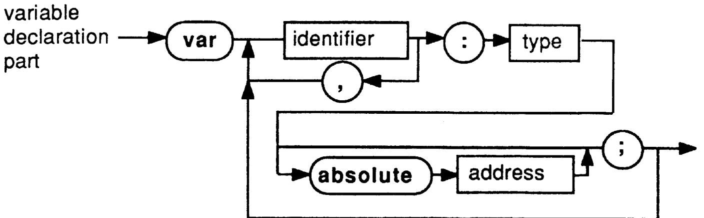


Figure 7-7 Syntax Diagram of Variable Declaration Part

As before, this syntax diagram hints at an advanced feature (**absolute** variables) that we are not ready to explain in detail just yet. For the moment, ignore the branch of the syntax diagram containing this reserved word.

OPERATORS

An *operator* is a special character, a group of special characters, or a reserved word that Turbo Pascal uses to indicate that an operation (arithmetic, for instance) is to be performed on one or more pieces of data. Some operators, like +, will already be familiar to you from arithmetic or algebra. Others, like * (for multiplication), represent familiar operations that have been adapted to the limited character set of a computer keyboard. (Most computers, like typewriters, do not have a multiplication symbol to represent multiplication.) And still others, like **div** or **mod**, are unique to the Pascal language.

Here are the operators you can use in Turbo Pascal:

`-`	Unary minus operator
`not`	Negation operator
`* / div mod shl shr and`	Multiplication operators
`+ - or xor`	Addition operators
`= < > <= >= <>`	Relational (comparing) operators
`in`	Set membership operator

The *unary minus operator*, which is the − sign that immediately precedes a number, does the same thing it does in familiar arithmetic: it changes a value sign. Thus, if the variable A has the value 5, writing $-A$ would yield the value -5.

The **not** operator takes a boolean value (TRUE or FALSE) and inverts it. Thus, **not** TRUE is the same as FALSE, and **not** FALSE is the same as TRUE.

The * and / operators indicate multiplication and division, respectively. The **div** operator represents a special kind of division operation, in which the remainder is thrown away; **mod** (short for *modulo*) divides two integers and simply returns the remainder.

The **shl** and **shr** operators are special low-level operations that shift the bits of a byte or integer. We'll describe exactly how they work in later chapters.

The + and − operators are the same operations you've seen in familiar arithmetic, and produce the sum or difference of two numbers.

The **and** and **or** operators work on pairs of boolean values. The value TRUE is produced by **and** if both of its operands are TRUE; **or** if either of its operands are TRUE. These two operators can also be used to do bit math on bytes and integers.

The relational operators (=, <, >, <=, >=, <>) compare numbers and return boolean values based on the results. They should all be familiar except for <>, which is the way the inventor of Pascal chose to represent the not-equal sign ($\neq$) on a computer.

Finally, the **in** operator determines whether or not an object is in a *set*. We'll discuss sets and how to use them shortly.

EXPRESSIONS

You've probably seen *expressions* before, perhaps in science and math classes where they were called formulas (or, if you want to be classical, formulae). Formulas that calculate such things as the circumference of a circle or the velocity of a falling object with respect to time are just expressions. In fact, FORTRAN (one of the earliest programming languages) stands for FORmula TRANslator.

In Pascal, expressions are combinations of identifiers, constants, and/or operators that describe how to produce a new piece of data from one or more existing ones. When an expression is evaluated, the calculations within it are carried out. The result is a single value.

Like all data used in a Turbo Pascal program, the result of an expression has a data type. This type may or may not be the same as that of the constants or variables within it. For instance, an expression such as

```
first_integer + second_integer + third_integer
```

which adds the contents of three integer-type variables, is said to be an integer expression. However, the expression

```
first_integer < second_integer
```

which compares two integers, yields a boolean value (TRUE or FALSE) that indicates whether or not the first_integer is less than the second_integer.

The Order of Operations in Expressions

If you've taken a course in algebra, you probably remember the *Order of Operations*—which means the order in which you perform operations in an expression. In this section, we will demonstrate why this ordering is necessary, and how Pascal handles expressions in which order makes a difference.

Suppose you asked the computer to evaluate the expression 3 + 4 * 2 (remember that * denotes multiplication). How would the computer calculate the result? The following shows two possible ways:

$$\begin{array}{r} 3 \\ +\,4 \\ \hline 7 \\ *\,2 \\ \hline 14 \end{array} \qquad \begin{array}{r} 4 \\ *\,2 \\ \hline 8 \\ +\,3 \\ \hline 11 \end{array}$$

The example on the left adds 3 to 4, totaling 7, then multiplies the total by 2 to get 14. The example on the right first multiplies 4 by 2 to get 8, then adds 3 to get 11. Which answer is correct? Well, a long time ago, mathematicians ran into this same dilemma, and set up a series of rules to determine how to evaluate expressions in an unambiguous way. Pascal follows these rules, and added a few more for its own unique operations.

The first rule to remember is that Pascal will always perform multiplication and division operations before addition and subtraction operations, unless the addition or subtraction operation is encased in parentheses, thereby causing the multiplication or division operation to follow. By applying this rule, we can see that the expression in the previous example will evaluate to 11, not 14. However, if we were to

add parentheses to the previous expression to make (3 + 4) * 2, the addition would be performed first and then the multiplication.

The second rule you should know is that operations of the same kind (multiplication/division, addition/subtraction) are performed from left to right. Thus, the expression 10 / 5 * 2 would be evaluated as shown:

```
  10                    5
 / 5                  * 2
 ---     and not    ------
   2                10/ 10
 * 2                ------
 ---                     1
   4
```

If you wanted the second answer, you would indicate that by writing the expression as 10 / (5 * 2). In this case, the parentheses indicate that the multiplication should occur first.

The third rule is that unary operations (that is, operations that operate on only one object) are performed before any of the others. For instance, in the expression −5 + 10, the unary minus before the number 5 applies only to the 5, not to the whole expression 5 + 10. The result of the expression is therefore 5, not −15!

Parentheses can be used to override the order of operations for unary operators, as well. The expression −(5 + 10) evaluates to −15, as you might expect.

Pascal extends the rules we just described to apply to the relational and set membership operations as well. The relational operations are performed after the addition operations, and the set membership operation follows. The complete table of operators given earlier was, in fact, intentionally laid out in order of precedence—that is, with the operations that are done first above those that are done afterward.

Exercises Evaluate each of the following expressions according to the Order of Operations used in Pascal.

1. 4 * 6 / 2 + 3
2. (4 * 6) / 2 + 3
3. 4 * (6 / 2 + 3)
4. (4 * (6 / 2) + 3)
5. 4 * ((6 / 2) + 3)
6. (4 * 6) / (2 + 3)

Now, check your answers by inserting them into the following short program and running it. The program is shown as ready to run the first example.

```
program calculate;
begin
  Writeln(4 * 6 / 2 + 3);
end.
```

STATEMENTS

A *statement* is a part of a program that tells the computer to perform an action. In the sample program in Chapter 6, the statement

```
Writeln('Hello, world, my name is Joe')
```

tells the computer to display a string on the screen. When statements occur in succession, they must be separated from one another by a semicolon (;). Here are some more examples of statements:

```
Value1 := Value2 + Value3;

Radius := 40.25;

if Value1 > 100 then
  Writeln('Value1 is greater than 100');
```

The first statement given is an *assignment statement.* It is called this because it evaluates the expression *Value2* + *Value3*, and assigns the result of the addition operation to the variable *Value1* (that is, it puts the result in the place in memory reserved for *Value1*). *Value1* retains this result as its variable unless it is specifically changed by the program. The second statement is also an assignment statement. Here, the constant value 40.25 is assigned to the variable *Radius.* The last statement is called an ***if*** *statement.* This kind of statement instructs the computer to perform a certain action only if a particular condition is met; in this case, if *Value1* is greater than 100. Note that this statement actually contains another statement (the *Writeln* statement). This is a common occurrence in Pascal, and we'll say more about it later.

COMMENTS

Last, but not least, there is the Pascal construct that causes the computer to do absolutely nothing at all: the *comment.* A Pascal comment is a string of characters starting with the character "{" and ending with the character "}". The space within the braces can contain any kind of text at all (except, of course, more braces), and everything within the braces will be completely ignored by the compiler. (Chapter 8 details another method of commenting.)

```
{ This is an example of a comment. Turbo Pascal will ignore
this text. }
```

At this point, you might well ask, "If comments are useless to the compiler, and it completely ignores them anyway, then why are they part of the language at all?" The answer is that comments are very useful to the reader who is trying to understand, change, and/or fix a program. Now, Pascal is designed to make programs more readable than many other languages; however, it is still possible to come back to your own code only a month after writing it, and discover that you no longer understand how it works!

In such situations, comments can save the day because they allow you to attach notes to anything and everything within your program. As you read through the following example, note the liberal use of comments to make the program easier to understand.

A PROGRAMMING EXAMPLE

Now, let's say you want to write a simple Pascal program to accept two numbers typed on the keyboard, add them together, and display the result. First, let's look at a complete program for doing this task, then let's discuss each part of the program. Before you read the discussion of the program, look at the program sample carefully and try to understand what it does and how it works. Here is the program:

```
program Simple; { This is the Program Heading.}
{
A simple Pascal program to display the sum of two numbers.
DATE: 17 June 1986
AUTHOR: put your name here
}
{ This is the beginning of the
  Declaration Part of the program,
  where our identifiers are declared. }

const
  YourName = 'Friend';
  { This is a string constant used in the greeting message.
    Change to contain your name if you'd like.}
var
  A,B,C : Integer;

{ This is the beginning of the Statement
  Part of the program. It contains
  statements--the parts of a Pascal
  program that tell the computer what to do. }

begin { Main body of program Simple }

  { Start by greeting the user. As in our very first
    program, we use a Writeln ("Write Line") statement
    to write a line to the terminal. }

  Writeln('Hello, ', YourName, '.');
```

```
  { Note that the Writeln statement can take a LIST
    of things to write on a line, as well as just
    one thing. In the statement above, we wrote
    three things: the constant string 'Hello, ',
    the value of the constant identifier
    YourName (another string), and a period
    (a character constant). }

  { We now write a string to the terminal asking
    the user for an integer. A message like this one,
    which requests a response of some kind, is often
    called a "prompt." }

  Writeln('Please type an integer, followed by a return.');

  Readln(A);
  { Wait for the user to type a number,
    then place that number in the variable
    A. "Readln," which is read as
    "Read Line," tells the computer to
    wait for the carriage return key to
    be pressed before assuming that the number is complete. }

  { Repeat the two steps above for a second number:}

  Writeln('Now please type another integer, followed ',
          'by a carriage return.');

  { Prompt for another number }

  Readln(B);
  { Read the number and place it in the variable B. }

  C := A + B; { Add A and B and place the result
                in the variable C. }

  Writeln('The sum of the two integers is: ', C);

  { Write a line containing a message and the value of
    the variable C. }

  { Putting an identifier (here, C) in the list of
    things that a Writeln statement is to write
    causes its VALUE to be written, rather than its
    name. If we wanted to just print the letter C, we
    would enclose it in single quotes as we did with
    the period in the first Writeln statement. }

end. { of program Simple }
```

The first thing you should notice about this program is that it is divided into three sections, each starting with a reserved word (**program**, **const**, and **begin**). We will now discuss the functions of each of these parts of the program.

The Program Heading

The first line (the one with the reserved word **program**) gives the program's name and indicates that the lines of code that follow comprise a program. This program is named "Simple." (The importance of program headings is discussed in detail in Chapter 8.)

```
program Simple; { This is the Program Heading. }
```

The Declaration Part

Next is the *declaration part*, where identifiers are declared. As mentioned previously, an identifier is a name you give to something (a constant, a variable, a place in your code, or a piece of your code). The declaration part must occur after the program header, but before the rest of the program. In this program, we *define* the constant *YourName* and *declare* the variables *A*, *B*, and *C*.

A constant definition consists of two pieces of information: the *name* of the constant and its *value*. These are separated by an equal sign (=), and the definition is followed by a semicolon (;). A group of one or more constant declarations is preceded by the reserved word **const**.

We declared the constant *YourName* as follows:

```
const
  YourName = 'Friend';
  { This is a string constant used in
    the greeting message. Change to
    contain your name if you'd like. }
```

A variable declaration also consists of two pieces of information: the *name* of the variable and its *type*. These two pieces of information are separated from one another by a colon, and each declaration statement must end with a semicolon. To let the compiler know that a group of one or more variable declarations is coming, we use the reserved word **var**.

```
var
  A,B,C : Integer; { Variables }
```

The variables in our program are *A, B,* and *C*, and they are all of the type integer. Thus, the only things that can be "contained" by the variables *A, B,* and *C* are integers (positive and negative numbers without fractional parts).

The Statement Part

The remainder of the program is enclosed by the reserved words **begin** and **end**. This part of the program consists of statements, and is therefore called the *statement part*. The computer starts with the first statement in the statement part and continues to perform (or execute) the statements, in order, until it reaches the final **end**.

```
begin { Main body of program Simple }
  .
  .
  (Statements go here)
  .
  .
end. { of program Simple }
```

Each pair of statements in the statement part is separated by a semicolon. Since the last statement is not followed by another statement, no semicolon is needed. Also, note that there is a period after the final **end** in the program. This is required to let the compiler know that the program is finished.

The first statement in the sample program is a *Writeln* statement:

```
{ Start by greeting the user. As in our very first
  program, we use a Writeln ("Write Line") statement
  to write a line to the terminal. }
Writeln('Hello, ', YourName, '.');

{ Note that the Writeln statement can take a LIST of
  things to write on a line, as well as just one
  thing. In the statement above, we wrote three things:
  the constant string 'Hello, ', the value of the
  constant identifier YourName (another string), and
  a period (a character constant). }
```

As we mentioned in the previous comment, the *Writeln* statement takes a list of variables, constants, or expressions and writes their value(s) to the screen. (We didn't show it in our program, but it is also possible to have a *Writeln* statement without a list of things to write. The statement *Writeln*; will simply write an empty line to the terminal.)

Our sample program will then *prompt* (that is, ask) for an integer value to place in the variable *A*.

```
{ We now write a string to the terminal
  asking the user for an integer. A message
  like this one, which requests a response
  of some kind, is often called a "prompt." }

Writeln('Please type an integer, followed by a carriage ',
        'return.');

Readln(A);
{ Wait for the user to type a number, then place that
  number in the variable A. "Readln", which is read
  as "Read Line," tells the computer to wait for the
  carriage return key to be pressed (starting a new line)
  before assuming that the number is complete. }
```

The *Readln* statement does the work of getting the integer we asked for from the user's keyboard. Like *Writeln*, *Readln* can take a list of values to get from the terminal, or can be used with no list at all. The statement

```
Readln;
```

just waits for the user to press ⏎. Our program then does a second *Writeln* and a second *Readln* to get another number and adds the first number to the second:

```
{ Repeat the two steps above for a second number:}
{ Prompt for another number}

Writeln('Now please type another integer, followed by a ',
        'carriage return.');

Readln(B); { Read the number and place it
             in the variable B. }

C := A + B; { Add A and B and place the result
              in the variable C. }
```

The last statement of the preceding group is an assignment statement. This one, $C := A + B$, evaluates the expression $A + B$ by adding the values contained in the variables A and B together. It then places the result of this operation in the variable C.

It helps to think of the assignment operator, :=, as an arrow pointing to the left, indicating the flow of information from the expression on the right-hand side to the variable on the left-hand side. When a program is read aloud, the assignment operator is usually read as "gets." The previous statement would be read as "C gets A plus B."

Now our sample program writes the value we saved in the variable C. To do this, we simply put C in the list of things to write:

```
{ Write a line containing a message and the value of the
  variable C. }

Writeln('The sum of the two integers is: ', C)

{ Putting an identifier (here, C) in the list of things that
  a Writeln statement is to write causes the VALUE of C to
  be written, rather than its name. If we wanted to just
  write the letter C, we would enclose it in single quotes
  as we did with the period in the first Writeln statement. }
```

Finally, the program reaches the final **end** and stops.

Note that throughout the sample program, everything has been arranged in an orderly fashion, along with copious comments. Since the compiler does not care at all about spacing between words (unless, of course, the spaces are within a quoted string of characters), we have spaced and aligned everything to make it easy for anyone to understand.

We can't emphasize enough the importance of clear and concise comments in a program. They can add much value to your code, especially if others will be reading and/or modifying it. Keep in mind that although comments take space in your uncompiled (source) program,

they never increase the size of the compiled program. Thus, the *performance* of your software will never suffer due to too many comments!

We've covered a lot of ground in this chapter. So, before you move on, please take the time to perform the following exercises. They are designed to reinforce what you have learned so far, and prepare you for the material in chapters to come.

Exercises

1. Review the previous sample program. How many identifiers can you find? (Hint: The names of data types, like integer, are identifiers). How many constants? How many statements?
2. Use Turbo Pascal to load and run this program in its original form. Change the value of the constant *YourName* so that the computer writes your name when the program is run.
3. Modify the sample program so that it prints not only the sum of the two numbers ($A + B$), but the difference ($A - B$) as well. Add a new variable, D, to the program to hold the difference.
4. Modify the program to return the following values:

    a. Twice the difference between A and B

    b. A minus twice B

    c. Five times A, minus the quantity three times B

    d. The product of A and B

    e. *A modulo B* (Watch out if you enter a value of 0 for *B*!)

    Check your programs by accumulating results for several values of A and B.
5. Try typing a number with digits to the right of the decimal point when asked for one by the program. What happens? Can you explain why?
6. Now, change the variables in the program so that they are all of the type real. Repeat Exercise 5 with this new program. Can you explain the results? (The compiler will complain if you try Exercise 4e with the variables A and B as real numbers; the **mod** operator, by definition, only works on integers.)

REVIEW

Each of the topics presented in this chapter deserves (and will get) more explanation. Our objective here has been to define some of the basic concepts of Pascal, and you now should have at least a basic understanding of the terms data type, identifier, reserved word, operator, constant, variable, expression, statement, and comment. If you feel comfortable using these terms, then you are ready to go on to the next chapter.

8 Program Structure

In Chapter 7, we took a brief look at the most important Pascal terms and concepts, and showed them at work in a simple program. In this chapter, we will examine the rules governing the structure of a Pascal program.

Let's quickly review what we have already learned about program structure. A Pascal program consists of three distinct parts: (1) the program heading, (2) the declaration part, and (3) the statement part. (Figure 8-1 shows the syntax of a program.)

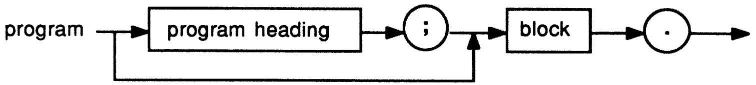


Figure 8-1 Syntax Diagram of Program

The program heading names the program and, for some compilers, gives information about what files the program will use.

The declaration part consists of declarations that give Pascal information about the pieces of data and code that fit together to make the program work. It is possible to have a Pascal program without a declaration part (program MyName in Part I, for example), but such a program is usually not very useful.

Finally, there is the statement part, which consists of one or more program statements that tell Pascal how to do the actual work, such as adding numbers, printing items on the screen, and deciding what to do next. The statements are enclosed between the reserved words **begin** and **end**, followed by a period indicating the end of the program.

THE PROGRAM HEADING

In standard Pascal, your program *must* begin with a program declaration that consists of the following in this order: (1) the reserved word **program**, (2) the name you want to give to your program (which can be any legal identifier), (3) a list of identifiers naming the files that the

program will use (if any), and (4) a semicolon. The syntax diagram for the program heading is shown in Figure 8-2.

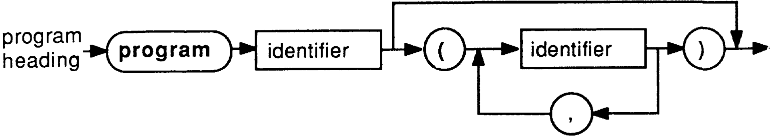


Figure 8-2 Syntax Diagram of Program Heading

In Turbo Pascal, unlike most compilers, the program heading is optional. Turbo Pascal is smart enough to figure out what files you will use from the text of your program, and doesn't need to know the name of your program in order to compile and run it.

However, since the Turbo editor always puts you at the beginning of the file when you start it up, a program heading is good for immediate program identification. If you have forgotten what the program in a particular disk file does, the heading can save you much time and exploration. Using a program heading also makes your programs more portable. We recommend you use one in every program you write (as we do), along with a meaningful program name.

Here are some sample program headings:

```
program BudgetAnalysis;
program Rutabaga (input, output);
program With_A_Very_Long_Name_Indeed (file1,file2);
```

THE DECLARATION PART

The declaration part consists of declarations, or definitions, of all labels, constants, types, variables, procedures, and functions that you will be defining in your program. (Procedures and functions are pieces of your program that are invoked by name.) Each declaration lists one or more identifiers and then gives information about the meaning(s) of these identifiers. (When declaring a label—a number or identifier used to mark a place in your program—you simply list the numbers and/or identifiers that will be used as labels.)

In Pascal, the declaration part is divided into five different subparts:

- Label declaration part
- Constant definition part
- Type definition part

- Variable declaration part
- Procedure and function declaration part

Standard Pascal requires that each of these subparts occur only once (if at all), and in the exact order listed. Turbo Pascal, however, is not as rigid about the order in which these things are declared, as long as they're declared *before they are used.* With Turbo, you can make any kind of declaration, any number of times, and in any order in the declaration part. This is shown by the syntax diagram of Turbo's declaration part (Figure 8-3).

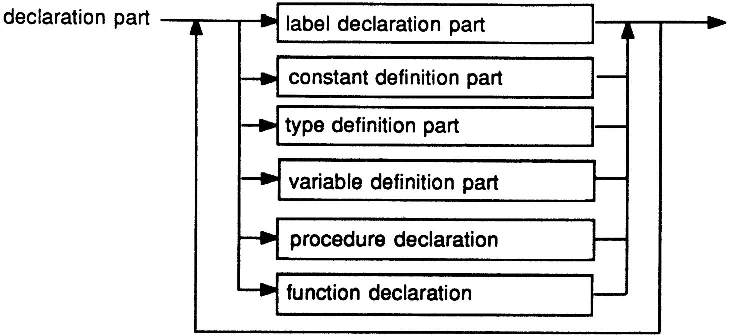


Figure 8-3 Syntax Diagram of Declaration Part

Since Pascal has no idea what an identifier means until you tell it, you *must* declare all of the identifiers you will use in your program (except, of course, the "standard" identifiers, like *Readln* and *Writeln*, which are already part of Turbo Pascal). This includes all labels, constants, types, variables, and procedures and functions that you create as you build your program. If you use an identifier that has not been declared, Turbo (like all Pascal compilers) will simply give an error message and refuse to compile or run your program.

This rule of identifier declaration is so fundamental to all Pascal programs that we need to call special attention to it. Memorize this rule well: *All identifiers must be declared before they are used.*

Formatting Your Declarations

As long as you remember to declare all of your identifiers, Pascal will allow you considerable freedom in the way you format the declarations. In this section we will present some sample variable declarations to illustrate this flexibility; as you read them, bear in mind that the same principles apply to constant and type declarations as well.

As you may recall, a group of variable declarations (a variable declaration part) is preceded by the reserved word **var**. Each declaration

consists of one or more names (identifiers) for variables, a colon, a type for the variable(s), and, finally, a semicolon. The result looks like this:

```
var
  A,B,C : integer;
```

This example defines three variables of type integer: *A*, *B*, and *C*. The reserved word **var** can be followed by several variable declarations without needing to be repeated. Thus, if you want to make the program more readable by declaring *A*, *B*, and *C* on separate lines, you can write

```
var
  A: integer;
  B: integer;
  C: integer;
```

without repeating the **var**.

As mentioned in Chapter 7, there are five predefined data types in Turbo Pascal: integer (positive and negative whole numbers and zero), byte (positive integers from 0 to 255—the range can be represented by one byte of data), real (also known as *floating point*—positive and negative numbers with fractional parts and optional exponents), char (one ASCII character enclosed in single quotes, and boolean (TRUE and FALSE). Here are some variable declarations for these different data types:

```
var
  Alive,Breathing    : boolean;
  Age,Height,Weight  : integer;
  Score              : byte;
  Ratio,Percentage   : real;
  First,Middle,Last  : char;
```

Pascal is what is known as a "free format" language—it doesn't care how the text of your program is broken into lines (unlike BASIC, FORTRAN, and some other languages). As long as the syntax of the declarations agree with the diagram in Figure 8-3, Pascal allows quite a lot of flexibility in how you arrange your declarations. If, for some reason, you want to arrange your declarations in a long, narrow column, you could rewrite the previous example as:

```
var
  Alive,
  Breathing
  : boolean;
  Age,
  Height,
  Weight
  : integer;
  Score
  : byte;
  Ratio,
  Percentage
  : real;
```

```
  First,
  Middle,
  Last
  : char;
```

And here's yet another variation:

```
var
  Alive : boolean; Breathing : boolean; Age : integer;
  Height : integer; Weight : integer; Ratio : real;
  Percentage : real; First: char; Middle : char; Last : char;
```

Note, however, that when we show declarations in this tutorial, we will use the format that we think is the easiest to read and understand.

THE STATEMENT PART

The statement part of a Pascal program consists of the reserved word **begin**, followed by any number of statements, followed by the reserved word **end** (see Figure 8-4). The statement part of a program is followed by a period (.) to indicate the program is finished. Execution always starts with the first statement after **begin**, and proceeds sequentially to the last statement before **end** (unless a statement directs the program flow elsewhere than to the next sequential statement). The period after the last **end** must be the last thing that appears in a Pascal program (with the exception of a comment).

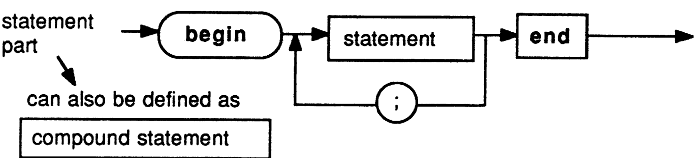


Figure 8-4 Syntax Diagram of Statement Part

Formatting Your Statements

Neither spaces nor line breaks affect the meaning of the program, unless, of course, they occur in the middle of a string. For this reason, we could have written our first program (MyName) like this:

```
program MyName;
begin ClrScr;
Writeln('Hello, world, my name is _______________') end.
```

Pascal also doesn't care about upper- and lower-case or line indentation. You have the freedom to format your programs in a variety of ways. The examples shown in this book (except, of course, for the previous one) are written in what we think is a readable, easy-to-use

format that can be understood by anyone knowledgeable in Pascal. (For more discussion about programming style, see Appendix E.)

Statement Types

A large portion of this tutorial will be devoted to discussing the different kinds of statements available to you in Turbo Pascal. Let's look at the syntax diagram of a statement (shown in Figure 8-5) to get an overview of what's to come.

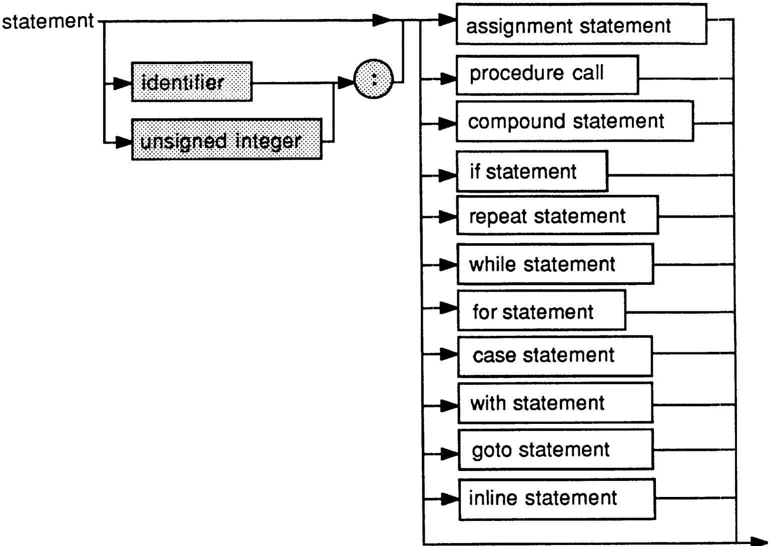


Figure 8-5 Syntax Diagram of Statement

You have already been introduced to the first kind of statement in the diagram, the assignment statement. The assignment statement causes expressions to be evaluated, and assigns values to variables.

A *procedure call* causes the named program part of the call to be run. Note that the *Readln* and *Writeln* statements in the previous examples are procedure calls.

Readln and *Writeln* are procedures that are predefined by Turbo Pascal. We'll talk more about procedures, and how to define your own, in chapters to come.

A *compound statement* is a number of statements grouped together between the reserved words **begin** and **end**. If you think this sounds similar to the statement part of a program, you're absolutely right—the statement part is an example of a compound statement. Compound statements have other uses, which we will discuss shortly. The **if** and **case** statements make decisions about what the computer will do next.

The **repeat**, **while**, and **for** statements all cause the computer to repeat certain actions until a particular condition is met. We'll say more about them in Chapter 11, "Control Structures."

The **with** statement helps the programmer by allowing variable names to be shortened. We'll cover this in Chapter 15, "Records."

The **goto** statement tells the computer to jump to a particular place in the program (indicated by a label) and start working. Unlike less-structured languages like FORTRAN and BASIC, Pascal never presents a situation where **goto** statements must be used; we recommend avoiding them altogether. For the sake of completeness, however, we've included a brief explanation of the **goto** statement in Part III.

If you have already programmed in BASIC, it is especially important to kick the "**goto** habit." First, learn to use Pascal's structured techniques for controlling a program. Once you have mastered these methods, then you can resort to **goto**s, if you must.

The **inline** statement is an advanced programmer's tool that allows machine language instructions to be placed directly within the text of a Turbo Pascal program. We recommend this feature for experts only; for more information, refer to Chapter 28 of this manual or the *Turbo Pascal Reference Manual*.

Finally, the *null statement* is represented by the arrow that completely bypasses all of the other statements in the previous syntax diagram (Figure 8-5). In some places where the Pascal language requires a statement, we may want to tell the computer to "do nothing at all." The null statement provides us with a way to convey that message to the compiler.

The path shown in gray in Figure 8-5 shows the format of a label—the target of a **goto** statement. As previously mentioned, we recommend avoiding **goto** statements entirely; this is shown simply for your information.

COMMENTS: THE REST OF THE STORY

An important part of any program is the documentation—text that explains what the program does and why. Documentation can exist at many levels, including help screens, online instructions, user manuals, and comments.

Comments are used to tell the reader of the program the uses of identifiers, the actions of the program, the situation that will occur when a certain condition is met, the date the program was written, the name of the author, and anything else that might be useful in understanding the program. Comments require a small amount of additional work, but are invaluable in debugging, maintaining, and enhancing

your software. You should always take into consideration the possibility of someone other than yourself modifying the text (source code) of your program. And while it is completely possible to write a program without comments, it is a bad practice to do so. Think about how hard it would be to cook a gourmet meal from a recipe that listed all the ingredients but gave no instructions on how they were to be combined.

As mentioned in Chapter 7, comments are ignored by the compiler, which means they have no affect whatsoever on your compiled program (they do not increase its size or affect its execution speed). The only computer-related effect they have is to increase the size of the source code (not the compiler output) of your program slightly, and this is well worth the benefit of being able to better understand the code, as well as to modify it later.

Pascal comments begin with a left brace ({) and end with a right brace (}). These symbols are known as *comment delimiters*. A comment can start and end almost anywhere, and occupy as many lines as needed. However, you must be careful of *nested comments*—comments embedded within comments.

Let's say you've written the following section of a program:

```
   .
   .
   .
Writeln (date); {Write the date}
old_a := a; {Save the old value of a}
Readln (a); {Read a new value for a}
   .
   .
   .
```

Now, suppose you want to temporarily remove the last two statements shown (that is, you don't want to delete them from the program, but you don't want them to be performed during this run). This is known as "commenting out" a section of code. At first glance, it appears that you could simply put comment symbols before and after the statements you want to comment out, as follows:

```
   .
   .
   .
Writeln (date); {Write the date}
{
old_a := a; {Save the old value of a}
Readln (a); {Read a new value for a}
   .
   .
   .
```

Unfortunately, if you use this approach, it will have a quite different effect than you might have intended. Pascal will recognize the first comment ({Write the date}) with no problem, and will understand that

the next left brace (at the beginning of the second line) is the beginning of another. However, when the compiler encounters the right brace at the end of the third line, it will think that the comment is over. Thus, the statement *Readln(a)* will be compiled into the program, even though we didn't want that to happen. In this particular case, the compiler will find the extra right brace in the last line of the example, and will signal that something is wrong with an error message. Sometimes, however, it is possible for this sort of error to go undetected, causing many debugging headaches.

One solution to this problem is to delete some of the comment symbols within the section that is commented out. But that would be more trouble (and probably cause more errors) than removing the entire section of code. Fortunately, there is a better way.

Earlier, we told you that Pascal comments begin with a left brace, ({) and end with a right brace (}), but there's also an alternate pair of comment symbols: a left parenthesis and an asterisk ((*) to begin a comment, and an asterisk and a right parenthesis (*)) to end a comment. Turbo Pascal allows you to place one kind of comment within the other.

Well, maybe "allow" isn't quite the right word; the situation is a natural result of the way comments work. If you begin a comment with a left brace ({), everything will be ignored, including the set of parentheses and asterisks ((* and *)), until the right brace (}) appears. The reverse is also true: When a comment begins with a left parentheses and an asterisk ((*), everything up to the next asterisk and right brace (*)) will be ignored, including a set of braces ({}).

So, the solution to the problem of nested comments is to always use one set of comment delimiters for descriptive comments and the other set of comment delimiters for commenting out sections of code. In this manual, we'll use braces as comment delimiters for ordinary text comments. If the situation requires that sections of code be commented out, we will use the parenthesis-asterisk comment delimiters for that purpose. Now, let's rewrite our latest example by using both sets of comment delimiters:

```
  .
  .
  .
Writeln (date); {Write the date}
(*
old_a := a; {Save the old value of a}
Readln (a); {Read a new value for a}
*)
  .
  .
  .
```

The program will run just fine now, except that the sections of code between the (* and the *) will be ignored by the compiler. You may want to use this technique for testing various parts of programs, to isolate problems, or to prove that a section of your program does what you intended.

As we said, you can insert comments almost anywhere. The exceptions to this rule are that you cannot insert a comment into the middle of an identifier or a reserved word, or inside a string. In the first case, the compiler will think that the reserved word or identifier ends where the comment begins; in the second, it will think that the whole comment was part of the string. The statement

```
Writeln('Hello, world, my name is {not} Joe.');
```

produces the output

```
Hello, world, my name is {not} Joe.
```

REVIEW

The structure of a Pascal program follows this form:

```
program Name ({optional file identifiers});
label
{ label declarations here }
   .
   .
   .
const
{ constant declarations here }
   .
   .
   .
type
{ type declarations here }
   .
   .
   .
var
{ variable declarations }
   .
   .
   .
{ subprograms (procedures and functions) declared here}
begin
{ main body of program }
end.
```

Turbo Pascal is more flexible than standard Pascal in that **label**, **const**, **type**, **var**, and subprogram declarations can be placed in any order and can occur more than once.

Pascal is a free-format language. It allows declarations and statements to be formatted in many ways, subject to certain simple constraints. Pascal provides a rich variety of statements from which to choose.

Comments are most often used for two reasons: to document programs and to prevent certain sections of code from being run during testing. The two sets of comment delimiters ({ and } and (* and *)) allow the programmer a limited ability to nest comments used for these purposes within one another. Comments can be used anywhere, except in the middle of a string, a reserved word, or an identifier.

Now you're ready to tackle the predefined data types of Pascal in Chapter 9.

9 Predefined Data Types

Every data object that appears in a Pascal program has a *data type*. There are two kinds of data types: *simple types*, which are used for data that is always manipulated as a whole, and *structured types*, which are used for data consisting of smaller pieces that can be manipulated individually.

In this chapter, we will explore Turbo Pascal's *predefined data types*—the ones that you will probably use most often. All of Turbo's predefined types are simple types; these include integer, byte, real, boolean, and char. We'll explain each type and give examples of how it is used.

In addition, we'll cover the group of structured types called *string types*, which allow you to manipulate groups of characters easily. String types are not a part of standard Pascal, but are one of Turbo's most useful extensions.

When you declare a constant, the data type of the constant is implicit in the constant declaration, and must be one of the predefined types. (*Typed constants*, which we will discuss in Part III, allow you to define a constant of any type.) When you declare a variable, you are required to explicitly state the type in the declaration. Remember our rule: *All identifiers must be declared before they are used.*

INTEGERS

Integers are whole numbers, negative and positive, including zero. The number 12 is an integer, as are 456 and −12000. On the other hand, 1.234, 2.0, and −0.54234312 are not integers (they have decimal points). You will use integer data types when your data is strictly numeric and does not contain fractions. Figure 9-1 shows the syntax diagram for an *unsigned integer* (an integer constant without a possible + or − sign).

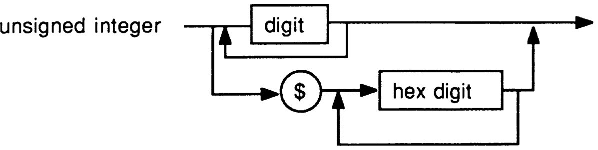


Figure 9-1 Syntax Diagram of Unsigned Integer

Note that in Turbo Pascal you have the option of specifying an integer constant in *hexadecimal* (base 16) *notation* as well as ordinary *decimal* (base 10) *notation*. A Turbo hexadecimal constant consists of a series of hex digits (0 through 9, or a letter *a* through *f*) preceded by a dollar sign ($).

In Turbo Pascal, an integer occupies 2 bytes of computer memory. Because there are only so many unique values that can be expressed by 16 bits, objects of type integer in Turbo Pascal are limited to the numbers in the range of −32768 through 32767. Keep in mind that each data object you declare as an integer will occupy 2 bytes no matter what (for example, the integer 0 occupies 2 bytes as does the integer 20000). Pascal has a predefined constant called *MaxInt* that always holds the maximum possible value of an integer. In Turbo Pascal, *MaxInt* has the value 32767. In some situations, you may want to test a data object with a greater range of values (such as a real) to see if it can be expressed as an integer. To do this, test the number to see if it falls between *MaxInt* and *−MaxInt−1*.

Integer Operators

When you do calculations with integers, you can use the operations that you might expect to work on any kind of number: +, −, *, and /. (The / operator actually does division on real numbers, but will work on integers because it converts both its operands to type real before it divides. The result of the / operator is always of type real.) If a value with a fractional part is assigned to an integer variable, the fraction is removed (truncated) and the integer assumes the value of the non-fractional part of the number.

The *relational* (or comparing) *operators* may also be used with integers. These include >, <, >=, <=, =, and <> (not equal to).

In addition to the preceding operations, Turbo Pascal has two special operations that can be applied only to integers: **div** and **mod**.

The **div** operator performs an operation known as *integer division* on two integers. Integer division works much like ordinary division, but the operands must both be integers and the result is always an integer

(any fractional part is dropped during the division). Use of the **div** operator is preferable to that of the / operator when the operands and the results are integers, since the **div** operation performs approximately ten times faster on most machines.

The **mod** or *modulo* operator divides its two operands (again using integer division) and returns the remainder. This operation is useful for "clock arithmetic." (As you already know, the hours on a clock go up to 12, then start at 1 again, so the next hour is the current hour plus 1, **mod** 12.) The **mod** operator also helps determine if a number is an exact multiple of another (if *A* is a multiple of *B*, then *A* **mod** *B* is zero).

Some other operators, such as **and**, **or**, **xor**, **shl**, and **shr**, also work on integers in Turbo Pascal. These are advanced features, however, and we will defer discussing them until later in the book.

Integers and Arithmetic Overflow

As we have already mentioned, integers in Turbo Pascal are limited to a very specific range of values. What happens, you may ask, if the result of an integer operation falls outside this range?

Arithmetic overflow is what happens when you calculate a value that is too big or small to fit and store in an integer that falls in the range −32768 to 32767. You can unintentionally introduce errors into your program if such an overflow occurs in the middle of a calculation.

For example, suppose your program tried to evaluate the expression 1000 * 100 **div** 50. This expression does *not* produce the answer you might expect—2000—because of arithmetic overflow. Turbo Pascal will attempt to multiply 1000 by 100, and since the result is too big to fit in an integer variable, you'll get the intermediate result of −31072. (This is the number corresponding to the lower 16 bits of the number 100,000—the rest of the number is lost because it doesn't fit.) This will then be divided by 50, producing an answer of −621.44. Since we're working with integers, the decimal fraction will be dropped, and the final result will be −621. This is hardly the correct answer!

Turbo Pascal expects you to be cautious when calculating with integers, and does not report an error when an arithmetic overflow happens. You must therefore take precautions to avoid this problem. In the previous example, writing the expression as

```
1000 * (100 div 50)
```

would have prevented arithmetic overflow from occurring.

Exercises Which of the following are valid constants of type integer? If not, why not?

1. 40000
2. −10,000
3. $b
4. Maxint
5. −32768
6. $A21H
7. 2.0
8. 0

BYTE

The data type byte includes all integer values that can be represented by 8 bits of computer memory. A limited set of a given type is called a *subrange*. Therefore, byte is a subrange of type integer. Data of type byte must be in the range of 0 through 255.

Byte values can be used in place of integer values and vice versa, but note that because of their limited range, it is even easier to generate an arithmetic overflow. Any value outside the allowable range for a byte data type will be represented as the lower 8 bits of the integer value. For example, if an arithmetic operation produces the result 256, and the result is assigned to a byte, the value of the byte becomes 0.

To explain, look at the individual bits of the integer result:

```
256     = 00000001         00000000
Decimal Most Significant Least Significant
Number  Byte             Byte
```

Despite this limitation, byte values are handy for numbers that will never have values outside the range 0 to 255. They also save space in memory, since they occupy half the amount of space required to store an integer.

Integers and bytes are fine for working with numbers without fractions, but there may be times when you want to divide, say, 11 by 4 and get a total of 2.75 instead of 2. There may also be times when you need to find the answer to 5234 * 21342. To do these things, you need to use the data type real.

REAL

A real constant, like all numeric constants, must begin with at least one digit. To be recognized as a real constant as opposed to an integer constant, it must then contain either a decimal point followed by one or more digits, an exponent, or both. Thus, 10.0 and 10.0000 are real

constants, while 10 is an integer constant. In Turbo Pascal, if the exponent is present, the digits following the decimal point may be omitted. The syntax diagram in Figure 9-2 shows an *unsigned real.*

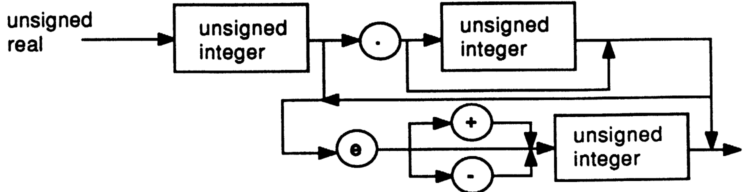


Figure 9-2 Syntax Diagram of Unsigned Real

Real Operators

When you do calculations with reals, you can use the basic four arithmetic operations: +, −, *, and /. It is important to note that if any of these operations is done on one value of type integer and another of type real, the result will be of the type real.

The relational operators may also be applied to values of type real. Again, if operands of types integer or byte are mixed with real numbers in such an expression, they are converted to type real before the operation is performed.

Exponential Notation

What is an *exponent*, and what is its purpose in representing a real number? Since real numbers must encompass very large and very small values, Pascal allows us to use *exponential*, or scientific, *notation* to represent them. For instance, the number

```
134,000,000,000,000
```

may be represented as:

```
1.34 * 10**14
```

And, since it is hard to show multiplication symbols and superscripts on many computer screens, we replace the "* 10" part of the number with the letter E to represent "exponent." Now, our number is represented as:

```
1.34E14
```

Exponential notation is also used to represent very small numbers. For example, the number:

```
0.0000023143
```

can be represented in exponential notation as:

```
2.3143 * 10**-6
```

and you can represent this same number in Pascal as

```
2.3143E-06
```

The number to the left of the *E* in the first example (1.34) is called the *mantissa*, and the number to the right of it is called the *characteristic*. To convert a number from exponential notation to ordinary notation, move the decimal point in the mantissa the number of places to the right indicated by the characteristic, and fill in any gap with zeroes. (If the characteristic is negative, you move the decimal point to the left.) To convert a number to exponential notation, move the decimal point until it is just to the right of the left-most nonzero digit in the number. The characteristic then becomes the number of digits you move the decimal point (negative if you move it to the right, positive if you move it to the left).

The range of real numbers in Turbo Pascal is 1E−38 through 1E38. The mantissa can have up to 11 digits. Of course, it takes more space to store this information than it does to store an integer, but there are many times when no other type will do.

The following are examples of real numbers:

```
1E5
3.1415926
3546.3
0.0034
32.E4
5.679E21
1.324E 2
21343.0
0.0
0.1
```

The following are examples of invalid representations of real numbers:

`-.123423`	No digit on the left of the decimal
`5.6231E42`	Exponent too large
`25.`	No digit on the right of the decimal
`E14`	No mantissa

The following are incorrect representations of real constants that will *not* cause compiler errors, but which may also *not* do what you expect:

`0`	Interpreted as an integer constant
`5654352311312.3`	More than 11 digits in mantissa

If you write a number that you intend to be a real constant without a decimal point or exponent, Turbo Pascal will *not* complain as long as the constant is smaller than the maximum integer. Instead, the compiler will interpret the number as an integer constant. Since Turbo

Pascal automatically converts integers to real numbers when necessary, this will never cause your code to execute incorrectly; however, the conversion will slow down the execution of the program slightly.

If you write a constant of type real that has more than 11 digits in the mantissa, Turbo will (once again) not complain. However, since the accuracy with which Turbo Pascal can represent such numbers is limited to 11 digits, the right-most digits (after the 11th) will be treated as zeroes. (If a calculation with reals produces a number too large or too small for Turbo to handle, the program will halt with an error message.)

Exercises Convert each of the following numbers to a legal Pascal constant in exponential notation.

1. 20000
2. −.000025
3. +42.77
4. −530000.5

Convert each of the following constants from exponential notation to standard notation.

1. 1.5E−10
2. −5.545454E12
3. 2E0

Type in the following program and run it. Why does Turbo Pascal print the value it does? What happened to the least-significant (right-most) digits?

```
program truncate;
  const a = 1234567890123456.0;
begin
  Writeln(a);
end.
```

BOOLEAN

Objects of the type boolean are limited to only two values: TRUE and FALSE. This type is named in honor of the 19th-century mathematician George Boole, who developed the rules that govern the operation of digital computers.

Boolean values are useful in recording whether certain conditions are true or not—especially when the computer must later decide what to do next. You may recall that the results of all of the relational operators are of type boolean—we'll show you how to use them to make decisions in Chapter 11, "Control Structures."

CHAR

The data type char includes the set of ASCII characters. Actually, a variable of type char can have 256 values, including the 128 characters in the standard ASCII set. The remaining 128 values depend on the type of computer you are using. On some computers, these high values are used to provide graphic characters for drawing boxes, lines, and tables. Other computers use these high values for alternate character sets. Some computers don't use them at all. Table 9-1 lists the complete ASCII character set.

As you may recall, the ASCII character set includes both printing and nonprinting characters. You represent printing characters by enclosing them in single quote marks, as follows:

```
'a' '$' ' ' 'J'
```

Notice that a space is a printing character and is represented as a space enclosed in single quotes, just like any printing character. Nonprinting characters are another matter, as are the characters prior to value 126 (those characters for which there is no key on your keyboard). Turbo Pascal provides special ways to represent these characters. Let's look at control characters first.

Control characters have ASCII values between 0 and 31. For example, the ASCII value of 7 represents Ctrl-G. However, you can't enclose a Ctrl-G in single quote marks because a Ctrl-G does not display anything on the screen.

So, how do you represent a control character? One way is to use the notation ^⟨char⟩, where ^ is the "caret" character (usually the shifted [6] key) and ⟨char⟩ is the corresponding printing character. You can determine the printing character that corresponds to a control character by looking at the third set of columns in Table 9-1. The table is set up so that a printing character in column 3 corresponds to a control character in column 1. For whatever purpose you may need this information, the printing character has a decimal value of 64 more than the corresponding control character.

That takes care of codes 0 through 31, but what about characters with ASCII codes 127 through 255? Turbo Pascal handles these characters by letting you use the notation #⟨val⟩, where # is the pound sign character (usually a shifted [3]) and ⟨val⟩ is a number between 0 and 255. In fact, you can represent *any* ASCII character using this method. Here are some examples of representing characters using the #⟨val⟩ method:

```
NUL          #0
Ctrl G (BEL) #7
ESC          #27
blank        #32
the digit 0  #48
```

Table 9-1 Complete ASCII Table

D	H	Ch	Ctrl	Mem	D	H	Ch	D	H	Ch	D	H	Ch	D	H	Ch	D	H	Ch	D	H	Ch	D	H	Ch
00	00		@	NUL	32	20	SP	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	└	224	E0	∝
01	01	☺	A	SOH	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	┴	225	E1	β
02	02	●	B	STX	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
03	03	♥	C	ETX	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	├	227	E3	π
04	04	♦	D	EOT	36	24	$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
05	05	♣	E	ENQ	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
06	06	♠	F	ACK	38	26	&	70	46	F	102	66	f	134	86	å	166	A6	ª	198	C6	╞	230	E6	μ
07	07	•	G	BEL	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
08	08	◘	H	BS	40	28	(	72	48	H	104	68	h	136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
09	09	○	I	HT	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
10	0A	◙	J	LF	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
11	0B	♂	K	VT	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
12	0C	♀	L	FF	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
13	0D	♪	M	CR	45	2D	–	77	4D	M	109	6D	m	141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
14	0E	♫	N	SO	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
15	0F	☼	^O	SI	47	2F	/	79	4F	O	111	6F	o	143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
16	10	►	^P	DLE	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	░	208	D0	╨	240	F0	≡
17	11	◄	Q	DC1	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
18	12	↕	R	DC2	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
19	13	‼	S	DC3	51	33	3	83	53	S	115	73	s	147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
20	14	¶	T	DC4	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
21	15	§	U	NAK	53	35	5	85	55	U	117	75	u	149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
22	16	▭	V	SYN	54	36	6	86	56	V	118	76	v	150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
23	17	↨	^W	ETB	55	37	7	87	57	W	119	77	w	151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
24	18	↑	^X	CAN	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
25	19	↓	^Y	EM	57	39	9	89	59	Y	121	79	y	153	99	Ö	185	B9	╣	217	D9	┘	249	F9	•
26	1A	→	Z	SUB	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	║	218	DA	┌	250	FA	•
27	1B	←	[	ESC	59	3B	;	91	5B	[	123	7B	{	155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
28	1C	∟	\	FS	60	3C	<	92	5C	\	124	7C	¦	156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
29	1D	↔	]	GS	61	3D	=	93	5D	]	125	7D	}	157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
30	1E	▲	^	RS	62	3E	>	94	5E	^	126	7E	~	158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
31	1F	▼	^_	US	63	3F	?	95	5F	_	127	7F	△	159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

```
the letter A #65
DEL          #127
ASCII 237    #237
```

Many of us are most comfortable working with decimal numbers, especially when we're doing arithmetic. However, if you are an experienced programmer, you are probably quite familiar with the hexadecimal system. As mentioned previously, this numbering system has 16 digits instead of the usual 10: 0 through 9 plus *A* through *F* (which represent the numbers 10 through 15). Thus in hexadecimal the number 15 (decimal) is written as $0F, while the number 16 is written as $10 (a 1 in the 16's place and a 0 in the 1's place). The dollar sign indicates the number is in hexadecimal (or, as programmers call it, "hex"). You can use the notation #$⟨val⟩, where ⟨val⟩ is the hexadecimal code for the desired ASCII value. Here is the list of previous characters shown with their hexadecimal values:

```
NUL          #$0
Ctrl G (BEL) #$07
ESC          #$1B
blank        #$20
the digit 0  #$30
the letter A #$41
DEL          #$7F
ASCII 237    #$ED
```

You now have at your disposal two ways of representing printing characters, two ways of representing control characters, and one way to represent the high value ASCII characters. Each character occupies 1 byte of memory.

STRINGS

As we mentioned before, *strings* are really not a predefined data type. They are, rather, a group of *structured types* that you can specify using a special shorthand. Unlike the data types integer, real, boolean, and char, string types weren't included in Wirth's original definition of standard Pascal. Because of this, many Pascal implementations have their own definitions of strings. Turbo Pascal has one such implementation. Strings are important enough to be the subject of an entire chapter (Chapter 14), but you need to learn a few things about them now to be able to follow the examples in the next few chapters.

Briefly, a string is a sequence of characters with a specific starting point, stopping point, and length. For example, 'Enter first value: ', is a string. The single quotes (') show where it starts and stops; it contains the characters *E*, *n*, *t*, *e*, *r*, and so on. Its length—the number of characters between the single quotes—is 20, including two spaces at the end.

String Constants

A string constant consists simply of a group of characters, enclosed in single quotes, and/or a group of special symbols that represent control characters. Figure 9-3 displays the syntax diagram for a string constant.

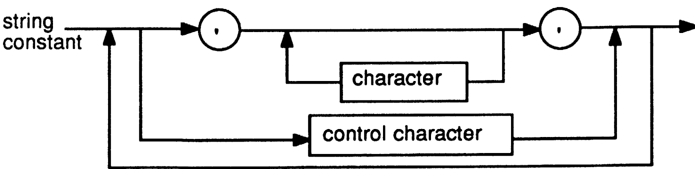


Figure 9-3 Syntax Diagram of String Constant

The following are examples of legal string constants:

```
'a'
'Hello, how''s it going?'
'WallawallawallawallawallaFAGROON!'
''
```

Note that in the second of these examples, we have used two single quotes immediately following one another to represent a single quote within the string. When this string is printed, the result is:

```
Hello, how's it going?
```

Any number of single quotes can be placed within a string in this way. (Of course, the beginning and ending single quotes still have their usual meaning.)

The last of these constants demonstrates the one case where two consecutive single quotes do not print as a single quote: the null string. The null string is a string of zero length, and does not show anything when written.

Putting Control Characters in a String

Turbo Pascal provides special facilities to let you place control characters within a string. Let's say you want to signal the end of a long processing job and prompt the user for additional input. To alert the user, you want to make the computer beep three times as it displays a message on the screen.

On most computers, the way to produce a beep is to print the control character ^G (Ctrl-G, or ASCII code 7). Since this is not a normal printing character, how do you tell Turbo Pascal to put it in the string?

In Turbo, you can put a control character in a string by specifying it in one of two ways: by number, or by the ASCII character that you would press along with the control key to generate it.

The general form for a control character in a Turbo Pascal string constant is shown in Figure 9-4.

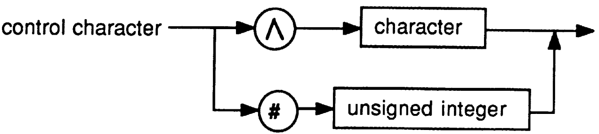


Figure 9-4 Syntax Diagram of Control Character

To represent Ctrl-G, you can use either the symbols ^G or #7, as shown in the following:

```
const
  alert = 'Please wake up!!!'^G^G^G' I need more data!!!';
```

or

```
const
  alert = 'Please wake up!!!'#7#7#7' I need more data!!!'
```

Note that we use a closing quote to indicate the end of the first group of ordinary characters, and another one to begin the second. (Had we not done this, Turbo would have printed our ^G or #7 symbols verbatim.) As long as there are no intervening spaces, Turbo Pascal considers this combination to be a single string.

In addition to placing control characters in strings, you can make a string of only control characters. To do this, you must list all control characters with no intervening spaces, like this:

```
const
  allcontrol = #27^U#20;
```

This string of control characters has a length of three characters.

Declaring String Variables

You can declare a variable of a string type as you would declare any other variable: identifier, colon, and type. In this case, the type will be a string type. The syntax diagram in Figure 9-5 shows how to specify a string type.

Figure 9-5 Syntax Diagram of String Type

The unsigned integer in the string type definition is a value from 1 to 255 (a string can be up to 255 characters in length), giving the maxi-

mum number of characters a string of that type can hold. When you declare a variable of type **string**[⟨*n*⟩], where ⟨*n*⟩ represents an integer, Turbo Pascal will set aside $n + 1$ byte of memory to hold the largest possible string: *n* bytes to hold the string, and 1 byte to hold the current size of the string.

Here is an example of an actual string declaration:

```
CompanyName : string[40];
```

In this case, 41 bytes are set aside for *CompanyName*. If you then execute the statement

```
CompanyName := 'Borland';
```

the current length of *CompanyName* is 7, even though the maximum length is still 40. If you think of the string being stored as a series of characters in successive bytes of memory, then the first byte would contain the current length of 7, the second byte would contain the character *B*, the third byte *o*, and so forth. The unused characters (if any) at the end of the string can contain anything at all; Turbo Pascal will not use them.

REVIEW

In this chapter, we introduced you to Turbo Pascal's predefined data types (integer, byte, real, boolean, and char). We also gave you a quick introduction to strings, which are really a Turbo Pascal data structure and not a type.

To summarize, an integer is a positive or negative whole number, including zero. Integers require 2 bytes of memory each, and can be in the range of −32768 through 32767.

A byte is a subrange of type integer, and is equivalent to the least significant 8 bits of an integer value. Byte values must be in the range of 0 through 255, and occupy 1 byte of memory each.

Real values consist of a mantissa and an optional positive or negative exponent. The mantissa may have up to 11 significant digits. The maximum and minimum values for reals are 1E−38 through 1E38.

Logical values of TRUE and FALSE are best handled by the data type boolean. A boolean value occupies 1 byte of memory.

The type char is one of 256 ASCII characters, including both printing and nonprinting characters. The printing characters are represented by enclosing them in quotes, while the nonprinting characters are represented by decimal or hexadecimal ASCII values preceded by a pound sign (#). In addition, control characters can be represented by a caret (^) followed by the corresponding printing character. Char data types occupy 1 byte of memory each.

Finally, the string data types can have a defined length of 1 to 255 characters and are declared with a maximum length. A string occupies 1 more byte in memory than its maximum length.

Now that you have a fundamental understanding of predefined data types, you are ready to expand your knowledge with declared scalar types in the next chapter.

10 Defined Scalar Types

You now know about the predefined types of Pascal: integer, byte, boolean, real and char; we have also introduced the notion of string types. In this chapter, we will explain the concept of a *scalar type*, and show you how to create one for use in your program.

A scalar type is a type in which all possible values can be said to be in order—from the first to the last—with no gaps. Some of the predefined types are scalar types; in particular, integer, byte, boolean, and char. Real is not considered to be a scalar type, because adding 1 to a real does not always produce the next real number. (In fact, adding 1 to a large real, such as 1.0E37, has no effect. The digit of the number that would need to be changed is more than 11 digits below the left-most digit of the number, and Turbo only remembers the 11 most-significant digits of a real). Similarly, string types are not scalar types. After all, it would be hard to decide how to find the "next" string after any given string.

In real life, there are many types of objects that have a limited range of values and a definite order. A day of the week, for example, can range from Sunday to Saturday, and the days come one after the other in a fixed sequence. Pascal lets you tell the compiler about such types and how to manipulate data of those types.

A *defined scalar type* is a data type with a user-defined range of values and a user-defined order. The range of values is defined by the number of elements you declare, and the order is defined by the order in which the elements are declared. Properly used, these types can improve the readability of your programs, save space in memory, and allow range checking to be performed to catch errors in your program. We'll discuss two kinds of defined scalar types: *enumerated scalar* and *subranges*.

ENUMERATED SCALAR TYPES

Good programming style means being considerate of other programmers who may have to read and understand your code. For example, if you want to use a variable to hold a day of the week in your program, you could declare a variable *DayOfWeek* of type integer, and then use numbers for the days. But which should be the first day of the week, Sunday or Monday? And, should the first day of the week be represented by a 0 or a 1? If another programmer tries to modify your program, that person will have to determine what number stands for which day.

The way to address these potential problems is *not* to declare *DayOfWeek* to be of type integer. Instead, you can declare a type specifically for the purpose of holding a day of the week, and *enumerate*, or list, the values it can have:

```
program Day_Of_Week_Example;
type
  Days = (Monday,Tuesday,Wednesday,Thursday,
          Friday,Saturday,Sunday);
var
  DayOfWeek : Days;
begin
  DayOfWeek := Thursday;
  .
  .
  if DayOfWeek = Saturday then
    Writeln('It''s Saturday. Why are you at work?');
  .
  .
end.
```

Now there is absolutely no confusion over which days are represented by which numbers; they are represented by their actual names. This way, any Pascal programmer who looks at your program will know that the value Thursday is assigned to the variable *DayOfWeek*.

As you can see from the example, to declare a scalar type you must first give an identifier for the type, then list, in order, identifiers for all of the values the type can have. You can declare scalar types for just about any set of values, including those with no inherent order. However, if order is important, you *must* declare your list of values from the first to the last.

In a Pascal program, you define user-defined types in the declaration part, in the section called the *type definition part*. The syntax of this part of the program is shown in Figure 10-1.

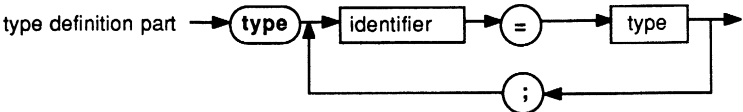


Figure 10-1 Syntax Diagram of Type Definition Part

Each type definition names the type to be defined, then tells the compiler the details of the type. An enumerated scalar type is a simple type (see Figure 10-2). The highlighted part of this diagram shows the syntax for the specification of an enumerated scalar type.

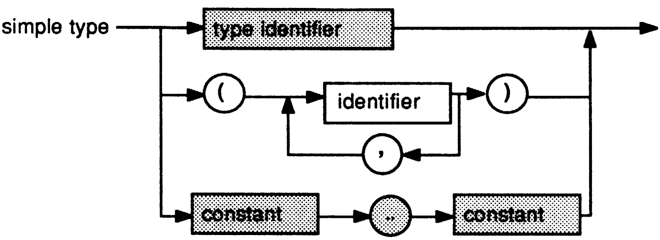


Figure 10-2 Syntax Diagram of Simple Type

It is important to remember that when you list the values of an enumerated type, you are in fact declaring the names of those values as identifiers. It is therefore *not* legal to declare:

```
type
  Days = (Monday,Tuesday,Wednesday,Thursday,
          Friday,Saturday,Sunday);
  DaysOff = (Saturday,Sunday);
```

because to do so would be to declare Saturday and Sunday twice. In Pascal, each identifier must have one, and only one, type.

Ordinal Values

All enumerated scalar types (and, indeed, all scalar types) are inherently ordered. In other words, there is a lowest value, a highest value, and a number of distinct values in between. In the data type *Days* in the previous example, Monday is the lowest value and Sunday is the highest value. Tuesday through Saturday are the values between the lowest and highest values. The lowest value is considered to have an

ordinal value of 0, while the highest value has an ordinal value equal to the total number of values defined, minus 1 (because we started counting at 0). For example, here are the ordinal values of the type *Days*:

```
Element     Ordinal Value

Monday           0
Tuesday          1
Wednesday        2
Thursday         3
Friday           4
Saturday         5
Sunday           6
```

Standard Functions for Scalar Types

To simplify the use of enumerated types, and scalar types in general, the Pascal language provides certain functions or operations that let you manipulate objects of these types. In particular, these functions help you to find:

- The value that comes before a value of the type (its predecessor).
- The value that comes after a value of the type (its successor).
- The position of a value in the list of values in the type definition.

Here's a scenario that will help explain why this information might be useful. Imagine that in your program you have defined the type *Days* and two variables of that type, *DayOfWeek* and *NextDay*:

```
  .
  .
  .
type
  Days = (Monday,Tuesday,Wednesday,Thursday,
          Friday,Saturday,Sunday);
var
  DayOfWeek, NextDay : Days;
  .
  .
  .
```

Now suppose that somewhere in your program you want to set *NextDay* equal to the day following *DayOfWeek*. How do you do it? Well, one thing you *cannot* do is attempt to add 1 to the value of *DayOfWeek*. The statement

```
NextDay := DayOfWeek + 1;
```

will cause the compiler to give you the error message "Type Mismatch Error" —and rightly so. The operation of addition is not defined for objects of the type *Days*. Yet, we know intuitively that we want an operation that somehow adds one to a day of the week, finding its

successor in the list of weekdays. Pascal comes to the rescue by providing exactly this operation in the form of the *Succ* (for successor) function. Using this function, we can write

```
NextDay := Succ(DayOfWeek);
```

and achieve the desired effect. If *DayOfWeek* has the value Sunday, *NextDay* is assigned the value Monday; if it's Monday, *NextDay* will become Tuesday, and so on. A *function* is an operation that takes one or more values (called *parameters*, or sometimes *arguments*) and uses them to produce a new value. (The sine function, for instance, takes an angle and returns a number that is related to that angle.) In the previous example, we made the function *Succ* work on the parameter *DayOf-Week* to produce (or return) a new value, which was the next value in the type *Days*. The new value was then assigned to the variable *NextDay*.

The *Succ* function is one of three predefined "standard" functions that manipulate objects of scalar types; the other two are *Pred* and *Ord*.

As you might have already guessed, the *Pred* function does exactly the reverse of what the *Succ* function does: it returns the predecessor of the value given to it as a parameter. The *Ord* function returns the ordinal value of its parameter.

Here are some examples of the results of using these functions. Given that at a particular instant *DayOfWeek* = *Wednesday*, then:

```
Pred(DayOfWeek)        = Tuesday
Succ(DayOfWeek)        = Thursday
Pred(Pred(DayOfWeek))  = Monday
Succ(Succ(DayOfWeek))  = Friday
Pred(Succ(DayOfWeek))  = Wednesday
Succ(Pred(DayOfWeek))  = Wednesday
Ord(DayOfWeek)         = 2
Ord(Pred(DayOfWeek))   = 1
Ord(Succ(DayOfWeek))   = 3
```

Note that *Ord* returns a value greater than or equal to 0 for all scalar types except integer. This is because all scalar types (except integer) start with an ordinal value of 0. For values of type integer, *Ord* returns the actual integer value, so there really is no reason to use *Ord* with integer types (or byte types, for that matter). After all, you can look at the integer and see its ordinal value! In the earlier examples, we also demonstrated that it is perfectly legal to *nest* functions—that is, to apply a function to the result of another. The expression *Pred(Pred(DayOf-Week))* is a convenient way to get the day two days before *DayOfWeek*.

Cyclical Enumerated Types—Avoiding Range Errors

One thing to watch out for when using *Succ* and *Pred* is the problem of searching for a nonexisting successor or predecessor of an existing value. For instance, suppose the variable *DayOfWeek* in the previous example had the value Sunday, and you tried to assign

```
NextDay := Succ(DayOfWeek);
```

What would happen? Well, Turbo Pascal has no way of knowing that the days of the week run in a cycle—that is, that the "successor" of the last value of the type is the first value of the type. So instead, it tries to assign to *NextDay* a value with the ordinal value 7 (one more than the ordinal value of Sunday, which is 6).

The result of this assignment depends on the conditions present when you compiled your program. If you tell the compiler to range-check (explained in the next section), it will realize that there is no "successor" for Sunday, and will stop the program and display an error message. If the compiler is not range-checking (which is the default), it will not catch the error, and your program will behave erratically.

Neither of these is a desirable condition. To achieve the correct result, use an **if** statement to handle the special case:

```
if DayOfWeek = Sunday then
  NextDay := Monday
else
  NextDay := Succ(DayOfWeek);
```

While we haven't described the **if** statement in detail yet, the preceding one is very close to what actually needs to be done. If *DayOfWeek* has the value Sunday, then we want to explicitly set *NextDay* to the value Monday; otherwise, the *Succ* function will provide the correct value.

Exercises Rewrite the previous code fragment to set the variable *Yesterday* to the day before *DayOfWeek*. How do you handle Sunday?(Solutions are in Appendix B.)

Of course, the same situation can arise even if the enumerated type in question is not cyclical—though it is less likely. If you defined the type

```
Rank = (Peon, Manager, SeniorManager, VicePresident,
        President, Chairperson);
```

and tried to find the next rank after Chairperson, the same type of error could occur. You must construct your program so that this does not happen.

Range-Checking

While defined scalar types are designed to help you restrict the values of your variables to fall within certain predetermined limits, Turbo Pascal does not enforce those limits unless it is specifically told to do so. For

instance, the previous example, in which we attempted to take the successor of the last value of an enumerated type, would not cause an error message in Turbo Pascal unless the feature called *range-checking* was enabled.

Range-checking is turned on by a special kind of comment called a *compiler directive.* A compiler directive consists of a comment in which the first character is a dollar sign ($) and the remaining characters fit the pattern(s) of one or more legal directives.

The compiler directive that turns on range-checking is {$R+}. We recommend that you put this directive before the program heading of every program you write; it will almost certainly save you many hours of frustrated debugging. Only if your program must run at maximum speed, or if it is about to overflow memory, should you omit this directive or use the directive {$R-} to turn range-checking off.

Note that your programs will compile and run faster with range-checking off. However, since turning range-checking off may hide bugs, we recommend it only for well-tested programs that are being published as finished products.

Undefined Values in Enumerated Types

Sometimes, when working with defined scalar types, it is useful to consider what will happen if no value is provided for a variable, or if your program must have a "none of the above" choice for a value. For example, there may be times when you need to begin a program with your variables set to a known value, but one that is not normally associated with your type. (Putting your variables into a known state before doing anything else is known as *initializing.*)

A good way to do this is to add an extra value to your type to reflect this "undefined" state. For instance, in the first example in this chapter, we could have written

```
program Day_Of_Week_Example;
type
  Days = (Noday,Monday,Tuesday,Wednesday,
          Thursday,Friday,Saturday,Sunday);
var
  DayOfWeek : Days;
begin
  DayOfWeek := Noday;
  .
  .
  .
end.
```

Now when the program begins, *DayOfWeek* has a known value: Noday. If later in the program you want to test whether or not you

have assigned a value to *DayOfWeek*, you can test to see whether *DayOfWeek* still equals Noday.

SUBRANGES

Another important kind of scalar type is called a *subrange*. A subrange is a group of consecutive values that is part of another scalar type. They are useful when you want to limit the possible number of values a variable can have to a subset of the original type.

A subrange is specified by the minimum and maximum values to be allowed in the subrange, separated by two periods (..). In the following example, we define an enumerated scalar type and two of its subranges:

```
type
  Days = (Noday,Monday,Tuesday,Wednesday,
          Thursday,Friday,Saturday,Sunday);
  Workdays = Monday..Friday;
  Weekend = Saturday..Sunday;
```

The type from which the subrange is derived is called the *base type* of that subrange. Therefore, *Workdays* is a subrange of *Days* and *Days* is the base type of *Workdays* and *Weekend*. An important consideration when declaring subranges is that the minimum value (the one specified first in the declaration) must not have a greater ordinal value than the maximum value (the one specified last in the declaration).

Here are more examples of subranges:

```
type
  CompassRange = 0..359; {Subrange of the base type integer}
  ValidEntry = 'A'..'F'; {Subrange of the base type char}
  MonthlyIncome = 10000..30000;
  Hours = 0..23; {Subrange of integer (or byte)}
  Minutes = 0..59; { " " " }
  {Here is a defined scalar type...}
  MusicType = (Notype,Classical,Jazz,Folk,RhythmBlues,Rock,
               HardRock,AcidRock,HeavyMetal);

  { ...and here is a subrange of that type.}
  MusicILike = Classical..Rock;
  {Subrange of the base type Musictype}
```

Subranges can be used to design menus (or any other human interface element of your program) in which the only valid entries are a subrange of an existing type.

Subranges are often used in defining other elements of Pascal, such as arrays and records; however, these uses will be covered later.

Subranges as Anonymous Types

So far we've defined subranges in **type** declarations, implying that to use a subrange type you must define it, then declare variables with it. However, there is an easier way—you can specify a subrange directly in the variable declaration.

For instance, after you have declared

```
Days = (Noday,Monday,Tuesday,Wednesday,Thursday,Friday,
        Saturday,Sunday);
```

you can write:

```
var
  Workday : Monday..Friday;
  {This is a specification of a subrange type that
   has no name.}
```

rather than

```
type
  WorkingDay = Monday..Friday;
var
  Workday : Workingday;
```

The ability to specify subranges in this way saves you from inventing a name for every subrange you want to use. Because the subrange type is never named, it is called an *anonymous type.*

Enumerated types can be defined anonymously as well, though it is not recommended. It is legal to declare

```
var
  Day1 : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
```

Note, however, that the identifiers *Mon* through *Sun* are now defined as part of the anonymous type, and may not be used for anything else. For this reason, you may *not* declare:

```
var
  Day1 : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
  Day2 : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
```

If you attempt to compile this, you will get a error when the compiler gets to the second declaration since the second anonymous, but distinct, type uses the same identifiers as the first. Instead, you should write

```
var
  Day1,Day2 : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
```

Two other restrictions also apply to anonymous enumerated types: (1) You cannot coerce variables to an anonymous scalar type (that is, convert them from other types to an anonymous type), because the name of the type is required to perform the operation; (2) You cannot pass such variables as typed parameters to a subroutine or procedure, since there is no data type to use in the declaration of the formal

parameter. These restrictions will be more important to you later, when you begin writing code using these advanced features.

INPUT AND OUTPUT

It would be convenient if you could read and write objects of enumerated scalar types directly, but Pascal won't allow it. For example, if you wanted to display the current value of *DayOfWeek*, it would be nice if you could write

```
Writeln('Today is ',DayOfWeek);
```

But this statement will produce an error when you compile your program. The same is true for a statement such as:

```
Readln(DayOfWeek);
```

To overcome this limitation of Pascal, you need to explicitly tell the compiler to write, or look for, specific strings. In Chapters 11 and 13, respectively, we'll show you how to use the **case** statement and *arrays* to accomplish this.

MEMORY USAGE

One of the advantages of defined scalar types is efficient use of memory. A variable of a defined scalar type that has less than 256 possible values uses only 1 byte of memory (since it always has a positive ordinal value and 256 different elements can be represented by a byte). Furthermore, if you define a subrange of the type integer that has a minimum value greater than or equal to 0 and a maximum value less than or equal to 255, only 1 byte of storage is required for a variable of that type.

REVIEW

Defined scalar types are data types that you define; they include enumerated types and subranges of existing scalar types. Defined scalar types can tremendously aid program development, documentation, and maintenance, especially if you apply your knowledge of them carefully and logically.

Now that you're clear on the use of types, we can move on to the next level of complexity, the use of control structures in Pascal.

11 Control Structures

So far, in almost every sample program we've shown, the statements have been executed sequentially, from the first **begin** to the last **end**. While this sort of execution is straightforward and easy to understand, it doesn't lend itself to repetitive tasks, or to making decisions and acting on them.

To allow a program to do these things, the Pascal language provides *control structures*—special statements that divert the flow of control from its usual sequence. In Pascal, these structures fall into four categories: *conditional* (the **if** statement), *iterative* (the **for**, **while**, and **repeat . . . until** statements), *case* (the **case** statement), and *goto* (the **goto** statement). We will cover all but **goto** in this chapter; see Chapter 25 in Part III for more detail on this statement type.

CONDITIONAL EXECUTION: THE IF STATEMENT

In earlier chapters, we touched briefly on the notion of an **if** statement, a statement that tells the computer to do something only if a certain condition is satisfied. We will now explain this construct in detail.

Chapter 10 showed this simple example of the **if** statement:

```
if DayOfWeek = Saturday then
  Writeln('It''s Saturday. Why are you at work?');
```

Here we tell the computer: "Compare the value of the variable *DayOfWeek* to the constant *Saturday*. If they are equal, then perform the enclosed *Writeln* statement; otherwise, do nothing." This is an example of the simplest form of the **if** statement: the reserved word **if**, followed by a boolean expression (that is, one that yields the value TRUE or FALSE), the reserved word **then**, and finally a statement. Note that the semicolon at the very end of this example marks the end of the **if** statement, not of the enclosed *Writeln* statement. Therefore, if the **if** statement were not followed by another statement, the semicolon could be omitted.

Another form of the **if** statement allows the computer to choose between two possible actions. In another example from the last chapter, we wrote

```
if DayOfWeek = Sunday then
  NextDay := Monday
else
  NextDay := Succ(DayOfWeek);
```

Here, the format of the statement is similar, but we have added the reserved word **else** and another statement—one to be done if the boolean expression *DayOfWeek = Sunday* does *not* yield the value TRUE. Since the **if** statement doesn't end after the enclosed (or nested) statement *NextDay := Monday,* and there can be *only one* nested statement, there is no semicolon there. This is an important part of the format of the **if** statement. A semicolon that occurs immediately before an **else** will cause the compiler to generate an error message, because it will cause the compiler to "think" that the **if** statement is over. The syntax diagram for the **if** statement is depicted in Figure 11-1.

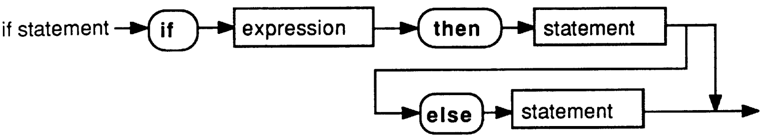


Figure 11-1 Syntax Diagram of If Statement

Compound Statements and the if Statement

As we mentioned earlier (and as you can see from Figure 11-1), the **if** statement allows only one statement in the **if** clause, and only one in the optional **else** clause. If this seems overly restrictive, we agree—in most cases, you would probably like to have the computer decide to do more than one thing if a certain condition is met.

The "brute force" way to do this might be to write a separate **if** statement for each and every statement you want to execute conditionally. However, this would make your program quite cluttered. It would also cause the program to test a condition repeatedly, when it really needs to be tested only once.

Fortunately, there is a better way. In Pascal, one can group a series of statements together in such a way that the **if** statement "sees" the group as one large statement; this is called a *compound statement.* It is constructed by enclosing a series of statements between the reserved words **begin** and **end**, creating a sort of "program within a program." The

syntax of a compound statement is in fact exactly the same as that of the statement part of a program, as shown in Figure 11-2.

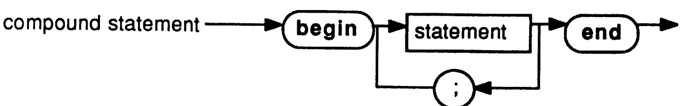


Figure 11-2 Syntax Diagram of Compound Statement

Thus, if we wanted to expand our first example to do more than one thing when it discovers that our user is working on Saturday, we could write:

```
if DayOfWeek = Saturday then
begin
  {The compound statement begins here...}
  Writeln('It''s Saturday. Why are you at work?');
  Writeln('Why not go home and watch some TV instead?')
  {...and ends here.}
end;
```

Furthermore, if we wanted to put more than one statement into the **else** clause, we could do that as well:

```
if DayOfWeek = Saturday then
  Writeln('It''s Saturday. Why are you at work?')
else
begin
  {This compound statement is in the "else" clause. }
  Writeln('It isn''t Saturday.');
  Writeln('Quit messing around and get to work!');
end;
```

You'll also find compound statements useful when working with the **while** and **for** statements (covered later in this chapter).

Boolean Expressions

The expression *DayOfWeek* = *Saturday*, as used in the previous example, falls into the class known as boolean expressions because it yields a value of the type boolean. Objects of type boolean may have only the values TRUE and FALSE, and are used to make decisions.

One of the most common boolean expressions contains a relational operator (an operator that compares two numbers or other objects). For example, suppose the integer variables *Score* and *Maximum* have the values 10 and 0, respectively. Here are some boolean expressions that apply relational operators to these variables and the results that they yield:

```
Score > Maximum  --> TRUE  (is greater than)
Score = Maximum  --> FALSE (is equal to)
Score < Maximum  --> FALSE (is less than)
Score >= Maximum --> TRUE  (is greater than or equal to)
Score <= Maximum --> FALSE (is less than or equal to)
Score <> Maximum --> TRUE  (is not equal to)
```

The relational operators don't just apply to numbers, however. All of the relations shown can be applied to objects of any scalar type, even enumerated types (since enumerated types have a definite ordering). Thus, it is legal to write

```
if DayOfWeek > Friday then
  Writeln('It''s the weekend!');
```

Relational operators are not the only operators that yield results of type boolean. The **not** operator, for example, "inverts" its boolean operand: **not** FALSE yields TRUE, and **not** TRUE yields FALSE. A boolean expression can, of course, also contain one or more variables of type boolean, and a boolean variable can be assigned the result of a boolean expression. For instance, if we declare

```
var
  NewMaximum : boolean;
```

then we can write the assignment statement

```
NewMaximum := Score > Maximum;
```

and use the variable *NewMaximum* to remember the result of the comparison.

Combining this with our earlier example, we can create a series of statements that determines if the player of a game has set a new high score, then prints an appropriate message.

```
NewMaximum := (Score > Maximum);
{NewMaximum is TRUE if new high score}
if NewMaximum then
begin {Compound statement to record
       score and congratulate the winner}
  Maximum := Score;
  Writeln('Congratulations!');
  Writeln('Your new high score is ', Maximum)
end
else
begin {Compound statement to print consolation message}
  Writeln('Your score was ', Score);
  Writeln('Nice try!')
end;
```

More Boolean Operators

You can create more complex expressions using boolean operators **and**, **or**, and **xor**. The **and** operator returns the value TRUE if (and only if) both of its operands are TRUE; thus,

```
FALSE and FALSE --> FALSE
FALSE and TRUE  --> FALSE
TRUE  and FALSE --> FALSE
TRUE  and TRUE  --> TRUE
```

Sometimes, it is convenient to show the results that a boolean operator returns in a *truth table*. This table shows the result of the operation for all possible combinations of values of the operands, much like a multiplication table shows the results for some of the operands of the multiplication operation. The truth table for the **and** operation looks like this:

and	F	T
F	F	F
T	F	T

You read a truth table exactly as you would a multiplication table: Find the values of the operands on the edges of the chart, then find where the row and column of the two operands intersect. The value in that box is the result of the operation. (F, of course, stands for FALSE, and T for TRUE.)

Exercises For practice, try reading the results listed for the **and** operator in the previous truth table. Do they agree with the results shown earlier?

The **or** operator, as you might have already guessed, returns the value TRUE if *either or both* of its operands are TRUE. Its truth table looks like this:

or	F	T
F	F	T
T	T	T

Finally, the **xor**, or *exclusive or*, operator returns the value TRUE if one, but not both, of its operands has the value TRUE. The truth table for the **xor** operation takes this form:

xor	F	T
F	F	T
T	T	F

Using these operators, you can create such expressions as

```
(Score > Maximum) or (Score > 30000)
```

TRUE if either *Score* is greater than *Maximum*, or *Score* is greater than 30000.

```
(Score > 10000) and (Score <= 20000)
```

TRUE if *Score* is both greater than 10000 and less than or equal to 20000.

or

```
not (NewMaximum or (Maxiumu = 0))
```

TRUE if neither *NewMaximum* nor the expression *Maximum* = 0 is TRUE.

If we assume that *Score*, *Maximum*, and *NewMaximum* have the values 10, 0, and TRUE, then the first expression yields the value TRUE, while the other two yield FALSE. Note the copious use of parentheses in these examples. Since Turbo Pascal allows some of the boolean operations to apply to integer as well as boolean values (a feature we'll discuss in the advanced part of this tutorial), it is very important to enclose your boolean expressions in parentheses to be sure they do what you intend them to.

REPETITIVE TASKS

Iteration

Conditional execution, as provided by the **if** statement, lets your programs make decisions; still, at this point, each statement in your program can execute, at most, only once. In Part I, we said that one of the things computers do best is perform tedious, repetitive tasks without tiring. We are now ready to describe the Pascal constructs that unlock this power.

The Pascal language provides you with three ways of telling the computer to repeat a series of statements until some condition is met. Using the **while**, **repeat...until**, and **for** statements, you will be able to tell your computer to "Do this 10 times," or "Do this until the task is completed." This kind of repetitive process is known as *iteration*, and the section of code that performs this activity is often called a *loop*.

This is a good time to warn you: Be careful when using repetitive statements. It's easy to start a loop that never reaches its ending condition. Read the section entitled "Endless Loops" before beginning your experiments.

The While Statement

The first kind of iterative statement we'll learn about is the **while** statement. The **while** statement tells the computer to repeat a nested statement (which, of course, can be a compound statement), as long as a certain condition holds true.

Here's a simple program that while not particularly useful (useful programs are generally complex) is an apt demonstration of how the **while** statement works:

```
program WaitForkey;
{ Program to wait for a keyto be struck on the keyboard }
{$R+,C-}
{ Range-checking on; Ctrl-C breaks off }
begin
  Write('Waiting for a keystroke');
        { Announce that we're waiting }
  while not KeyPressed do
  { Write dots continuously until a key is hit. }
  Write('.'); { KeyPressed is a Turbo Pascal function }
              { that returns TRUE only after a key has
                been struck.}
end.
```

We suggest that you type this program into Turbo Pascal, then watch it run. Notice how quickly the dots appear on the screen! The **while** statement is a very fast way of repeating one or more statements many times.

Note that the first line in the program contains the {$R+} compiler directive and a {$C-} directive as well. The {$C-} directive tells Turbo whether to allow your program to be interrupted when a user presses Ctrl C (or Ctrl Break on some systems). If the *C* directive is not specified, Turbo will make your program interruptable. Here we've turned *C* off because a side effect of {$C+} (the default) is that an occasional keystroke is "lost" in a tight, fast loop like the sample program. The syntax of the **while** statement is shown in Figure 11-3.

Figure 11-3 Syntax Diagram of While Statement

The **while** statement consists of the reserved word **while**, followed by an expression (which must be of type boolean), the reserved word **do**, and then a statement to be repeated. When the **while** loop is encountered during execution of the program, the boolean expression is evaluated. If the expression yields the value FALSE, the nested statement inside the **while** statement is never executed at all; if the expression returns TRUE, the nested statement is executed. The expression is

then evaluated again, and the process of test-execute-test-execute continues until the expression returns the value FALSE.

The Repeat...Until Statement

Like the **while** statement, the **repeat...until** statement causes a process to repeat until a condition is satisfied. However, its syntax and its effects are very different.

The syntax of the **repeat...until** statement is shown in 11-4.

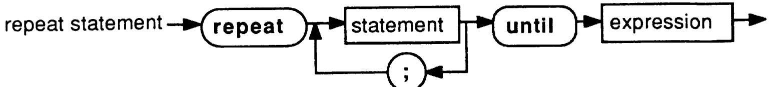


Figure 11-4 Syntax Diagram of Repeat Statement

Unlike either the **if** or the **while** statement, the **repeat...until** statement can enclose as many other statements as desired—separated by semicolons if there is more than one. Thus, there is no need to use a compound statement to make the **repeat...until** statement execute more than one statement on each iteration.

The **repeat...until** statement also differs from the **while** statement in that the statements within the loop are always executed at least once. The boolean expression that occurs after the **until**, at the end of the loop, is evaluated *after* the enclosed statements are executed. If the expression yields the value FALSE, the loop is repeated; otherwise, execution continues with the next sequential statement.

The following program illustrates the use of the **repeat...until** statement. It plays a game in which it asks the user to guess a number from 1 to 10. The player is always asked to guess at least once, but the game only continues until the user gets the number right. It is therefore a problem for which a **repeat...until** loop is an ideal solution.

```
program GuessingGame;
{$R+}
{ Turn range-checking on }
const
  Answer = 3; { For the purposes of this demonstration,
                let's make the required answer 3 always. }
var
  Guess : integer;
begin
  Writeln('In this program, you will guess an integer ',
          'from 1 to 10.');
  repeat
    Writeln('You have not guessed the number yet.');
    Write('Type an integer from 1 to 10 as your guess: ');
    Readln(Guess);
  until Guess = Answer;
end.
```

The For Statement

So far, we've discussed two kinds of iterative statements: one that says, "Do this *while* the following condition is TRUE" and one that says, "Repeat this *until* the following condition is TRUE." We now come to the third type of looping statement in Pascal—one that corresponds to the idea "Do this *n* times."

The **for** statement in Pascal accomplishes this task by allowing you to make a variable of a scalar type into a *counter* (or, in more formal terms, a *control variable*), which keeps track of how many times you have passed through a loop. You tell Pascal the value at which the counter will be started, and what the final value will be—the computer does the rest.

The syntax of the **for** statement is depicted in Figure 11-5.

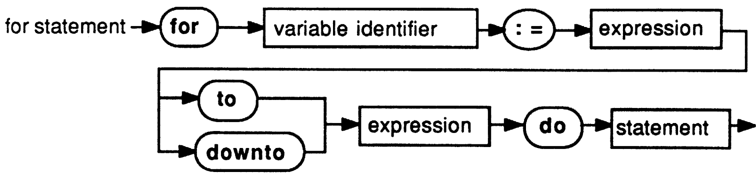


Figure 11-5 Syntax Diagram of For Statement

When the **for** statement is first encountered in a program, the variable is assigned the value of the first expression. If the ordinal value of the variable is less than or equal to that of the second expression, then the statement is executed. The variable is then assigned the value of its successor (if **to** is used) or its predecessor (if **downto** is used). This process continues until the ordinal value of the variable is greater than that of the second expression. (If the ordinal value of the first expression is greater than that of the second expression, then the statement is never executed.)

Here's a simple example. If the variable *Index* is declared to be of type integer, then the loop

```
  .
  .
  .
for Index := 1 to 10 do
  Writeln('n = ',Index,' n*n = ', Index * Index);
  .
  .
  .
```

will write out the integers from 1 to 10 in increasing order, each followed by its square. If we were to write the same loop like this:

```
  .
  .
for Index := 10 downto 1 do
  Writeln('n = ',Index,' n*n = ', Index * Index);
  .
  .
```

then the same values would be printed, but in decreasing order, from 10 down to 1.

Note that there is no rule that says you must use your control variable as anything other than a counter. In the statement

```
for Index := 1 to 10 do
  Writeln('Hello!');
```

we merely tell the computer to write the string 'Hello!' 10 times, and never write out, or use, the value of the variable *Index* within the loop.

There is one important point to remember about **for** loops that catches many novice programmers unaware. The value of the control variable is undefined after the **for** statement has finished executing; that is, it can have any value at all. This feature, which is part of the standard Pascal language, allows Turbo to write faster machine code under some circumstances. Because this is not what you expect, it is important to remember to never assume the control variable of a **for** statement has any specific value after the loop has completed.

If you are familiar with other programming languages, such as BASIC, you may be wondering how you can specify a "step" value for the **for** statement. (By step value we mean a value other than 1 by which the control variable is to be incremented or decremented after each iteration.) The answer, in Pascal, is that you can't. You must create loops that increment or decrement a counter by more than 1 using the **while** or **repeat...until** constructs. However, in return for this inconvenience, Pascal gives you the ability to use a variable of *any scalar type* as the loop index, including char, byte, boolean, or any enumerated or subrange type you define. Given appropriate definitions and declarations, the following loops are all valid:

```
for Ch := 'A' to 'Z' do
  Writeln;

for Flag := TRUE downto FALSE do
  Writeln;

for Day := Mon to Fri do
  {Assuming you have previously defined the type }
  Writeln;
```

Now you can see why there is no step capability. It wouldn't make sense to write a statement like

```
for Month := January to December step 2 do
  Writeln;
```

since 2 is not of the same data type as January or December, and can't be "added" to them. Similarly, the statement

```
for Month := January to December step February do
  Writeln;
```

implies that you can somehow add February to another month, which of course is not so. For this reason, there is no step capability in the **for** statement.

As with the **if** and **while** statements, a compound statement can be used with the **for** statement to make more than one thing happen each time through the loop.

```
for Index := 20 to 30 do
begin
  { First statement here }
  ...
  { Last statement here }
end;
```

Simulating a Step Size Larger than 1

The **for** statement is actually a notational convenience; it is possible to simulate any **for** statement with a **while** statement. The statement

```
for A := B to C do
  Writeln('Hello!');
```

can be simulated exactly by the statements

```
A := B;
while A <= C do
begin
  Writeln('Hello!');
  A := Succ(A);
end;
```

except, of course, that *A* is defined upon exiting from the **while**. From this, we see how to create a loop with a step size greater than 1. If the variables *A*, *B*, and *C* are of any scalar type, we can write

```
A := B;
while A <= C do
begin
  Writeln('Hello!');
  A := Succ(Succ(Succ(A)));
end;
```

and the ordinal value of *A* will be increased by 3 on each iteration. In such cases, however, it is important to watch for overflow—there may not be a legitimate value of that scalar type with an ordinal value 3 greater than that of *A*. Again, we advise you to turn range-checking on in all your programs to prevent this kind of mishap from going undetected. If the control variable of your loop is of type integer, or a subrange thereof, you will be able to use the addition operation (instead

of the *Succ* function) to create large step sizes even more easily. Assuming, now, that *A*, *B*, and *C* are of type integer, we could write

```
A := B;
while A <= C do
begin
  Writeln('Hello!');
  A := A + 50;
end;
```

giving a step size of 50.

Endless Loops

One of the problems that you will doubtless encounter in your programming career is that of an *endless loop*—a loop that for some reason or another never finishes executing. Occasionally, such a loop may occur due to bad data, or you may even have a reason for programming one intentionally. Most often, however, an endless loop will appear in your program when you least expect it. In the following program, which was intended to list all the numbers from 1 to 20 evenly divisible by 3, an error in the placement of a statement caused an endless loop. Can you find the bug before going on to the next paragraph?

```
program Whoops;
var
  i : Integer;
begin
  i := 1;
  repeat
    if (i mod 3) = 0 then
    { If this is true, then i is evenly divisible by 3 }
    begin
      Writeln(i); {Write i out}
      i := Succ(i); {Increment i by 1}
    end
  until i = 20;
end.
```

The problem with this program is that *i* is only incremented if it is divisible by 3, and not otherwise. Thus, the program will consider the number 1 again and again, and never move on to test the number 2 for divisibility. To stop the program, you'll have to shut down the system and reboot, losing everything you've done. Thus, it is good practice to save your program text before running it, to prevent your losing valuable time and text in case of a reboot.

To solve this program's problem, place the statement *i := Succ(i)* after the **if** statement and before the **until**, so that *i* is incremented on every pass through the loop.

THE CASE STATEMENT

In many of your programs, you will want the computer to perform one of a list of actions, depending upon the current value of a variable. For example, you might want to display a menu, accept the user's choice, and then perform the action that was chosen. We already have the capability to do this with the **if** statement. The following program fragment accepts a one-character command from the user, and does something appropriate:

```
...
Write('Enter your choice: U)p, D)own, L)eft, R)ight:');
Readln(Ch) { Ch is of type Char }
if (Ch = 'U') or (Ch = 'u') then
  Y := Y + 1
else if (Ch = 'D') or (Ch = 'd') then
  Y := Y - 1
else if (Ch = 'L') or (Ch = 'l') then
  X := X - 1
else if (Ch = 'R') or (Ch = 'r') then
  X := X + 1;
else Writeln(^G'Invalid command!');
  ...
```

Note that we have nested a number of **if** statements, one inside the other; each time a condition is not satisfied, we try the next until we find the appropriate action or give up.

However, the **if..then..else** chain can get a little tiresome (and difficult to follow) if there are a lot of different conditions to check. So, Pascal helps to reduce the complexity by including an additional flow-of-control structure: the **case** statement.

In the **case** statement, you provide Pascal with an expression (which must be of a scalar type), followed by a list of values and actions (constants). When the program runs, the action associated with the current value of the variable is performed.

Using the **case** statement, we could express the intent of the previous example much more clearly, like so:

```
...
Write('Enter your choice: U)p, D)own, L)eft, R)ight:');
Readln(Ch);  { Ch is of type Char }
case Ch of
  'U','u' : Y := Y + 1; {Do this if Ch is 'U' or 'u'}
  'D','d' : Y := Y - 1; {Do this if Ch is 'D' or 'd'}
  'L','l' : X := X - 1; {Do this if Ch is 'L' or 'l'}
  'R','r' : X := X + 1; {Do this if Ch is 'R' or 'r'}
else
  Writeln(^G'Invalid command!');
  {Do this if Ch is none of the above}
end;
  ...
```

The syntax of the **case** statement in Turbo Pascal is shown in Figure 11-6.

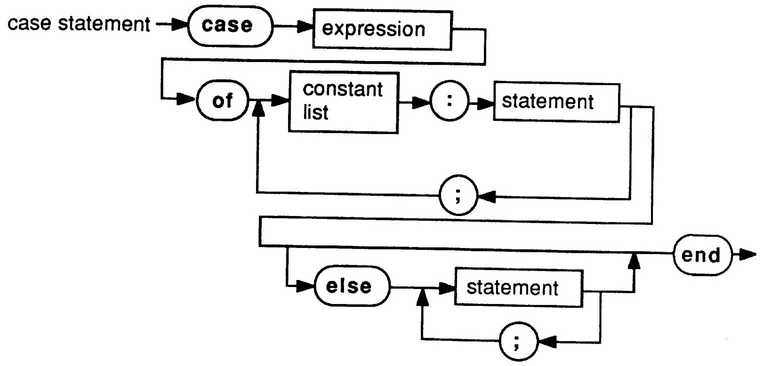


Figure 11-6 Syntax Diagram of Case Statement

Figure 11-7 shows the syntax diagram of a constant list.

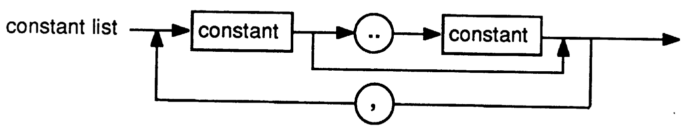


Figure 11-7 Syntax Diagram of Constant List

When the **case** statement is executed, the computer inspects the lists of constant values preceding each action. If the value of the expression is present in one of these lists, the specified action is taken. Each action must be expressed as a single statement, with compound statements allowed if more than one statement needs to be executed. The end of the **case** statement is marked by the reserved word **end**.

What if the expression has a value other than those listed in the **case** statement? Under the original definition of the Pascal language, the result is "undefined," that is, the outcome is uncertain. Turbo Pascal, however, extends this definition in a logical way.

In Turbo Pascal, you can specify the "default" action (that is, an action to be taken if no match is found) in the **else** clause of the **case** statement. In an earlier example, we used the **else** clause to make a noise and write an error message. Unlike the **if** statement, there must be a semicolon before the **else**. There may also be as many statements between the **else** and the **case**'s final **end** statement as desired.

If a **case** statement does not have an **else** clause (and none of the specified **case** conditions apply), Turbo Pascal skips *all* of the actions listed in the case statement and continues with the statement following the **case**.

The Constant List

In the **case** statement, the list of constants for each possible action can be specified in a number of ways: as individual values, as a subrange, or as a combination of the two. A subrange is specified exactly as it is in the definition of a subrange type: the lower bound (that is, the one with the smallest ordinal value in the subrange), followed by two periods and the upper bound (the one with the largest ordinal value in the subrange).

Using subranges can make the specification of a case much easier. The following **case** statements are completely equivalent:

```
case Age of {Age is of type Byte or Integer}
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10, 11, 12, 13, 14, 15, 16, 17:
  Writeln('You are not old enough to vote yet.');
else
  Writeln('You are old enough to vote. Who will you',
          ' vote for?');
end; {case}
```

or

```
case Age of { Age is type Byte or Integer }
  0..20: Writeln('You are not old enough to vote yet.')
else
  Writeln('You are old enough to vote. ',
          'Who will you vote for?');
end; {case}
```

To mix subranges with individual items in the constant list, you need only separate them with commas. The following is perfectly legal:

```
case Age of
  0, 1, 2..5, 6, 7, 8, 9,
  10..17:
  Writeln('You are not old enough to vote yet.');
else
  Write('You are old enough to vote. ',
        'Who will you vote for?');
end; {case}
```

REVIEW

In this chapter, we introduced you to Turbo Pascal's control structures—the statements that enable a Turbo Pascal program to make decisions and perform repetitive actions. We also discussed boolean

expressions and operations in some detail, and showed how the compound statement can be used to make one statement out of many.

Now that you know how to control the order in which your program's statements execute, you are ready to learn how to divide your code into logical groups to make your program more readable and save unnecessary typing. The mechanisms to do this, procedures and functions, are the subject of our next chapter.

12 Procedures and Functions

In previous chapters, you've seen procedures and functions (known collectively as subprograms) used in some of the sample programs. You've been given a general idea of how they're used, and you're probably anxious to learn more. (Appendix C provides a complete list of standard Pascal procedures and functions.)

This chapter will teach you more. First, we'll describe the general concept of a subprogram, and then show you how (and where) to place them in your main program. We'll then cover the idea of *scope*, which determines what the identifiers used in a subprogram mean, and from where a subprogram may be called. We'll end with a discussion on *parameters*—the data objects you tell a subprogram to work on—and give you a taste of what you can do with them.

SUBPROGRAMS

You've learned that a Pascal program must have a program heading, a declaration part, and a statement part. You've also learned that each statement in the statement part is normally executed in order—from beginning to end—unless a conditional (**if**) or iterative statement (**for**, **repeat**...**until**, or **while**) alters the flow of control.

The iterative statements that we've learned about are perfect for repeating a statement, or group of statements. However, what do you do if you want to repeat a group of statements in several distinct places in your program, rather than only within the same loop?

Here's a simple example of a situation where a subprogram might come in handy. The following sequence of statements asks the user for a number, then checks the number to see if it is in range. If the number is in range, it is assigned to the variable *NewNumber*.

```
Writeln('Please enter an integer from ',
        'Minimum, ' to ', Maximum,': ');
Readln(Temporary);
{Temporary is of type integer, or a subrange thereof}
while (Temporary < Minimum) or (Temporary > Maximum) do
begin
  Writeln('The integer you have entered is out of range.');
  Writeln('Please try again: ');
  Readln(Temporary);
end;
  NewNumber := Temporary;
  {NewNumber is of type integer, or a subrange thereof}
```

Now, suppose there are several places in your program where you want to perform this same function—widely separated by tens or even hundreds of lines. You could copy all of the statements given here, exactly as shown, into each place; however, this would increase the size of both your source and executable files. Also, if you wanted to make a change, you might need to find and alter every occurrence. Fortunately, the Pascal language provides a solution. Instead of copying the group of statements, you can give it a name, and cause the entire group to be executed by merely mentioning that name. Such a group of statements is called a *subprogram.*

The simplest kind of subprogram in Pascal is called a *procedure.* To create a procedure, you declare its name as an identifier (as you declare all identifiers), and present the compiler with the code that is to go by that name. The syntax of a *procedure declaration* is shown in Figure 12-1.

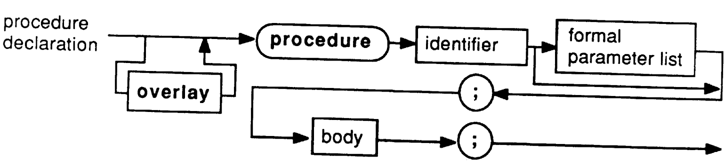


Figure 12-1 Syntax Diagram of Procedure Declaration

Each procedure declaration contains the reserved word **procedure,** followed by the identifier that is to be the name of that procedure. Then, there appears an optional list of parameters (described more fully later), a semicolon, a procedure body, and a closing semicolon.

A *procedure body* is usually just a block—the exact same sequence of declaration part and statement part that comes after the heading of a program. This sequence reinforces the idea that procedures—and subprograms in general—are really small programs within a larger program. The syntax of the body is shown in Figure 12-2.

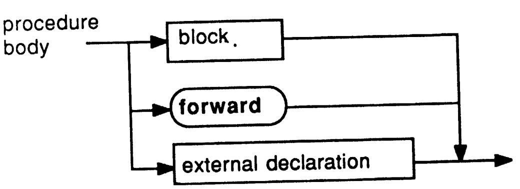


Figure 12-2 Syntax Diagram of Procedure Body

Where do you put subprograms in your program? In standard Pascal, you place them in the declaration part, after all other declarations and before the opening **begin** of your program. (Turbo Pascal, as you may remember, is more flexible, and lets you place them anywhere in the declaration part.) A subprogram can declare its own constants and variables, just like the main program. Can you declare a subprogram within a subprogram? Yes, you can, just like you would in a regular program. And you can declare subprograms within *that* subprogram, and so on, ad infinitum (well, not quite; all compilers have their limits).

Now, let's rewrite our earlier sequence of statements as a procedure and show how we might build a small program around them.

```
program Sample;
const
  Ten = 10;
var
  NewNumber, Index : integer;

procedure GetNumber;
{ Get a number from the user and store }
{ it in the global variable NewNumber }
const
  Minimum = 0;
  Maximum = 25;
type
  Response = Minimum..Maximum;
  { A type for a legal response }
var
  Temporary : Response;
  { A temporary place for the user's integer }
begin
  { Statement Part of procedure GetNumber }
  Writeln('Please enter an integer from ',
          'Minimum, ' to ', Maximum, ': ');
  Readln(Temporary);
  while (Temporary < Minimum) or (Temporary > Maximum) do
  begin
    Writeln('The integer you have entered is not '
            'between 0 and 25, ');
    Writeln('inclusive. Please try again.');
    Readln(Temporary);
  end;
```

```
    NewNumber := Temporary;
  end; { procedure GetNumber }

begin { Statement Part of program Sample }
  GetNumber;
  .
  .
  for Index := 1 to 10 do
  begin
    GetNumber;
    .
    .
  end;
  .
  .
  GetNumber;
  case NewNumber of
    0: Writeln('You have selected option 0');
    .
    .
    25: Writeln('This is option 25: ');
  end;
  .
  .
end. { of program Sample }
```

In this example, **const**, **type**, and **var** declarations have been deliberately included in the subprogram to demonstrate that the subprogram can indeed have these parts.

Besides avoiding code duplication, subprograms help to make a program easier to understand. Procedure names that specify a particular action to be taken can clarify a program, whereby the main actions of a program are shown in the statement part and the details of those actions are "hidden" in the functions and procedures. Changes and improvements that are made to subprograms can instantly have an effect on many other parts of the code—without any additional rewriting. Also, when many people are at work on a program, each programmer can write one or more subprograms, then allow them to be combined into the finished product. This modular approach allows each author to perfect his or her part of the program without having to directly modify the work of others.

SCOPE

Since it is possible to define and declare types, constants, and variables in the declaration part of a procedure, you may well ask: "What happens if an identifier is declared in the main program, and then another with the same name is declared in the subprogram? Or, what if two subprograms declare identifiers with the same names?"

Well, your first guess might be that the compiler would indicate an error, since each identifier can only refer to one thing if the program is to be unambiguous. However, if this were true, then the modular approach to programming would be difficult. Each programmer would have to know all of the names that every other programmer used, and combining their work might cause unexpected conflicts.

To avoid this problem, the concept of *scope* was developed. The Pascal scope rules specify, in a rigorous way, what object any identifier refers to at a given time. We'll cover the first two scope rules right now, and the third when we get to the section on recursion later in this chapter.

Scope Rule #1: Each identifier has meaning only within the block in which it is declared, and only after the point in that block at which it is declared.

Thus, in the previous example, the variable *Temporary* only has meaning within the procedure *GetNumber*, and even then only after its declaration. By the same rule, the declarations

```
const
  Minimum = 0;
  Maximum = 25;
```

and

```
type
  Response = Minimum..Maximum;
  {A type for a legal response}
```

could not have occurred in the reverse order. If they had, the constants *Minimum* and *Maximum* would not have been defined when the type *Response* was defined using them, and the compiler would have indicated an error.

Another consequence of the first scope rule is that the procedure *GetNumber* can refer to, and in fact assign a value to, variables that were declared before *GetNumber* in the declaration part of the program Sample. Thus, the statement

```
NewNumber := Temporary;
```

does what one would expect, and assigns the value of *Temporary* to the variable *NewNumber*, which was declared in the main program. An identifier like *NewNumber*, which is declared at a higher level and is accessible within *GetNumber*, is said to be global to the procedure *GetNumber*. The converse, however, is not true. In the main program, we could not write the statement

```
Temporary := NewNumber;
```

because the variable *Temporary* is not defined outside of *GetNumber*. *Temporary* is said to be local to *GetNumber*, and is not visible outside of that block.

Scope Rule #2: If a global identifier is redefined within a block, then the innermost (most deeply nested) definition takes precedence from the point of declaration until the end of the block.

What does this mean? The following "skeleton" program will help to illustrate.

```
program A;
const { These are the "global" identifiers of program A.}
  J = 1; {They are visible everywhere within the program,}
  K = 2; {unless hidden by local symbols with the same name.}
var
  R, S : integer;

procedure B;
const
  L = K; { L is defined to be 2 (NOT 3!) }
  K = 3; { K is now defined locally to be 3, "hiding"
           the K defined in A }
begin { Statement Part of procedure B }
  { Within the Statement Part of procedure B, the
    following identifiers are visible:
          identifier ! defined in
          ---------- + ----------
          B, J, R, S ! A
                K, L ! B

    The local constant L derives its value from the
    GLOBAL constant K, not the local one, since the
    global identifier was not yet "hidden" when L was
    defined. Note that there is no identifier A
    visible. Turbo Pascal, unlike most other
    compilers, ignores the program heading entirely,
    including the program name. }
end; { procedure B }

var
  T, U: integer; { These identifiers are not visible
                   within procedure B! }

procedure C;
var
  V : integer;

procedure D; { local to procedure C }
var
  R, T : integer; { These declarations "hide" the
                    R and T declared in A }
begin { Statement Part of procedure D }
  { Within the Statement Part of procedure D, the
    following identifiers are visible:
          identifier ! defined in
    ---------------- + ----------
    B, C, J, K, S, U ! A
                D, V ! C
                R, T ! D
```

```
    Note that the constant K is seen as having the value 2
    here, since the local K (with a value of 3) defined
    in B is visible only there. }
end;

var
  B : integer; { This declaration "hides" procedure B
                 within the Statement Part of procedure
                 C. However, procedure B is still callable
                 from procedure D, and this integer is
                 not visible to D. }
begin { Statement Part of procedure C }
  { Within the Statement Part of procedure C, the
    following identifiers are visible:
              identifier | defined in
    -------------------- + ----------
     C, J, K, R, S, T, U | A
                 B, D, V | C
  }
end; { procedure C }

begin { program A }
  { Within the Statement Part of program A, the
    following identifiers are visible:
                   identifier | defined in
        --------------------- + ----------
        B, C, J, K, R, S, T, U | A
  }
end. { program A }
```

In each procedure within this program (and also within the statement part of the main program), we've listed which identifiers are visible and which objects they refer to.

To fully understand how this information is derived, it helps to "pretend" that you are the compiler, scanning the program from top to bottom. Each time you encounter a definition that overrides a previous one, you make a note of the new definition, and use it until the current block is exited. When you exit a block, all the identifiers declared locally within that block become undefined, and all the identifiers that were temporarily "hidden" by definitions in that block become visible once more.

Exercises Consider the following program. At each of the points marked {1}, {2}, and {3}, list the identifiers that are accessible and the procedure (or program) where they were defined. For the constant identifiers, also list their values; for the variables and types, list their base types (that is, the predefined type from which their types are derived).

```
program ScopeTest;
type
  A = integer;
  B = real;
```

```
const
  J = 5;
  K = 14;
var
  Q : A;
  R : B;
procedure First;
type
  B = A;
var
  R : B;
const
  K = J;
  J = K;
begin { procedure First }
  {1}
end; { procedure First }

procedure Second;
var
  First : A;
const
  L = K;
  { 2 }
  K = 3;
type
  A = B;

procedure Third;
var
  First : A;
begin { procedure Third }
end; { procedure Third }

begin { procedure Second }
end; { procedure Second }

var
  S : A;

begin { program ScopeTest }
  {3}
end. { program ScopeTest }
```

Now consider the following program. Will it compile without errors? What will it write when run? Why? Test your answer using Turbo Pascal.

```
program Scope2;
var
  A : integer;

procedure SetA;
var
  A : integer;
```

```
begin { Statement Part of procedure SetA }
  A := 4
end; { procedure SetA }

begin { Statement Part of program Scope2 }
  A := 3;
  SetA;
  Writeln(A)
end. { program Scope2 }
```

THE LIFETIME OF LOCAL VARIABLES

Unlike the global variables that are declared in the declaration part of the main program, the local variables of a subprogram are created (assigned places in memory) each time the subprogram is entered, and destroyed each time control returns to the calling program or subprogram. Thus, the *lifetime* of a local variable—that is, the time during which it will be able to retain its assigned value—is said to be limited to the current invocation of the subprogram. What does this mean? Well, suppose you have a procedure *A* that declares the local variable *X* in its declaration part. Now suppose you call *A* and cause it to set *X* to the value 3. When you call *A* again, can you assume that *X* will still be 3? No! And, if you do, you may be inviting disaster, since *X* may have *any value* the next time you make a call. Thus, the rule of thumb is that all local variables should be considered undefined upon entrance to a subprogram.

PARAMETERS

In our first example of a procedure (procedure *GetNumber*), each call to the procedure causes a number input by the user to be assigned to the global variable *NewNumber*. Since the variable *NewNumber* is overwritten each time *GetNumber* is called, it will most likely be necessary to save the value of *NewNumber* in some other variable immediately after the call, like this:

```
   .
   .
   .
GetNumber;
MenuChoice := NewNumber;
   .
   .
   .
```

While this is a perfectly valid way of getting a number, it has two potential problems. First, you must always remember to perform the

assignment; if you don't, the value the user entered will be lost. Second, the variable *NewNumber* must be declared globally and must not be used for anything else! If the procedure *GetNumber* is called within a procedure that defines its own variable *NewNumber*, then it will be impossible to retrieve the result.

Earlier we mentioned that one of the purposes of using subprograms is to avoid this sort of unexpected naming conflict when many programmers are working on parts of the same program. Local variables solve this problem for data objects that are used entirely within the subprogram. Parameters handle the problem of naming data that is passed to and from the subprogram.

You have already seen and used parameters when you have called the built-in procedures *Readln* and *Writeln*. In the statement

```
Readln(A);
```

the variable *A* is passed as a parameter to the procedure *Readln*, which in turn gets data from the keyboard or a file and places that data in the variable. In the statement

```
Writeln('Hello, world, my name is Joe.');
```

the constant string 'Hello, world, my name is Joe' is passed as a parameter to the *Writeln* procedure. To show how to declare procedures with parameters, let's rewrite our *GetNumber* procedure to return a number in a parameter, rather than in a global variable:

```
procedure GetNumber(var NewNumber : integer);
  { Get a number from the user and return
    it in the variable parameter NewNumber. }
const
  Minimum = 0;
  Maximum = 25;
type
  Response = Minimum..Maximum;
  { A type for a legal response }
var
  Temporary : Response;
  { A temporary place for the user's integer }
begin  { Statement Part of procedure GetNumber }
  Writeln('Please enter an integer from ',
          'Minimum, ' to ', Maximum, ': ');
  Readln(Temporary);
  while (Temporary < Minimum) or (Temporary > Maximum) do
  begin
    Writeln('The integer you have entered is not ',
            'between 0 and 25,');
    Writeln('inclusive. Please try again.');
    Readln(Temporary);
  end;
    NewNumber := Temporary;
end; { procedure GetNumber }
```

Note that the only change made to this procedure (other than to the introductory comment) was in the very first line, where we changed the procedure heading to read

```
procedure GetNumber(var NewNumber : integer);
```

What does this accomplish? First, it tells the compiler that when the procedure is called, it can expect to find the name of a data object of type integer in parentheses following the procedure name. Second, it says that inside the procedure that data object will be referred to by the name *NewNumber*, regardless of what it might have been named in the part of the program calling the procedure. Finally, the **var** preceding the name *NewNumber* indicates that a variable (rather than an expression or a constant) must be passed for that parameter, and that the procedure will have the ability to alter the value of that variable. It is important to note that *NewNumber* is not a variable itself; rather, it is an identifier that represents another variable whose identity is decided by the procedure call (and can change from call to call). Such an object is called a *formal*, or *dummy, parameter*. If we were to call our new procedure *GetNumber* as follows:

```
GetNumber(A);
```

then the procedure will act exactly as if the variable *A* were present everywhere *NewNumber* was mentioned. In this situation, *A* is said to be the actual parameter of the procedure. If we were then to make the call

```
GetNumber(B);
```

the actual parameter would be *B*, and any assignments made to the formal parameter *NewNumber* would actually be made to the variable *B*.

A procedure can also have formal parameters that are not declared using the reserved word **var**. If this is the case, two things change. First, *any* expression can be passed as the actual parameter, rather than only as a variable. Second, any changes (such as assignments) that are made to the formal parameter in the subprogram do *not* affect the value of the actual parameter. (After all, it does not make sense to assign a value to an expression.) Instead, they are made to a *copy* of the value of the expression, which is created when the procedure is entered.

Such parameters are called *value parameters*, and the "one-way" information flow they provide is useful for two reasons. First of all, they allow the values of constants and expressions, as well as variables, to serve as input to a subprogram. Without them, statements such as

```
Writeln('Hello, world, my name is Joe.');
```

would be impossible to write without assigning the string to a variable. The second use of value parameters is as a precautionary measure. Because a subprogram works on a copy of a value parameter, rather

than on the value parameter itself, it will never make unwanted modifications to that parameter. Be forewarned, however, that one of the most common mistakes made by beginning programmers is to do the reverse, and forget to declare a parameter that should be changed as a **var** parameter. If this happens, the compiler will not complain, but it will appear as if the subprogram is not working properly. Figure 12-3 shows the syntax of a *formal parameter list*—the part of a subprogram declaration where you specify the names and types of the subprogram's formal parameters.

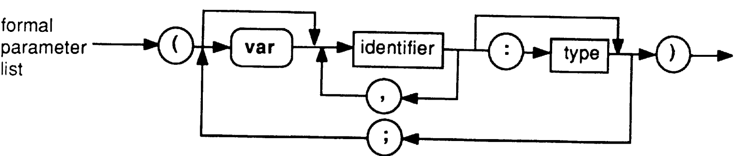


Figure 12-3 Syntax Diagram of Formal Parameter List

A subprogram can have almost any number of parameters, subject only to the limitations of the particular compiler, and those parameters can be of almost any type.

One more brief point about parameters. Because formal parameters are considered to be declared as identifiers within the block of their procedures, they can "hide" identifiers declared at a higher level of the program just as other local identifiers can. For the same reason, a locally declared constant, type, or variable may not have the same identifier as a formal parameter.

FUNCTIONS

As we've seen, a procedure can change the values of variable (**var**) parameters passed to it. The calling program (or subprogram) can then continue to use the modified parameters for whatever purpose it wants. However, it often happens that the programmer wants only a single value back from the subprogram, and it is inconvenient (and messy) to set aside a variable just to hold that value.

Suppose, for example, that you were to write a procedure to find the square root of an integer (approximated to the next lowest integer):

```
procedure ISqrt(Val : integer; var Root : integer);
var
  OddSeq,Square : integer;
begin { procedure ISqrt }
  OddSeq := -1;
  Square := 0;
```

```
  repeat
    OddSeq := OddSeq + 2;
    Square := Square + OddSeq
  until Val < Square;
  Root := Succ(OddSeq div 2);
  if Val <= Square-Root then
    Root := Pred(Root)
end; { procedure ISqrt }
```

This procedure would take *Val*, find its square root, and set *Root* to that value. The calling program might use it as follows:

```
repeat
  Write('Enter value: '); Readln(Square);
  ISqrt(Square, Root);
  Writeln('The square root is ', Root)
until Square = 0;
```

In the previous example, the variable *Root* is only used to carry the value of the square root from the call to *ISqrt* to the *Writeln* statement. If there are many variables like this in a program, it can become clumsy to keep track of them all—and most of them will not be in use most of the time (a poor use of memory space). We mentioned functions earlier (in Chapter 10, "Defined Scalar Types"), and as you may recall, a function is used in an expression in place of the value that it produces, with the value becoming part of the expression when it is evaluated. When you declare a function, you create a heading similar to that of a procedure, except that you specify the type of the returned value. (This is necessary so that Turbo Pascal knows how to "fit" the returned value into the expression properly.)

The syntax of a function declaration is shown in Figure 12-4.

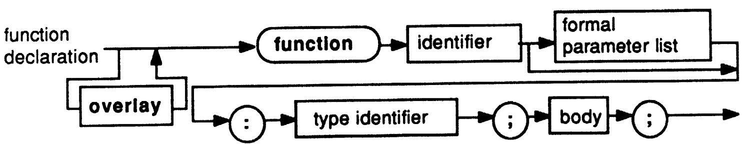


Figure 12-4 Syntax Diagram of Function Declaration

As we just mentioned, the only difference between a function declaration and a procedure declaration is that in the function declaration a type is required.

Here is an example of a function that is equivalent to the square root procedure shown earlier. Note that there is no parameter to hold the root—the name of the function itself is used to represent the value.

```
function ISqrt(Val : integer) : integer;
var
  OddSeq,Square,Root : integer;
```

```
begin { Statement Part of function ISqrt }
  .
  . { same code as before }
  .
  ISqrt := Root { The value is returned by assigning to the
                  function name as if it were a variable }
end; { function ISqrt }
```

The function can now be used in a program like this:

```
repeat
  Write('Enter value: '); Readln(Square);
  Writeln('The square root is ',ISqrt(Square))
until Square = 0;
```

Now, the main program (or any subprogram that wants to use this subprogram) does not need to declare a variable to hold the square root. This means a simpler program, with less room for errors and naming conflicts.

A function can be used anywhere that a constant or an expression of the same data type could be used. Suppose you wanted to find the fourth root of a given integer value. You could rewrite the program this way:

```
repeat
  Write('Enter value: '); Readln(FourthPower);
  Writeln('The fourth root is ',ISqrt(ISqrt(FourthPower)))
until FourthPower = 0;
```

When writing a function, take care to ensure that you have set the function identifier to some value before exiting. As shown previously, you do this by assigning some value to the function name, as if it were a variable. Actually, you can assign a value to the function identifier many times; however, if you do, the last value you assign before the function terminates will be the value returned.

While you may always make assignments to the function identifier, you may *not* retrieve the value you assigned to that identifier by including it in an expression. Why? Because the compiler will interpret that use of the function name as another call to the function. This is why we made the identifier *Root* a local variable, rather than eliminating it entirely from our function in the previous example. Had we tried to use the identifier *ISqrt* to hold the intermediate values of the root as it was being computed, we would have had no way of getting them back.

Since an attempt to get the value of the function identifier within a function is construed by the compiler as another call to the same function, it follows that functions (and procedures as well) can call themselves. Subprograms that do this are called *recursive subprograms*, and are discussed in detail in the next section.

RECURSIVE SUBPROGRAMS

Sometimes, the easiest way to describe a task is to describe it in terms of a subprogram that calls itself to get a job done. This is called a recursive subprogram. For instance, there is a function in mathematics called the factorial function, which gives the product of all the positive integers up to, and including, that integer. For instance, 5 factorial (written as 5!) is $5 \times 4 \times 3 \times 2 \times 1$, and 8 factorial is $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$.

How can we write a function to compute the factorial of a number? Well, we could write a loop to accomplish that job, counting from a given number down to 1 and multiplying at each step. However, a very simple and clever way to do the calculation is to notice that for any integer n greater than 1, $n!$ (n factorial) is equal to $(n - 1)! \times n$. Thus, we can describe the factorial function in terms of itself, as follows:

```
function Factorial(N : Byte) : Real;
begin { function Factorial }
  if N <= 1 then
    Factorial := 1 { If N <= 1, no recursion occurs. }
  else
    Factorial := N * Factorial(N - 1)
    { Here is the statement that causes the recursion! }
end;
```

When this function is called, it looks to see if the value of the parameter N is less than or equal to 1. (Because 256! is much too large to hold even in a variable of type real, we've restricted the parameter to the type byte. This way, we also need not worry about negative numbers.) If N is 1 or 0, there is no multiplying to be done, since $1! = 1$ (and, incidentally, 0! is considered to be 1 as well). Otherwise, we set the function identifier *Factorial* equal to N times the *Factorial* of $N - 1$.

Note well the different uses and meanings of the function identifier on the left- and right-hand sides of the assignment operator. On the left, the identifier is used without a list of parameters, and represents a place where the result of the function is to be stored. On the right, the same identifier is used with a list of parameters to represent the result of a call to that function. The calls will "nest" more and more deeply until *Factorial* is called with the value 1; at that point, the series of calls will unwind, doing the necessary multiplication at each step.

Since a procedure can call other procedures (including itself), procedures can be recursive subprograms as well. It is also possible to build more complex recursive structures where one procedure or function calls a second, which calls the first, which calls the second, and so on.

Recursive subprograms often make a calculation very simple to program. But beware, like loops (another repetitive process), it is possible for such structures to get out of control and run forever. It is possible

for the calls to nest indefinitely, or until the machine no longer has the memory available to keep track of them. (As we mentioned before, space for local variables is allocated each time a subprogram is called. Thus, each call can potentially use up a large block of memory, and the available space can be exhausted very quickly.)

So always use recursive subprograms with care, and make sure that there is a statement that will not cause more recursion (like the previous statement *Factorial:* = 1) in every recursive subprogram.

FORWARD DECLARATIONS

Occasionally, the rule that all identifiers must be declared before they are used can keep you from making your program do what you want. For instance, as we mentioned earlier, you may want to write a subprogram that calls another, which in turn calls the first, and so on. The problem with writing such a structure, however, is which subprogram do you declare first? No matter which one you choose, the other subprogram will not have been defined yet, and therefore you will not be able to call it. Here is a simple program illustrating the problem.

```
program Example;
var
  Alpha : integer;

procedure Test1(var A : integer);
begin { procedure Test1 }
  A := A-1;
  if A > 0 then
    Test2(A);
  Writeln(A);
end; { procedure Test1 }

procedure Test2(var A : integer);
begin { procedure Test2 }
  A := A div 2;
  if A > 0 then
    Test1(A);
  Writeln(A);
end; { procedure Test2 }

begin { Statement Part of program Example }
  Alpha := 15;
  Test1(Alpha)
end. { program Example }
```

As you can see, *Test1* calls *Test2* and *Test2* calls *Test1*. As it stands, this program won't compile; you'll get an "Unknown identifier" error when it finds the reference to *Test2* within *Test1*. If you swapped *Test1* and *Test2*, you'd get a similar error within *Test2*.

The solution to this problem is to tell the compiler, *before* it gets to the procedure *Test1*, that the procedure *Test2* will be declared later. This is done with a **forward** declaration, as shown in the following example:

```
program Example;
var
  Alpha : integer;
procedure Test2(var A : integer); forward;

procedure Test1(var A : integer);
begin { procedure Test1 }
  .
  .
  .
end; { procedure Test1 }

procedure Test2 {(var A : integer)};
  { We've commented out the parameter list; }
    it was supplied earlier. }
begin { procedure Test2 }
  .
  .
  .
end; { procedure Test2 }

begin { Statement Part of program Example }
  Alpha := 15;
  Test1(Alpha)
end. { program Example }
```

The **forward** declaration of *Test2* contains only the procedure or function heading and the reserved word **forward** (the information necessary for the compiler to check any calls to it for a correct name and parameter list). The actual body of *Test2* occurs after Test1. Now *Test1* can call *Test2* (because of the **forward** declaration) and *Test2* can call *Test1* (since the latter precedes the former).

Note that when *Test2* is finally declared, its parameter list is omitted (though we recommend showing it in a comment as a reminder of what the parameters and their types are). The parameter list *may not be repeated*; Turbo Pascal already "knows" what the parameter list is, and does not need the redundant (and possibly inconsistent) information.

SCOPE AND RECURSION

The subject of recursion brings us to the final scope rule of Pascal. As you will remember, Scope Rule #2 stated that if an identifier is declared in an outer block and then again in an inner block, the inner declaration will take precedence until the end of the inner block. This is true regardless of which subprograms call which others; it is the position of the variables in the text of the program at compile time that decides which symbol refers to which object.

The problem of scope becomes more complex, however, when recursion is involved. For instance, suppose we wrote the following set of nested procedures, and then called the function *A* with the parameter 5. What would be written? And what would the function *A* return?

```
function A(G : integer): integer;
var
  X : integer;

procedure B;
begin
  Writeln(X);
end;

begin
  if G > 1 then X := A(G 1)
  else
    X := 0;
  B;
  A := G;
end;
```

To help in understanding the answer to this question, we need to explain the third (and last) scope rule of Pascal.

Scope Rule #3: When procedures are invoked recursively, a reference to a global variable always refers to the instance of the variable in the most recent invocation of the procedure in which that variable is defined.

This rule applies to the previous procedure *B*, when it references the variable *X* defined in function *A*. So when *B* is invoked and executes the statement *Writeln(X)*, the *X* that is written is the one that exists in the storage area allocated by the most recent call of the procedure *A*. The correct answer to our question then is that *A*(5) would return the value 5, and would write the numbers 0, 1, 2, 3, and 4.

THE EXIT PROCEDURE

Sometimes when you are writing a Turbo Pascal procedure or function, you will reach a point in the middle of the body of the subprogram at which the subprogram may be ready to return *immediately*—without executing the rest of the subprogram. In standard Pascal, there is no way to accomplish such an exit, and you must structure the subprogram (using **if** statements, perhaps) so that all statements from then on to the end of the subprogram can be skipped.

In Turbo Pascal, there is a special feature that allows a quick return from any point in a procedure or function. It takes the form of a procedure call to a predefined procedure called (appropriately enough) *Exit.*

Here's a sample program demonstrating the usefulness of *Exit.* Suppose you want to write a routine that accepts numbers from the

keyboard, one at a time, then returns the total. A function to do this might look like so:

```
function RunningTotal : Real;
var
  Subtotal, NewNumber : Real;
begin
  Subtotal := 0.0;
  repeat
    Write ('Enter a number to be added to the total: ');
    Readln (NewNumber);
    if NewNumber <> 1.0 then
      { Only add if number is not 1 }
      Subtotal := Subtotal + NewNumber;
  until NewNumber = 1.0;
  { Exit the loop if number is 1 }
  RunningTotal := Subtotal;
end;
```

Note that in this example the function returns when the user enters a special value: −1. Such a value, used as a signal to the program to do something, is known as a *sentinel*. Here, the sentinel value −1 indicates that there are no more numbers to be entered.

Now, while the previous subprogram will certainly do the job we need it to do, it is not as efficient as it could be. In particular, we test *twice* to see if the variable *NewNumber* has the value −1: once to determine whether to add it to the running total, and once to see if we should exit the subprogram.

The second test wouldn't be necessary if we could put a statement in the **repeat...until** loop that says, "If *NewNumber* is −1, return the total immediately without doing anything else!"

Here is how to use the *Exit* procedure to accomplish this:

```
function RunningTotal : Real;
var
  Subtotal, NewNumber : Real;
begin
  Subtotal := 0.0;
  repeat
    Write ('Enter a number to be added to the total: ');
    Readln (NewNumber);
    if NewNumber = 1.0 then
    begin
      RunningTotal := Subtotal;
      { Set the function result and exit }
      Exit; { right here! }
    end
  until False; { Since we exit the loop from the middle,
                 we'll never want the "until" to be
                 satisfied }
end;
```

This technique becomes even more useful when the point from which you want to exit is deeply nested in structured statements, such as **if**s,

whiles, and **fors**. It is also likely to make your program more readable, since the reader will be able to recognize immediately where the exit occurs and what value is returned. If the *Exit* procedure is called from the body of your main program, it causes the program to stop running immediately. *Exit* should be used in this manner with care—the program must remember to perform cleanup operations, such as closing files, before halting.

REVIEW

In this chapter, we introduced the two types of subprograms in Pascal: procedures and functions. We described the format of procedure and function declarations, and explained Pascal's rules about the scope and lifetime of identifiers declared within subprograms. We touched on the topic of recursion, and explained how to declare forward subprograms and how to determine the scope of identifiers during recursion. Finally, we presented the predefined procedure *Exit*, which causes an immediate exit from a subprogram or main program. In the next chapter, we'll cover the concept of arrays and how to use them.

13 Arrays

Previously, we taught you about the five predefined data types—integer, byte, real, boolean, and char—as well as declared scalar types. A variable of one of these types can hold only one value at a time. For example, if you define:

```
var
  Index : integer;
```

Index will have only one specific value at any moment. However, there are situations where you'd like to have a single identifier represent a list of values, such as a list of numbers or characters. That's where *arrays* come in. Suppose, for example, you want to write a program to balance your checkbook. Well, one thing your program will need is a list of all your checks and their respective amounts. To reserve space for this information, you could declare a variable for each check:

```
var
  Check1 : Real; { Amount of check 1 }
  Check2 : Real; { Amount of check 2 }
  Check3 : Real; { Amount of check 3 }
  Check4 : Real; { Amount of check 4 }
  . . .
```

Of course, this could quickly become tedious if you write a lot of checks. Also, it would be difficult to write a loop to go through all the checks and do something with each—say, add them to a running total. You could *not* write the following:

```
for Check := Check1 to Check25 do
  Total := Total + Check;
```

because, first of all, all of the variables you have declared are of the type real, and only scalars can be used as the indices in a Pascal **for** statement. Second, and more importantly, we really want to step through the locations where the amounts of the checks were stored, which is not what this statement would do. Even if this loop would compile, it would only increment the variable *Check* from the amount of the first check (*Check1*) to the amount of the last (*Check25*), rather than looking at the amount of each check and adding it to the total. The

result would bear no resemblance to the correct answer. How, then, do we accomplish the simple task we've described? The answer, as you've probably guessed, is to store the list of amounts in an array.

An array is a list of variables of identical type, each one of which can be referred to by telling the compiler the name of the list and its position in the list. Suppose, for instance, that you were to declare

```
var
  Check : array[1..10] of Real;
```

This tells the compiler that the identifier *Check* refers to a list of 10 variables of the type real, each with a number (called its *index*) from 1 to 10.

Each item of an array is referred to by the name of the array (*Check*), followed by its index enclosed in square brackets ([]). Thus, the array *Check* contains the variables *Check[1]*, *Check[2]*, *Check[3]*, *Check[4]*, *Check[5]*, *Check[6]*, *Check[7]*, *Check[8]*, *Check[9]*, and *Check[10]*. You can use any of these variables anywhere you would use any real variable. Furthermore—and this is what gives arrays their true power—the index value does not have to be a constant. In fact, it can be any expression that yields an integer in the range 1..10. For example, if the variable *Index* is of the type integer, the statement

```
for Index := 1 to 10 do
  Check[Index] := 0.0;
```

would set each variable to 0. Now can you see how to solve the problem of adding up the amounts of all the checks? Since we can now refer to each check by its index, we can write

```
Total := 0.0; { Remember to set the total to 0 before we
                start. }
for Index := 1 to 10 do
  Total := Total + Check[Index];
```

The diagram in Figure 13-1 shows the syntax of an array type.

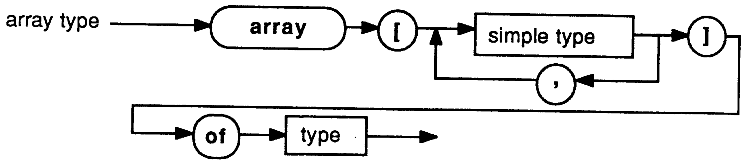


Figure 13-1 Syntax Diagram of Array Type

It's important to note that specifying an array type requires you to tell the compiler about two other types: the array's *index type* and the type of each of the items of the array, the *base type*.

The index type, which appears between the square brackets, must be a simple type; that is, it cannot consist of more than one data object (like an array). Moreover, the index type must be a scalar type.

Most often, you will want to use a subrange type as the index type of an array, as we did. However, sometimes you may decide that another type is appropriate. For example, if you are writing a program to encrypt a secret message using a cipher, you might want an array that holds the code for each possible object of the type char. In which case you could declare the array

```
var
  Cipher : array[char] of char;
```

filling each location of the array with the replacement character for the corresponding index character. Then, to encode a character, you could write:

```
MsgChar := Cipher[MsgChar];
```

and each character in the secret message would be replaced by the code for that character. There are other limits to the index type of an array. One of them is that the type cannot have so many possible values that the array is too big for Turbo Pascal to handle. The declaration

```
var
  BigArray : array[integer] of char;
```

would cause a memory overflow error. Why? Because it would try to reserve enough room for 65536 characters—one for each possible value of the type integer. In Turbo Pascal, no single data object or variable can be over 65535 bytes in length. And, especially in a CP/M machine, such an object can easily overflow available memory. Thus, it is best to declare arrays with an index type of as small a subrange as possible. The definition

```
var
  NotSoBigArray : array[byte] of char;
```

compiles just fine, since byte, as you will recall, is the subrange of the type integer that goes from 0 to 255. Often, you will want to use a declared scalar type as an index type for an array type like this:

```
type
  Days = (Sun,Mon,Tues,Wed,Thur,Fri,Sat);

var
  Regular  : array[Mon..Fri] of integer;
  Overtime : array[Days] of integer;
  Present  : array[Days] of boolean;
```

The array *Regular* has a subrange of the type *Days* as its index type, while the arrays *Overtime* and *Present* have the entire type *Days* as their index type. The array *Regular* consists of 5 integers. If a variable of the type integer takes up 2 bytes, and a variable of the type boolean takes

up 1 byte, can you tell the total amount of memory (in bytes) taken up by each array?

The base type of any array can be almost any data type at all—as long as the total size of the array does not exceed 65535 bytes. In fact, you can declare arrays that contain other structured types, including other arrays. Arrays of arrays (usually called *multidimensional arrays*) are useful for describing groups of objects that need to be located using two indices, such as a cell in a spreadsheet (which is located by its row and column) or a point on a piece of graph paper (located by its *X* and *Y* coordinates).

Suppose, for a moment, you are writing a computer program to play a game of checkers. One thing the computer would need to do is keep track of what piece was on each square of the board. One way to represent the board might be like this:

```
type
  Square = (Empty, Red, Black, RedKing, BlackKing);
  {Type for a square}

var
  CheckerBoard : array[1..8] {outer array}
                 of array[1..8] {inner array} of Square;
```

How would you access each square of the board under these conditions? In Pascal, the way to do this is to give the subscript of the *outer array*, followed by the subscript of the *inner array*. Thus, if we let each of the inner arrays be a column (horizontal file) of the board, we could locate a square in the third column from one player's left, and in the fourth row (vertical rank) from the same side of the board (see Figure 13-2).

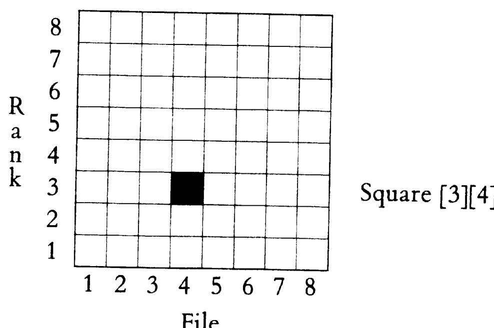


Figure 13-2 Accessing Game Board

Because it can get inconvenient to write all of those square brackets, and since we don't always need to point out explicitly that a two-dimensional array is an array of arrays, Pascal provides a more convenient notation to express both array types and array indices. Instead of

writing *Square[3][4]*, Pascal lets us write *Square[3,4]* as well. Similarly, when specifying the array type for the checkerboard, we can write

```
type
  Square = (Empty, Red, Black, RedKing, BlackKing);
  {Type for a square}
var
  CheckerBoard : array[1..8, 1..8] of Square;
```

which is equivalent to what's shown in Figure 13-2.

In some programs, you may wish to use arrays with a dimension of 3 or more (for instance, if you are playing three-dimensional checkers, or if you need to specify the location of a point in space). There is no theoretical limit to the number of dimensions that an array can have, although some compilers impose practical limits. Turbo Pascal will allow an array to have as many dimensions as you specify, until the capacity of memory is exceeded.

ARRAY ASSIGNMENTS

In standard Pascal, you can only perform assignment operations on the individual objects within an array, not on a whole array. Thus, if we declare

```
var
  A, B: array[1..10,1..10] of integer;
```

and we want to transfer all the elements of *A* into *B*, we would have to write the double loop:

```
{Assume I and J are of type integer}
for I := 1 to 10 do
  for J := 1 to 10 do
    B[I,J] := A[I,J];
```

This loop will transfer each element of *A* into *B*, one at a time. Can you see how it works? While this complex statement gets the job done, we really just want to tell the compiler to take everything from *A* and move it to *B*. Turbo Pascal, therefore, allows you to assign whole arrays to other arrays of the same type. With Turbo, you can write:

```
B := A;
```

and the whole job is done. Turbo Pascal also lets you work with arrays that are nested within other arrays. Thus, if you wanted to transfer just one row of array *A* to a row of array *B*, you could write

```
A[8] := B[8];
```

What does this mean? Remember that the declaration

```
var
  A, B: array[1..10,1..10] of integer;
```

is convenient shorthand for

```
var
  A, B: array[1..10] of array[1..10] of integer;
```

so that *B[8]* means, in this case, "the eighth 10-element array of *B*." These special features of Turbo Pascal make working with arrays a lot easier.

RANGE-CHECKING AND ARRAYS

One of the most common errors that novice (and even advanced) programmers make when using arrays is to try to access an array element that does not exist. For instance, if you declare

```
var
  Check : array[1..10] of real;
```

and then write the statement

```
for i := 1 to 11 do
  Writeln(Check[i]);
```

What will happen? As you would expect, the first 10 numbers in the array *Check* would be written to the terminal. But what would be written when the computer tried to find *Check[11]* the last time through the loop? The answer depends on whether or not range-checking is enabled. If range-checking is off, Turbo Pascal will look in memory where it thinks *Check[11]* ought to be—immediately after *Check[10]*. It will then assume whatever is there to be a real number, and write that number to the console. Of course, this place in memory could be part of another variable, or even part of a program, so the number would be meaningless.

To keep Turbo Pascal from ignoring this error, and possibly compounding it by using such a value in further calculations, turn range-checking on. Use the compiler directive {$R+} to turn range-checking on. (Chapter 10 covers this in more detail.)

We can't emphasize enough how important it is to make sure that *subscripts* (as well as variables of subrange and enumerated types) do not go out of range without your knowledge. Again, we advise you to turn range-checking on in every program you write.

INITIALIZING AN ARRAY

Before you use an array you must initialize it, that is, set all of its elements equal to some set of values. (Remember that before a variable is assigned a value it can have any value at all.) If all the values are the same, the process is simple. For example, suppose you want to set all elements in the array *A* (defined earlier) to 0, so that you can later set those desired to other values. One way to do this would be:

```
for X := 1 to 10 do
  for Y := 1 to 10 do
    A[X,Y] := 0;
```

This takes a while to do and uses up a bit of space for code and variables. Turbo Pascal, however, provides you with a faster way: the predefined procedure *FillChar.* A call to *FillChar* looks like this:

```
FillChar(Dest,Length,Data);
```

where *Dest* is the variable (of any type) to be filled, Length is the number of bytes to initialize, and Data is the value to which to set each byte (to be expressed either as a character or as a byte value). We know what you want to fill—*B*—and we know what you want to fill it with—the integer 0, which is represented as 2 bytes of zeroes. Thus, we use *B* for *Dest,* and 0 for the data byte.

Now we just need the length in bytes. You can obtain this value by using the built-in function *SizeOf.* The function *SizeOf* takes as its parameter either a variable or a data type, and returns the size of that variable (or of a variable of that type) in bytes. So to initialize *B*, we could write:

```
FillChar(B,SizeOf(B),0);
```

This statement will set all bits and bytes in *B* to 0. The combination of *FillChar* and *SizeOf* is the fastest way to initialize an array variable in Turbo Pascal. Be warned, however, that this will not always work when you wish to initialize all the elements of an array to a value other than 0. If you tried to set all of the elements of *B* to 1 by using the statement

```
FillChar(B,SizeOf(B),1);
```

you would discover that each element of *B* had the value 257, not 1. This is because *B* is an array of type integer, and an integer with both of its bytes set to 1 equals 257. Thus, it is important to be careful when using *FillChar* to initialize arrays (and other structures) whose components are larger than 1 byte.

REPRESENTING AN ARRAY IN MEMORY

The elements of an array are stored in a specific order. If the array is one-dimensional—that is, if it has only one index—then the elements are stored in ascending order. For example, the array *Check* (defined as **array**[1..10] **of** real) stores its elements in the order *Check[1]*, *Check[2]*, and so on, as you might expect. But what about multidimensional arrays? The array *CheckerBoard* is defined as

```
var
  CheckerBoard : array [1..8, 1..8] of Square;
```

So, the question is, are the elements in *CheckerBoard* stored as *CheckerBoard[1,1]*, *CheckerBoard[2,1]*, *CheckerBoard[3,1]*, and so forth, or are they stored as *CheckerBoard[1,1]*, *CheckerBoard[1,2]*, *CheckerBoard[1,3]*, and so on? The definition of Pascal itself hints at the answer to this question. Remember that the previous definition is just shorthand for

```
var
  CheckerBoard : array[1..8] of array[1..8] of Square;
```

Thus, the first index of *CheckerBoard[3,4]* (which can also be written as *CheckerBoard[3][4]*) does not select a square. Rather, it selects a column of the board, which is an **array**[1..8] **of** Square. The second index selects a square within that array, and those elements are stored sequentially, just as in *Check*. *CheckerBoard[1,1]* says to pick the first element of the first array; *CheckerBoard[1,2]*, the second element of the first array, and so on. Thus, the squares are stored in the order

```
CheckerBoard[1, 1]
CheckerBoard[1, 2]
CheckerBoard[1, 3]
  ...
CheckerBoard[2, 1]
CheckerBoard[2, 2]
  ...
CheckerBoard[8, 7]
CheckerBoard[8, 8]
```

Remember that the index furthest to the right—the last index—changes the fastest, regardless of the number of dimensions in the array. Thus, the array

```
var
  BigOne : array[0..3,0..4,0..5,0..2] of byte;
```

is stored as

```
BigOne[0,0,0,0]
BigOne[0,0,0,1]
BigOne[0,0,0,2]
BigOne[0,0,1,0]
BigOne[0,0,1,1]
  ...
BigOne[3,4,5,1]
BigOne[3,4,5,2]
```

If you're using a CP/M-80 system, you need to be aware of one very important difference in how arrays are stored. Suppose you've declared the following array:

```
var
  List : array[0..7] of byte;
```

If *List[0]* is at address X, then *List[1]* is at $X - 1$, *List[2]* is at $X - 2$, and so on. In other words, array elements are stored in descending memory locations for CP/M-80 systems, while 16-bit systems (IBM PC, MS-DOS, CP/M-86) store arrays in ascending memory locations (*List[1]* is at $X + 1$, and so forth).

PACKED ARRAYS

The discussion on storage space brings up another issue. Standard Pascal defines two kinds of arrays: regular arrays and *packed arrays*. It also provides two procedures, *Pack* and *Unpack*, to convert between the two types. This feature was useful when Pascal was implemented on certain large computers, where arrays could be stored in two different ways. Packed arrays were slower to access, but took up less storage space. For unpacked arrays, the reverse was true.

In Turbo Pascal, arrays are always stored as described in the preceding section. For compatibility reasons, Turbo Pascal allows the reserved word **packed** to be used as shown in the syntax diagram, but ignores it. The procedures *Pack* and *Unpack* aren't defined.

REVIEW

In this chapter, we have learned how to declare and access arrays—a structured type that consists of a list of variables of any type. We also showed you how to work with multidimensional arrays and how to initialize arrays. We also described how arrays are stored in memory.

That's all on arrays for now, though you'll see them used heavily throughout the rest of this book.

Chapter 14 will discuss in detail Turbo Pascal's string types and how to use them.

14 Strings

When Niklaus Wirth designed the Pascal language, he did so on a very large computer (known as a "mainframe") that accepted programs and data in the form of punched cards and magnetic tapes. On these systems, programs were submitted as "batch jobs"; the program and input data went in, the system worked on them, and the output data came out. Users could not communicate with the system while the program was running.

In time, this situation began to change. "Timesharing" systems, in which a computer served many users at once, became common. CRT terminals became available to more users. Minicomputers became popular, and users began to insist on *interactive programs*—programs that communicated with the user while being run. Many of these programs performed such functions as word processing, unlike the older systems that worked mostly with numbers.

Unfortunately, because it originated in the world of number-crunching machines, the standard Pascal language was not given facilities for working with strings (groups of ASCII characters). In standard Pascal, strings could only be stored in fixed-length arrays of characters, and no special operations for reading, writing, or processing them were provided. This made the design of programs to handle text (as opposed to numbers) difficult. Turbo Pascal, like many modern extensions of the Pascal language, has solved this problem by adding special features for handling strings, which make writing text-oriented programs simple and quick.

STRING TYPES

A string is a list of ASCII characters (that is, data objects of the type char); Turbo Pascal keeps track of both their contents and length. Using Turbo Pascal's built-in operators and subprograms, you can copy parts of the string, add to it, take away from it, combine it with other strings, write it out, or read it in.

In Turbo Pascal, you create a string variable by declaring it to be of a string type. To specify a string type, you tell the compiler the maximum length that a string of that type can be (so it can reserve enough memory). The syntax of a string type is shown in Figure 14-1.

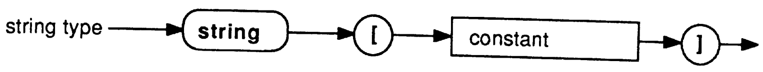


Figure 14-1 Syntax Diagram of String Type

The constant in the specification of the string type must be in the range 1..255, giving you a choice of 255 possible string types. Here is a sample declaration of a string type:

```
const
  MaxstringSize = 255;
type
  Bigstring    : string[MaxstringSize];
  Littlestring : string[15];
```

and here are some examples of string variable declarations:

```
var
  MyName       : string[80];
  Token        : Littlestring;
  MyBigString  : Bigstring;
```

The constant used in the specification of a string type sets the maximum number of characters each string can hold. The variable *MyName* could hold up to 80 characters; *Token* could only hold up to 15 characters. Thus the statement

```
Token := 'this is too long a string for Token';
```

would only store the first 15 characters ('this is too lon') into *Token*. The last variable, *MyBigString*, has the maximum length possible for a string in Turbo Pascal: 255 characters.

STRING OPERATORS, FUNCTIONS, AND PROCEDURES

Turbo Pascal's string extensions include more than just the **string** data types. Turbo also provides you with a rich assortment of operators, procedures, and functions to work on strings (see Table 14-1).

Table 14-1 String Procedures and Functions

Procedures/Functions	Definition
Concat(St1, St2 {, St3,...,Stn})	Returns string composed of St1 through Stn concatenated together; the plus sign (+) can also be used.
Copy(St, Position, Len)	Returns string composed of St[Position].. St[Position+Len−1].
Delete(St, Position, Num)	Deletes Num characters from St starting at St[Position].
Insert(Source, Destination, Position)	Inserts Source into Destination starting at Destination[Position].
Length(St)	Returns current length of St.
Pos(Pattern,Target)	Returns position (index) of Pattern within Target.
Str(Value, St)	Converts Value (integer or real) into a string and stores it in St.
Val(St, Value, Index)	Converts St into Value (integer or real) and sets Index to the position of any error occurring (0 means no error or St = '').

String Assignments

The most common operator you are likely to use with strings in Turbo Pascal is the *assignment operator*. Assignments to string variables work as they do with any other type of variable, with one difference: If the destination string is too small to hold all the characters in the value assigned to it, those characters are dropped or truncated. For instance, if the string variable *Fruit* is of the type **string**[5], then the assignment statement

```
Fruit := 'Watermelon';
```

would cause *Fruit* to get the value 'Water.'

The Length Function

One of the most frequently used functions that Turbo provides for working with strings is *Length*, which returns the current length of a string. (This is not to be confused with the maximum possible length of that string.) If *St* is a string variable, then the expression

```
Length(St)
```

gives the number of characters in *St*. The following is a program that demonstrates how this function works.

```
program LengthTest;
type
```

```
  SmallStr = string[15];
var
  Test : SmallStr;
procedure ShowLength(St : SmallStr);
  { Write out a string and its length }
begin
  WriteLn('The length of "',St,'" is ',Length(St))
end;  { procedure ShowLength }

begin
  Test := 'hello, there';
  ShowLength(Test);
  Test := 'hi';
  ShowLength(Test);
  Test := '';  {This is a null string--it has a length of 0.}
  ShowLength(Test)
end.  { program LengthTest }
```

In this program, we define a procedure, *ShowLength*, which accepts a string as its parameter. It then writes out the string, followed by its length. When this program is run, it will produce the output:

```
The length of "hello, there" is 12
The length of "hi" is 2
The length of "" is 0
```

The Concat Function and the + Operator

Another useful string function is called *Concat*. As the name might suggest, *Concat* is used to concatenate, or combine, strings to make one large string. If *St1* is a string variable with the value 'Joe,' then the expression

```
Concat('Hello, world, my name is ', St1, '.')
```

yields the string 'Hello, world, my name is Joe.'. The general syntax of the *Concat* function is

```
Concat(St1, St2 {, St3,...,Stn})
```

where *St1*, *St2*, and so on, are all variables of any string type.

Like some other built-in procedures, such as *Writeln*, *Concat* can take any number of parameters (indicated by the curly brackets). Unlike *Writeln*, however, it must have at least two parameters. Beyond this, its only restriction is that the total length of all the concatenated strings must be less than 255. If not, the program will halt at that statement and write out an error message. (This sort of error, which occurs when the program is run rather than compiled, is called a *runtime error.*)

Besides explicitly calling the function *Concat*, Turbo Pascal also lets you concatenate strings using the plus sign (+) as an operator. The expression

```
'Hello, world, my name is ' + St1 + '.'
```

is exactly equivalent to the expression shown earlier using *Concat*.

The Copy Function

The next string function that you may find useful is called *Copy*, which allows you to make a copy of any part of a string (that is, a substring). It takes as parameters the string, the number of the first character at which to begin copying, and the number of characters to copy. The expression

```
Copy('This string has no characters', 20, 10);
```

returns the value 'characters' (which, incidentally, does have characters after all). The syntax of the *Copy* function is

```
Copy(Source, Position, Len)
```

where Source is a string and Position and Len are integers.

There are a few restrictions on the parameters passed to *Copy*. First of all, the second parameter, which indicates the position at which to start copying from the string, must be in the range 1..255 or a runtime error will occur. Secondly, if you try to use *Copy* to copy beyond the end of a string, only the characters within the string will be returned. If the starting position is already beyond the end of the string, then *Copy* will return a *null string*, a string with a length of 0 containing no characters at all.

The Pos Function

Another powerful Turbo Pascal string function is the *Pos* function. This function looks for the first occurrence of one string inside another string, and tells you where it begins. *Pos* takes the string to search *for* as its first parameter, and the string to search *in* as its second parameter. If the string is found, *Pos* returns an integer giving the location in the string where the matching string begins; otherwise, *Pos* returns a value of 0.

Suppose, for example, you wanted to see if the word "to" occurred in the string variable *St1*, which happened to contain the string, 'To be or not to be.' You could use the *Pos* function to look for the word and also to see where it occurred, by writing

```
St1 := 'To be or not to be';
Location := Pos('to', St1);
```

The variable type *Location* (assumed to be of type integer) would get the value 13. (Note: Case is significant, so the "To" at the beginning of *St1* will not be matched.) You could then use this information to make changes to the string. If the target string is not found, *Pos* returns the value 0. The syntax of the *Pos* function is

```
Pos(Pattern, Target)
```

where Pattern and Target are strings.

The Delete and Insert Procedures

There are two things you may want to do with strings: delete characters from them, or insert characters into them. The *Delete* procedure lets you remove a section of a string; like *Copy*, it requires the string, the starting position, and the number of characters to delete.

For example, suppose you find the location of the word "to" in a string, and want to remove it and the blank following it from the string. You could do this by writing the statement

```
Delete (St1, Location, 3);
```

which removes 3 characters from *St1*, starting at the character indicated by *Location*. Alternatively, you could perform the entire operation in one step by writing

```
Delete (St1, Pos('to', St1), 3);
```

omitting the use of the variable *Location* entirely. In either case, the variable *St1* would have the value 'To be or not be.' The syntax of the *Delete* procedure is

```
Delete(St, Position, Num)
```

where *St* is a string and Position and Num are integers.

If Position is beyond the last character of the string, no characters are removed; if it is not a value from 1 to 255, a runtime error occurs. If an attempt is made to delete past the end of the string, only characters in the string are removed.

Combined with *Pos* and *Copy*, *Delete* can be used to separate a string into words. The following is a procedure that will get the first word from a line of text, where a "word" (for our purposes) is like any substring starting with a non-space character and followed by a space.

```
procedure GetWord(var Line, Word : BigStr);
  { Get the next word from the string Line }
const
  Space = ' ';
var
  Len : integer;
begin
  while Pos(Space,Line) = 1 do     { remove leading blanks }
    Delete(Line,1,1);
  Len := Pos(Space,Line) - 1;              { look for blank }
  if Len = 0 then begin                    { no blanks left }
    Word := Line;
                 { get word might be null string if none left }
    Line := ''
                              { now make line the null string }
  end
  else begin                  { get word and delete from line }
    Word := Copy(Line,1,Len);            { get all but blank }
    Delete(Line,1,Len + 1)         { delete word plus blank }
  end
end;  { procedure GetWord }
```

The next procedure, *Insert*, is the reverse of the *Copy/Delete* operation: It takes one string and stuffs it inside another. The first parameter of *Insert* is the string to insert, the second is the string into which it's to be inserted, and the last is the location where the insertion will occur. For example, if you want to put the "to" back in the string *St1*, you could write

```
Insert(St1, Location, 'to ');
```

The syntax of the *Insert* procedure is

```
Insert(Source, Destination, Position)
```

where Source and Destination are strings and Position is an integer.

If Position is outside the range 1..255, a runtime error occurs. If the result is longer than the maximum length of Destination, the extra characters at the end will be truncated. Finally, if an attempt is made to insert a string at a position after the end of Destination (that is, Position is greater than Length(Destination)), then Source is concatenated onto the end of Destination.

Insert and *Delete* together can be used to substitute one substring for another. Suppose you were writing a program that takes a form letter and inserts the appropriate names, dates, and so on. Within the form letter, these strings might be represented by tokens (groups of symbols that show you where to put the information). For example, the salutation might look like this:

```
Dear <title> <last name>:
```

where we've represented the information to be added using tokens of the form ⟨...⟩. The following procedure, then, might be used for replacement purposes:

```
procedure Replace(var Line : BigStr; Token,Sub : TokStr);
  { Look for Token in Line and replace with Sub }
var
  Index,Len : integer;
begin
  repeat
    Index := Pos(Token,Line);
    if Index > 0 then begin
      Delete(Line,Index,Length(Token));
      Insert(Sub,Line,Index)
    end
  until Index = 0
end;  { procedure Replace }
```

The statements

```
Line := 'And so, <title> <last>, the entire <last> family';
Replace(Line,'<title>','Dr.');
Replace(Line,'<last>','Lewis');
Writeln(Line);
```

would produce

```
And so, Dr. Lewis, the entire Lewis family
```

This should give you a clue as to how "personalized" junk mail is generated.

Miscellaneous Character Functions

There are two other functions that are often useful when working with strings: the *Chr* and *UpCase* functions.

The *Chr* function takes a number of the type byte or integer and returns an ASCII character (a data object of type char) that corresponds to that ASCII code. For instance, if you wanted to see all of the characters in the ASCII code on your screen (or at least those that will print out), you could use the following program:

```
program PrintASCII;
{Print the characters for all the ASCII codes, 0 to 255.}
var i :integer;
begin
  for i := 0 to 255 do
    Writeln(i, ' --> ', Chr(i));
end.
```

The *Chr* function has only one restriction: Its parameter must be between 0 and 255. If it is not, no runtime error will occur, but the parameter *modulo* 255 (that is, the lower byte of the parameter) will be used. The *UpCase* function takes a parameter of type char and, if it is a lower-case letter, converts it to its upper-case equivalent. Here are some examples of calls to the *Upcase* function and what they return:

```
UpCase('a')  --> 'A'
UpCase('A')  --> 'A'
UpCase('?')  --> '?'
UpCase('x')  --> 'X'
```

Note that if the parameter is not in the range *a..z*, it is returned unchanged.

REPRESENTING STRINGS IN MEMORY: STRINGS AS ARRAYS

The data type **string**[n] can be thought of as an **array**[0..n] **of** char. You can reference individual characters in a string variable using the same notation you would for an array; for instance, the first character of *Token* is *Token[1]*, the second is *Token[2]*, and so on. The first location of the array—the one with an index of 0—contains the current length of the string. If you execute the statement

```
Token := 'this string';
```

then *Token[0]* contains a character with the ASCII value 11, since there are 11 characters in 'this string.' However, you could not do something like this:

```
program MisMatch;
var
  Token : string[15];
  Len   : integer;
begin
  Token := 'this string';
  Len := Token[0];      { The compiler will indicate
                          an error here.}
  Writeln('The length of Token is ',Len)
end.
```

because *Token[0]* is of type char, and you can't assign a character to an integer. You could, however, substitute the statement

```
Len := Ord(Token[0]);
```

which would return the ordinal (numeric) value of *Token[0]*, which happens to be 11. Of course, the *Length* function is designed to do this for you, so there is usually no need to use this technique.

Using the array notation, you can access any individual character of a string. As mentioned previously, each element of a string is a variable of type char, and you can treat it as such. For example, you might want a procedure to convert all letters in a string to upper case ('A'..'Z'):

```
type
  Maxstring = string[255];
procedure UpperCase(var Str : Maxstring);
var
  Index : integer;
begin
  for Index := 1 to Length(Str[0]) do
    Str[Index] := UpCase(Str[Index])
end; { procedure UpperCase}
```

One caveat is in order: Unless you know what you're doing, you should avoid messing with any elements beyond the current length of the string. Turbo Pascal won't give you any sort of error, but you need to be aware that you've just changed a portion of the string that won't print unless you change the length as well. For example, the sequence

```
Token := 'Hello';
Token[6] := '!';
WriteLn(Token);
```

will produce the output "Hello" rather than "Hello!", because *Token[0]* will still contain a length of 5. If you add the statement *Token[0] := Chr(6)*; before the *Writeln* command, you'll write out the complete string. Usually, though, this sort of "string surgery" is not necessary. Similarly, be careful when using the standard procedure *FillChar* on strings. In the following example

```
FillChar(Token, Sizeof(Token), 32);
```

the string variable *Token* is filled with spaces and the length byte is set to 32. The best way to initialize a string is with a null assignment, for example:

```
Token := '';
```

This statement simply sets the length byte of ASCII to 0. If you need to fill a string with spaces, your *FillChar* statement should look like this:

```
FillChar(Token, Sizeof(Token), 32);
Token := '';
```

STRING COMPARISONS

Just like numbers, strings can be compared to each other. In Turbo Pascal, you use the same operators to compare strings that you use to work on any other type of data object: =, <, >, <=, >=, and <>.

How does the test for equality work? First, the lengths of the two strings are compared. If they're different, then the strings are not equal. If they're the same, then the characters in the two strings are compared, starting with the first one and continuing until two characters are different or all characters have been compared. If no characters are different, the strings are equal.

Similar comparisons may occur if you want to see if a string is "greater than" or "less than" another. For example, let's suppose you're sorting a list of names into alphabetical order. At some point, you'll compare two strings to find out which one comes before (is less than) the other. The statement

```
if Str1 > Str2 then ...
```

will take some action if, and only if, *Str2* comes before *Str1*. The test is done exactly as if you were sorting the strings by hand: The first characters are compared, then the second, and so forth. The test is completed when either

- One of the strings has a character that is different from the corresponding character in the other string; in this case, the string that has the character with the smaller ASCII code comes first.
- The end of one of the strings is reached before a difference is found; in this case, the shorter string comes first.
- The strings are exactly the same; in this case, either string can be said to come first.

When doing string comparisons, it is important to remember that all characters (spaces, symbols, and control characters as well) are included in the comparison according to their positions in the ASCII code.

Because all upper-case letters come before lower-case letters in ASCII, capitalization is important. (You may find the procedure *UpperCase* helpful to make sure that the capital letters in a name like "MacGregor" don't cause unexpected results.)

NUMERIC CONVERSIONS

Turbo Pascal provides two procedures for converting numbers to strings and vice versa: *Str* and *Val*. These two procedures work much like read and write; however, instead of reading from the keyboard or writing to the screen, these procedures move information from one data object to another.

Str converts a number into a string, formatting it as if it were going to write it on the screen. The number can be either integer or real, and you can specify the number of columns that the string will take up. (You can do this with *Write* as well, which you'll learn shortly.)

The syntax of a call to *Str* is

```
Str(Value, St);
```

St must be a variable of any string type, and *Value* must be a *Write* parameter of type integer or real.

A *Write* parameter is an expression that is followed by special formatting commands to control how the value is converted into a string. The complete set of rules for specifying *Write* parameters is included in Chapter 14 of the *Turbo Pascal Reference Manual*; we'll briefly discuss how they work for integers and real numbers here.

If *Value* is just an expression that yields a value of the type integer, then *St* gets the decimal representation of that integer. If *St* is too small to hold the string containing this representation, the right-most part of the number is truncated.

However, if *Value* is an integer expression followed by a colon and another integer expression, then the second integer expression gives the number of columns in which to represent the number. The number is "right-adjusted," which means the right-most digit is always in the last column of the string. If the number of columns is too small to hold the converted integer, Turbo Pascal places the whole integer in the string anyway (room permitting).

In case this is confusing, here are some examples to help clear things up. Suppose the integer *I* is equal to 14916, and we have a string *S* with a maximum length of 12. Here are some sample calls to *Str* and their results:

```
     Call              Value of S       Length of S

Str( I, S);         "14916"                 5
Str( I:12,S);       "       14916"         12
Str( I: 7,S);       "  14916"               7
Str( I: 5,S);       "14916"                 5
Str( I: 3,S);       "14916"                 5
Str( I:15,S);       "          14"         12
```

If the last result seems surprising, remember that Turbo Pascal performs the conversion first, and then attempts to assign the result to the string. If there is not enough room, the right-most characters of the result are truncated, leaving the string shown.

For expressions of type real, one can specify either 1 or 2 integers, separated by colons, to tell Turbo Pascal how to convert the number to a string. If no formatting information is present, Turbo Pascal writes the number out in exponential format, using 18 columns. If a single integer is present, then exponential format is still used, but the integer tells Turbo Pascal how many columns to use for the output string. (If the number given is less than 8, Turbo Pascal will use 7 or 8 columns anyway.)

Most of us, however, like to read real numbers in an ordinary decimal format, with digits separated by a decimal point. To get this format, place 2 integers, separated by colons, after the real parameter of *Str.* The first integer will be used as the length of the output string, and the second will specify how many digits below the decimal point are shown.

Here are some calls to *Str* with a real parameter, along with the resulting strings. Let *X* (a real) = 4.281953E3, and let *S* be of the type **string**[18].

```
     Call                Value of S              Length of S

Str( X, S);          " 4.28195300000E+03"          18
Str( X:14, S);       "4.28195300E+03"              14
Str( X:12:3,S);      "    4281.953"                12
Str( X:12:0,S);      "        4282"                12
Str( X:10:7,S);      "4281.9530000"                12
Str( X:12:5,S);      "  4281.95300"                12
Str( X: 5:4,S);      "4281.9530"                    9
```

In general, note that the string length is set equal to the field width (the first number specified). If the width is too small (such as X:10:7, X:5:4, or I:3), it is increased to fit the number. If the field is wider than is necessary, then the number is right-justified; that is, blanks are put in front of the number to fill out the remaining space. In the case of real numbers, rounding off is done when needed.

The second procedure, *Val*, converts from a string to a number (again, either real or integer). The string itself must contain only a number and no characters other than +, −, ., and *E* in the appropriate places. And,

of course, the number in the string must be of the same type as the variable to which *Val* is converting it. Since there are so many chances for error, *Val* has a third parameter, a result value, which tells you whether or not there are any problems. If the result is greater than 0, then this last parameter indicates the character of the string at which it ran into problems. If the result is 0, then either there were no problems during the conversion, or the string is completely empty. In any case, if there is a problem, the numeric variable that is to be the destination of the conversion operation is unchanged. Here are a few examples:

```
Val(S, I, Result);

Contents of S         I            Result

"14916"              14916         0
" 32"              <unchanged>     1 {space}
""                 <unchanged>     0 {error, but no chars}
```

```
Val(S, X, Result);

Contents of S         X            Result

"4281.953"          4281.953        0
" 332.3"             332.3          0
" 332.3"           <unchanged>      7 {space}
"4,281"            <unchanged>      2 {comma}
""                 <unchanged>      0 {error, but no chars}
```

A word of caution for those of you using 8-bit (CP/M) systems: Do not use *Str* or *Val* within a function that is itself called within a *Read*, *Readln*, *Write*, or *Writeln* statement. Strange and undesirable things will happen as a result. Instead, call the function beforehand, assigning its value to a variable, then use that variable in the input or output statement. (Those of you with 16-bit machines (CP/M-86, MS-DOS) needn't worry about any of this.)

STRINGS AS PARAMETERS

You've probably noticed in these examples that whenever we've passed a string to a procedure or function, we've given that parameter a named type, like *BigStr* or *TokStr*, rather than **string**[255] or the like. For example, we used

```
procedure GetWord(var Line, Word : BigStr);
```

instead of

```
procedure GetWord(var Line, Word : string[255]);
```

In Turbo Pascal, you cannot directly declare a parameter as a string of an anonymous string type. Instead, you must declare a named data type that is equivalent to a string of some length, then use that data type in the parameter declaration, like this:

```
program ParseText;
type
  BigStr = string[255];
procedure Parse(var Line, Word : BigStr);
begin
end; { procedure Parse }
```

When strings are passed as parameters to procedures, it is usually a good idea to pass them as **var** parameters, rather than as *value* parameters (parameters declared without the reserved word **var**). As we mentioned in Chapter 12, parameters that are passed by value are copied when the subprogram is called—requiring more time and space than **var** parameters. However, when a string is passed as a **var** parameter, the Turbo Pascal compiler imposes a restriction that may at first seem like an inconvenience: It requires that the formal and actual parameters have the same maximum length. Thus, if you try to call the procedure *Parse* from the previous example, with a string whose maximum length is 80, Turbo Pascal will indicate a "type mismatch" error.

The reason for this restriction is to prevent you from placing more characters into the parameter than it can hold, possibly writing over other important memory locations in the process. However, if you are careful (and especially if you are not going to change the parameter but are making it a **var** parameter to save time and space), you can disable this kind of error checking. The compiler directive {$V-} turns it off, and the directive {$V+} turns it back on.

With this directive disabled, Turbo Pascal will no longer check to see if the string lengths match. Like most "disable" options, you should use this with caution; if you aren't careful with passing different length strings (that is, strings with different defined maximum lengths) to the same procedure, you can get some bizarre errors. We recommend that if you do disable checking of string parameters, turn the check off for each specific call, and then turn it right back on again:

```
{$V-}     { turn off string checking }
Parse(TLine,Tword);
{$V+}     { turn on string checking  }
```

This way you will turn it off only where you actually need it.

REVIEW

In this chapter, we have learned about Turbo Pascal's string types, and how to use them to manipulate strings of characters. We've shown the syntax of a string type declaration, and discussed such operations as string comparison, string assignment, and Turbo Pascal's built-in string procedures and functions. We ended with a discussion of Turbo's

string conversion routines and some of the finer points of using string parameters.

In the next chapter, we'll discuss records, another structured type that can combine data of many different types into one unit.

15 Records

We've told you about arrays and strings, both of which allow you to create collections of objects of the same type. But what if you want a collection of objects of *different* types, such that, like an array, the entire collection can be referred to by a name? For this purpose, Pascal offers another kind of structured type: *records*.

Let's return for a moment to our checkbook example from Chapter 12, to see how records might be useful. Originally, we declared an array of real numbers to hold the check amounts:

```
var
  Check : array[1..10] of real;
```

Now suppose that, along with the amount of each check, we wanted to store the date it was written and to whom it was written. How could we do this? One way might be to declare many arrays, each using the check number as an index, to hold each piece of information about a check. The arrays might look something like this:

```
type
  CheckNumType  = 1..1000;
  MonthType     = (January, February, March, April, May, June,
                   July, August, September, October, November,
                   December);
  DayType       = 1..31;
  YearType      = 1980..2000;
  PayeeType     = string[40];
var
  CheckAmt      : array[CheckNumType] of real;
  CheckMonth    : array[CheckNumType] of MonthType;
  CheckDay      : array[CheckNumType] of DayType;
  CheckYear     : array[CheckNumType] of YearType;
  CheckPayee    : array[CheckNumType] of PayeeType;
```

This technique works, and is often used in languages like FORTRAN, where there is no other alternative. However, these data structures don't really accomplish grouping all of the information about a single check together as a unit. Instead, the information on each check is spread among many arrays, and must be "pieced together" when a complete set of information about the check is desired. For instance, to

make a working copy of the information for one check, say, check *N*, you might have to write the statements

```
CheckCopyAmt   := CheckAmt[N];
CheckCopyMonth := CheckMonth[N];
CheckCopyDay   := CheckDay[N];
CheckCopyYear  := CheckYear[N];
```

instead of being able to write something like

```
CheckCopy      := Check[N];
```

and transferring all of the information with a single assignment statement. Record structures solve this problem by allowing you to create a data object that consists of smaller objects of different types bundled together. These smaller objects, called *fields*, can be accessed individually, or the entire record can be referred to by name.

We'll now demonstrate how to create a record type for the checkbook example. We want each variable of this type to contain all information related to a single check: amount, date, and to whom it was written. Here's how the record type might be declared:

```
type
  Check = record
     { The reserved word "record" identifies a record type }
    Amt   : real;          { Each field of the record
    Month : MonthType;       has a name and a type }
    Day   : DayType;
    Year  : YearType;
    Payee : PayeeType;
  end;  {The reserved word "end" marks the end of the record}
```

What does this definition mean? It defines a type called *Check*, which is a record type. Each object of the type *Check* consists of a group of smaller data objects (fields), each with a name and a type. The field *Amt*, which holds the check amount, is of type real. The next field, *Month*, contains an object of the type *MonthType*, which holds the month the check was written, and so on.

Now, suppose you have a variable called *MyCheck*, which you have declared to be of the type *Check*. How do you access the individual objects within *MyCheck*? You do this by writing the name of the record variable (*MyCheck*), followed by a period, followed by the name of the field. Thus, you can refer to the amount of *MyCheck* by writing

```
MyCheck.Amt
```

Here is a program fragment that shows how you might declare the variable *MyCheck* and fill its fields with values.

```
        ...
var
  MyCheck  :  Check;
        ...
begin
  MyCheck.Amt   := 100.00;
  MyCheck.Month := February;
  MyCheck.Day   := 10;
  MyCheck.Year  := 1986;
  MyCheck.Payee := 'Philippe Kahn';
        ...
```

As we suggested previously, however, one of the powerful features of records is that they can be copied all at once by a single assignment statement. If *MyCheck* and *YourCheck* are both variables of type *Check*, then the statement

```
YourCheck := MyCheck;
```

will copy all of the fields from *MyCheck* into *YourCheck*.

What kinds of data objects can you use as the fields of a record? Almost anything, subject, of course, to the restriction that no Turbo Pascal data structure can be larger than 65535 bytes. Arrays, strings, scalars, and even other records may be the fields of a record.

If a record contains another record, you can access the subrecord and its fields exactly the way you access the arrays contained within a multidimensional array. Suppose you declared a record type like this:

```
type
  Transaction = record
    Purpose : string[80];
    Payment : Check;
  end;
```

The record type *Transaction* contains two fields: a string of 80 characters, whose field name is *Purpose*, and a record of the type *Check*, whose field name is *Payment*. If *Sale* were a variable of type *Transaction*, then you could refer to the *Payment* field of *Sale* as *Sale.Payment*, and to the *Amt* field of *Sale.Payment* as *Sale.Payment.Amt*. You can also use record types as the components of other structured types; in particular, you can have an array of records. With this knowledge, we can rewrite the data structures for our checkbook-balancing program as

```
type
  CheckNumType =  1..1000;
  MonthType    = (January, February, March, April, May,
                  June, July, August, September, October,
                  November, December);
  DayType      = 1..31;
  YearType     = 1980..2000;

  PayeeType    = string[40];
  Check        = record
    Amt   : real;
```

```
    Month : MonthType;
    Day   : DayType;
    Year  : YearType;
    Payee : PayeeType;
  end;

var
  CheckBook  : array[CheckNumType] of Check;
```

which is exactly what we wanted in the first place.

To work with an individual field of a check within the array *CheckBook*, we could write statements of the form

```
for N := 1 to NumChecks do  {Total the amounts of the checks}
  Total := Total + CheckBook[N].Amt;
```

Figure 15-1 and Figure 15-2 depict the formal syntax diagrams for a record type declaration.

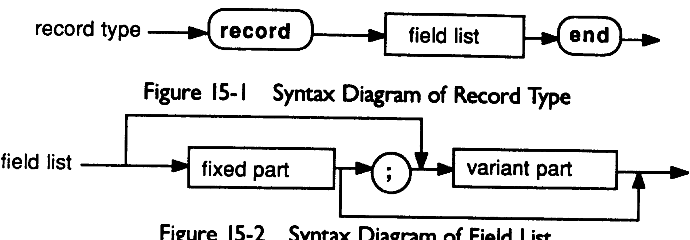


Figure 15-1 Syntax Diagram of Record Type

Figure 15-2 Syntax Diagram of Field List

(For the moment, disregard the box labeled "variant part"; we'll cover this feature later in the chapter.)

THE WITH STATEMENT

Previously, we showed you how to access each field of a record individually in order to assign values to those fields. You had to type in the name of the record variable, plus the name of the field, for every assignment:

```
MyCheck.Amt   := 100.00;
MyCheck.Month := February;
MyCheck.Day   := 10;
      ...
```

This can get rather tedious, especially if many fields are to be assigned. For this reason, Pascal provides you with a shortcut that eliminates typing the name of the record variable repeatedly: the **with** statement.

Using the **with** statement, you can tell the compiler the name of the record variable you are using and then refer to its fields using the field names. The following is an example of a **with** statement that performs the same assignments as the previous statements:

```
with MyCheck do
begin
  Amt   := 100.00;        { Assigns to MyCheck.Amt }
  Month := February;    { Assigns to MyCheck.Month }
  Day   := 10;          { Assigns to MyCheck.Day }
       ...
end;
```

The syntax diagram of the **with** statement is shown in Figure 15-3.

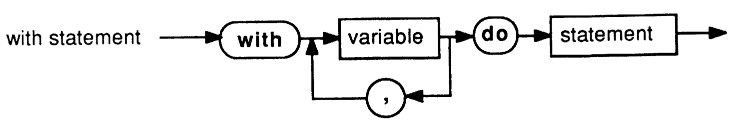


Figure 15-3 Syntax Diagram of With Statement

While it isn't good practice to do it, Pascal allows you to have field identifiers that are the same as the names of variables, types, constants, procedures, and so on. Thus, when using the **with** statement, remember you are creating a special scope in which the field names will take precedence over, and perhaps "hide," other identifiers. In the previous example, if you had had a variable called *Month*, it would not have been accessible within the **with** statement.

With statements can also be nested; for instance, it is legal to do the following:

```
var
  Sale : Transaction; { as defined in the earlier example }
   ...
begin
   ...
  with Sale do
    with Payment do
    begin
      Amt   := 100.00;        { Assigns to Sale.Payment.Amt }
      Month := February;    { Assigns to Sale.Payment.Month }
      Day   := 10;          { Assigns to Sale.Payment.Day }
           ...
    end;
```

Of course, the variables specified in the nested **with** statements need not have any relationship to each other at all. For example, we can write:

```
var
  FirstRecord : record          { An "anonymous" record type }
    Field1, Field2  : integer;  { If fields are of the same
                          type, we can declare them together }
    Field3          : real;
  end;
```

```
  SecondRecord : record  { Another "anonymous" record type }
    Field4     : real;
    Field5     : integer;
  end;

begin
    ...
  with FirstRecord do
    with SecondRecord do
    begin
      ...  { Within this compound statement, the fields of
             both records (Field1...Field5) can be accessed
             by their field names }
    end;
```

However, if **with** statements are nested, and two of the **with** variables have fields of the same name, the most-deeply nested **with** takes precedence. If we modify the previous example so that some of the field names are the same, we can see what happens:

```
var
  FirstRecord : record
    Field1, Field2  : integer;
    Field3          : real;
  end;

  SecondRecord : record
    Field1     : real;
    Field2     : integer;
  end;

begin
    ...
  with FirstRecord do
    with SecondRecord do
    begin
      ...  { Within this compound statement, the identifiers
             Field1 and Field2 refer to SecondRecord.Field1
             and SecondRecord.Field2, respectively. Field3
             still refers to FirstRecord.Field3, since there
             is no overlap. The "hidden" fields of FirstRecord
             can still be accessed by their full names:
             FirstRecord.Field1 and and FirstRecord.Field2 .}
end;
```

The Pascal language allows one more "trick" to make **with** statements more convenient to use. Instead of nesting **with** statements, you may specify a list of variables in a single **with** statement to accomplish the same thing. The statement

```
with FirstRecord, SecondRecord do
     ...
```

is precisely equivalent to

```
with FirstRecord do
  with SecondRecord do
     ...
```

There's one more point to remember when working with the **with** statement: When the **with** variable is an element of an array, do not change the value of the index inside the **with** statement. The code shown in the following example demonstrates what *not* to do:

```
    ...
var
  CheckBook  : array[CheckNumType] of Check;
  I : integer;
    ...
begin
    ...
  I := 1;
  with CheckBook[I] do
  begin
    ...
    I := 5; { Depending on the compiler, this may or may not
              change what location the fields of the "with"
              variable refer to. The Pascal language does
              not define what will happen in this case. }
    ...
  end;
```

While we haven't discussed *pointer variables* yet, it's important to note that the same restriction applies when the **with** variable is pointed to by a pointer: The pointer may not be changed. In general, the rule is *don't do anything that might change the identity of the **with** variable.*

VARIANT RECORDS

Occasionally you may run into a situation where the same record, or part of a record, may need to store different kinds of data. For instance, suppose you are creating a record type to keep track of sales transactions in a store. There is some information that you will want to track under all conditions and in the same manner: the amount of the purchase, how it was paid for, and the date and time of the transaction. However, the way you record other information may vary. For instance, if the person paid by credit card, you'd want to record the kind of credit card, the credit card number, and the expiration date. And if the person paid by check, you'd want the check number, the amount of the check (in case the customer got cash back from the transaction), and the customer's driver's license number.

One way to design such a record type is to allocate a separate field for every possible piece of information, leaving the unused fields empty. The record type definition might look something like this:

```
type
  MonthType    = (January, February, March, April, May,
                  June, July, August, September, October,
                  November, December);
  DayType      = 1..31;
```

```
  YearType     = 1980..2000;
  PaymentType  = (Cash, Check, CreditCard);
  CardType     = (Amex, Visa, MC);

  Purchase = record
    Amount : real;
    Month  : MonthType;
    Day    : DayType;
    Year   : YearType;
    Hour   : 0..23;
    Minute : 0..59;
    MethodOfPayment: PaymentType;
    CheckNumber: integer;      { These fields used for
                                 check purchases only }
    CheckAmt: real;
    LicenseNumber: string[20];
    Card   : CardType;         { These fields used for
                                 card purchases only  }
    ExpMonth: MonthType;
    ExpYear: YearType;
  end;  { record Purchase }
```

While this sort of definition will do the job adequately, it can waste large amounts of storage space. No matter what kind of purchase is made, some of the fields are guaranteed to be left empty.

To help economize on storage space, Pascal provides a feature called a *variant record*, which allows mutually exclusive fields (fields that will never be used at the same time) to share the same storage space within the record. The result can be a dramatic reduction in memory consumption.

Here is how to define a variant part for the previous record, allowing the mutually exclusive fields for check and credit card information to overlap. Note that the variant part of a record must come after all normally existing fields (the *fixed part*).

```
Purchase = record
  Amount : real;
      ...
  Minute : 0..59;
  case MethodOfPayment : PaymentType of
     { Beginning of variant part }
    Check      : ( CheckNumber: integer; { First variant }
                   CheckAmt: real;
                   LicenseNumber: string[20] );
    CreditCard : ( Card   : CardType; { Second variant }
                   ExpMonth: MonthType;
                   ExpYear: YearType );
  end;  { record Purchase }
```

The variant part of a record begins with the reserved word **case**, followed by the name and type of a special field of the record, called the *tag field*. Here the tag field is *MethodOfPayment*. This field, besides carrying information about the purchase, serves another purpose: By

looking at *MethodOfPayment*, your program can decide what information it expects to find in the rest of the record.

Following the definition of the tag field comes the reserved word **of**, followed by one or more lists of field definitions. Each of these lists, called a *variant*, describes how the remaining space in the record will be used for a different value of the tag field.

If the value of *MethodOfPayment* is *Check*, then the rest of the space in the record will hold the fields *CheckNumber* (the number of the check), *CheckAmount* (the amount of the check), and *LicenseNumber* (the driver license number of the issuer of the check). On the other hand, if *MethodOfPayment* has the value *CreditCard*, then the same space is used to hold the fields *Card* (the kind of credit card used), *ExpMonth* (the expiration month of the card), and *ExpYear* (the expiration year of the card). What happens, you may ask, if *MethodOfPayment* has the value *Cash*? The answer is that there are no additional fields in the record, and the remaining space contains no useful information.

The syntax of this optional part of a record definition, the variant part, is shown in Figure 15-4.

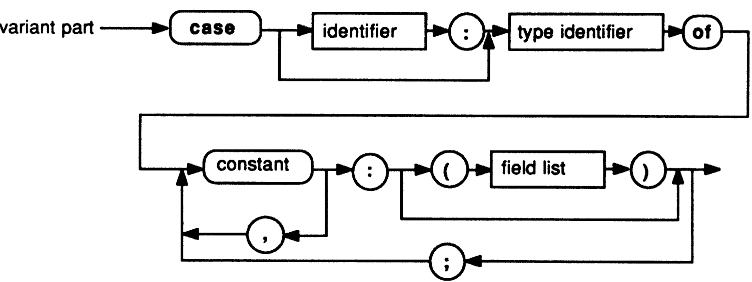


Figure 15-4 Syntax Diagram of Variant Part

As you can see in the diagram, the same variant might be used for more than one value of the tag field. To illustrate this, suppose that we added the value *TCheck* (for traveler's check) to the type *PaymentType*, and we wanted the same information for this form of payment as we did for a check. We could redefine things as follows:

```
type
  PaymentType  = (Cash, Check, CreditCard);
    ...
  Purchase = record
    Amount : real;
      ...
    Minute : 0..59;
    case MethodOfPayment : PaymentType of
      Check, TCheck : ( CheckNumber: integer;
```

```
                     CheckAmt: real;
                     LicenseNumber: string[20] );
    CreditCard     : ( Card    : CardType;
                     ExpMonth: MonthType;
                     ExpYear: YearType );
  end;  { record Purchase }
```

When using variant records, it is important to keep track of which variant you are using at any given moment. Despite the fact that Pascal lets you define different field names and types for different values of the tag field, it does *not* check to make sure you are using the right ones. Careless use of variant records can lead to scrambled data and disastrous results. Usually, your program will have a **case** statement to handle the different variants of a record. There are many possibilities; this is just one technique that's worth including.

```
   ...
case ThisPurchase.MethodOfPayment of
     { ThisPurchase is of type Purchase }
  Cash:  begin
           ...
         end;
  Check, TCheck:
         begin
           ...
           CheckAmt := ...etc.
           ...
         end;
  CreditCard:
         begin
           ...
           Card := ...etc.
           ...
         end;
   ...
```

FREE UNIONS: OMITTING THE TAG FIELD

The computer science term for a variant record that includes a tag field is a *discriminated union*, which is a combination, or union, of field definitions that are discriminated from one another by the value of the tag field. Pascal also allows you to use another kind of variant record, or union, called a *free union*.

In the syntax diagram for a variant part (Figure 15-4), you may have noticed a path that we did not mention earlier: There is an arrow that goes around the identifier (and the subsequent colon) for the tag field. As you might guess, this means that it is possible to define a variant part with *no* tag field, though a type must still be given.

A free union is a variant record with no tag field. Usually, it is used in one of two cases: (1) when the correct set of fields to be used can be determined some other way than from a tag field, or (2) when the programmer intentionally wants to look at a location in memory that is of two different types simultaneously—depending on which field name is being used.

The latter is a sophisticated programming technique that is not to be used by the unwary. We'll show you a little about how this might be useful in the advanced section of this tutorial.

REVIEW

In this chapter, we introduced a group of structured types, called records, that can be used to hold collections of variables of other types. Records consist of fields that hold these variables, with each field having a distinct name and type.

The **with** statement can be used to make referring to the fields of records easier. Variant records can save storage space by using the space in a record variable to hold more than one data object (depending on the value of a tag field).

Free unions, or variant records without tag fields, can be used when there is no need for a tag field, or for when there is a need for some advanced programming techniques.

In the next chapter, we'll discuss one more kind of structured type: the set type.

16 Sets

You may remember the concept of sets from your early math classes; if you do, you're ahead of the game, because sets in Pascal are very similar. In Pascal, a set is a collection of zero or more objects of the same scalar type (called the base type), which has some properties that make it an especially efficient way to store information.

There may be times when you want to check to see if a value of a scalar type (integer, byte, char, boolean, and so on) belongs to a set of values of that type. For example, suppose you had a program in which you needed to test whether *Ch*, a variable of type char, contained a vowel or not. Without the use of sets, how would you accomplish that? You could write out a long **if** statement:

```
if (Ch = 'a') or (Ch = 'e') or (Ch = 'i') or (Ch = 'o')
      or (Ch = 'u') or (Ch = 'y') or (Ch = 'A') or (Ch = 'E')
      or (Ch = 'I') or (Ch = 'O') or (Ch = 'U') or (Ch = 'Y')
then
      . . .
```

Unfortunately, the statement structure of this test is somewhat hard to read. Also, the characters you're testing for are fixed, and are not easy to change without rewriting the statements.

Fortunately, the Pascal set affords you a better way to describe to the compiler the list of allowable characters. To rewrite the previous test using a set, we could create a set that contained all the vowels in the alphabet by listing them between square brackets. Then, we could use the Pascal set operator, **in**, to see if *Ch* was there. With sets (and with a little help from the *UpCase* function), the test would look like this:

```
if UpCase (Ch) in ['A','E','I','O','U','Y'] then...
```

which is much easier to read and understand. This set is a set of objects of the type char, but a set can be defined to hold objects of any scalar data type. There is one restriction, though: The base type of the set (the type of objects contained within it) must not have more than 256 possible values. Thus, you could define a set of objects of the type byte, which has exactly the maximum number of possible values, or a set of objects of the subrange type 75..98. But you could *not* define a

set of objects of the type integer, because there would be 65,536 possible values!

Pascal sets are *proper sets*; that is, no object can be contained in such a set more than once. Thus, a Pascal set can have at most 256 members. To remember whether or not an object is in a set, Pascal sets or clears a single bit in memory. Thus, a set of the maximum size is only 32 (256/8) bytes long—a very efficient use of storage.

BUILDING A SET: THE SET CONSTRUCTOR

A *set constructor* consists of a list of expressions of the same scalar type, separated by commas and enclosed by square brackets ([and]). If there are many elements, and some of them have consecutive ordinal values, you can use the same notation used for subranges (that is, two expressions separated by two periods ('..')). Here are some examples of set constructors:

```
[]                  { empty set contains nothing }
[1,3,5,7,9]                        { set of byte }
['A'..'Z']                         { set of char }
[Mon,Wed..Fri]                     { set of Days }
[Jan..Aug,Oct..Dec]              { set of Months }
```

It's important to remember that the objects within a set constructor need not be constants. They can, in fact, be any kind of expression whose result is of the base type of the set. This feature (which is unknown even to some experienced users of Pascal) can make the use of sets especially convenient. For instance, suppose *A* is a variable of the type char. To construct a set that consists of all the characters from the character stored in *A* to the letter '*w*', you could write the set constructor

```
[A..'w']
```

Of course, if there are no characters between *A* and '*w*' (that is, *A* is past '*w*' in the ASCII character set), the result is the empty set.

DEFINING A SET TYPE

To define an object of a set type, use the reserved words "**set of**," followed by the name of the base type, or you can also supply an anonymous type—usually a subrange—as the base type. Here are some examples of set types:

```
type
  CharSet = set of char;
       { Set of objects of the type char }
  MonthDays = set of 1..31;
       { Set of objects of the
         (anonymous) subrange type "1..31" }
```

```
  DayType = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
  Days = set of DayType;
       { Set of objects of DayType }
  WorkWeek = set of Mon..Fri;
       { Set of objects of the
         (anonymous) subrange type "Mon..Fri". }

  Colors = set of (Red, Green, Blue);
       { Set of objects of the (anonymous) declared
         scalar type (Red, Green, Blue) }
```

The syntax diagram in Figure 16-1 shows precisely how to specify a set type.

Figure 16-1 Syntax Diagram of Set Type

Here's a simple example that shows one very good use for sets: It makes sure that a character input from the user is a legal command for a program.

```
program charTest;
{$V-}          { to avoid any problems passing strings }
type
  CharSet = set of char;
  Prompt  = string[80];
var
  Cmd     : char;
procedure Getchar(var Ch : char; Msg : Prompt;
                  OKSet : CharSet);
                  { Write a message, then get a character
                    from the user. Ignore any character that
                    is not in the set OKSet. }
begin
  Write(Msg);
  repeat
    Read(Kbd,Ch);          { read the character from the
                             keyboard, but do not echo it }
    Ch := UpCase(Ch)           { force Ch to be upper case }
  until Ch in OKSet;
  WriteLn(Ch)
end;  { procedure GetChar }

begin  { Statement Part of program charTest }
  repeat
   Getchar(Cmd,'CharTest>  S)peak, C)ount, Q)uit:  ',
           ['S','C','Q']);
   case Cmd of
     'S'   : Writeln('Woof! Woof!');
     'C'   : Writeln('1, 2, 3, 4, 5, 6, 7, 8, 9, 10')
   end
  until Cmd = 'Q'
end.  { program charTest }
```

A procedure like the previous *GetChar* prompts the user with a message, then lets him or her type in a single character. It converts the character to upper case, then checks to see if it's a valid command. If not, it waits until a valid character is entered. The **in** operator is used to see if the character is in the set of legal characters. The **in** operator is one of a large selection of operations available for use on Pascal sets, as you will see in the next section.

SET OPERATIONS

The Pascal language provides operators to form the union, intersection, and set difference of any two sets. Also available are the membership operator, **in**; the subset operators, <= and >=; and the equality operators, = and <>.

Set Membership: The In Operator

In the sample program we used the operator **in** to determine whether a character was part of a set of characters. The expression

```
Object in SetOfObjects
```

returns the boolean value TRUE if and only if *Object* is of the base type of *SetOfObjects* and *Object* is a member of *SetOfObjects*. Note that *Object* can also be represented by an expression as long as it is of the proper type.

Set Equality and Inequality

The equality and inequality operators, = and <>, do precisely what you might expect when used on sets: They return the value TRUE whether the two sets that they operate on have exactly the same members (for =) or not (for <>). In either case, it is not required that the sets have exactly the same base type, as long as the base types are compatible. For instance, suppose we executed the following program fragment. What would the output be?

```
program EqualityTest
var
  Set1 : set of char;
  Set2 : set of 'a'..'x';
begin
  Set2 := ['a', 'b', 'g'..'w'];
  Set1 := Set2;
  Writeln(Set1 = Set2);
end.
```

If you guessed TRUE, you are correct. Since all of the elements of *Set1* and *Set2* are the same, they are considered to be equal.

Set Union, Intersection, and Difference

The set union operator (+) returns a set that contains any member that is in either of its operands. The set intersection operator (*) returns the elements that are common to both of its set operands. And the set difference operator (−) returns the elements that are in its first operand, but not in its second.

Here are some examples that illustrate the use of these operators. Given the sets *A*, *B*, and *C*, all of the type **set of** char, suppose that

```
A = ['A'..'Z']
B = ['A', 'C', 'E', 'G']
C = ['A'..'D', 'Z']
```

then

```
A * B = ['A','C','E','G']       A + B = ['A'..'Z']
A * C = ['A'..'D','Z']          A + C = ['A'..'Z']
B * C = ['A','C']               B + C = ['A'..'D','E','G','Z']
B - A = []                      A - B = ['B','D','F','H'..'Z']
A - C = ['E'..'Y']              C - A = []
B - C = ['E','G']               C - B = ['B','D','Z']
```

Set Inclusion Operators

The operators <= and >= have special meanings when used with sets. The >= operator returns the boolean value TRUE when its second operand is a proper subset of the first; that is, when all the elements of the second operand are included in the first. Similarly, the <= operator returns TRUE if and only if the first operand is a proper subset of the second. Thus, given *A*, *B*, and *C* from the previous example

```
A <= B , B => A  is  False      B <= A , A => B  is  True
A <= C , C => A  is  False      C <= A , A => B  is  True
B <= C , C => B  is  False      C <= B , B => C  is  False
```

Set Disjunction

Finally, the condition of *set disjunction*, in which two sets have no members in common, can be tested by evaluating the truth of the expression

```
A * B = []
```

that is, if the set of elements in common between the two sets is the empty set, then they are disjoint.

REVIEW

In this chapter, we explained the concept of a Pascal set: how it is constructed, how it is defined, and the operations that can be performed on it.

In the next chapter, we'll begin our discussion of pointers—the facility that allows your program to create new data objects while it is running.

17 Pointers and Dynamic Allocation

Up to this point, whenever you have created a data structure such as an array, you have had to determine the size of such a structure in advance. For instance, in the checkbook example, we reserved space for an array of records, each holding the information about a single check:

```
type
  CheckNumType =  1..1000;
  MonthType    = (January, February, March, April, May,
                  June, July, August, September, October,
                  November, December);
  DayType      = 1..31;
  PayeeType    = string[40];
  Check = record
    Amt   : real;
    Month : MonthType;
    Day   : DayType;
    Year  : YearType;
    Payee : PayeeType;
  end;

var
  CheckBook  : array [CheckNumType] of Check;
```

In this case, there are as many records in our array as there are elements in the index type *CheckNumType*.

The problems that can arise when an approach like this is used are threefold. First, the program may be called upon to balance a very large checkbook, in which case the 1000 records we allocated might not be enough. Second, the number of checks written might be very small, in which case a vast amount of space would be wasted (here, 50 bytes of memory per unused record). And finally, there is the problem of exceeding the memory capacity of the machine you are using. If each of these records takes up 50 bytes of memory, and the array is 1000 checks long, then the array will take up 50K of space. This is more than what will be available on a CP/M system, and comes close to the size limit for a data object on even a large MS-DOS system.

How, then, do we solve the problem of allocating enough, but not too much, space in memory for our data? And how can we "change our minds" about how much memory we need while the program is running? Pascal's *pointer* and *dynamic memory allocation* features provide answers to these questions.

POINTERS

So far, every Pascal data object we've created has had a name or an identifier (possibly accompanied by array indices and/or field names) that can be used to access it. However, if we are going to create brand-new data objects while the program is running, then we have a problem. Since we don't know in advance what objects we are going to create, or how many, how can we give the compiler names for them? (Remember, the compiler, which keeps track of the names of objects, has finished its job and may no longer even be in memory by the time the program runs.)

While the Pascal language is very powerful, one thing it can't do is go back in time and tell the compiler what to name these objects. In fact, the objects that we are going to create will not have their own names at all. How, then, can we manipulate data objects if we don't know their names? The answer is that we will "point" to them, and by doing so show the program where they are in memory. For this purpose, we are going to define a new kind of variable, a pointer.

A pointer is a variable used to record the address in memory where a dynamically created data object (one created while the program is running) is stored. When the program creates a new data object, it finds an unused place in memory to put that object. It then gives you a pointer that points to the location in memory that it chose. Besides holding a memory address, each pointer has a type associated with it (a type of object it can point to). This way, when you use the pointer to refer to an object, Pascal knows what kind of object it is as well as where it is.

Suppose, for instance, you wanted to be able to create new data objects of the type integer while your program is running. To do this, you'd need at least one pointer variable to keep track of them. So, you might declare a pointer variable as follows:

```
var
  IntPointer : ^integer;
```

The notation "^integer" is called a *pointer type*, and is read as "pointer to integer." When we declare *IntPointer* of the type ^integer, we are saying that *IntPointer* is a pointer, and that it may only point to objects of the type integer. Another way of saying this is that *IntPointer* is bound to the type integer. You can create a pointer type that points to any type

of data object at all, and define as many pointers of that type as you like. For instance, here's how we'd define a pointer type to point to a record of the type *Check*, and also create a few pointers of that type:

```
type
  CheckPointerType = ^Check;
  { A type of pointer which is to be used
    to point to objects of the type Check }
var
  { Three pointer variables, each of which can point
    to an object of type Check }
  CheckPointer1 : CheckPointerType;
  CheckPointer2 : CheckPointerType;
  CheckPointer3 : CheckPointerType;
```

The syntax used to specify a pointer type is very simple (see the syntax diagram in Figure 17-1.

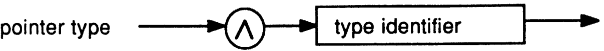


Figure 17-1 Syntax Diagram of Pointer Type

DYNAMIC ALLOCATION: THE NEW PROCEDURE

Now that you understand what a pointer is, let's look at how to use them to create and manipulate data objects. To create a new data object, Pascal provides a built-in routine that does the work for us: the procedure *New*.

In the first example, we'll use the pointer variable *IntPointer* (defined earlier) to create a new object of the type integer. We can do this by making a call to the procedure *New*:

```
New(IntPointer);
```

This call causes a new variable of the type integer to be created in an unused portion of the computer's memory. The pointer variable *IntPointer* is then changed so that it points to the new variable.

What would happen if we made the call *New*(*IntPointer*) a second time? Well, Pascal would oblige us by reserving space for yet another variable of the type integer, and change *IntPointer* to point to that object. So what happens to the first integer allocated? Does it go away? The answer is no, its place in memory is reserved just as before. However, the second call to *New* destroyed the old value of *IntPointer*, which was the only record of that variable's location. The variable is still there, but we have both literally and figuratively "lost its address."

For this reason, it is important to keep a pointer to each dynamically allocated variable that you create. We'll show how to "destroy" such a variable and free the space it occupies for other purposes later in this chapter.

DEREFERENCING POINTERS

After you have created a variable using a pointer, how do you access it? As we've mentioned, the way to do this is to use the pointer variable to "point" to a location in memory. The process of finding the object to which a pointer variable points is called *dereferencing* the pointer; this is signified in Pascal by the caret symbol (^).

For instance, to refer to the variable pointed to by *IntPointer*, we use the notation *IntPointer^*. The following program fragment sets *IntPointer^* to 6, then prints it out:

```
   ...
IntPointer^ := 6;
Writeln(IntPointer^);
   ...
```

A dereferenced pointer can be used anywhere you can use an object of the type that it points to, and the dereferencing symbol can be followed by indices (for array types), field selectors (for record types), or both. For instance, if we wanted to write out the amount of the check pointed to by *CheckPointer1*, we could write

```
Writeln(CheckPointer1^.Amt);
```

Similarly, if we wanted to write out the fifth letter of the name of the payee of the same check, we could write

```
Write(CheckPointer1^.Payee[5]);
```

Pointers, like all variables, can be assigned the value of other variables of the same type. The assignment statement

```
CheckPointer2 := CheckPointer1;
```

does what you would expect it to do: It assigns the value of *CheckPointer1* to *CheckPointer2*, so that both pointers will point to the same data object. Of course, to try to assign the value of a pointer that points to one type to a pointer that points to another type is illegal. The statement

```
IntPointer := CheckPointer1;
```

will cause an error when you try to compile it.

THE NIL POINTER

When you dereference a pointer, make certain that the pointer contains the legitimate address of a variable in memory. Like all variables in

Pascal, a pointer variable that has not been assigned a value (either by the *New* procedure or by assigning it a value from another pointer) can have any value at all. This means that dereferencing an undefined pointer can be particularly destructive. Changing the memory location pointed to by such a pointer can alter any part of memory at all, including the operating system, your program, or any data. One incorrect pointer reference can instantly crash your entire system.

For this reason, Pascal has provided a special value to which you can set a pointer. This value is represented by the reserved word **nil**, which in essence means that the specified pointer does not point to any valid object.

Whenever you use pointers, it is a good idea to set them to **nil** if you know they do not currently point to valid data. It is also good practice to test pointers to see if they are equal to **nil** before using them.

The value **nil** can be assigned to any variable of a pointer type. You can test to see whether a pointer is equal to **nil** or not by using a comparison of the form

```
IntPointer = nil
```

or

```
IntPointer <> nil
```

Equality and inequality, by the way, are the only relational operators that may be used on pointers.

LINKED LISTS

Now we're ready to show you how to solve the problem of maintaining a list of any number of checks. The last piece needed to solve the puzzle is a data structure you can build from dynamically allocated records and pointers: the *linked list*.

To illustrate how useful linked lists can be, let's show two more approaches to the checkbook problem. First, we'll show a better solution that uses arrays of pointers, and then we'll show the linked list approach.

One way to store checkbook information might be to use an array of pointers, one for each check. Then, as needed, we could create new variables to hold the check information itself. To do this, we'd declare:

```
type
  CheckNumType =  1..1000;
  MonthType    = (January, February, March, April, May,
                  June, July, August, September, October,
                  November, December);
  DayType      = 1..31;
  YearType     = 1980..2000;
  PayeeType    = string[40];
```

```
  Check = record
    Amt   : real;
    Month : MonthType;
    Day   : DayType;
    Year  : YearType;
    Payee : PayeeType;
end;

  CheckPointer = ^Check;

var
 CheckBook : array[CheckNumType] of CheckPointer;
  { Now an array of pointers. }
```

The array *CheckBook*, which used to be an array of records of the type *Check*, is now an array of pointers, variables of the type *CheckPointer.* How does this save us space? As you may recall, a record of the type *Check* consumes about 50 bytes of memory. However, a pointer in Turbo Pascal consumes 4 bytes of memory on a 16-bit system, and only 2 bytes of memory on an 8-bit system. Thus, we have allocated a much smaller amount of space initially (by a factor of 25 or 12.5, depending on the system), and can allocate the rest only if it's needed.

To keep track of the checks in the book using this data structure, you would probably want to initialize all of the elements of the *CheckBook* array to **nil**. In Turbo Pascal, the value **nil** is represented by a pointer whose bytes are all zero, so we can use the array-initializing trick we mentioned in Chapter 13, and write

```
FillChar(CheckBook, SizeOf(CheckBook), 0);
```

which will set all of the pointers in the array to **nil**.

From that point on, we perform the *New* procedure on each element of the array *CheckBook* that we need to store information on. Then, if we need to access the information for check number *I*, we can perform an operation like this:

```
if CheckBook[I] = nil then
  Writeln('No information on this check!')
else
  Writeln('The amount of check #',I,' is ',
          'CheckBook[I]^.Amt');
```

Again, before using it, we double-check that the pointer we are about to dereference is not **nil**. In doing this, we may be able to save a little more memory with a simple convention: A **nil** pointer can stand for an outstanding check. Thus, we need not consume space for checks for which we are not storing any information.

While this method is much better than allocating a record for every check we might (or might not) want to track, it could still waste space if some of the pointers in the array *CheckBook* were not used. Also, if the number of checks turn out to be larger than the number of pointers

in *CheckBook*, we'd once again face the problem of running out of room in the array. What we'd really like to do is dynamically allocate not only the records, but the pointers to them as well, so that no space will go unused.

To do this, we can string the *Check* records together in a linked list, a list of records in which each record contains a pointer to the next record on the list. Thus, each time we allocate a new record, we allocate a new pointer as well, so that there is always a pointer available to point to the next object added.

To clarify, let's look at the declaration for a new kind of record of the type *Check*, one which includes a pointer that can point to the next check on the list.

```
type
CheckPointer = ^Check;  { This definition is allowed to
                          precede the definition of the
                          type Check. }
Check = record
  CheckNumber : CheckNumType;
  Amt         : real;
  Month       : MonthType;
  Day         : DayType;
  Year        : YearType;
  Payee       : PayeeType;
  NextCheck   : CheckPointer;
              { This new field can point to another record
                of the type Check. }
end;
```

In this example, we show the only exception to the rule that all Pascal identifiers must be declared before they are used. We declare the pointer type *CheckPointer* before we declare the type it is bound to (namely *Check*), so that we can use it as the type of a field in *Check* itself. Since all pointers are the same regardless of what they point to, Pascal will know, without knowing all about *Check*, how a pointer to this type should be handled. However, if the type *Check* were to remain undefined, Pascal would flag the definition as an error.

Having defined the type *CheckPointer*, we can place a field of this type in the record type *Check*. Now, we can define a variable of the type *CheckPointer* to point to the first record of the list. We'll want to set it to **nil** to begin with, as well, so that the program can easily tell that there are no items on the list.

```
var
  CheckBook : CheckPointer;
       ...
begin
       ...
  CheckBook := nil;
       ...
```

Until we need to store the information of the first check, this is all the storage space needed. When it comes time to create a record for the first check, we can use the statement

```
New(CheckBook);
```

to create the first record. We'll also want to set the *NextCheck* field of the new record to **nil**, to indicate that there is no next check on the list.

As we need to reserve storage for each new check, we can make the *NextCheck* pointer of the previous check on the list point to the new check, and set the *NextCheck* pointer of the *new* last check to **nil**. As the list grows, it will look like this:

```
CheckBook--> (Check Info) !
           !  NextCheck   !--> (Check Info) !
                             !  NextCheck   !--> nil
```

Using this method, there is no wasted space and, again, checks for which no information is stored need not have memory allocated for them. In the preceding record definition, we added another field to the record *Check* to contain the number of the check, so that we would not have to allocate records (or *nodes*, as they are sometimes called) for unused checks.

To find a check with a particular number on the list, we can scan the list from beginning to end. This process of reading through a linked list is called *traversing* the list. The following shows a function that will take a check number and a pointer to a list of checks, returning a pointer to the check with that number:

```
function FindCheck (Num: CheckNumType;
            FirstCheck : CheckPointer) : CheckPointer;

{ Given Num, the number of a check, and FirstCheck, a pointer
  to the first of a linked list of checks, return a pointer
  to the first check found on that list with the given
  number. If no check with that number is found, return nil.}

begin  { FindCheck }
  FindCheck := nil;      { Start by assuming failure. }
  while FirstCheck <> nil do    { Stop if end of list }
    if FirstCheck^.CheckNum = Num then
{ Check found? }
    begin
      FindCheck := FirstCheck;  { If so, set the function }
      Exit                      { result and exit from the }
                                { routine right away }
    end
    else
      FirstCheck := FirstCheck^.NextCheck;
      { Number doesn't match; point to next check, if any.
        Note that since FirstCheck is not a var parameter, we
        only change our local copy.  }
end;  { FindCheck }
```

In this function, we play a number of "tricks" to save storage and make the code as efficient as possible. First, we make use of the fact that *FirstCheck* is not passed as a **var** parameter, and use it as the "moving" pointer to scan the list. (If *FirstCheck* were a **var** parameter, we couldn't change it without losing the main program's only pointer to the list.) We also use the special built-in procedure, *Exit*, to exit from the middle of the function. If we did not use *Exit*, we would have to either traverse the rest of the list unnecessarily, or force the loop to terminate by setting *FirstCheck* to **nil**. The use of *Exit* here is simpler and more efficient than the previously described techniques.

THE HEAP

The *New* procedure finds a place in memory for any dynamically allocated variable you choose to create. For instance, if we use the *New* procedure on the pointer *CheckPointer1* (as defined earlier), we would allocate 50 bytes for the new variable and return the address of that variable in *CheckPointer.*

Where do these 50 bytes come from? When a Turbo Pascal program runs, it reserves a large chunk of memory that remains after space has been set aside for all of your code and your non-dynamic variables. This area is called the *heap*. If you're using a CP/M system (which has, at most, 64K of memory), have loaded Turbo Pascal, and are running a large program, then the heap may be very small, possibly as small as 1K (enough to hold about 50 check records). On the other hand, if you're using a 16-bit system (PC-DOS, MS-DOS, CP/M-86, and so forth) with lots of RAM, then the heap can be very big. How big? A program running under PC-DOS version 2.0 on a 512K IBM PC system has over 430K of memory in the heap, or enough space for about 8,800 check records.

How is the heap used? When your program starts running, a special predefined variable called *HeapPtr* points to the first location in the heap, sometimes called the "bottom" of the heap. When you call *New*(*IntPointer*), *IntPointer* gets the value of *HeapPtr,* and *HeapPtr* is increased by the size of the data structure that *IntPointer* will point to. The process continues, as necessary, until all of the space is allocated or there is no further need for dynamically allocated variables.

THE MAXAVAIL FUNCTION

How can you make sure that there is enough room on the heap to allocate a variable? You can discover the size of the largest free block of space on the heap via the predefined function *MaxAvail*. In 8-bit systems, *MaxAvail* returns the number of bytes left on the heap; in 16-bit systems, it returns the number of *paragraphs* (16-byte chunks).

On both kinds of systems, it is possible to have a number of free bytes (or paragraphs) larger than 32767. Since this is the largest value that can fit in an integer, Turbo Pascal provides a special way of returning a larger result from this function. If the result of *MaxAvail* is greater than 32767, the values will continue upward from the value −32768 (−32768, −32767, −32766, and so on). You can find the true number of bytes or paragraphs free by testing to see if the result is negative; if it is, assign it to a varible of type real and add 65536.0. The resulting real will contain the correct number. Here is a simple program to do the conversion:

```
program WriteFree;
{ Write the amount of available heap space.  The result is
  given in bytes for a CP/M system, and in paragraphs
  (blocks of 16 bytes) for a 16 bit system. }
var
  TrueFree : real;
begin
  TrueFree := MaxAvail;       { convert to real value }
  if TrueFree < 0.0 then
    TrueFree := TrueFree + 65536.0;
  Writeln('Space available:  ',TrueFree:7:0)
end.
```

DEALLOCATION OF DYNAMIC VARIABLES: MEMORY MANAGEMENT

Once the procedure *New* creates a dynamically allocated variable, the storage space allocated for it remains reserved until the program terminates, or until it is explicitly freed. There are two ways to free space on the heap: using the standard Pascal procedure *Dispose*, or the Turbo procedures *Mark* and *Release*.

Dispose

The *Dispose* procedure provides a simple, programmer-friendly form of memory management. When a dynamically allocated variable is no longer needed, you pass Pascal a pointer to it using the procedure *Dispose*. The memory allocated to this variable will then be freed up so that it can be used for other purposes. For instance, to free the memory allocated for an integer object pointed to by the variable *IntPointer*, you would write

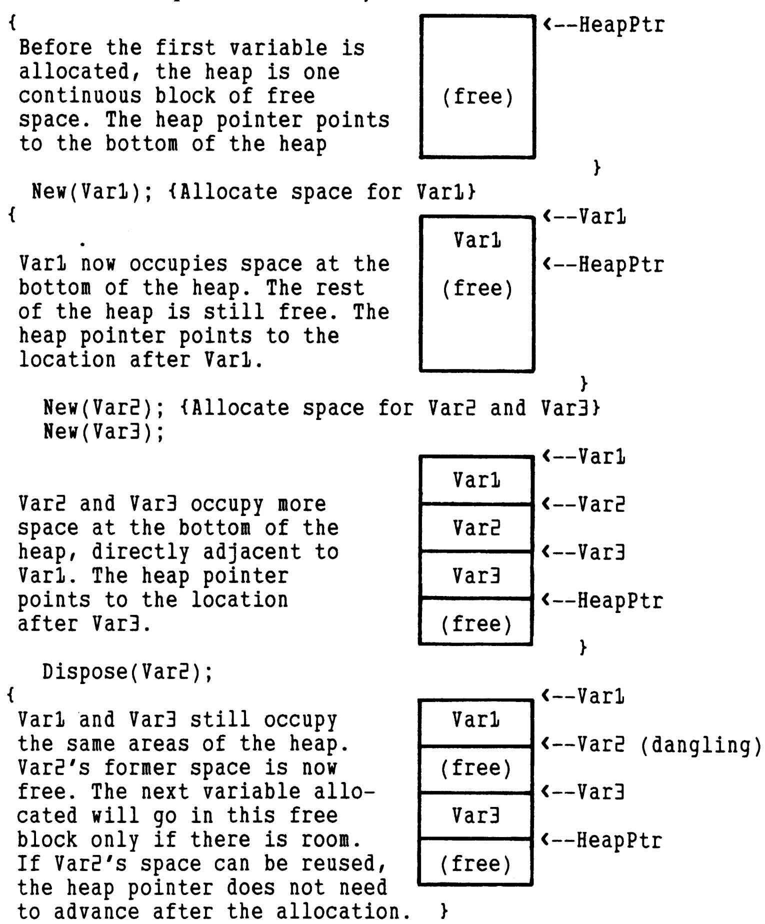
```
Dispose(IntPointer);
```

However—and this is an important point—the value of the pointer variable *IntPointer* does not change after a call to *Dispose*. Thus, until it is assigned another legitimate value (or the value **nil**), *IntPointer* points to a location in memory that is no longer reserved.

Such a pointer is called a *dangling pointer.* Dangling pointers can be every bit as dangerous as uninitialized pointers—they can destroy data, Turbo Pascal's internal data structures, or both. Worse yet, if you attempt to *Dispose* an unitialized or dangling pointer, you will almost certainly cause your computer to crash. For this reason, never attempt to dereference a dangling pointer.

What happens to the heap when you use *Dispose* on a dynamically allocated variable? In Turbo Pascal, a list is kept of all areas of the heap available for re-use. When you free a block of memory, Turbo Pascal attempts to put it back to work as soon as *New* is called again. If the block is big enough to hold the next variable created with *New*, then the space (or as much of it as is needed) is allocated to that variable.

Here is a code fragment, along with some pictures of the heap, that shows how *Dispose* frees memory.

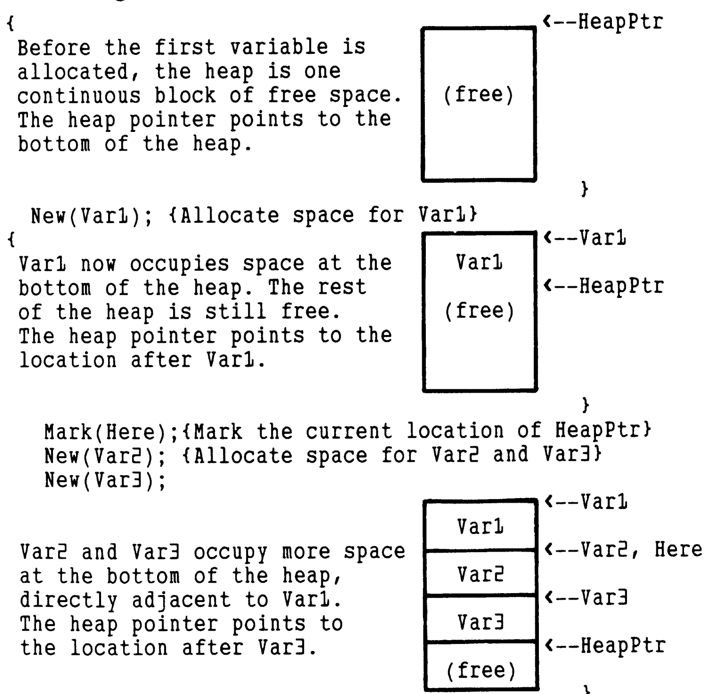


Mark and Release

When *Dispose* is used to free space on the heap, it is possible to create a situation in which many small chunks of unused memory are left in the middle, chunks that are not big enough for some things the program might want to allocate. This phenomenon is called *fragmentation* and can sometimes cause memory to be wasted. For this reason, Turbo Pascal provides another method of managing space on the heap, one that guarantees that all of the free space is in one big block at the top. This method uses the two procedures *Mark* and *Release*.

The procedure *Mark* takes a pointer variable as its parameter, and records in that variable the current location of the top of the heap. A program that uses *Mark* and *Release* will call *Mark* before allocating a group of objects on the heap, saving the value that is returned. Later, the program can call *Release* using the same pointer variable, causing all of the heap space allocated since the call to *Mark* to be freed. Because space is always freed from the top of the heap back to a certain point, no fragmentation is possible.

For the sake of comparison, let's show how the earlier example would work using *Mark* and *Release*.



```
   Release(Here); {Release(Var2) would give same result}
{                                            ------- <--Var1
                                            | Var1  |
 Var1 still occupies the same               |       |<--HeapPtr, Here,
 area of memory. The space allo-            |       |     Var2 (dangling)
 cated to both Var2 and Var3 is             |       |<--Var3 (dangling)
 now free, and the heap pointer             |(free) |
 is moved back to its location              |       |
 before Var2 is allocated.                  |       |
                                             -------
                                                          }
```

At this point, a brief warning is in order. Because *Dispose* and *Mark/Release* represent such different approaches to memory management, both techniques should not be used in the same program. Using them together may produce unpredictable results. For most programs, the *New* and *Dispose* procedures are sufficient to fulfill your memory management needs.

REVIEW

In this chapter, we introduced pointers, variables that hold the addresses of other variables. We explained the notion of a pointer type, a type that is bound to the type of object it points to. We also explained the use of the *New* procedure to allocate anonymous variables that are referenced using pointers. We then showed how to create a linked list, and discussed the two different schemes for reclaiming and re-using memory: *Dispose* and *Mark/Release*.

In the next chapter, we'll discuss how to handle Turbo Pascal files and operating system devices like terminals and printers.

18 Files

By this point you should be able to write programs that create any number of different types of data structures and manipulate them. But what happens when the program ends, as it sooner or later must? All your data structures will quietly vanish, and the information they hold will be lost.

What if you want to save that data until the next time you run the program. For instance, if you have just entered all the check information from your checkbook, you wouldn't want to retype it the next time you reconcile your checkbook. Or, if you are writing a book (like this one), you wouldn't want to lose your work if the program ends or the computer is turned off.

What if you also want to know how to make the computer write to a particular place on the screen or the printer, or how to input different kinds of data from the keyboard.

Files are the answer to these queries. A file is a collection of data structures, all of the same type, that can be written to or read from the peripheral devices attached to your computer. And so, in this chapter, we'll discuss in detail how Turbo Pascal uses files to communicate.

TURBO PASCAL'S I/O PROCEDURES

In virtually all of our programming examples, we have used the built-in procedures *Read*, *Write*, *Readln*, and *Writeln* to exchange information with the computer's screen and keyboard. In fact, these four procedures are used by Turbo Pascal to exchange information with *any* device attached to your computer, including a disk drive, the screen, a printer, or even a modem. This communication process is often called I/O, short for Input/Output.

To Turbo Pascal each of these devices appears as a file, a data structure that represents one of these devices (or a part of one) within your program.

So far, however, you have been able to use *Read*, *Write*, *Readln*, and *Writeln* without knowing anything at all about files. This is because Pascal has arranged it so that when these procedures are called and you don't explicitly tell them what file to read or write, they automatically use two standard files: Input (the keyboard) for *Read* and *Readln*, and Output (the screen) for *Write* and *Writeln*.

Before we get into a discussion of how to open other kinds of files, let's first discuss the properties of these built-in procedures.

Read and Readln

The *Read* and *Readln* procedures read one or more data objects from a file. Their syntax is as follows:

```
Read({FileVariable,} {Var1, Var2...VarN});
Readln({FileVariable,} {Var1, Var2...VarN});
```

The parameter *FileVariable*, which is optional (optional parameters are shown in curly brackets), specifies the file that the procedure is to work with. If no file is specified, the procedure acts as if the standard file variable *Input* had been used, with the keyboard as the source of input. The procedure *Read* reads data and returns it immediately, while *Readln* waits for the end of a line. When reading data from the terminal, however, both of these procedures wait for the end of the line. (If *Read* didn't wait for the end of the line, it would have no way of knowing when you were finished.) When reading from other places, like the disk or one of Turbo's special devices, *Read* doesn't wait for the end of the line.

Both *Read* and *Readln* can be called without any parameters at all, in which case the parentheses must be omitted. When this happens, Turbo simply waits for a carriage return to be input at the terminal. If *Read* is called with a file variable but no other parameters, it skips over the next object in that file and returns to the same line. If *Readln* is called in the same fashion, it skips to the next line in the file.

Write and Writeln

The *Write* and *Writeln* procedures write one or more data objects to a file. Their syntax is as follows:

```
Write({FileVariable,} {, Var1, Var2...VarN});
Writeln({FileVariable,} {, Var1, Var2...VarN});
```

The optional parameter *FileVariable* again specifies the file the procedure is to work with. If no file is specified, the procedure acts as if the standard file variable *Output* had been used, and the output goes to the screen. The *Write* procedure writes the objects listed to the file. *Writeln* does the same, but follows the information written with a carriage return.

When *Write* is called with no parameters, or with only a file variable as a parameter, nothing happens. However, when *Writeln* is called with no parameters, a new line is started on the screen. When *Writeln* is called with only a file variable as a parameter, a new line is started in that file (that is, a carriage return and line feed are output to that file). As with *Read* and *Readln*, when a procedure is called without parameters, the parentheses must be omitted.

Write Parameters

We introduced write parameters, and some examples of them, near the end of Chapter 14. Write parameters are a special kind of parameter used by the *Write* and *Writeln* procedures (and also by the *Str* procedure) to specify what a value will look like when it is output to the screen or a text file (or, in the case of *Str*, when a number is converted to a string). In general, a write parameter consists of a value of the type char, a string type, the type boolean, the type integer, or the type real, followed by one or more colons and integer-valued expressions. The integer expressions give the procedure information on how many columns to use to format the parameter when it is printed and, in the case of real numbers, the number of digits that are to appear below the decimal point.

Table 18-1 (on page 208) provides a complete list of the different kinds of *Write* parameters and what is output for each.

In the following examples, we'll show how field widths and right justification work. First, we'll write an integer using three different field widths: one a bit too small, one about right, and one too large. When the field width is too small to contain the entire number, the whole number is written anyway. When the field width is greater than the number of spaces needed to write the number, the number is right-justified (the number is flush against the right-hand side of the field) and blanks are added to the left. Thus, the statements

```
Int := 16421;
WriteLn('This field is too narrow:',Int:3);
WriteLn('This field is just right:',Int:6);
WriteLn('This field is too wide  :',Int:15);
```

would produce

```
This field is too narrow:16421
This field is just right: 16421
This field is too wide  :          16421
```

Table 18-1 Write Parameters and Output

Parameter	Function
Ch (char)	The character Ch is output.
Ch:n (char:integer)	The character Ch is output, right-adjusted in a field *n* characters wide (that is, Ch is preceded by *n* − 1 blanks.)
St (string)	The string St is output.
St:n (string:integer)	The string St is output, right-adjusted in a field *n* characters wide (that is, St is preceded by *n* − Length(St) blanks.)
B (boolean)	The word TRUE or the word FALSE is output, depending on the value of *B*. (Note that this is the only enumerated scalar type for which *Write* and *Writeln* produce a string automatically.)
B:n (boolean:integer)	The word TRUE or the word FALSE is output, depending on the value of *B*. The word is right-adjusted in a field *n* characters wide.
I (integer)	The decimal representation of *I* is output.
I:n (integer:integer)	The decimal representation of *I* is output, right-adjusted in a field *n* characters wide. If *n* characters are not large enough to hold the decimal representation, the additional digits are written anyway.
R (real)	*R* is output in exponential notation, in a field 18 characters wide. The format is: `_S#.##########E*##` where: _ is a space, S is a minus sign or a space, # is a digit, and * is a plus or minus sign.
R:n (real:integer)	*R* is output in exponential notation. If *n* is greater than or equal to 18, then the previous format is used, right-justified in a field *n* characters wide. If *n* is between 8 and 18, digits below the decimal point are omitted to make the number fit into a field *n* characters wide. If *n* is 8 or less, then *R* is printed with a minimum of one character below the decimal point, regardless of the value of *n*.

Table 18-1 Write Parameters and Output, Continued

Parameter	Function
R:n:m (real:integer:integer)	*R* is output in fixed-point decimal notation (that is, as digits separated by a decimal point), right-adjusted in a field *n* characters wide. The *m* digits are output after the decimal point. If *m* is 0, no digits below the decimal point are output, and the decimal point itself is omitted. The number is rounded correctly for the number of digits output. If *m* is greater than 24, it is ignored.

The following code demonstrates Turbo's special extension for boolean values, allowing them to be written as the strings TRUE and FALSE. The statements

```
Flag1 := True; Flag2 := False;
Writeln('Flag1:  ', Flag1:5, '  Flag2:  ' ,Flag2:5);
```

produce the output

```
Flag1:   TRUE  Flag2:  FALSE
```

(Note, however, that you cannot read in TRUE and FALSE as boolean values. Instead, you have to read them in as strings and convert them appropriately.)

Finally, here's a program that shows the different ways a real number can be formatted.

```
program FormatDemo;
begin
  WriteLn(Pi);                { The predefined constant }
  WriteLn(Pi:8);              { Pi = 3.1415926535       }
  WriteLn( Pi:8);
  WriteLn(Pi:12);
  WriteLn(Pi:16);
  WriteLn(Pi:20);
  WriteLn(Pi:8:0);
  WriteLn(Pi:8:4);
  WriteLn(Pi:12:10)
end.
```

When run, this program will write

```
   3.1415926535E+00
 3.14E+00
  3.1E+00
 3.141593E+00
 3.1415926535E+00
     3.1415926535E+00
       3
   3.1416
 3.1415926535
```

Get and Put

In Standard Pascal, the two procedures *Get* and *Put* are also used to perform operations on files. Because Turbo Pascal's methods of performing I/O differ from those used on the mainframes where Pascal was developed, these procedures are not available in Turbo.

FILE TYPES

Files, like all other data objects in a Pascal program, have types; and file types are a special class of types designed to represent files. File types, like arrays, consist of elements of a single type. For instance, if you had a file on disk that consisted of a series of characters, you could refer to it in your program as follows:

```
var
  MyDiskFile : file of char;
```

and then use Turbo's built-in procedures to associate this variable within your program with a file on the disk. Files are unlike arrays in that their size is not specified in the type definition. However, like arrays, files can have as their *component* type, or *record* type, any Pascal type at all. There is only one exception: The component type of a file type may *not* be another file type. Here are some examples of file-type definitions:

```
type
  CheckFile = file of Check;
    { File of objects of the type Check }
  SetFile = file of set of char;
    { File of objects of the anonymous type "set of char" }
  ScreenFile = file of array[1..25] of string[80];
    { File in which each record consists of 25 strings of
      max length 80  (possibly used to hold copies of a
      CRT screen) }
```

Because text files, or files of characters, are commonly used, Pascal provides a predefined identifier, *Text*, for the type **file of** char. (In CP/M-80 and CP/M-86 Turbo Pascal, *Text* and **file of** char actually mean two different things; for details on file formats, see the *Turbo Pascal Reference Manual*. In all cases, however, the file type *Text* is compatible with text files written by the operating system and non-Turbo programs.) The syntax of a file type is similar to that of an array type or a set type (see Figure 18-1).

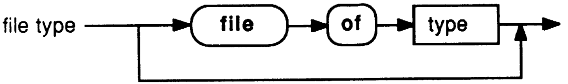


Figure 18-1 Syntax Diagram of File Type

READING AND WRITING TEXT FILES

A text file is a file that consists of ASCII characters, and is usually designed to hold readable information. As just mentioned, Turbo Pascal provides a predefined file type, *Text*, for this kind of file.

The characters within a text file are divided into *lines*, which are sequences of consecutive characters of any length. The end of each line is marked by the special ASCII characters for carriage return (^M or ASCII code 13) and a line feed (^L, or ASCII code 10). Traditionally, the end of a text file has been marked by a ^Z character (ASCII 26), and many older programs (such as word processors) still require it. However, newer versions of MS-DOS keep track of a file's size and do not need a ^Z as an end-of-file character. Despite this, if you are creating or modifying text files that will be used by other programs, you should probably place a ^Z at the end of the text file:

```
{open the file}
Write(Textfile, ^Z);    {place a ^Z at file's end}
Close(Textfile);                          {all done}
```

(Note that Turbo will handle text files correctly whether a ^Z is present or not.)

When you perform operations on text files, however, you don't need to remember all of these special characters, because Turbo Pascal recognizes and handles them for you. You can read or write text files using the four procedures already mentioned: *Read*, *Write*, *Readln*, and *Writeln*. There are also other special functions in Turbo Pascal to handle text files.

Here's a sample program that writes a few lines of text to a text file, then reads it back. In this example, we'll introduce Turbo's built-in subprograms that assign a name to a file, "open" it (prepare it for use by the program), return information on it, and "close" it (tell Turbo and the operating system that we are finished using it).

```
program FileTest;
var
  MyFile : Text;   { The text file we'll use }
  Line   : string[255];
           { A string to hold a line we read from the file }
  LineCounter : integer;  { A counter for the lines we read }
  I : integer;  { A counter for writing numbers to the file }
begin { program FileTest }

  Assign(MyFile, 'MYFILE.TXT');
{            ↑           ↑
             |           |
   A file variable       |
       A Pascal string giving the name of a file on the disk

    The Assign procedure associates a file variable inside
    your program with a file on the disk in the outside
    world. The names can be completely different; the name of
```

```
    the disk file, however, must be legal for the
    operating system you are using. }


  Rewrite(MyFile);
{            ↑

    A file variable

    The Rewrite procedure "opens" a file that you intend to
    write to. If there was already a fle on the disk having
    the name given in the Assign statement, it is destroyed.
}

  Writeln(MyFile, 'Hello, World!');
    { Write some text to the file }
  Writeln(MyFile, 'This is my first file!');
for I := 1 to 10 do  { Write some integers to the file }
  Writeln(MyFile, i:2);
Writeln(MyFile, '-- End of file --');
{ Write one more line of text }


  Reset(MyFile);
 {         ↑

    A file variable

    The Reset procedure "opens" a file that you intend to
    read from. Note that since the name associated with
    MyFile is still the same, we don't need to call Assign
    again. If the file was already open, Reset closes it
    and reopens it for reading. }

  LineCounter := 0;      { We'll count the lines and display a
                                line number with each one. }
  while not Eof(MyFile) do
                 { The built in function Eof returns True }
                 { if the end of a file has been reached. }
  begin
    LineCounter := LineCounter + 1;   { Count the next line }
    Readln(MyFile, Line);  { Read it into the variable Line }
    Writeln('Line ', LineCounter:2, '--> ', Line);
     { Write it to the screen    }
  end;


  Close(MyFile);
 {         ↑

    A file variable

    The Close procedure "closes" a file after you have fin-
    ished using it. When you close a text file you have just
    written, Close will put a ^Z (end of file mark) on the
    end of the file. You must always close your files after
```

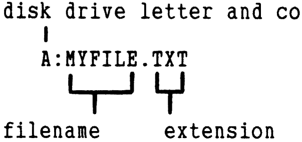
```
    using them, or other programs may not be able to read or
    write them.    }
end.
```

The Assign Procedure

The *Assign* procedure "connects" a file variable *inside* your program with a file on the disk *outside* your program. *Assign* must be called with a file variable before the file variable is used, unless it is one of Turbo's predefined files (which we will list shortly). The first parameter of *Assign* is the file variable, the second is a string giving the file name:

```
Assign(FileVariable, St);
```

The exact format of the file name St depends on your operating system. In CP/M and CP/M-86, the file name is an optional disk drive letter (followed by a colon if present), a name with up to 8 characters, an optional period, and an optional extension (up to 3 more characters):



Usually, the extension gives information on what kind of data is in the file. Here, the extension .TXT hints that there is probably text in the file. Case style is ignored in file names.

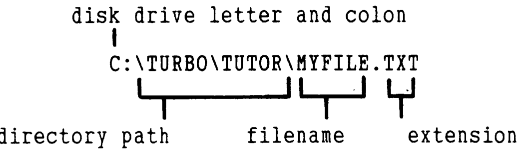
Here are some more examples of acceptable file names:

THISFILE
thx1138
Simple.pas
simple.exe
STARS.dat

If a file name has no drive number preceding it, Turbo Pascal will look for it (or create it) on the *current* drive (the operating system prompt often tells you which drive that is; if the prompt is X〉, then *X* is the logged drive). You can explicitly state which drive the file is on by appending the drive name in front of the file name, like so:

```
a:THISFILE
B:thx1138
c:STARS.dat
```

In PC-DOS and MS-DOS, file names have the same format, but may have a *directory path* in them as well; a file name without one is looked for on the *current directory* of the drive specified (or the current drive). A directory path gives information about the directory in which the file is located. The following is how an MS-DOS file name with a directory path looks:



Here are more examples of file names with directory paths.

```
a:\mydata\THISFILE
\films\lucas\thx1138
C:\ASTRO\GAMES\TREK\STARS.dat
```

(See your operating system manual to find out the rules regarding specification of file names.)

Reset and Rewrite

After you have used *Assign* to associate a file name with a file variable, you can use the procedures *Reset* and *Rewrite* to open the file (prepare it for reading or writing, respectively). When accessing a text file, you may only do one or the other (read or write) each time the file is opened; with other types of files, you may do both. In this case, you would want to use *Reset* to open an existing file and *Rewrite* to create a new one or overwrite an existing file.

```
Reset(FileVariable);
Rewrite(FileVariable);
```

If the file variable passed to *Reset* or *Rewrite* refers to a currently open file, the file is closed and then the *Reset* or *Rewrite* operation is performed. If *Rewrite* is called on an existing file, that file is erased and replaced with the new information you write.

Read and Readln with Text Files

It is convenient to think of files as having an associated *file pointer*, an imaginary pointer that points to the next object in a file. When a file is first written, that pointer always points just past the end of the data in the file. When the file is being read, it points to the next object to be read (if any).

When *Read* is called with a text file and a variable as its arguments, it reads the next object (if any) on the current line of the file into the variable, and advances the file pointer past that object. However, when the data on that line is exhausted, *Read* does not advance the file pointer to the next line. Attempting to use *Read* at the end of a line in a text file has no effect at all; you must call the procedure *Readln* to advance to the next line (if it exists).

The procedure *Readln* moves the file pointer to the next line immediately after it reads the information it requires from the current line; any other information on the line is ignored.

Eof, Eoln, SeekEof, and SeekEoln

The *Eof* and *Eoln* functions are used to find out whether the program has read up to the end of a file, or up to the end of a line. Both return boolean values.

```
Eof(FileVariable);
Eoln(TextFileVariable);
SeekEof(TextFileVariable);
SeekEoln(TextFileVariable);
```

The *Eof* function works on all kinds of files, including text files, and is TRUE when the file pointer is at the end of the file. *Eoln*, which indicates whether the file pointer is at the end of a line in a text file, has no meaning for non-text files. When *Eof* is TRUE for a text file, *Eoln* is also TRUE for that file.

SeekEof and *SeekEoln* have the same syntax as *Eof* and *Eoln*, but only have meaning for text files. When used on text files, they skip over spaces and tabs before they test for the end of the file or line. These functions are useful when you don't know the number of objects on a line or in a file. By calling these functions, you can avoid getting the *Read* procedure "stuck" at the end of a line or a file and find out whether there is another object available or not. You can then call *Readln* or finish processing the file, if necessary.

All of these procedures can be called without parameters; if so, the file variable is taken to be the standard file Input. However, before using these functions on standard Input, you should specify a {$B-} compiler directive to disable Turbo's buffering of Input.

The Close Procedure

The *Close* procedure tells the operating system that you are finished using a file. Whenever you are done with a file, especially if you are writing to it, it is vitally important to call *Close*. Failing to do so may cause the operating system to prevent other programs or other users from accessing your file. Worse yet, in the case of a file you are creating or extending, the operating system may not make a record of the size of the file. (In PC-DOS or MS-DOS, such an action can leave the recorded file size at 0.) The moral: Always close your files when you reach a point in your program where you might exit.

RANDOM ACCESS FILES

So far, we've discussed text files and the file type *Text*; however, there is an infinite variety of file types you can use to store your data.

In text files, all data is stored in an easily readable form. Numbers are represented as ASCII digits, boolean values are represented as TRUE

and FALSE, and strings are represented as strings. However, translating every non-character value to and from ASCII text takes time, which may not be time well spent if only a program (rather than a user) needs to read the file.

Text files have another restriction: Because the lines of a text file are of varying length, the only way to find the next line is to read through the previous one to the end. Thus, text files are inherently sequential—the objects in the file must be read and written in order.

However, other types of files consist of objects of a fixed length (like arrays). In these files, the objects (or records) do not have to be read in sequence; the position of any object in the file can be calculated by multiplying the size of an object by the number of objects that come before it. Because records within it can be accessed in any order, this kind of file is called a *random access* file.

In a random access file, data is stored exactly as it appears in the computer's memory; no translation is necessary. This saves processing time both when the file is written and when it is read; it also (usually) saves space.

Creating a Random Access File

Here's an example of how to use a random access file in a program. Suppose, once again, that you are writing a program to maintain your checkbook, and you want it to handle any number of checks (more than your computer's RAM could keep track of at one time). To store the data, you could use a random access file in which each record contains the data for a check, so that the number of checks you could record would be limited only by the size of your disk. To do this, you'd first define the record type for a check, then create a file with that type as its component type:

```
type
  CheckNumType =  1..10000;
  MonthType    = (January, February, March, April, May,
                  June, July, August, September, October,
                  November, December);
  DayType      = 1..31;
  YearType     = 1980..2000;
  PayeeType    = string[40];

  Check = record
    CheckNum : CheckNumType;
    Amt    : real;
    Month  : MonthType;
    Day    : DayType;
    Year   : YearType;
    Payee  : PayeeType;
end;
```

```
var
  CheckFile  : file of Check; { Keep check info in a file }
  ThisCheck  : Check;     { A variable to hold a record read
                            from the file }
```

Starting with these definitions and declarations, you could then write a routine to get the checkbook information from the user and write it into a file . Here's a procedure that would do the job:

```
procedure MakeCheckFile;
var
  CheckNumber : CheckNumType;
  MonthNumber : 1..12;
begin
    { Associate a name with the file }
  Assign(CheckFile, 'CHKBOOK.DAT');
  Rewrite(CheckFile);      { Open it as a new file }
  CheckNumber := 1;        { Start with check #1 }
  with ThisCheck do
  repeat  { Start our input loop here }
    Writeln('Enter information for check #', Num:0, ': ');
      { Note that "Num:0" will always print
        Num with no spaces around it }
    Write('Amount (< 0 to exit): ');  { Get amount of check }
    Readln(Amt);
    if Amt < 0 then  { User is done if amt is negative }
    begin
      Close(CheckFile);     { Always close the file!!! }
      Exit;                 { This is one way to exit from
                              the procedure }
    end;
    Write('Month (1-12): ');
      {Get month by number, then convert}
    Readln(MonthNumber);
    Month := MonthType(MonthNumber - 1);
    { When the name of a scalar type is used as if it were
      a function, and applied to a value of another scalar
      type, the result is a value of the first type with
      the same ordinal value. Here, we convert
      1 --> January (ordinal value 0), 2 --> February, etc.
      This conversion process is called "type conversion." }
    Write('Day (1-31): ');
    Readln(Day);
    Write('Year (1980-2000): ');
    Readln(Year);
    Write('Payee (40 characters max): ');

    Buflen := 40;
    { Buflen is a predeclared variable in Turbo Pascal that
      determines the maximum number of characters that will
      be accepted from the user the next time information is
      input from the terminal. It is reset to the default of
      127 after every read. By using BufLen, we make sure
      that the user cannot type more characters than we can
      handle!  }
```

```
    Readln(Payee);
    Write(CheckFile, ThisCheck);
    { Got all the information, write it out }
  until False;
  { We'll always exit from the middle of the loop }
end;
```

The preceding example points out a number of "tricks of the trade" in programming routines involving files and I/O in general. We'll go through these one at a time, so that you may use them in your own programs. First of all, we used an "endless" loop (that is, a **repeat**... **until** FALSE statement), with an *Exit* statement in the middle to exit the procedure. This structure is useful in I/O routines (and in other types of routines as well) because it ensures that there is only one exit point from the routine (which, in turn, lets us make sure that the file is closed before exiting) and it prevents us from having to use a more awkward structure (such as an **if** statement, plus another test in the **until** at the bottom of the loop).

Another useful trick is the specification of the write parameter *Num:0* to write out the value of the integer Num to the screen. This takes advantage of the fact that whenever the field length specified for an integer is too small, the whole integer is written with no space on either side. Thus, we can fit the number cleanly into the rest of the prompt, regardless of its size.

The next technique we used is called *type conversion*, which is a special extension of Turbo Pascal that makes inputting defined scalar types simple. In the statements

```
Readln(MonthNumber);
Month := MonthType(MonthNumber - 1);
```

we read in the month as a number from 1 to 12, but want to convert it to the type *Month* (January..December). We could have used a long **case** statement to do this:

```
case MonthNumber of
  1: Month := January;
  2: Month := February;
      ...
```

but entering such a statement every time you wanted to input a defined scalar type could become tedious. Instead, Turbo Pascal takes advantage of the fact that all values of scalar types are represented the same way internally: as a number (1 or 2 bytes long) containing the ordinal value. Turbo Pascal therefore lets you convert from one scalar type to the other by simply allowing the same value to be of a different type, and temporarily suspending type-checking. We indicate the type that we want the value to have by using the name of that type as if it were the name of a function; in this case, *MonthType(MonthNumber −1)*. This converts the number *MonthType* − 1 to the object of the type *MonthType* with the same ordinal value, giving us the result we need.

More often, of course, objects of defined scalar types won't lend themselves to being represented by a number to the user. In this case, a menu is helpful so that the user can type in a number, even though that number has no intrinsic relationship with the object he or she picked. For instance, if you wanted a value of the defined scalar type

```
type
  Color = (Black, White, Red, Orange, Yellow, Green, Blue,
           Indigo, Violet);
```

you could display a menu:

```
Select a color:
 0) Black         3) Orange        6) Blue
 1) White         4) Yellow        7) Indigo
 2) Red           5) Green         8) Violet
```

and use type conversion to change the number into a value of the type *Color.*

Finally, the last trick we used was to set the predefined variable *Buflen* to limit the number of characters the user could type when entering data. In the statements

```
Buflen := 40;
Readln(Payee);
```

we limited the user to typing in a string of no more than 40 characters to avoid truncation when the string was assigned to the variable *Payee*. Setting *Buflen* tells Turbo to limit the number of characters typed for one *Read* or *Readln* statement only; immediately after that statement, the limit returns to 127 (the default).

Properties of Random Access Files

As we demonstrated in the previous example, many of the same operations that apply to text files also apply to random access files. For instance, the procedures *Assign*, *Reset*, *Rewrite*, and *Close* work exactly the same on random access files, as does the function *Eof.* The procedures *Readln* and *Writeln*, of course, have no meaning when applied to random access files, since there is no concept of a "line." Also, *Read* and *Write* should only be used with one or more values of the component type.

When a random access file is opened, the file pointer is positioned at the beginning of the file, just as it is in a text file. If successive reads or writes are performed on the file, the file is accessed sequentially (the file pointer advances to the next record after each operation). Unlike text files, however, random access files allow either read or write operations to be performed at any time. Furthermore, by using the *Seek* procedure, it is possible to read or write the components of the file in any order.

The Seek Procedure

The *Seek* procedure allows you to position the file pointer at the beginning of any record in a random access file, so that the next read or write operation is performed on that record.

```
Seek(FileVariable, RecordNumber);
```

The first record of a file is considered to be record 0, and the last is the size of the file, in records, minus 1. Thus, if we use the procedure *MakeCheckFile* to create a file of checks, we could get the information on check *CheckNumber* by writing

```
Seek(CheckFile, CheckNumber - 1);
Read(CheckFile, ThisCheck);
```

(In this example, we assume that there is one record for every check number, and that the variables are all defined and the file is open).

A random access file can be made larger by writing new records at the end of the file, with the file pointer pointing just past the end of the last component. To position the file pointer at this location, you can read through the file until the *Eof* function returns TRUE for that file. Better yet, you can use Turbo's predeclared function *FileSize* (described in more detail later) with the *Seek* procedure to move the file pointer directly to the end. *FileSize* returns a value of type integer giving the number of components in a file; therefore, to append new records to the end of the file, you can perform the call

```
Seek(FileVariable, FileSize(FileVariable));
```

and then write the new information.

In MS-DOS and PC-DOS, it is possible to have a file with more than 32767 components. Thus, the *Seek* procedure, which takes an integer record number, will not suffice to access every component of the file. Therefore, Turbo Pascal provides an additional procedure, *LongSeek*, to deal with this situation.

```
LongSeek(FileVariable, RecordNumber);
```

The *LongSeek* procedure functions the same as *Seek*, except that it takes a real parameter for the record number. This allows access to any record of an MS-DOS file.

The FilePos and LongFilePos Functions

The *FilePos* function, which works only on random access files, returns the number of the component at which the file pointer is currently positioned, as a value of the type integer. The first component, or record, of a file is considered to have the number 0. The file must be open at the time of the call.

```
FilePos(FileVariable);
LongFilePos(FileVariable);
```

The companion function *LongFilePos*, which is only provided in the PC-DOS and MS-DOS versions of Turbo Pascal, is used when a file may contain more than 32767 records. *LongFilePos* returns the position of the file pointer as a real value.

The FileSize and LongFileSize Functions

FileSize and *LongFileSize* return the size of a file as a number of components, not as a number of bytes. For this reason, they work only on random access files. After a *Rewrite(FileVariable)*, *FileSize(FileVariable)* will always return 0. The file must be open at the time of a call to *FileSize*.

```
FileSize(FileVariable);
LongFileSize(FileVariable);
```

Like *LongFilePos*, *LongFileSize* returns a value of the type real so that the larger file sizes of PC-DOS and MS-DOS can be accommodated. *LongFileSize* is not defined in other implementations of Turbo Pascal.

The Append Procedure

In MS-DOS and PC-DOS, a random access file can be opened strictly for appending by making the call

```
Append(FileVariable);
```

This call replaces the *Reset* and *Seek* calls; thus, the file need not be open at the time of the call. Once it is made, the only allowed operation (until the file is closed or *Reset* or *Rewrite* is called) is appending new components to the file.

The Truncate Procedure

In MS-DOS and PC-DOS, the *Truncate* procedure will truncate a random access file at the current file-pointer position; that is, components beyond the file pointer are cut away.

```
Truncate(FileVariable);
```

The *Truncate* procedure works only on random access files that are already open. After a call to *Truncate*, the file is left open, and the file pointer is positioned at the new end of the file.

OTHER BUILT-IN FILE PROCEDURES AND FUNCTIONS

The following procedures and functions are provided by Turbo Pascal to help your program manage and manipulate files of different kinds.

Erase. With the *Erase* procedure, you can erase files from the disk by calling

```
Erase(FileVariable);
```

The file variable must be associated with a file name by a call to *Assign*, but must not be open when *Erase* is called.

Rename. This procedure allows you to rename a file on the disk; you can do so by calling

```
Rename(FileVariable, Str);
```

where *Str* is a string containing the new name. The name in *Str* must be a legal file name for your operating system, and *FileVariable* must be associated with a file name by a call to *Assign*. The file must not be open when it is renamed.

Flush. In some cases (CP/M and CP/M-86 random access files, MS-DOS and PC-DOS text files), Turbo Pascal waits until it has a large chunk of information to write to a file (or until the file is closed) before it actually performs a requested write operation. At times, however, you will want to make sure that all the information has actually been written to the file by calling

```
Flush(FileVariable);
```

Refer to the *Turbo Pascal Reference Manual* for information on the effects of this procedure on different file types and operating systems.

FOR MS-DOS AND PC-DOS ONLY: DIRECTORY MANAGEMENT PROCEDURES

The following procedures are provided by Turbo Pascal to manipulate directories in PC-DOS and MS-DOS. With them, you can change or find out the default directory, or create and destroy directories.

ChDir. The *ChDir* procedure operates the same way as the ChDir or CD (Change Directory) command in MS-DOS/PC-DOS. The call

```
ChDir(St);
```

changes the current directory to the name given in the string *St*, provided that it is a legal directory name. The logged drive is also changed if *St* specifies a drive name.

MkDir/RmDir. The *MkDir* and *RmDir* procedures perform the same functions as the MkDir/MD (Make Directory) and RmDir/RD (Remove Directory) commands in MS-DOS/PC-DOS:

```
MkDir(St);
RmDir(St);
```

The string parameter contains the name of a directory to create or destroy. Note that DOS will not permit directories with files in them to be removed.

GetDir. The *GetDir* procedure is used to determine the currently logged drive and/or the currently active directory on that drive. In the call

```
GetDir(Drive,St);
```

the parameter *Drive* must be set to an integer value indicating a disk drive, where 0 represents the currently logged drive, 1 represents drive *A*, and so forth. *GetDir* returns the drive name, followed by the active directory in *St*.

TALKING TO YOUR COMPUTER'S PERIPHERALS: DEVICE I/O

When you use disk files, you are communicating with one of your computer's peripherals, the disk drive. In this section, we'll explore in-depth how you can exchange information with the other peripheral devices attached to your computer: the keyboard, the screen, a printer, or a serial port.

Logical Devices

In Turbo Pascal, you are provided with a number of *logical devices*, special file names that are used to talk to the peripherals of your computer. To use them, use the *Assign* procedure exactly as if you were opening a file, and then perform input or output to them as appropriate. Note that since you can't erase a device, *Reset* and *Rewrite* perform the same functions on a logical device, *Close* does nothing, and these files cannot, of course, be erased.

Here is a list of the logical devices Turbo Pascal recognizes:

CON: Console I/O (that is, read from the keyboard and write to the screen). Echoes input, allows correction with backspace. Expands tabs on output. Echoes CR as CR/LF for both input and output.

KBD: Keyboard input with no echo or interpretation.

TRM: Console output with no interpretation.

LST: The printer. Can only be used for output. No interpretation.

AUX: Input and output device, usually an RS-232 port. Corresponds to PUN: and RDR: in CP/M, and COM1: in MS-DOS or PC-DOS.

USR: User I/O device. Advanced programmers can write their own I/O drivers for specific devices.

In practice, however, you will rarely (if ever) need to use these logical devices directly. To save you having to create file variables and use *Assign* with these logical devices, Turbo Pascal also provides a set of standard files, predefined file variables that are already associated with each of these devices.

Standard Files

Turbo Pascal's standard files allow you to treat each of the previously defined devices as a predefined text file (which you don't need to *Assign*, *Reset*, *Rewrite*, or *Close*). In fact, the use of any of these procedures on a standard file is illegal. Here is a list of Turbo's standard files and the logical devices to which they correspond.

Input This is the primary input file used for all calls to *Read* and *Readln* that do not specify a file (or that specify Input explicitly). It is normally equivalent to the CON: device.

Output This is the primary output file used for all calls to *Write* and *Writeln* that do not specify a file (or that specify Output explicitly).

Con The Console Device (CON:).

Trm The Terminal Device (TRM:).

Kbd The Keyboard Device (KBD:). This standard file is most often used to input single characters from the keyboard without echoing them to the screen.

Lst The List Device (LST:).

Aux The Auxiliary Device (AUX:), sometimes used to control a serial port.

Usr The User Device (USR:). This device can be used with special code written to direct output anywhere. (See the *Turbo Pascal Reference Manual* for details.)

Advanced Keyboard Handling: KeyPressed and the Standard File Kbd

In certain applications (such as games), you may want the program to continue to run while it is waiting for user input. For instance, in a Space Invaders game, you'll want the invaders to continue to advance while the program tests to see if a key has been pressed.

So far, all the input methods we have described are of the *blocking* type; that is, the program is "blocked" from doing any other tasks while it is

waiting for the required input to come in. Thus, when you execute the statement

```
Read(Kbd, Ch);
```

where *Ch* is assumed to be of the type char), all activity will cease until the user types a character. How can you avoid this? Well, one way might be to avoid making the call to *Read* until you know that there is a character waiting to be read. Then, the character will be read in immediately, with no waiting. For this reason, Turbo Pascal provides the function *KeyPressed.*

The built-in boolean function *KeyPressed* returns TRUE only if there is a character immediately waiting to be read from the keyboard. The following routine, *CheckCommand,* is an example of how to use it. *CheckCommand* might be called periodically from a game program to see if a key has been pressed, and if so, act on it.

```
{$C-}
procedure CheckCommand;
var
  Cmd : char;
begin
  if KeyPressed then begin
    Read(Kbd,Cmd);        { read key w/out echo }
    UpCase(Cmd);           { force to upper case }
    case Cmd of
      ...                  { handle commands }
    else
      Write(Chr(7));       { beep at illegal cmd }
    end  { case }
  end  { ifs }
end;  { procedure CheckCommand }
```

UNTYPED FILES

So far, we've discussed two kinds of files: text files, which consist of ASCII characters and give special meaning to some of those characters, and random access files, which consist of many components of the same type that can be accessed in any order.

There are, however, some situations in which neither file-handling method is suited to the job at hand. In the CP/M and CP/M-86 versions of Turbo, some of the bytes in a random access file created by Turbo have special meanings: The first 2 bytes of the file contain the number of records in the file and the second 2 bytes contain the record length. Thus, files not created by Turbo cannot be opened as random access files by the CP/M and CP/M-86 versions of Turbo Pascal. (The PC-DOS and MS-DOS versions of Turbo Pascal do not have this restriction; they can open any file as a random access file). Another problem arises where a file has records of many different

lengths mixed together but is not a text file. (Remember, Pascal random access files have fixed-length records).

An *untyped file* allows you to avoid all of these restrictions by working with files as blocks of data that can be interpreted any way desired. In CP/M and CP/M-86 versions of Turbo Pascal, these blocks are 128 bytes long; in MS-DOS and PC-DOS versions of Turbo, they may be of any length up to 32 Kbytes, but are 128 bytes long unless a different length is specified.

Declaring Untyped Files

To declare an untyped file variable, you use a declaration of the form

```
var
  BigFile : file;
```

The absence of a component or record type indicates to Turbo Pascal that this is an untyped file.

Using Untyped Files

The *Assign*, *Reset*, *Reset*, *Rewrite*, and *Close* procedures will work on an untyped file the same way they work on random access files. The difference, however, lies in the way these files are read and written.

Untyped files are always read or written one or more whole blocks at a time. The two procedures used to do this are

```
BlockRead(FileVariable,Buffer,NumBlocks);
BlockWrite(FileVariable,Buffer,NumBlocks);
```

in which *FileVariable* is an untyped file variable, *Buffer* is any variable of any type, and *NumBlocks* is the number of blocks to be written from (or read into) *Buffer*.

These routines do no checking whatsoever to make sure that *Buffer* is large enough to contain the number of blocks of data written or read; this is the responsibility of the programmer. If *Buffer* is too small, then a *BlockRead* will copy the requested blocks to memory, overwriting whatever code and/or variables follow *Buffer*. A *BlockWrite* is not necessarily as disastrous in such a case, you merely copy extra bytes out to the file. However, this may cause trouble later if your program attempts to interpret the extra bytes as valid data.

The standard procedures *Seek* and *LongSeek* can be used with untyped files to provide random access. For untyped files, these procedures act as if each block were a record of the file. The standard function *Eof* also works on untyped files.

As with random access files, if you are opening an existing untyped file to update it, you must use the *Reset* procedure. A call to *Rewrite* erases any existing file and creates a new one.

Untyped files are the fastest way to read and write disk files. One way to maximize this speed is to tell *BlockRead* or *BlockWrite* to transfer many blocks at once, perhaps an entire file, in one big gulp. To let you know exactly how many blocks were transferred during such an operation, Turbo allows you to call *BlockRead* and *BlockWrite* with an additional parameter that supplies this information, as shown in the following:

```
BlockRead(FileVariable,Buffer,NumBlocks,Result);
BlockWrite(FileVariable,Buffer,NumBlocks,Result);
```

The integer parameter *Result*, if present, is returned with the number of blocks written or read.

MS-DOS/PC-DOS Only: Specifying a Block Size

In the PC-DOS and MS-DOS versions of Turbo Pascal, you can specify the block size of an untyped file by using a special optional parameter in the *Reset* or *Rewrite* procedure:

```
Reset(FileVariable, BlockSize);
Rewrite(FileVariable, BlockSize);
```

where *BlockSize* is an integer expression giving the block length to use.

I/O ERROR HANDLING

Suppose while in your program you tried to call *Reset* on a nonexistent file. What would happen? As soon as the program tried to open that file for input, the program would stop running and an error message would be written to the screen. While this is okay for small or experimental programs, in a large program it can cause large amounts of data stored in memory to be lost. For this reason, Turbo Pascal provides you with the option of disabling these error messages and handling the error condition within your program.

To turn off I/O error checking, you can insert the compiler directive {$I-} into your program. This will disable I/O error checking (but not other kinds of error checking) until you re-enable it using {$I+}. For example, suppose your program asks the user for the name of a file to read. There's always a chance that the user could mistype the file name, or that the disk with that file might not be in the drive. To keep the program from stopping immediately as a result of such an error, you could write:

```
{$I-}
Write('Enter input file:  ');
ReadLn(InFileName);
Assign(InFile,InFileName);
Reset(InFile);
{$I+}
```

Of course, this alone doesn't solve the problem. The program won't stop here, but it might when you attempt to read from the file. In any case, it would be desirable to have your program detect the error and take action at once.

To detect an error that occurs with I/O error-checking turned off, Turbo Pascal provides the built-in integer function *IOResult*. When *IOResult* is called, it returns a code that indicates the result of the last I/O operation. If the operation is successful, it returns the value 0; otherwise, it returns a value indicating what the problem was. If we were to rewrite the previous example using *IOResult*, we might come up with something like this:

```
{$I-}
repeat
  repeat
    Write('Enter input file:  ');
    ReadLn(InFileName);
    Assign(InFile,InFileName);
  until IOResult = 0;
  Reset(InFile)
until IOresult = 0;
{$I+}
```

Why are there two **repeat...until** loops in this example? Because both the *Assign* and *Reset* procedures can cause an I/O error, and we want to repeat the process of prompting for the file name and attempting to open it if *any* error occurs. It is also important to note that the function *IOResult* only returns a nonzero code once for each error that occurs. Successive calls to *IOResult* will return 0 until another error occurs.

Another caution is also in order when using {$I-}/{$I+} and *IOResult*. If an error does occur, your program must call *IOResult* before attempting additional I/O. If you continue to do I/O on a file when *IOResult* has returned a nonzero result for that file, unpredictable results may occur.

Here is a program fragment that includes some routines to check for and report I/O errors. You may wish to use this code, or something similar to it, in your own programs.

```
program MyProgram;
...
var
  IOErr : boolean;
...
```

```
type
  Prompt = string[80];
...
procedure Error(Msg : Prompt);
{ Write error Msg out on line 24 and wait for a key }
var
  Ch : char;
begin
  GoToXY(1,24); ClrEol;
  Write(^G,Msg, 'Hit any key to continue')
  Read(Kbd,Ch)
end;  { procedure Error }

procedure IOCheck;
   { Check for I/O error; print message if needed }
var
  IOCode : integer;
begin
  IOCode := IOResult;
  IOErr := (IOCode <> 0);
  if IOErr then begin
    case IOCode of
      $01 : Error('File does not exist');
      $02 : Error('File not open for input');
      $03 : Error('File not open for output');
      $04 : Error('File not open');
      $10 : Error('Error in numeric format');
      $20 : Error('Operation not allowed on logical device');
      $21 : Error('Not allowed in direct mode');
      $22 : Error('Assign to standard files not allowed');
      $90 : Error('Record length mismatch');
      $91 : Error('Seek beyond end of file');
      $99 : Error('Unexpected end of file');
      $F0 : Error('Disk write error');
      $F1 : Error('Directory is full');
      $F2 : Error('File size overflow');
      $F3 : Error('Too many open files');
      $FF : Error('File disappeared');
    else
      Error('Unknown I/O error:  ');
      Write(IOCode:3)
    end  { case }
   end
 end;  { procedure IOCheck }
      ...
```

The procedure *IOCheck*, which is designed to be called after any I/O operation, does a number of useful things. First, it calls *IOResult* and recognizes any error. Second, it sets the global flag IOErr, so that other parts of the program will know whether or not there has been an error and can act accordingly. Third, it prints out an error message on line 24, pausing until the user hits any key (hence *Read(Kbd,Ch)*, which will read a single character without echoing it back to the screen). Finally, it uses the **else** clause of the **case** statement to handle any undefined I/O errors.

The *Turbo Pascal Reference Manual* recommends that *IOResult* be consulted after all of the following procedure calls:

Append	Close	MkDir	RmDir
Assign	Erase	Read	Seek
BlockRead	Execute	Readln	Write
BlockWrite	Flush	Rename	Writeln
Chain	GetDir	Reset	
ChDir	LongSeek	Rewrite	

The {$I-} compiler directive, along with *IOResult*, can be used in many places to develop "bulletproof" programs, programs that will not fail when given improper or inconsistent input.

REVIEW

In this chapter, we introduced the concept of a file, a mechanism through which a Pascal program can communicate with the outside world. We discussed procedures, functions, and techniques for handling three different categories of files: text files, random access files, and untyped files. We explained output formatting, and showed how to use write parameters to control the way numbers, boolean variables, and strings are output. Finally, we discussed I/O error checking and how to handle I/O errors within a Turbo Pascal program.

Chapter 19 will provide a program example to tie together many of the concepts discussed in the preceding chapters.

19 A Sample Program

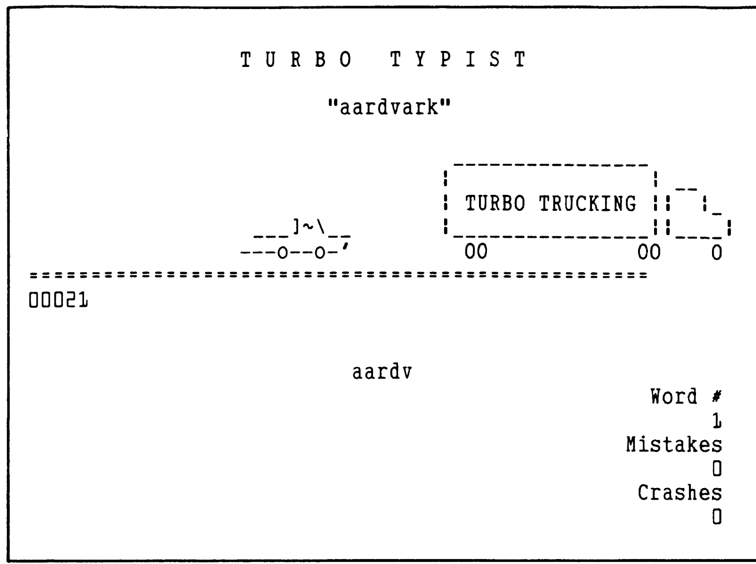
We've spent Part II of this manual filling your head with the fundamentals of Pascal programming. Since this is the last chapter of this section, we'd like to use all of the previously discussed concepts to walk you through a program that combines as many data structures and Pascal statements as can be naturally (comfortably) crammed into 600 lines.

TURBO TYPIST

The Turbo Typist program is a typing game that will work on any machine running Turbo Pascal 3.0. Typist will read a word (or phrase) from a text file, display the word and then wait for the user to type it correctly before retrieving the next word from the file. To make the game more fun, we've added a primitive scoring system and—the most difficult aspect of the program—some real-time animation. There are three major components of our program:

- File I/O to get words from a disk file
- Animation to draw and refresh the cartoon
- Real-time keyboard handling to get characters from the keyboard while simultaneously updating the entire screen (drawing the cartoon, displaying the word to be typed, displaying the characters typed so far by the user, and updating the scoring information).

If you're seated at your computer while reading this, try running the program (TYPIST.PAS). If you're far from the nearest computer, here's what the main screen looks like:



The goal of the playing screen is to type the word "aardvark." The car in the middle of the screen drives forward until it rear ends the truck or until the target word is typed correctly. The car has logged 21 "miles" this round, and the user has already typed the characters "aardv." When the user thinks the target word ("aardvark") has been typed correctly, s/he will press ↵ and the score will be updated accordingly (either a new word will display or the mistake count will be incremented).

In order to manage each of these tasks, Typist makes use of the following data structures and statements:

Data Structures/Types	**Pascal Features**
Sets (of characters)	Procedures/functions
Strings	case, if then else
Arrays (of strings)	Loops (for, while, repeat)
Records	File I/O
Booleans (both functions and variables)	Keyboard and video I/O
Enumerated scalars	Typed constants
Files (for reading text)	

The source code for this program is on your Tutor disk in the file named TYPIST.PAS. Rather than include the entire source code in this chapter, we will introduce the overall structure of the program and then focus on some specific areas.

STEP 1: THE PROGRAM BODY

The main body of the program allows the user to open text files and play the game. Here is a step-by-step outline of its logic (often called pseudocode):

1. Assume that the data file is called TYPING.DTA unless a command-line parameter has been specified.
2. Open the test file.
3. If the file is not empty, play a round; otherwise, display an error message.
4. Repeat steps 2 through 4 for as long as the user wants to play.

STEP 2: FILE I/O

The disk-file handling in Typist is quite straightforward. The task is simply to open a text file and read and display each word until the word list has been exhausted. The program can read words from any ASCII file. Procedure *InitProgram* is used to paint the welcoming screen and retrieve the name of the text file that contains the vocabulary. Once a file is successfully opened, control is returned to the program body. A boolean function, *GetWord*, retrieves the next word from the text file. It returns a TRUE value if a word is retrieved; otherwise, it returns a value of FALSE.

STEP 3: THE MAIN CONTROL LOOP

A procedure named *PlayOneRound* contains the most important control structure of the program. Here is the pseudocode for *PlayOneRound*:

1. Get a word from the text file and display it.
2. Erase the car, increment its x coordinate, and increment and display the odometer.
3. If the car collides with the truck, then:
   a. simulate a collision and back up the car to mile 1.

   Otherwise:

   a. display the car and read characters from the keyboard.
4. If the user presses Esc or Ctrl C, exit the program.
5. If the user presses ↵, process the word.

a. If the word typed matches the target word:

   1) If more words remain in the file, then get the next word, display it, update the scoring information, and back up the car.

   Otherwise, if the word typed does not match the target word:

   1) update and display the score.

6. Repeat steps 2 through 5 until all the words have been retrieved, displayed, and typed.

ANIMATION

What does real-time animation mean? In this case, Typist reads words from a file, displays them, updates the score, and so on, while simultaneously reading characters typed on the keyboard and acting upon them. In this sense, the program manages several processes concurrently and each process flows in a realistic, believable manner.

To accomplish this, a few tricks are used.

- At several places in the main loop (in *PlayOneRound*), a *GoToXY* statement is used to place the cursor near the characters the user has already typed. In this way, the cursor is used to draw attention to the user's goal—that of typing more characters. If the cursor were not explicitly moved several times in the loop, the cursor would be left haphazardly wherever the last screen write had occurred.
- The animation is minimal—14 characters are alternately drawn, erased, then re-drawn, a little to the left or right. Running this simple "animation" program will help you to understand the discussion that follows:

```
program SimpleCartoon;
{ Simple animation program.  Run it a few times and
  understand it. Then try making the object string longer.
  Try incrementing the row as well as the column.  Play
  with the delay interval and try turning the Ctrl-C
  compiler switch off:  (*$C-*).  You might even try
  to convert this program to simulate a bouncing ball.  }
var
  column : integer;                          { loop variable }
  Object : string[10];                 { object being animated }
begin
  ClrScr;
  Object := 'o';                       { start simple, try this
                                         string next: o/o }
  for column := 1 to 80 do
  begin
```

```
    GoToXY(column, 1);           { current column, first row }
    Write(Object);                        { display the object }
    Delay(10); { pause to give your eye a sporting chance }
    GoToXY(column, 1);
                      { place cursor back on top of object }
    Write(' ':Length(Object));    { erase previous object }
  end; { for }
end.
```

- The string editing routine is unusual. Most string editing routines exit when the user types a terminating character (for example, ⏎, Esc, Ctrl Z). Typist's *GetString* function, however, exits for two other reasons as well: 1) the user has used up his/her quota of keystrokes; 2) the allotted time for editing between "cartoon cycles" has expired. The following pseudocode demonstrates a major dilemma in real-time programming:

```
 1:   procedure GetString;
      { This routine is called to edit a string.
        The routine terminates when the user types a
        carriage return. }
      begin
 2:     repeat
 3:       Read a character;
 4:       Process the character;
                             { legal char? backspace? etc. }
 5:     until character=carriage return;
      end;  {GetString}

 6:   repeat
 7:     Move the car one "mile" forward;
 8:     GetString;       { the string}
 9:     Process the string;
                          { legal word? update score, etc. }
10:   until no_more_words;
```

At first glance, the repeat loop on line 6 looks reasonable: The car will advance one "mile" each time the loop is executed. However, several problems remain unaddressed.

What if the user holds down one key and never presses ⏎ (the beast!). Line 7 would move the car, and then the editing routine would be called. While *GetString* is waiting for the tyrannical typist to·press ⏎, the car will be written up for parking in the red zone! Instead of driving smoothly across the screen, the car will stay in the same place. And if this crazy keyboardist holds down the ⏎ key, the car will zip across the screen in one meaningless motion.

The solution here is to pass a parameter to *GetString*, instructing it to return after a certain number of keystrokes have been typed—even if the user doesn't press ⏎. Of course, pressing ⏎ will still cause *GetString* to return to the main loop.

```
*1:   procedure GetString(KeyQuota : integer);
      { This routine is called to edit a string. The routine
```

```
          terminates when the user types a carriage return or
          when KeyQuota keystrokes have been typed.
        }
        begin
 2:       repeat
 3:         Read a character;
*3a:        Decrement KeyQuota;
                        {subtract one from number remaining}
 4:         Process the character;
                        {legal char? backspace? etc.}
 5:       until (character = carriage return) or
*5a:             (KeyQuota <= 0);
        end; {GetString}

 6:     repeat
 7:       Move the car one "mile" forward;
*8:       GetString(5 keystrokes);   { edit the string,
                                       allow 5 keystrokes }
 9:       if user typed carriage return then
 9a:        process the string;
                        {legal word? update score, etc.}
10:     until no_more_words;
```

We have modified five lines (each one is marked by an asterisk (*)) and completely overhauled the program's animation. Now *GetString* returns to the main loop every five characters or when ↵ is pressed, whichever comes first. Of course, we only want to check the string for validity if ↵ has been pressed. Either way, we can update our cartoon often enough to please the eye.

Now we're getting somewhere. Except that the car will still go nowhere if this obstinate operator refuses to press a single key! Think about it. We display the car, then call *GetString*. *GetString* waits for a carriage return or five characters from the keyboard. If they never come, *GetString* never returns to the loop from whence it came.

The solution? Well, look at lines 6–10 of our original **repeat** loop. What we really want is quite reasonable. We want *GetString* to always take about the same amount of time, whether a carriage return was typed, whether zero characters were typed, or whether a key was held down the entire time. If "processing the string" also requires a fixed amount of time each time it is called, then our car will run smoothly from one side of the screen to the other.

```
!1:     procedure GetString(KeyQuota, TimeQuota : integer);
        { This routine is called to edit a string.  The
          routine terminates when the user types a carriage
          return, when KeyQuota keystrokes have been typed,
          or when TimeQuota milliseconds (or some other
          unit) have elapsed.
        }
        begin
 2:       repeat
 3:         if a character is in the buffer:
!3a:          read a character;
```

```
!3b:         Decrement KeyQuota;
                      {subtract one from number remaining}
!3c:       Decrement TimeQuota;
                      {subtract one from number remaining}
 4:        Process the character;
                      {legal char? backspace? etc.}
 5:      until (character = carriage return) or
*5a:          (KeyQuota <= 0);
!5b:          (TimeQuota <= 0);
       end; { GetString }

 6:    repeat
 7:      Move the car one "mile" forward;
!8:      GetString(5 keystrokes, 100 milliseconds);
                      {edit the string}
 9:      if user typed carriage return then
 9a:       process the string;
                             {legal word? update score, etc.}
10:    until no_more_words;
```

The modified lines are marked with an exclamation mark (!). They do a decent job of accomplishing our goal. If we add any other routines anywhere in the main loop, we must carefully test and re-adjust our keystroke and time quotas.

DARE TO EXPERIMENT

Be daring and modify a copy of the program (you will probably find ways to optimize the code). If you can follow this discussion, you can understand the source code.

If you had difficulty understanding this material, look at the source and try getting through a few routines. Then, modify the simple cartooning program we introduced earlier (program SimpleCartoon). Change it so it lets you type characters while it is moving its object across the screen; keep making the program more and more sophisticated. By enhancing that simple example, you will soon encounter and solve the same problems we addressed earlier.

REVIEW

The Turbo Typist demonstrates the use of many Pascal structures and statements. Our discussion began with a description of the overall program structure. Next, we did a best case/worst case analysis (the slowest typist refused to touch a single key; the fastest simply held down one key continuously), and then focused on some solutions to some programming issues that were raised.

Now is the time to read the source code itself. The best way to master this program is to read all of the procedure and function headings and

try to comprehend the explanatory comments associated with each one. Then step through procedure *PlayOneRound* line-by-line, ignoring fine details like scoring and cursor placement. Once you understand the major steps in the loop, fine-tune your focus a little and make another pass; repeat this process until you're satisfied.

This concludes Part II of this tutorial. In the next section, we'll discuss some advanced topics in Turbo Pascal programming, including complex data structures and more exotic features of the Turbo Pascal programming environment.

PART III

Advanced Topics in Turbo Pascal

20 Stacks, Queues, Deques, and Lists

Chapter 17 covered the basics of pointers, dynamic allocation of variables, and linked lists. But you still may not realize just how important and powerful these concepts are. They provide you with the ability to create a variable out of "thin air" and then link it to other such variables, giving you a lot of freedom to do what you want.

LINKED LISTS

To talk about linked lists, we first must talk about *nodes*, the data structures (usually records) that contain all the information to be stored in a given location in the linked list and all the pointers necessary to point to other nodes in the list. Let's say that the information to be stored is a single integer. The simplest node definition you could construct would be like this one from Chapter 17:

```
type
  NodePtr = ^Node;
  Node = record
    Data : Integer;
    Next : NodePtr
  end;
```

This example shows the one exception to the Pascal rule that all identifiers must be defined before declaring the type it is bound to. Here you can use the data type *Node* before it has been defined, so that you can define *NodePtr* and use it within the definition of *Node*.

Single vs. Double Links

Note that the preceding node definition has only one pointer in it (Next), which points to the next node in the list. Imagine this list looking like this:

```
...-->Node-->Node-->Node-->...
```

It is possible to use two pointers instead of one, with the extra pointer linking back to the previous node in the list, like so:

```
...<-->Node<-->Node<-->Node<-->...
```

The Pascal definition would then change to something like this:

```
type
  NodePtr = ^Node;
  Node = record
    Data      : Integer;
    Last,Next : NodePtr
  end;
```

A list using nodes like this is known as a *doubly linked list.* A doubly linked list is usually easier to work with because it allows you to freely move back and forth; a *singly linked list* only allows you to move forward.

There is, of course, a price for this capability: each node takes up more memory. The additional size is most significant if the data section is small, and if you're running on a 16-bit system (where each pointer is 4 bytes long). For example, the previous node definition would increase in size from 6 bytes to 10, nearly doubling in size. However, if the data section is fairly large, then the additional pointer represents a small increase in size and a great increase in power and flexibility.

STARTING A LINKED LIST

Since all nodes in a linked list are created as needed, you have to make some decisions about how to get the whole thing started, how to create that first node. There are a number of approaches, but the easiest is to use a *Header node.* A Header node is created at the start of a program; its pointer(s) is either set to **nil** (which means it isn't pointing to anything) or to itself (if you're using a *circular linked list*). The data field(s) often isn't used, at least not in the same way it's used in other nodes in the list.

You usually declare the Header node as a pointer variable. At the start of the program, you create the node and set the fields accordingly, as shown here:

```
program LinkedLists;
  type
    NodePtr = ^Node;
    Node = record
      Data : Integer;
      Next · NodePtr
    end;
  var
    Header : NodePtr;
  ...
begin   { main body of program LinkedList }
```

```
  New(Header);
  with Header^ do begin
    Data := 0;
    Next := nil
  end;
  ...
end.  { of program LinkedList }
```

This code assumes you have a singly linked list (*linear* or *non-circular list*). If you have a circular list, things change a little.

Circular Linked Lists

A linear list has the Header node pointing to the first node, while the last node points to nothing (a **nil**). Such a list looks like this:

```
Header-->Node-->Node-->...-->Node-->Node-->[nil]
```

The problem with this type of linked list is that it can make search loops messy. For instance, suppose you have a list (unsorted) of nodes like the ones already defined, and you want to see if there's a node with a given value in it. Your first impulse might be to write a loop like this (assuming the value you're looking for is in Val):

```
TPtr := Header^.Next;
while (TPtr <> nil) and (TPtr^.Data <> Val) do
  TPtr := TPtr^.Next;
```

The problem, of course, is that if you hit the end of the list (*TPtr* = **nil**), you'll get a runtime error when the program tries to evaluate the expression *TPtr^.Data* <> *Val*. This problem can sneak up on you again and again (such as when doing insertion and deletion with a doubly linked list).

One solution is to use a circular linked list. Instead of having the last node point to nil, you can point it to the Header node instead, as shown in the following:

```
Header-->Node-->Node-->...-->Node-->Node-->Header
```

When you create your Header node, you point it to itself:

```
New(Header);
with Header^ do begin
  Data := 0;
  Next := Header
end;
```

Thus, your search loop will run without any difficulty:

```
TPtr := Header^.Next;
while (TPtr <> Header) and (TPtr^.Data <> Val) do
  TPtr := TPtr^.Next;
```

If you were using a doubly linked list, then you would start the Header with both its pointers pointing to itself (add the statement "Last := Header" to the previous code). As you add nodes, Header will point to both the first and last nodes in the list.

Insertion

After setting up the Header, the first function you need to perform is adding nodes to the list. This is called *insertion*—you add the node between two other nodes, or between a node and the end of the list.

Before inserting, you must do two things. First, create the node to be inserted (using the *New* command) and set all the data fields to their appropriate values. We'll call the pointer to this node *NPtr.* For the sample node already given, the program might look like this:

```
New(NPtr);
NPtr^.Data := NewValue; { whatever it happens to be }
NPtr^.Next := nil;
```

Note that setting *NPtr^.Next* equal to **nil** usually isn't necessary, but it's not a bad idea to initialize all fields of a node. Second, find the node in the list after which you wish to insert the new node. If the new node belongs at the start of the list (or if it's the first node in the list), then it will be inserted after the Header node. Typically, you start at the Header and continue until you've found the node you're looking for. We'll call this node *TPtr.*

The process of insertion itself is simple. Given *NPtr* and *TPtr* (as defined earlier), and assuming this is a singly linked list, do the following:

- Set *NPtr^.Next* equal to *TPtr^.Next.* This ensures that *NPtr* and *TPtr* are both pointing to the same node.
- Set *TPtr^.Next* equal to *NPtr.* This makes *TPtr* point to *NPtr.*

The final result is that *TPtr* points to *NPtr,* and *NPtr* points to the node that *TPtr* used to point to. The Pascal code for this is simple:

```
NPtr^.Next := TPtr^.Next;
TPtr^.Next := NPtr;
```

For a doubly linked list, you must do a little more work since you have twice as many pointers to change. Here's the code for a linear list:

```
NPtr^.Next := TPtr^.Next;
NPtr^.Last := TPtr;
TPtr^.Next := NPtr;
if NPtr^.Next <> nil
  then  NPtr^.Next^.Last := NPtr;
```

First, note that both of *NPtr*'s pointers have changed, pointing to the nodes that now precede and follow it. *TPtr* is then changed to point to *NPtr.* Finally, the node following *NPtr* (which you can reference as *NPtr^.Next^*) is changed to point back to *NPtr.* If the list is circular, then you can drop the test for **nil** and always make the assignment to *NPtr^.Next^.Last.*

Deletion

The process of deletion is similar to insertion: First, you find the node you want to delete and then make the preceding node point to the following node. If you have a singly linked list, then you must "remember" which node precedes the one you want to delete (otherwise, you'll have no way to get back to it). If *NPtr* points to the node to be deleted and *TPtr* points to the preceding node, then the statement

```
TPtr^.Next := NPtr^.Next;
```

removes *NPtr* from the list.

With a doubly linked list, you only need to know the node to be deleted:

```
NPtr^.Last^.Next := NPtr^.Next;
if NPtr^.Next <> nil
  then NPtr^.Next^.Last := Nptr^.Last;
```

The first statement points the preceding node to the following one; the second points the following node to the preceding one. As with insertion, if you're using a circular list, you don't need to check for *NPtr^.Next <> nil.*

Having changed the pointers, you must now decide what to do with *NPtr.* If you're using the *New/Dispose* method, then the statement

```
Dispose(NPtr);
```

will reclaim the memory used by *NPtr^*.

If you're using the *Mark/Release* method, then your best bet is to maintain a second, separate linked list (*Avail*) of nodes available for "recycling." Then, when you need a new node, you can check the *Avail* list first and re-use a node from there. If none are available, you can create a new one using *New*. If you don't use the *Avail* method, you run the risk of slowly using up all your memory (until, of course, you call *Release*).

STACKS

A *stack* is a list of objects that follows certain rules about how you add to or remove from it; you cannot simply insert or delete nodes anywhere in the list. Instead, all nodes added to the list must be added at the very front; likewise, all nodes taken from it must be taken from the very front. If you think about this for a minute, you'll realize that any node you remove will always be the one (of all those left in the list) most recently added. Because of this, a stack is sometimes known as a *last-in-first-out* (or LIFO) *list*. When you add to a stack, you're usually said to be "pushing" a node onto it; when you remove from it, you're "popping" a node.

Stacks can be handy in any situation where you need to remember what you're doing (and what you've done), go perform some other action (often the same action on a different set of data), then pick up where you left off. Usually, you can just call procedures and functions (sometimes recursively), and Pascal takes care of all that for you. However, situations may arise where you need (or want) to handle that explicitly, either to avoid recursion or to direct control over the stack.

The code to manipulate a stack is straightforward. The following code implements a stack as a linear, singly linked list:

```
type
  NodePtr = ^Node;
  Node = record
    Data : Integer;
    Next : NodePtr
  end;

var
  StackPtr   : NodePtr;      { Header for stack }
  StackEmpty : Boolean;      { flag for empty stack }

procedure CreateStack;
begin
  New(StackPtr);
  with StackPtr^ do
  begin
    Next := nil;
    Data := 0;
  end;
  StackEmpty := True;
end;  { of proc CreateStack }

procedure Pop(var Val : Integer);
var
  NPtr : NodePtr;
begin
  if not StackEmpty then
  begin
    NPtr := StackPtr^.Next;
    StackPtr^.Next := NPtr^.Next;
    Val := NPtr^.Data
    Dispose(NPtr);
    StackEmpty := (StackPtr^.Next = nil);
  end;
end;   { of proc Pop }

procedure Push(Val : Integer);
var
  NPtr : NodePtr;
begin
  StackEmpty := False;
  New(NPtr);
  NPtr^.Data := Val;
  NPtr^.Next := StackPtr^.Next;
```

```
  StackPtr^.Next := NPtr;
end;   { of proc Push }

procedure DeleteStack;
var
  TVal : Integer;
begin
  while not StackEmpty do
    Pop(TVal);
  Dispose(StackPtr);
end;   { of proc DeleteStack }
```

The routine *CreateStack* must be called before any other stack routines. It creates the Header node (called *StackPtr*) and sets the *StackEmpty* flag to TRUE. *Push* adds a value to the stack. Note that it just passes a value; you don't have to worry about what the node data structure looks like. Likewise, *Pop* just returns a value, hiding the node removal and deletion from you. Finally, the routine *DeleteStack* disposes of all the nodes in the stack, and then gets rid of *StackPtr* itself. You then must call *CreateStack* again before using the stack.

QUEUES

As we've just shown, a stack follows the LIFO principle. Sometimes, though, you want to treat nodes on a first-come, first-served basis. In programming, a *queue* is a list of nodes treated in the *first-in-first-out* (FIFO) manner: You always add a node to the end of the list, and you always remove a node from the front of the list.

Like stacks, queues are simple to implement. However, since you have to deal with both ends of the list, you'll probably find it easier to use a circular, doubly linked list rather than a linear, singly linked one. You should probably use a queue whenever things keep cropping up faster than you can handle them, and you want to look at them in the exact order they appeared. Here's an implementation:

```
type
  NodePtr = ^Node;
  Node = record
    Data : Integer;
    Last,Next: NodePtr
  end;

var
  Header       : NodePtr;     { Header for stack }
  QueueEmpty   : Boolean;     { flag for empty stack }

procedure CreateQueue;
begin
  New(Header);
  with Header^ do
```

```
  begin
    Next := Header;
    Last := Header;
    Data := 0;
  end;
  QueueEmpty := True;
end;  { of proc CreateQueue }

procedure GetValue(var Val : Integer);
var
  NPtr : NodePtr;
begin
  if not QueueEmpty then
  begin
    NPtr := Header^.Next;
    Header^.Next := NPtr^.Next;
    Header^.Next^.Last := Header;
    Val := NPtr^.Data;
    Dispose(NPtr);
    QueueEmpty := (Header^.Next = Header);
  end
end;  { of proc Pop }

procedure PutVal(Val : Integer);
var
  NPtr : NodePtr;
begin
  QueueEmpty := False;
  New(NPtr);
  with NPtr^ do
  begin
    Data := Val;
    Next := Header;
    Last := Header^.Last;
  end;
  Header^.Next := NPtr;
  NPtr^.Last^.Next := NPtr;
end;  { of proc PutVal }

procedure DeleteQueue;
var
  TVal : Integer;
begin
  while not QueueEmpty do
    GetVal(TVal);
  Dispose(Header);
end;   { of proc DeleteQueue }
```

The procedure *CreateQueue* must be called before any of the other queue procedures. *PutVal* creates a new node and inserts it between the Header and the end of the queue. *GetVal* gets the value from the node at the start of the queue, removes that node, and then disposes of it. And, of course, *DeleteQueue* cleans up the whole thing.

DEQUES

Donald Knuth, in his classic work *Fundamental Algorithms*, talks about yet another list type: a *deque*, or double-ended queue. While a queue adds nodes on only one end and removes them only from the other, a deque lets you add and remove nodes from either end. Implementing a deque is not much harder than implementing a queue. The only real difference is that the *GetVal* and *PutVal* routines now have to know whether to use the front (*Header^.Next*) or the rear (*Header^.Last*) of the list. Also, you will almost certainly want to use a circular, doubly linked list for a deque. Here's an implementation:

```
const
  Front = True;
  Rear = False;

type
  NodePtr = ^Node;
  Node = record
    Data     : Integer;
    Last,Next: NodePtr
  end;

var
  Header      : NodePtr;          { Header for stack }
  DequeEmpty  : Boolean;       { flag for empty stack }

procedure CreateDeque;
begin
  New(Header);
  with Header^ do
  begin
    Next := Header;
    Last := Header;
    Data := 0
  end;
  DequeEmpty := True;
end;  { of proc CreateDeque }

procedure InsertNode(var NPtr,TPtr : NodePtr);
begin
  NPtr^.Next := TPtr^.Next;
  NPtr^.Last := TPtr;
  TPtr^.Next := NPtr;
  NPtr^.Next^.Last := NPtr;
end;  { of proc InsertNode }

procedure RemoveNode(var NPtr,TPtr : NodePtr);
var
  TPtr : NodePtr;
begin
  NPtr := TPtr;
  NPtr^.Next^.Last := NPtr^.Last;
  NPtr^.Last^.Next := NPtr^.Next;
end;  { of proc RemoveNode }
```

```
procedure GetValue(var Val : Integer; theFront : Boolean);
var
  NPtr : NodePtr;
begin
  if not DequeEmpty then
  begin
    if theFront then RemoveNode(NPtr,Header^.Next)
    else RemoveNode(NPtr,Header^.Last);
    Val := NPtr^.Data;
    Dispose(NPtr);
    DequeEmpty = (Header^.Next = Header);
  end;
end;    { of proc Pop }

procedure PutVal(Val : Integer; theFront : Boolean);
var
  NPtr : NodePtr;
begin
  DequeEmpty := False;
  New(NPtr);
  NPtr^.Data := Val;
  if theFront then InsertNode(NPtr,Header)
  else InsertNode(NPtr,Header^.Last);
end;  { of proc PutVal }

procedure DeleteDeque;
var
  TVal : Integer;
begin
  while not DequeEmpty do
    GetVal(TVal,Front);
  Dispose(Header);
end;  { of proc DeleteDeque }
```

As you can see, we've gone on to general routines for *InsertNode* and *RemoveNode*. These routines are then called by *GetVal* and *PutVal*, with the boolean parameter *theFront* indicating whether to access the front or end of the deque.

LISTS

In the linked list data structures we've looked at, we've given *GetVal* and *PutVal* greater access to the list; but in every case, that access has been at one end or the other. What if you want to insert or delete nodes in the middle of the list?

You can do just that, and you can do it easily. With the creation of the *InsertNode* and *RemoveNode* procedures, you can now get to any node you want to by stepping through the list. Given a circular list, the code to do that is as follows:

```
TPtr := Header^.Next;
while (TPtr <> Header) and ({ whatever condition }) do
  TPtr := TPtr^.Next;
if TPtr = Header then { node not found }
else { node found }
```

The test (*TPtr* <> *Header*) keeps you looking until you've gone through the loop; the test ({ *whatever condition* }) determines what you're looking for. For example, if you were looking for a node with a particular value (*theVal*), then the **while** statement might look like this:

```
while (TPtr <> Header) and (TPtr^.Data <> theVal) do
  TPtr := TPtr^.Next;
```

General lists have all kinds of uses. As shown in Chapter 17, they can hold a list of data structures (such as records) in a more flexible form than an array (although you can index into an array faster than you can search through a list). Another good use is to create and maintain a sorted list of items, especially if you don't know ahead of time how many items you'll need to sort. With a linked list, you just insert the item in its proper place as you read it in.

REVIEW

In this chapter we have discussed linked lists in detail, elaborating on how they can be used to build different data structures (both linear and circular). Chapter 21 will look at linked lists and how to use them to build non-linear structures.

If you're interested in learning more, here are two books to guide you.

- Horowitz, E., and Sahni, S. *Fundamentals of Data Structures in Pascal.* Rockville: Computer Science Press, Inc., 1984.
- Knuth, D. E. *Fundamental Algorithms.* Vol. 1 of *The Art of Computer Programming.* 2d ed. Reading: Addison-Wesley, 1975.

21 Trees, Graphs, and Other Non-linear Structures

In Chapter 20, you learned more about linked lists and how to use them to build different data structures such as stacks, queues, and deques. All of these structures have one thing in common: The nodes in them are strung together. Sometimes, though, you need a different kind of structure, one that is not so linear.

Several such structures exist. The most common is the *tree*, which allows a node to point to several other nodes. There's also the *graph*, which allows *rings* and other intricate paths to form. *Sparse arrays* let you implement large, multidimensional arrays without wasting lots of space. Let's take a look at each of these non-linear structures.

INTRODUCTION TO TREES

A tree is like a linked list with branches. In a linked list, each node points ahead to, at most, one more node (though it may point back to the previous node as well). In a tree, each node can point ahead (or "down") to more than one node.

The *root* of a tree is the first (or topmost) node. (Notice that this kind of tree has the root at the "top" and the leaves at the "bottom.") A *subtree* consists of a non-root node and all the nodes (if any) below it. A *terminal node* (or *leaf*) is a node with no other nodes below it. A given node has a *parent* (the node directly above it), unless it's the root. A node may have *siblings* (other nodes pointed to by its parent) and *children* (nodes directly below it that it points to). A tree can now be defined as a root with zero or more subtrees; a *forest* is a set of zero or more unconnected trees.

Trees are useful to relate data in a hierarchical fashion, that is, in order of grade or class. Each leaf in a tree then represents a small component of the whole.

Another common use for trees is in the area of artificial intelligence (AI), where a complex goal or task can be broken down into small, performable subgoals or subtasks. Game-playing programs often use game trees to "look ahead" for good (or bad) moves. The root represents the current board position. Possible moves by one side generate new board states, or children to the root. Moves by the other side produce the next generation of children (grandchildren to the root), and so on.

These uses are beyond the scope of this book, but the following sections take a look at some simpler uses of trees.

BINARY TREES

The most commonly used tree is known as a binary tree; it is one in which each node has no more than two subtrees attached to it. Usually, these subtrees are labeled as left and right. You could use the following node definition for a binary tree:

```
type
  NodePtr = ^Node;
  Node = record
    Data          : Integer;
    Left,Right    : NodePtr
  end;
```

Note that we're using a minimum of pointers. Each node points only to its children (the nodes directly below it), with a value of **nil** indicating an unused link. A leaf, then, is a node whose left and right pointers are both **nil**. For this example, we'll assume the tree is *sorted*; that is, values are added according to certain rules. In this case, we'll assume that lower values are stored to the left and higher values to the right.

The program ANIMAL.PAS on your disk uses a binary tree to organize data about the animal kingdom. It prompts you to think of an animal and tries to "guess" it using its binary tree of animal data. If it does not know your animal, it will give up and let you "teach" it a new animal.

Searching Binary Trees

To add or remove any value from the tree, we must first see if the value is in the tree. The following boolean function (based on the algorithm described in *How to Solve it by Computer*; see the end of this chapter)

looks for a given value in the tree. If found, the function returns TRUE and passes back both the node containing the value as well as its parent; otherwise, it returns FALSE.

```
function FoundInTree(var TPtr,Parent : NodePtr;
                             Val : Integer) : Boolean;
var
  Found : Boolean;
begin
  TPtr := Root;
  Parent := nil;
  Found := False;
  while (TPtr <> nil) and not Found do
  with TPtr^ do begin
    if Data = Val
      then Found := True
    else
    begin
      Parent := TPtr;
      if Data > Val
        then TPtr := Left
        else TPtr := Right;
    end;
  end;
  FoundInTree := Found;
end; { of func FoundInTree }
```

This function works its way down the tree until it either finds the given value or runs into a **nil** pointer. There are two reasons why both the node itself and its parent are passed back. First, if you want to add a node, then you'll automatically have the parent node to add it to. Second, if you want to delete a node, then you also have the parent node to patch things back up.

Inserting Into Binary Trees

To have a tree to search, you must first build it, add values to it, and place them in the proper locations. For starters, you need a pointer, Root, of type *NodePtr*, which is initialized to **nil**. When you add your first value, **nil** is checked for (see the following routine) and Root is given the value. From then on, each value goes either to the left or right, depending upon whether it's less than or greater than Root's data.

To add a new value, you must check to see if it's already in the tree (using the function *FoundInTree*). If it's present in the tree then you needn't do anything; otherwise, you have the parent node to add it to, and you need only decide whether it goes to the left or the right. This routine does it all:

```
procedure AddToTree(Val : Integer);
var
  TPtr,Parent,NPtr    : NodePtr;
  Done                : Boolean;
```

```
begin
  if not FoundInTree(TPtr,Parent,Val) then
  begin
    if GetNode(NPtr) then
    begin
      NPtr^.Data := Val;
      if Root = nil then
        Root := NPtr
      else
        with Parent^ do
          if Data > Val then Left  := NPtr
          else Right := NPtr;
    end;
  end;
end; { of proc AddToTree }
```

Note: The boolean function *GetNode* creates the node, checking for sufficient memory and initializing the different record fields. If there isn't enough memory, it returns FALSE, preventing the node from being created and added.

Traversing Binary Trees

There's only one way to move through a linked list: straight ahead. There are at least three ways to traverse a binary tree: *preorder*, *inorder*, and *postorder*.

Preorder prints out the data of the current node *before* printing out that of either subtree. Inorder prints the current node *between* the left and right subtrees. Postorder writes out the current node *after* both subtrees. All are recursive, described in terms of *visiting* a root and its subtrees. Visiting a node means handling it in some way (for example, printing a value or comparing it to another value), since you will often pass (and ignore) nodes on your way to visit other ones.

The following set of routines traverse the binary tree you've created, writing out the values in the appropriate order. As you can see, it's easiest to define the traversal recursively. (The only difference between the three traversal methods is the point at which the data of the current node is written out.) Given the previous insertion routine, the procedure *InOrder* will print out the values in the tree in numerical order as shown here:

```
procedure WriteData(Data : Integer; var Row,Col : Byte);
begin
  GoToXY(Col,Row);
  Write(Data:9);
  Row := Row + 1;
end; { of proc WriteData }

procedure PreOrder(Node : NodePtr; var Row,Col : Byte);
begin
  if Node <> nil then
```

```
    with Node^ do
  begin
    WriteData(Data,Row,Col);
    PreOrder(Left,Row,Col);
    PreOrder(Right,Row,Col);
  end;
end ; { of proc PreOrder }

procedure InOrder(Node : NodePtr; var Row,Col : Byte);
begin
  if Node <> nil then
    with Node^ do
    begin
      InOrder(Left,Row,Col);
      WriteData(Data,Row,Col);
      InOrder(Right,Row,Col);
    end
end;  { of proc InOrder }

procedure PostOrder(Node : NodePtr; var Row,Col : Byte);
begin
  if Node <> nil then
    with Node^ do
    begin
      PostOrder(Left,Row,Col);
      PostOrder(Right,Row,Col);
      WriteData(Data,Row,Col);
    end;
end;  { of proc PostOrder }
```

Deleting Nodes And Subtrees

The easiest deletion to do on a binary tree is to remove an entire subtree. You can disconnect it by setting the appropriate pointer (Left or Right) on its parent to **nil**. However, you must also track down and dispose of all nodes in the subtree to recover the memory used by them. This simple procedure does just that:

```
procedure PruneTree(var TPtr : NodePtr);
begin
if TPtr <> nil then
  with TPtr^ do
  begin
    PruneTree(Left);
    PruneTree(Right);
    if (Left = nil) and (Right = nil) then
    begin
      Dispose(TPtr);
      TPtr := nil;
    end;
  end;
end;  { of proc PruneTree }
```

If you wanted to remove, say, the entire left subtree of Root, you could simply call *PruneTree(Root^.Left)*, which would dispose of all the nodes

and set *Root^.Left* equal to **nil**. And if you wanted to remove the entire tree, you'd just call *PruneTree(Root)*.

A far trickier matter is to remove a single node, especially if that node has subtrees below it. Think for a moment, you have removed a single node that frees up exactly one pointer (either Left or Right) on its parent, but you might have two subtrees to graft back in somewhere. Where do you put them? If there's only one subtree then there's no problem, but two can make things messy.

There is a well-defined, if complex, solution. The basic rule is this: If the node deleted is to the left of its parent, then its left subtree gets grafted in its place; likewise, if the node is deleted to the right, its right subtree is grafted in. The root of the ungrafted subtree is then added to the grafted subtree, with its (the root's) subtree still hanging below.

Non-Binary Trees

Not all trees are binary; nothing says that a given node can have only two children. There are applications with nodes of three or more, allowing you to have finer distinctions between subtrees. We will not describe these applications here, but we will show you how you might implement non-binary trees.

In implementing such a tree, your first problem is the node data structure. If you allow exactly three (or four, or five) subtrees, you can simply declare that many pointers. But what if you want a more general tree structure? What if you don't know ahead of time the maximum number of children a given node will have?

Believe it or not, you can implement a general tree using a binary tree node. Let's redefine our earlier data structure like this:

```
type
  NodePtr = ^Node;
  Node  = record
    Data          : Integer;
    Child,Sibling : NodePtr;
  end;
```

As you can see, the node is identical in size and content; we've just renamed Left and Right to be Child and Sibling. For a given node, Child points to the first (left-most) child of that node, while Sibling points to the first sibling to the right.

Since we no longer have a binary tree, our concept of order has disappeared to a certain extent. A given node can have several subtrees below it. How then do their relative positions correspond to the values they contain? A number of approaches can be taken. Each child can have some sort of cut-off value, or a range of values. The child itself can hold that value (or values) so the tree becomes self-regulating.

Manipulation of non-binary trees tends to be specific to a given implementation; thus, we won't discuss the topic in any more detail. However, the books listed at the end of this chapter and Chapter 20 deal with non-binary trees in more depth.

GRAPHS

You may remember that our definition of a tree included the provision that all subtrees of a given node were disjoint; that is, the nodes in one subtree were not found in any other subtree. This guaranteed exactly one path from the root to any given node. Cases could exist where two different nodes contained the same information, but they were still distinct nodes with different parents and in different subtrees.

What if we let a given node have parents? This seems like a small change, but it can have dramatic effects. Imagine multiple paths from one node to another, with one path better than another. Or imagine a node as a parent to one of its own ancestors, forming a ring of nodes that could be looped through indefinitely.

Such a data structure is generally called a *graph*. You can think of a graph as a tree with fewer restrictions; or better yet, think of a tree as a special, restricted graph. (Often, the distinction between the two is blurred, and what one person might call a graph is accepted by someone else as a tree.) In any case, a graph is a set of nodes that point to one another; different limitations may exist on how the nodes point. If a pointer goes one way only—like a tree, where the parent points to the child but not vice versa—then you have a *directed graph*. However, if a given pointer links two nodes equally, that is, you can't tell which is pointing to which, then you have an *undirected graph*. The link between two nodes in a graph is called an *edge*. In some graphs, the edge may have a value (or *weight*) assigned to it, in which case you have a *weighted graph*.

Graphs are used much like trees, and are particularly common in artificial intelligence (AI) where their greater flexibility is advantageous. Directed, weighted graphs are particularly useful in goal decomposition (breaking up a large, difficult goal into many small, easy goals). Researchers in the physical sciences, such as biology and chemistry, often use graphs to represent systems or molecules.

Again, it is beyond the scope of this book to treat graphs in any depth, particularly since the subject is rather esoteric. If you're interested, you might look at the book *Fundamentals of Data Structures in Pascal* (referred to at the end of Chapter 20), which devotes an entire chapter to the subject.

SPARSE ARRAYS

From time to time, you may need to work with large arrays that are possibly multidimensional. If you're not careful, you may quickly exceed the available memory, even on 16-bit systems. Take, for example, the following array:

```
var
  PicData : array[0..1023,0..1023] of Integer;
```

At first glance this may not seem like a large array. A little math, however, will quickly show that you need 2 megabytes of RAM just to hold it. If you try to compile a program with a declaration like this, you'll get a memory overflow error on this statement.

Suppose, though, that you needed an array like this and you didn't need it to hold too many values. (We'll talk later about how many values are "too many.") Maybe you had a thousand or so non-zero values to place in the array; the remaining values would all be zero. Such an array is known as a *sparse array*, since the number of significant values is small compared to the total number of storage locations. Is there some way you could store only the non-zero values with their coordinates?

There is indeed a way to implement a sparse array—by using a linked data structure. Suppose you defined the following node:

```
type
  NodePtr = ^Node;
  Node = record
    Val       : Integer;
    X,Y       : Integer;
    Next,Last : NodePtr;
  end;
```

Now, for each non-zero entry in the array, we use the Node data structure. The non-zero integer value is held in Val, the coordinates are kept in *X* and *Y*, and Next and Last point to the adjacent nodes along the X-axis. In the following example, each node has the value *(X,Y=Val)*:

```
-->(99,110=205)<---->(99,375=-10321)<---->(99,1001=32032)<--
```

The general idea is that nodes with the same *X* value form a doubly linked list using the pointers Next and Last. Furthermore, the list is sorted by the Y coordinate along the X axis; in other words, given two nodes, *A* and *B*, if *A^.Next* = *B*, then *A^.Y* <= *B^.Y*. This makes searches faster, since you may not need to search all nodes in a given list to find the one you're looking for.

You could implement this using a singly linked list, but it makes insertions and deletions a little more difficult. If you're tight on space, or if you're going to build the array and not change it, a single link reduces space.

So now you have several doubly linked lists, where each list represents all values having the same X coordinate. How do you find a given list, and how do you find a node within that list? One solution is to define a Header node for each list, and then link all the Header nodes together. To do that, we'll need a few extra pointers (for the Header nodes only). (The Header nodes don't need the *Y* and Val values, only the *X*.) Note that everything must link together. Let's modify our definition of node as follows:

```
Node = record
  Next,Last     : NodePtr;
  case Header : Boolean of
    False : (   Val      : Integer;
                X,Y      : ARange);
    True  : (   XVal     : Integer;
                Up,Down : NodePtr)
  end;
```

We've created a variant record. The field Header decides what type of node this is. If *Header = True*, then the node has the fields *XVal* (X coordinate for the axis), *Up* and *Down* (pointers to other Header nodes). Using *Up* and *Down*, you can implement a doubly linked circular list of Header nodes. Each Header then uses *Next* and *Last* to form a doubly linked circular list of data nodes (*Header = False*).

Here is a routine containing the boolean function *NodeFound*, which takes (X,Y) coordinates and returns a pointer to the corresponding node (if it exists):

```
function NodeFound(TX,TY : Integer;
                   var TPtr : NodePtr) : Boolean;
var
  Found : Boolean;
begin
  TPtr := theHead^.Up;
  Found := False;
  while (TPtr^.XVal < TX) and (TPtr <> theHead) do
    TPtr := TPtr^.Up;
  if TPtr^.XVal = TX then begin
    TPtr := TPtr^.Next;
    while (TPtr^.Y < TY) and not TPtr^.Header do
      TPtr := TPtr^.Next;
    Found := (TPtr^.Y = TY);
  end;
  NodeFound := Found;
end; { of func NodeFound }
```

The global variable *theHead* is of type *NodePtr*, and is the master Header node for the entire structure. This function first seeks to find a Header node for the X coordinate desired. If it is found, it searches the list of data nodes until the Y coordinate desired is found. In either case, the search ends (fails) if a coordinate greater than the one sought is found, or if the list circles back to the initial Header.

Even when *NodeFound* returns FALSE, it produces valuable information. In such cases, *TPtr* points to the closest node. For example, suppose you're looking for (259,321). It doesn't exist, but the nodes (259,17) and (259,421) do. *TPtr* will return from *NodeFound* pointing to (259,421), which means that if you want to add (259,321), you're at the correct point for insertion. Even if there are no nodes with X coordinate 259, *TPtr* would still help: It would point to the Header node of the list just above 259. So, once again, you are pointing to the appropriate spot to insert a new Header node.

Mixed Sparse Arrays

There are many variations on sparse arrays, depending on how much space you have and how fast the program must be. Suppose you need the program to run faster and have memory to spare. You might keep your original definition of *Node* and declare the following array:

```
var
  Header : array[ARange] of NodePtr;
```

Thus, instead of a linked list of Headers, you would have a fixed array of all 1024 Headers, one for each possible X coordinate. You would initialize all elements in this array to be **nil**, then create linked lists as needed. The function *NodeFound* would then look like this:

```
function NodeFound(TX,TY : Integer;
                  var TPtr : NodePtr) : Boolean;
var
  Done : Boolean;
begin
  TPtr := Header[TX];
  NodeFound := False;
  if TPtr <> nil then begin
    Done := False;
    repeat
      if (TPtr^.Y >= TY) then Done := True
        else TPtr := TPtr^.Next
    until Done or (TPtr = nil);
    if Done
      then NodeFound := (TPtr^.Y = TY);
  end;
end ; { of func NodeFound }
```

Note that the inner search loop has been changed. Our linked lists are no longer circular, since the nodes can't point to the elements in the array Header. This forces us to make the test *TPtr^.Y >= TY* inside the loop, since at the **until** statement there's a chance that *TPtr* = ***nil*** (which makes the other comparison illegal). To use this routine, you would have to do some benchmarks to see if the increase in speed is worth the additional memory required.

When to Use Sparse Arrays

The toughest question about sparse arrays is when to use them. If you've got an array that is impossible to compile under Turbo Pascal (like the one at the start of this chapter), then you have no choice. Well, almost no choice. It is possible to emulate a very large array using pointers and a few other tricks, provided you have sufficient memory on the heap. And, if your array isn't sparse (if it's not only very big but has lots of values as well), then you're in real trouble and maybe you should look for a larger computer to perform your data manipulation.

Here's the hard decision: What if you could implement a regular array, but want to save memory? How can you tell if a linked-list implementation will be smaller? The easiest way is to define your "normal" array, and find out its size using the *SizeOf* function. Then define a node for your linked list structure, and find out its size as well. Divide the array's size by the node's size. This will tell you the point (in terms of number of nodes) at which your linked list version is eating up more memory than the normal array. In much the same manner, you can divide the dynamic memory size (given at the end of a compilation) by the size of a node and find the maximum number of nodes allowable.

You must also realize that pointers can eat up a lot of space, especially on 16-bit machines where each pointer is 4 bytes long. So if your actual data is small compared to the rest of the data structure, you may be better off with a regular array. On the other hand, if the data is rather large, such as a complete record or something similar, then the linked-list approach looks better and better, since the additional overhead for pointers becomes less significant and the wasted space in a regular array becomes very significant.

REVIEW

As a balance to Chapter 20, here we have examined linked lists using non-linear structures such as graphs, trees, and arrays. Here are two books (in addition to those mentioned in Chapter 20) that will provide you with more detail about trees, graphs, and other linked structures.

- Dromey, R. G. *How to Solve it by Computer.* Englewood Cliffs: Prentice-Hall International, 1982.
- Knuth, D. E. *Searching and Sorting*. Vol. 3 of *The Art of Computer Programming*. Reading: Addison-Wesley, 1973.

22 Sorting and Searching

The ultimate function of a program is to solve problems. It presents a solution to you in the form of graphics, hardcopy, spreadsheets, hex dumps, and so on. However, to arrive at and represent a solution in any form, you usually must rely on sorting and searching techniques. While it is possible to write an entire book on these topics, we'll only review a few examples here.

SORTING

Sorting can be done on various levels. For instance, you can sort by grouping similar items together (all two-story homes would comprise a group). And, you can also sort and order a group of similar items by predetermined ascending or descending values (all two-story houses valued from $125,000 – $195,000, listed in ascending order). Let's look at a few sorting methods, using a list of integers as the data to be sorted.

Insertion Sort

In Part I, we presented a sample program to sort a list of integers. Here's the routine from that program, using the *insertion sort* method:

```
procedure InsertSort(ListMax : Integer);
{  purpose: sort list using insertion algorithm }
var
  Indx,Jndx,Val : Integer;
begin
  for Indx := 2 to ListMax do begin
    Val := List[Indx];
    Jndx := Indx;
    while List[Jndx - 1] > Val do begin
      List[Jndx] := List[Jndx - 1];
      Jndx := Jndx - 1
    end;
    List[Jndx] := Val
  end
end;  { of proc InsertSort }
```

This procedure assumes that List is declared as an **array**[*1..ListMax*] **of** integer. An insertion sort takes each number in the list and moves it toward the top of the list, shuffling the other numbers as it goes, until all the numbers preceding it are of a lower value. Starting at the top of the list, the sort works its way down the list, so that the upper portion of the list is always sorted. For example, suppose that partway through the sort the list looked like this:

```
-10 -2 15 19 55 69 0 -20 42 100
```

The next number we would look at would be 0, and that number would keep moving left until it found a number less than itself (which would be −2). Then, the list would look like this:

```
-10 -2 0 15 19 55 69 -20 42 100
```

The next number to be moved, −20, illustrates a special case in insertion sorts. The number −20 is of the lowest value in the list of numbers; you must know how to stop when you hit the top of the list since you'll never encounter a lower value in the list itself.

Three possible solutions present themselves. First, if you're working with a list of N elements, then declare the array to have $N + 1$ elements (0..N) and store the lowest possible value in location 0. For example, you might do the following:

```
const
  ListMax = 10;
var
  List : array[0..ListMax] of integer;
begin
  List[0] := -MaxInt - 1;
  ...
end.
```

The values you want sorted are stored in locations 1 through *ListMax*. The location *List[0]* acts as a stopper or *sentinel*; since it's the lowest possible integer value, nothing can move beyond it.

In some cases, you will not be able to use this extra location. For example, when sorting part of a list, there may not be a "free" location there. You can then use the second approach: Before starting the sort, find the lowest value in the list and move it to *List[1]* as the sentinel, trading it with the present sentinel value. For example, if our original list to be sorted looked like this:

```
-10 19 15 -2 69 55 0 -20 42 100
```

After the search and swap, the list would look like this:

```
-20 19 15 -2 69 55 0 -10 42 100
```

As you can see, the values −20 and −10 traded places. Now, all other values in the list are greater than −20, thus all values will stop moving when they reach it (if not before).

The third solution is to put a **goto** statement in the inner loop, jumping out of the loop if *Jndx* gets down to 1.

Shellsort

The insertion sort, while an effective method, suffers from the large number of compares and exchanges needed to move a number from its starting position to its final one. A computer scientist named Donald Shell suggested that since most numbers sorted are going to move a fair amount anyway, it might be more efficient to compare and swap numbers some distance from each other first, shrinking the distance between numbers until you return to the ones in closest proximity to each other. This method is known as *shellsort*, or sorting by diminishing increment.

Two issues immediately arise in considering shellsort. First is what sort method to use, since shellsort only states to sort those numbers that are some distance from each other, not how to sort them.

The second issue is deciding what incremental value to use. Shellsort has proven hard to analyze; most studies of its effectiveness are based on trial-and-error testing. The literature suggests at least three approaches. Assuming a list of *N* integers, you might use one of the following sets of diminishing increments:

- Start with *Inc* = *N div* 2; divide *Inc* by 2 each time.
- Start with *Inc* = (2***P*) − 1, where *P* = *Trunc(N log 2)*; decrement P by 1 each time.
- Set *Inc* = 1, then continue to set *Inc* := 3 * *Inc* + 1 until *Inc* > *N*; divide *Inc* by 3 at the start of each loop.

The following example of shellsort (adapted from *Algorithms* by Robert Sedgewick; see the end of this chapter) uses the third method:

```
procedure ShellSort();
 { purpose: sort list using shell algorithm }
label
  ExitLoop;
var
  Indx,Jndx,Val,Inc : integer;
begin
  Inc := 1;
  repeat
    Inc := 3*Inc + 1
  until Inc > ListMax;
  repeat
    Inc := Inc div 3;
    for Indx := Inc+1 to ListMax do begin
      Val := List[Indx];
      Jndx := Indx;
      while List[Jndx-Inc] > Val do begin
```

```
        List[Jndx] := List[Jndx - Inc];
        Jndx := Jndx - Inc;
        if Jndx <= Inc then goto ExitLoop
      end;
ExitLoop:
      List[Jndx] := Val
    end
  until Inc = 1
end; { of proc ShellSort }
```

At the center of the program is the insertion sort routine (using the **goto** solution) with one major change: The number 1 has been replaced by the variable *Inc*. Because of the way *Inc* has been initialized, it will equal 1 the last time through the loop and will do a regular insertion sort. However, by that time the list will be mostly sorted; very few numbers will have to be moved, and those that do won't need to move far.

Quicksort

Shellsort is a simple, efficient sorting method, one that will satisfy most casual needs. However, you may find yourself in a time-critical situation where you need the list sorted as quickly as possible. In such cases, your best bet is probably *quicksort*, an algorithm developed by C.A.R. Hoare.

The basic idea of quicksort is a simple one. Using the list of integers *List[1..ListMax]*, we'll first perform the following steps:

- Pick some number in *List*, which we'll call *Val*;
- Move all the numbers in *List* so that *Val* is in its correct location, *List[Indx]*. This means that all the numbers in *List[1..Indxl−1]* are less than *Val* (though not necessarily sorted), and that all the numbers in *List[Indx+1..ListMax]* are greater than *Val* (also not necessarily sorted).

Now perform the same operation for each of the sublists, *List[1..Indx−1]* and *List[Indx+1..ListMax]*. This continues until each sublist is too small to sort.

The simplest implementation of quicksort is a recursive one. Adapting again from Sedgewick, you have the following implementation:

```
function Partition(Left,Right : integer) : integer;
{ partition list into two sublists }
var
  Val,Indx,Jndx,Temp : integer;
begin
  Val := List[Right];
  Indx := Left - 1; Jndx := Right;
  repeat
    repeat Indx := Indx + 1 until List[Indx] >= Val;
    repeat Jndx := Jndx - 1 until List[Jndx] <= Val;
```

```
    Temp := List[Indx];
    List[Indx] := List[Jndx];
    List[Jndx] := Temp
  until Jndx <= Indx;
  List[Jndx] := List[Indx];
  List[Indx] := List[Right];
  List[Right] := Temp;
  Partition := Indx
end; { of func Partition }

procedure QuickSort(Left,Right : integer);
{ recursive implementation of Quicksort }
var
  Indx : integer;
begin { main body of proc QuickSort }
  if Left <= Right then begin
    Indx := Partition(Left,Right);
    QuickSort(Left,Indx - 1);
    QuickSort(Indx + 1,Right)
  end
end;  { of proc QuickSort }
```

The procedure *Partition* uses the last value (*List[Right]*) in the current sublist as *Val*, the one to be moved to its correct location. It then starts at both ends of the list and moves toward the center, swapping numbers on the left greater than *Val* with those on the right less than *Val*. Once it hits the center, it picks the last number greater than *Val* *(List[Indx])* and swaps it with *Val* (which is still sitting at *List[Right]*). *Val* is now in its final location, and *Indx* becomes the new dividing point. The procedure *QuickSort* calls *Partition* for the current list, then calls itself for the left and right sublists that *Indx* divides.

However, you may not want to use a recursive approach in some situations. Most notably, if the list is large, your recursion stack might overflow, especially on 8-bit systems. You can avoid recursion by implementing your own stack, either with an array or with a linked list (as shown in Chapter 20).

Here's a non-recursive version of *QuickSort*, assuming you have a set of stack routines (*ClearStack, Push, Pop, StackEmpty*):

```
procedure QuickSort;
{ non-recursive implementation of Quicksort }
var
  Left,Right,Indx : integer;
  Done            : boolean;
begin
  Left := 1; Right := ListMax;
  ClearStack;  Done := False;
  repeat
    if Left <= Right then begin
      Indx := Partition(Left,Right);
      if (Indx - Left) > (Right - Indx) then begin
        Push(Left); Push(Indx - 1);
        Left := Indx + 1
```

```
      end
      else begin
        Push(Indx+1); Push(Right);
        Right := Indx - 1
      end;
    end
    else if not StackEmpty then begin
      Pop(Right); Pop(Left)
    end
    else Done := True
  until Done
end; { of proc QuickSort }
```

This version always pushes the larger sublist on the stack, looping back to partition the smaller one first. This helps reduce the number of values pushed onto the stack; the upper limit is about *lg ListMax* (*lg* means log base 2). If *ListMax* = 10, then the stack need only hold four sets of values. Your best bet is probably to use an array-based stack, since the overhead for a linked-list stack isn't worth it.

SEARCHING

As mentioned at the start of this chapter, searching techniques help you determine the location (if any) of specific data, in order to retrieve, modify, or verify its existence, or place more data nearby (as in sorting). Usually, you're searching for a *key*, some part of the information that identifies the rest of it. In the simplest case, such as a list of integers, the key is the information itself. In more complex settings, such as a list of records, the key might be an ID number, a name, or something even more complex. In every case, you know the key, and you want to find its location. Let's look at a few methods of searching.

Sequential Search

Given a list of values (such as integers), the most straightforward way of finding a given value (or key) is to start at the top of the list and search it sequentially until the end of the list. With certain data structures, such as linear linked lists, this may be your only option, since you (usually) have no way of jumping into the middle of the list. (Arrays provide more options, which we'll examine momentarily.) Right now, let's assume we're looking for a given integer value in the same integer array, *List[1..ListMax]*. Our routine might look like this:

```
function Found(Val : integer; var Indx : integer) : boolean;
var
  Flag : boolean;
begin
  Flag := False;
  Indx := 1;
```

```
  while not Flag and (Indx <= ListMax) do
    if List[Indx] = Val
      then Flag := True
      else Indx := Indx + 1;
  Found := Flag
end; { of func Found }
```

This function returns TRUE if *Val* is found in *List* and sets *Indx* to the appropriate location; otherwise, it returns FALSE and *Indx* equals *ListMax + 1*. If *List* has been sorted, then it can be made a bit more efficient:

```
function Found(Val : integer; var Indx : integer) : boolean;
var
  Done : boolean;
begin
  Found := False;
  Done  := False;
  Indx := 1;
  while not Done and (Indx <= ListMax) do
    if List[Indx] = Val then begin
      Done := True;
      Found := True
    end
    else if List[Indx] > Val then Done := True
    else Indx := Indx + 1
end; { of func Found }
```

Now the search function knows to quit as soon as it encounters a number greater than *Val*, since the rest of the list will then be greater than *Val*, too. However, unless *List* is short, you're better off using a binary search, the next technique we'll discuss.

Binary Search

Think for a moment about looking for a given value in a sorted list of numbers. Instead of starting to the left and going through the list, suppose you start in the middle. If by chance you find the value you want, you're done. If the value is too large, then you know you'll have to search the left half; otherwise, you'll have to search the right half. Repeat this process on the appropriate half until you find the value or run out of lists. You've just performed a *binary search*. Here's an implementation:

```
function BFound(Val : integer; var Indx : integer) : boolean;
var
  Left,Right : integer;
begin
  Left  := 1; Right := ListMax;
  repeat
     Indx := (Left+Right) shr 1; { = div 2 }
     if Value < List[Indx]
       then Right := Indx - 1
```

```
      else Left  := Indx + 1
  until (Value = List[Indx]) or (Left > Right);
  BFound := (Left <= Right)
end; { of func BFound }
```

The nice thing about a binary search is that the most comparisons you'll have to do is *lg* of *ListMax*, while with a sequential search, you'll average *ListMax* / 2 comparisons, and your worst case is *ListMax*. If *ListMax* = *MaxInt* (32767), then here are the best, worse, and average performances of sequential and binary searches:

	Best	Worst	Average
Sequential	1	32767	16383
Binary	1	16	8

As you can see, there's quite a difference between the two methods. However, if you will be changing the list a lot, keeping it sorted could be quite time consuming, unless you're using a linked list (and if you're using a linked list, you can't use the binary search method). However, there is yet another technique that combines the two approaches: *hashing*.

Hashing

You can easily maintain (add to and remove from) a sorted linked list, but you must search it sequentially. You can easily search a sorted array, but you must do much to maintain it. Hashing allows you to combine both approaches. For example, suppose you have the following definitions:

```
const
  HMax = 63;
type
  NodePtr = ^Node;
  Node = record
    Next : NodePtr;
    Data : integer
  end;
var
  HList : array[0..HMax] of NodePtr;
```

HList is an array of header nodes, each one possibly pointing to a linked list. To add a value to the list, you must first use a hashing function to choose among the header nodes. This function will take the data subfield (on which we're sorting) and return some result in the range 0..*HMax*. It will always return the same result for the same data value (otherwise, you couldn't find the stored data). Also, it will try to spread the hash values evenly over the range 0..*HMax*, so that most of your values don't end up in only a few locations.

Once you have the hash value, you can use it as an index into *HList*. You now have a linked list to which you can add your value. This list can be sorted or not, as you prefer. Finding a number involves the same process: You must use the hash function to find the Header node in *HList*, then search the linked list until you find the value or reach the end.

If you think about it for a minute, you'll see how this is a compromise between pure arrays and pure linked lists. If *HMax* = 0, then you've got a simple linked list; if *HMax* = *the number of values to be stored* (and you drop the hash function), then you've got a regular array. The value of *HMax* determines just how much you're leaning toward one or the other. A large *HMax* reduces collisions (when two or more values map to the same index) and speeds up the search, while a small *HMax* reduces the amount of memory initially allocated.

External Search

Sometimes, you may want to find data stored in a file. The file may be too large to have in memory, so you pull in selected records as you need them. However, each time you read a record from the file, it costs you a certain amount of time. Your goal is to read as few records as possible when searching for the one desired.

The worst case is to start at the beginning of the file and read each record until you find the one you want. This method would be necessary if you could only read files sequentially. Luckily, Turbo Pascal does allow random access of files (via the *Seek* procedure), so if a file is sorted, you can use a variation of the binary search to look for a given record.

However, sorting a file is even messier than sorting an array, and you still might have to read several records before finding the record (or verifying that it isn't there). More efficient techniques are needed.

One simple approach is to maintain a separate list, an *index table* of keys and record numbers for all the records in the file. Let's suppose that you want to retrieve a given record based on a star's name (assuming you have already established a table of star names). You might make the following definition:

```
type
  StarIndex = record
    Name : NameStr;
    Index : integer
  end;
```

You now have a data type for each entry in the index table, associating a name with a record number. The question is, what data type do you use for the table itself? The answer depends on how big the table will be and if it will change in size a lot. Essentially, you have the choices we've already presented: array, linked list, or hash table. You can even add some twists, such as "faking" a dynamically sized array. And you'll probably want to write the index table itself out to disk, to avoid recreating it each time you read through the entire file of star records. Given the choice of an index table, you can then use the searching and sorting options to find the name, get the index, and read the appropriate record in from the file. Assuming that *TabMax*, *StarRec*, and *NameStr* are defined :

```
var
  ITable : array[1..TabMax] of StarIndex;
  ICount : 0..TabMax;
  SFile  : file of StarRec;
  ...
function RecFound(FName : NameStr;
               var Star : StarRec) : Boolean;
var
  Tndx : Integer;
begin
  Tndx := 1; RecFound := False;
  while (Tndx <= ICount) do with ITable[Tndx] do
    if FName = Name then begin
      RecFound := True;
      Seek(SFile,Index);
      Read(SFile,Star);
      Tndx := ICount + 1
    end
    else Tndx := Tndx + 1
end; { of func RecFound }
```

This routine does not assume that the index table (*ITable*) is sorted. It does a sequential search for the name. If it is found, it reads in the appropriate record and returns TRUE; otherwise, it returns FALSE.

REVIEW

In this chapter, we have covered only a bit more than the rudiments of sorting and searching. And after our discussion of the insertion sort method, shellsort, and quicksort, then hashing, sequential, binary, and external searching, you may thirst for more knowledge. Many books deal with the topic; the following book, along with the ones mentioned in previous chapters, will provide you with more detailed information.

- Sedgewick, R. *Algorithms*. Reading: Addison-Wesley, 1984.

23 Writing Large Programs

If you are like most programmers, the majority of programs that you write will be smaller than about 30K. It is possible, however, to run out of memory when trying to write a larger program. (This can also happen when writing a small program on a computer that does not have much main memory.) What kinds of programs use up a lot of memory? And how do you handle them? In this chapter, we discuss the memory structure of a Turbo Pascal program, the way the compiler manages memory, and some ways to deal with "out of memory" errors.

A PROGRAM'S MEMORY REQUIREMENTS

There are essentially two parts to every program that you create: code and data. Once you have typed in your source code, you use the compiler to translate your Pascal instructions into machine language code and to reserve space in the data area for the variables that you declare. Turbo Pascal allows you a maximum of 64K space for your code, and another 64K space for your data.

Of course, unless your computer has more than 128K total memory, you cannot run a program that will use the maximum amount of code and data space; and if you have a CP/M-80 computer, both your code and data must fit into 64K of memory.

There are four main ways to run out of memory when trying to write Turbo Pascal programs:

1. *Too much everything*. The combination of your operating system, the run-time library, the compiler/editor, and your program text doesn't leave enough room to compile your program. First, try making a .COM file (see Chapter 6). Next, try reducing the amount of text Turbo has to keep in memory by using Include files

(explained later). If an error still occurs, your problem is number 2, 3, or 4.

2. *Too much code.* Your program source compiles to more than 64K of code (runtime library included). (Refer to the discussions about overlays and chaining that follow.)
3. *Too much data.* You have declared more than 64K worth of global variables. You need to learn how to use the heap, or perhaps you can make some of your variables local to the procedures that need them (discussed later in this chapter).
4. *Too much heap/stack use.* The stack and heap are bumping into each other. Check *MaxAvail* (see Chapter 17) before allocating memory on the heap, or perhaps some procedure/function is eating up too much stack space (especially if you are using recursion). Finally, reducing the amount of code will leave more room for the heap and stack; refer to the discussions about overlays and chaining that follow.

Now we'll discuss each of these in a bit more detail.

TOO MUCH DATA

Let's pretend that the following program will not generate any code at all (actually, it generates a very small program that loads and then exits—a small amount of space is always reserved in the data area as well):

```
program Example;
begin
end.
```

In addition, because we did not declare any variables, we'll also imagine that no space was reserved in the data area. On a 16-bit computer, we would still have 64K memory available for our code, and another 64K available for our data. If we now declare a character variable (characters use 1 byte), we would still have 64K for our code and 64K minus 1 byte for our data:

```
program Example1;
var
  ch : char;
begin
end.
```

The next example declares a 32K array and therefore leaves us another 32K for more variables:

```
program Example2;
var
  a : array[0..MaxInt] of char;  { MaxInt equals 32K - 1 }
begin
end.
```

What would happen if we changed the array type to real? Real numbers use 6 bytes, so we would be trying to reserve 192K (6 bytes * 32K) of data area and the compiler would give us a MEMORY OVERFLOW error. And we haven't even written any code!

TOO MUCH CODE

Now that we've succeeded in running out of data space, let's intentionally run out of code space as well. Before we discuss the next example, however, you need to know about the Turbo runtime library. Every Turbo program that you write has access to many "built-in" features: file-handling routines, routines to clear the screen, facilities to write data to several devices, and so on. These routines are written and stored in a 12K machine language library that is automatically placed at the beginning of your programs. Since your entire program (library included) is allocated a maximum of 64K, this leaves about 52K for your Pascal code:

```
program Example3;
var
  ch : char;
begin
  ch := ch;
end.
```

The statement *ch := ch;* uses about 8 bytes on an IBM PC. If you compile this program to a .COM file, the size of the .COM file will be about 12K (for the library) plus 8 bytes (for the statement *ch := ch*).

Now, let's try to use too much code so that we generate a compiler error. Since we are trying to use up 52K bytes (64K minus 12K library), we'll need to repeat our assignment statement more than 6,600 times in order to run out of memory.

```
program Example4;
var
  ch : char;
begin
  ch := ch;        { Line    1 }
  ...
  ch := ch;        { Line 7000 }
end.
```

This program generates a MEMORY OVERFLOW error on line 6702. What have we learned? We have learned how to write simple programs that ask for too much data or code space and therefore generate compiler errors.

TOO MUCH TEXT

Actually, you will run into a slight problem when trying to compile program Example4. If you try to type the assignment statement 7,000 times, you will end up with an 88K text file. Since the Turbo Pascal editor has a text buffer of only 64K, the file will be too big for the compiler/editor to load.

Let's take a look at how quickly memory is eaten up on a 16-bit system. When you boot your computer, DOS will take up to 40K of memory (depending on which version you're running). Load Turbo Pascal, and the program takes up about 40K memory; it also loads the 12K Pascal runtime library. We've already used 92K, and we haven't loaded your program text (up to 64K) or generated any code (another 52K):

```
         640   TOP OF MEMORY
                     ...    ...
               data, heap and stack areas
         208
               program code generated by compiler (52K)
         156
    K          program text                       (64K)
    B     92
    Y          Turbo Pascal compiler              (40K)
    T     52
    E          Pascal run time library            (12K)
    S     40
               DOS                                (40K)
           0
                                            MAX. BYTES

    Simplified DOS memory map. Figures are approximate.
```

If you are working on a 128K system, there is obviously not enough room for all of this. And if you run any RAM-resident programs—such as SideKick, SuperKey, or Turbo Lightning—there is even less room available. If you are using a 64K CP/M-80 system, all of the preceding (except DOS) must fit into one 64K chunk of RAM.

THE STACK

After memory has been reserved for the runtime library, your code, and the global data you declared, any remaining memory is set aside for the stack and heap. The stack is a scratch area of memory used by your program (we'll discuss the heap in a moment).

You don't have to worry about the details of what the stack does or how it works; just keep in mind that parameters passed to procedures/ functions, locally declared variables, and other similar objects are all

placed on and removed from the stack automatically *each time a procedure or function is called.* You also don't have to worry about keeping this scratch area clean and tidy—Turbo takes care of that for you. Given this general introduction about the stack, consider the following examples:

```
program GlobalData;
type
  BigArrayType = array[0..MaxInt] of char;  { 32K array }
var
  BigArray : BigArrayType;
begin  { program body }
end.
```

This program is perfectly reasonable and will compile; however, we have already used up half of our data area on one variable. If we are only using the variable during part of the program, we should declare it locally (inside a procedure, for example) and space will be set aside for it on the stack (rather than waste room in the data area):

```
program LocalData;
type
  BigArrayType = array[0..MaxInt] of char;  { 32K array }
procedure UseTheArray;
var
  BigArray : BigArrayType;
begin
end;  { UseTheArray }

begin { program body }
  ...
  UseTheArray;
  ...
end.
```

Now we still have 64K worth of data space available. In general, declare your variables locally unless

- You need access to them globally (a perfectly valid reason).
- You are writing a video game or other real-time application and speed is essential (it takes slightly longer to manipulate stack data).
- Your local variables will overflow the stack (the stack is limited to 64K).

On CP/M-80 systems, even local variables are declared statically; though no memory is saved by using local variables, it is still good programming practice to use them whenever possible. Of course, if a TOO MANY VARIABLES warning or an OUT OF MEMORY error occurs, you will need to economize memory usage using one of the methods discussed in this chapter. (There is an excellent memory map of Turbo Pascal on CP/M-80 systems in Chapter 22 of your *Turbo Pascal Reference Manual.*)

THE HEAP

As explained in Chapter 17, Turbo Pascal supports an unlimited heap. All the extra memory on your system is "heaped" into a large continuous chunk after (higher in memory than) your data area. On a 640K system, you may end up with over 512K of heap space available. You can use all this RAM to store and manipulate variables. Instead of referring to these variables with an identifier, however, you must declare a pointer variable and refer to this pointer when you want to access a variable:

```
program PointerData;
type
  BigArrayType = array[0..MaxInt] of char;  { 32K array }
var
  one, two, three : ^BigArrayType;
begin   { program body }
  New(one);
  New(two);
  New(three);
end.
```

This program uses three 32K arrays (the *New* statement reserves memory from the heap at runtime). Of course, there has to be room on the heap or an error will occur.

If there isn't enough room on the heap and a *New* statement is called, a heap/stack collision error will occur (RUNTIME ERROR FF). Similarly, if there isn't enough room on the stack and you call a routine that uses a lot of stack space, the same error will occur. Note that this error will cause your program to crash. If you ever encounter one, your stack has overflowed onto your heap or vice versa. (Refer to Chapter 17 for more information about using pointer variables.)

SOLUTIONS

Now that you have a better understanding of the perils of memory management, let's discuss some techniques for dealing with "out of memory" errors.

INCLUDE FILES

There are two good reasons to use Include files: (1) they reduce the amount of text held in memory and therefore free up more RAM for Turbo to work with, and(2) they make it easy to break your program into modules of related routines. When you compile, each module is pulled in as needed. This not only solves the overflow problem, but also lets you edit program files that would be too big to edit in memory all at once.

The {$I} (Include) compiler directive allows you to divide your program into smaller sections for editing purposes. These sections of code are rejoined when the program is compiled. The compiler directive takes this format:

```
{$I filename.ext}
```

The file name must follow the standard conventions of your operating system (drive, path name, allowable characters, and so on). If no extension is specified, then .PAS is assumed; if you really want a file name with no extension, then use *filename.*, which explicitly puts a period (.) after the name.

Note: The Include compiler directive is not to be confused with the I/O compiler directive that enables/disables I/O checking. The I/O compiler directive is {$I+} to turn I/O checking on and {$I-} to turn I/O checking off (refer to Chapter 18 for more about file I/O). When the compiler sees a file name after the *I*, it knows to include that file.

MODULAR PROGRAMMING

A common programming practice is to put all global declarations into a separate file and pull them in via the Include option. Likewise, the procedures and functions can be grouped together in another file. Your main program might look like this:

```
program Whatever;
{$I whatever.def }                { pull in declarations }
{$I whatever.prc }            { procedures and functions }
  { note:  only 1 include directive is allowed on a line }
begin  { main body of program Whatever }
  Initialize;
  repeat
    GetCommand(Cmd);
    HandleCommand(Cmd);
    UpdateStatus
  until Done;
  Cleanup
end.   { of prog Whatever }
```

The file WHATEVER.DEF has all the **const**, **type**, and **var** statements for this program, and WHATEVER.PRC has all the subroutines (including *Initialize*, *GetCommand*, *HandleCommand*, *UpdateStatus*, and *Cleanup*).

Don't worry about keeping track of what you're editing and what you want to compile—the Turbo Pascal main menu makes it easy. Use the **M**ain file command and give it the name of the file containing your main program (such as WHATEVER.PAS). Now, you can use the **W**ork file command to freely select between WHATEVER.DEF, WHATEVER.PRC, and WHATEVER.PAS, editing whichever one

you'd like. When you compile, Turbo will automatically save the file you were working on before starting the compilation. What's more, if there is a compiler error, Turbo will bring in the correct file and go to where the error occurred.

Note that the main file, WHATEVER.PAS, contains all the Include file statements; Include modules are not allowed to include other modules. Don't let the idea of Include files confuse you. They are simply a way to keep most of the source code for a program on disk. This helps free up RAM, and it also makes it easy to adopt some good programming habits (keeps related routines in the same file, encourages the use of "libraries," makes it easy to find a routine—instead of searching through lines and lines of code, simply load the right module).

LIBRARIES

Once you start writing large programs, or even just a lot of programs, you can save yourself time and effort by creating *subroutine libraries.* A library is a collection of procedures and functions (with any accompanying declarations) you can use repeatedly in different programs.

Using a library has a number of advantages. First, it allows you to solve a certain problem, then go on to your debugged routines. Second, your programs will tend to be more consistent with one another, performing the same functions in the same way. Third, you'll be able to write programs more quickly, since much of the "grunt work" will already be done. Fourth, you'll save disk space, since one file of routines may be included in several programs.

Note: Although source code libraries aid in development speed, they must be recompiled for each program that uses them. You can include libraries just like any other Include file, although you usually place the Include statements right after the **program** statement so that subsequent declarations and routines can make full use of the libraries.

USING OVERLAYS TO SAVE CODE SPACE

Libraries, Include commands, and .COM files can all help to solve the problem of insufficient memory during compilation (called compiler overflow). But what if the resulting program is just too big? What if your .COM file, together with the data structures you declare, exceed your available memory? If this happens, then you have memory overflow, and the previously described methods offer no solution because they do not affect the size of the generated program code.

However, one method for saving code space is to use *overlays.* Overlays are portions of a program that are loaded into the same area of

memory, but not at the same time. Since only one chunk is loaded at any one time, the program only has to set aside enough memory to accommodate the largest one. For example, if you have five procedures occupying a total of 20K and the largest is only 6K in size, then you've reclaimed a total of 14K. Consequently, you can only save code space if there is more than one overlay procedure.

In Turbo Pascal, overlay groups are formed by sets of procedures and functions. Each overlay group has its own area of memory; the procedures and functions found in that group share that area in memory. This is an important point: The procedures and functions in a given group all compete for that area of memory, thus only one can be loaded in at any time. It also means that subprograms in an overlay group must not call each other (even indirectly), since the calling routine must be flushed out before the called routine can be loaded in. This would wipe out the rest of your calling routine and prevent you from continuing.

To create an overlay group, you must group together all procedures and functions into one part of the program. Each subprogram must have the key word **overlay** at the start of its procedure statement, like so:

```
overlay procedure Initialize;
  { declarations }
begin
  ...
end;  { of proc Initialize }

overlay procedure Pilotage;
  { declarations }
begin
  ...
end;  { of proc Pilotage }

overlay procedure MoveInShip;
  { declarations }
begin
  ...
end;  { of proc MoveInShip }

overlay procedure DoRepairs;
  { declarations }
begin
  ...
end;  { of proc DoRepairs }

overlay procedure HandlePod;
  { declarations }
begin
  ...
end;   { of proc HandlePod }

overlay procedure Cleanup;
```

```
  { declarations }
begin
  ...
end;   { of proc Cleanup }
```

The six procedures—*Initialize*, *Pilotage*, *MoveInShip*, *DoRepairs*, *HandlePod*, and *Cleanup*—form one overlay group. Only one of the six can be in memory at any given time. The amount of memory set aside is that needed by the largest of the six. An overlay group ends when the first non-overlay procedure (or the main body of the program) is encountered. You can have multiple overlay groups, separated from one another by regular subprograms (which can be "do nothing" procedures, such as a procedure statement **begin** and **end**). Note that procedures in one overlay group can call a procedure in another overlay group.

When you compile a program with overlay groups, each group is written out to a separate file. The first group is written to the file ⟨filename⟩.000, the second to ⟨filename⟩.001, and so on, up to ⟨filename⟩.099. These files are created whether or not you've selected the .COM-file option; otherwise, you couldn't test your program while still in the Turbo environment.

To get a given procedure from an overlay group into memory, you must call it. If it's not in memory, it'll be loaded in, overwriting whatever procedure from the group that is currently in memory. This means that you'll probably want to design your program to avoid constant, successive calling of procedures in the same overlay group; otherwise, the program will be constantly loading in those procedures from the disk. For example, the main body of the previous program might look like this:

```
program StarShip;
{ declarations }
type
  States = (InPilotage,InShip,InRepair,InPod,GameOver);
var
  GameState : States;
  { more declarations, including overlay group }

begin  { main body of program StarShip }
  Initialize;
  repeat
    case GameState of
      InPilotage : Pilotage;
      InShip     : MoveInShip;
      InRepair   : DoRepairs;
      InPod      : HandlePod
    end
  until GameState = GameOver;
  Cleanup
end.   { of program StarShip }
```

This program uses the overlay group we've already defined to break up the major functions into manageable chunks and keep only one chunk in memory at any one time. The global variable *GameState* controls which ship function you're running at the moment.

Menu Program Example One: Overlays

Since many programs are menu-driven, let's look at how to overlay a menu program. A simple method is to have each menu option carried out by an overlay procedure that has all of its utility routines nested inside of it. These subroutines are considered part of the overlay procedure and are all loaded simultaneously when the option is selected. The following sample program has two options: to enter data and to generate reports.

```
program MENU;
{$I Share.def} { type, variable and procedure definitions }
                     { available to both overlay procedures }
overlay procedure DataEntry;

procedure InitDataValues; { nested procedure }
begin
  ...
end;  { InitDataValues }
  ...
procedure GetUserData; { nested procedure }
begin
  ...
end;  { GetUserData }

begin  { DataEntry }
  InitDataVals;
  ...
  GenUserData;
end;  { DataEntry }

overlay procedure PrintReports;

procedure InitPrintVals;  { nested procedure }
begin
end;  { InitPrintVals }

procedure GenerateReports  { nested procedure }
begin
  ...
end;  { GenerateReports }

begin { PrintReports }
  InitPrintVals;
  ...
  GenerateReports;
end;  { PrintReports }
```

```
var
  choice: char;
begin   { main }
  repeat
    GetChoice(Choice);  { Prompts the user     }
    case Choice of      { and reads the choice }
      'D': DataEntry
      'P': PrintReports;
    else;
    end;
  until Choice = 'Q'
end. { Menu }
```

This method is simple and effective for a number of reasons. First of all, you only need to overlay as many procedures as there are user options. Second, only one overlay file is created. Third, you generally do not have to worry about different menu selections calling each other. And fourth, program execution is slowed by disk access only when the option is selected (and not even then if the option was chosen previously), rather than at various points in the option's execution.

Note: This program only generates one overlay file, yet saves a great deal of code space. Remember, it is not the total number of overlay files (or groups) that is important, rather, it is how much space can be saved in a given overlay file.

Location of Overlay Files

Normally, Turbo Pascal expects overlay files (⟨filename⟩.000, and so on) to be on the logged drive. If you're using version 2.0 (or later) of MS-DOS/PC-DOS, then the logged path name is used.

You can, however, tell your program where to look for the overlay files. If you're using MS-DOS/PC-DOS, then you can use the built-in procedure *OvrPath(PathName)*, where *PathName* is a string giving the drive and/or directory path name where the overlay file is to be found. The string '.' selects the current logged drive.

In the same way, you can use the procedure *OvrDrive(DNum)* to specify which drive to use when running under CP/M or CP/M-86. *DNum* is an integer value indicating the drive: 0 = logged drive, 1 = A:, 2 = B:, and so on.

Overlay Restrictions

In Chapter 18 of the *Turbo Pascal Reference Manual*, there is a discussion of overlays and some of their restrictions. Most notably, it is stated that you cannot have recursive procedures within overlay groups (something else to watch for if you're building libraries). The reference manual suggests making a regular subroutine recursive and have it call the overlay procedure. You're probably better off finding a non-

recursive implementation (remember *QuickSort* in Chapter 22 of this manual), or moving the recursion out of the overlay group altogether.

Likewise, you can't declare an overlay procedure as **forward**, though the solution does work easily. Simply declare a regular subroutine as **forward**, declare the overlay procedure, then put in the original (**forward**) subroutine. For example, suppose you have two procedures in two different overlay groups that need to be able to call each other. You might set up something like this:

```
procedure CallTest2; forward;
overlay procedure Test1;    { start of first overlay group }
begin
  ...
  CallTest2;
  ...
end;  { of overproc Test1 }

procedure Dummy;    { Dummy to end the first overlay group }
begin
end;

overlay procedure Test2;  { start of second overlay group }
begin
  ...
  Test1;
  ...
end;  { of overproc Test2 end of second overlay group }

procedure CallTest2;
begin
  Test2
end;  { of proc CallTest2 }
```

Since Test1 is declared before Test2, there's no problem with Test2 calling Test1. However, the procedure *CallTest2* is declared **forward** before Test1 and actually created after Test2. That way, Test1 can call *CallTest2*, and *CallTest2* can call Test2. Simple, huh? Remember, though, that this assumes that Test1 and Test2 are in two separate overlay groups. If they're in the same group, they can't call each other regardless of what you do. (Actually, they can call each other, but your program may die a quick and messy death.)

CHAINING

There are alternatives to overlays. For example, your program may actually be several distinct and independent programs (each of which can be 64K) sharing the same data (and data structures), but with the need to be run at different times. Turbo Pascal lets you *chain* between programs.

Chaining lets a currently running program load another program in its place. So if you're running the program Test1 and chain to the program Test2, all of Test1 is thrown away, Test2 is loaded into memory, and execution starts at the top of the main body of Test2. Likewise, if Test2 chains back to Test1, Test1 is pulled in while Test2 is discarded, and execution starts at the top of the main body of Test1. (Note execution of Test1 does *not* begin where Test1 had previously chained to Test2.)

If you want to run a group of chained programs, the first program executed must be a regular .COM file (that is, one created using the **C**om-file command under the compiler **O**ptions menu). All others must be .CHN files, created by the c**H**n-file option in the same menu. .CHN files omit library routines (about 12K in size) and so are generally much smaller. The runtime routines are loaded when the .COM file is executed and remain in memory as each .CHN file is loaded in.

While chained programs don't share code (except for the runtime routines), they can share data if you declare things properly. The global declarations in each program are created in the same area of memory, in the order of declaration and are not initialized (except for typed constants). So, if you have the identical set of global variable declarations at the start of each chained program, then those variables will retain their values during the chaining process. The easiest way to ensure this is to declare said variables in a separate file, then include that file at the start of each program:

```
program Test1;
{$I test.def }
  ...
end. { of program Test1 }

program Test2;
{$I test.def }
  ...
end. { of program Test2 }
```

Note that the file TEST.DEF can include all types of declarations: constants, types, variables, subprograms, and so on. Be aware, though, that any typed constants used by chained programs will be reinitialized.

For one program to chain to another, you must first assign the .CHN file to a file variable (which can be of any type), then chain to it. You might want to define something like this:

```
type
  FileStr : string[80];
procedure ChainTo(FileName : FileStr; var IOCode : Integer);
var
  CFile  : file;
```

```
begin
  Assign(CFile,FileName);
  {$I }
  Chain(CFile);
  {$I+}
  IOCode := IOResult
end; { of proc ChainTo }
```

This routine will attempt to chain to the file name passed to it. If an error occurs, it will return an I/O error code (0 = no error) in the parameter *IOCode*; otherwise, contol will be passed along to the chained program.

One immediate caution to note: The procedure *Chain* can only chain to a .CHN file; you cannot chain back to the original (.COM) file. This is important to remember, especially if you want to chain back and forth between programs. (Read the section "Executing Files" for a way around that restriction.)

One more caution is in order. When you select c**H**n mode on 16-bit systems, another menu will appear allowing you to set code and data segment sizes and specify minimum and maximum memory size required. You can ignore this for now. However, after each .CHN file has compiled, make a note of the code and data printout that comes up on the screen. When you have finished compiling all the .CHN files, select the **C**om option to compile your main, executable program. This time when the menu appears, you must set

```
minimum cOde segment size:    XXXX paragraphs
minimum Data segment size:    XXXX paragraphs
mInimum free dynamic memory:  XXXX paragraphs
```

If the code or data is smaller than any of the .CHN values, choose the largest value from the data acquired when compiling the chain files. When you enter these values the compiler will add overhead space and return the computed values. **Note**: The module with the largest code segment size may not necessarily have the largest data size, so pick the largest value for each code and data segment.

EXECUTING FILES

The *Execute* procedure works exactly like the *Chain* procedure, with one important distinction: The file being executed must be a Turbo Pascal .COM file, rather than a .CHN file. This means the executed program can be run as a standalone program, which may or may not be an advantage. **Note:** You may have to set the minimum code and data segment sizes discussed in the previous section when using *Execute* as well.

Menu Program Example 2: Chain and Execute

In the section on overlays, we showed you how to save code space in a menu-driven program. In this example, we'll show you how to save code space using *Chain* and *Execute*. *Chain/Execute* is especially tailored to the structure of a menu program, since each option is usually a distinct self-contained task. The following shows how you can chain back and forth between programs. Menu serves as the main program, chaining to the separate programs DataEntry and GenReports:

```
program Menu;
{$I Share.DEF }

var
  Choice:char;
  OptionFile: File;
  ...

begin
  repeat
    GetChoice(choice);          { Prompt the user and }
    case choice of              { read the selection   }
      'P': Assign(OptionFile,'GenReport.CHN');
      'D': Assign(OptionFile,'DataEntry.CHN'),
      else;
    end;
    if Choice in ['D','P'] then
      Chain (OptionFile);
  until Choice = 'Q';
end.

program DataEntry;
{$I Share.Def }
  ...
procedure InitDataValues;
begin
  ...
end;

procedure GetUserData;
begin
end;

var
  MainFile: File;
begin
  GetUserData;
  Assign(MainFile,'Menu.COM');
  Execute(MainFile);        { re-execute main menu program }
end. { DataEntry }
```

```
program GenReports;
{$I Share.Def }
procedure InitPrintVals;
begin
  ...
end;
  ...
procedure GenerateReports;
begin
  ...
end;

var
  MainFile: File;
begin
  GenerateReports
  Assign(MainFile,'Menu.Com');
  Execute(MainFile);
end.  { GenerateReports }
```

In program MENU.COM, when an option is selected it chains to the corresponding .CHN program (DataEntry or GenReports). The chain program completes its task and re-executes the main program. This is necessary because, unlike a subroutine call, when the task is completed the control of the program does not automatically return to the calling procedure. Instead the main program is re-executed from the beginning. However, since our main program is a repeat loop, it doesn't matter if you restart it at the top of the loop.

OVERLAY VS. CHAIN/EXECUTE

We have shown you two ways to implement a menu-driven program, and each has its strong and weak points. The overall solution generally uses a smaller number of files since all the options are in one overlay file. Also, if you choose the same option repeatedly, an overlay program will probably run faster than a menu-driven one because the option doesn't need to be loaded in from disk. Also, in the *Chain/Execute* solution, the main program must be loaded in from disk when it is re-executed, further slowing execution time.

Non-overlay utility routines can be used by several overlay procedures, reducing the amount of duplicated code. In *Chain/Execute* on the other hand, utility procedures must be duplicated in each .CHN file.

Chain/Execute has the advantage in compile time. Once you have compiled the .CHN programs, they do not have to be recompiled with the main program. Also, a *Chain* file need only be recompiled as a .COM program in order to run as a stand-alone program. It is also easier to incorporate this module into a different menu-driven program.

REVIEW

You can use overlays, the heap, and Include modules to create large, fast Turbo Pascal programs. The heap and stack can be used to augment the static data storage area, overlays (and, less commonly, chaining) can be used to supplement the code area, and include files serve the dual purpose of allowing modular, portable programming while freeing up RAM previously used for storing text for the editor.

24 Typed Constants

In Chapter 7, we introduced the concept of a Pascal constant. As you may recall, constants are fixed values associated with identifiers and have the following properties:

- They must be of a scalar type (the one exception being string constants, which are strings of characters).
- They cannot be changed during the course of a program.

Many other computer languages, such as C, do not restrict constants to string and scalar types. In fact, constants of "structured" types—especially arrays and records—can be very useful in many programs.

For instance, suppose you were writing a program to play a game of cards. You might define the types:

```
type
  SuitType = (Clubs, Diamonds, Hearts, Spades);
  RankType = (Ace, Deuce, Three, Four, Five, Six, Seven,
              Eight, Nine, Ten, Jack, Queen, King);
  Card = record
    Suit : SuitType;
    Rank : RankType;
  end;
```

Now suppose you wanted a constant value in your program for the Ace of Spades, so that you could make a card the Ace of Spades, or test to see whether a card has that suit and rank. In standard Pascal you could not define such a constant and would be forced to work on the individual fields of the record:

```
ThisCard.Suit := Spades;
ThisCard.Rank := Ace;
```

In Turbo Pascal, however, you can use a structure called a *typed constant* to make the manipulation of records easier to program and understand. You could define the typed constant as

```
const
  AceOfSpades : Card = (Suit : Spades; Rank : Ace);
```

Then, to make the variable *ThisCard* become the Ace of Spades, you could simply write

```
ThisCard := AceOfSpades;
```

Typed constants are a Turbo Pascal specialty. In this chapter, we'll explore the properties and uses of this powerful feature.

DEFINING A TYPED CONSTANT

To define a typed constant, you must place a typed constant definition in the constant definition part of your program. The syntax of a typed constant definition is shown in Figure 24-1.

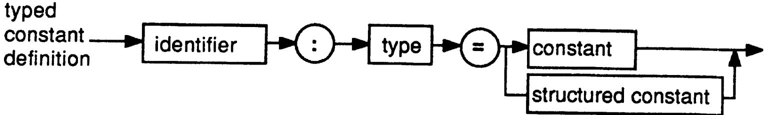


Figure 24-1 Syntax Diagram of Typed Constant Definition

Note that this definition looks like a combination of a variable declaration and a constant definition. Like a variable declaration, it specifies explicitly the type of the constant (hence the name "typed constant"). Like a constant definition, however, it specifies the value of the typed constant.

The value of a typed constant can be a value you might give an ordinary constant, that is, a scalar or a string of characters. It can also be a *structured constant*, a constant specifying the fields of a record, the elements of an array, or the members of a set. The syntax diagrams showing the notation for a structured constant are displayed in Figures 24-2 through 24-5.

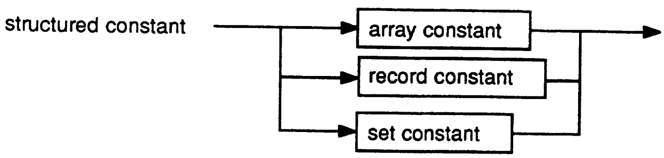


Figure 24-2 Syntax Diagram of Structured Constant

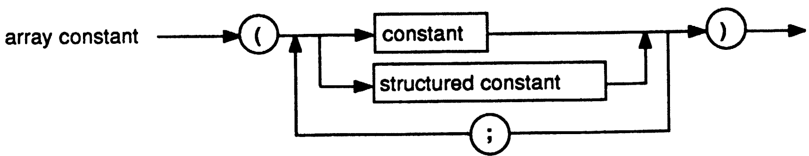


Figure 24-3 Syntax Diagram of Array Constant

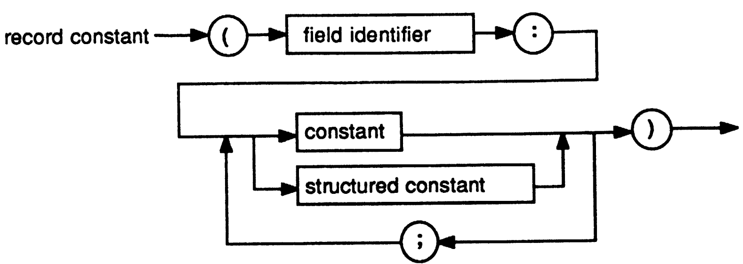


Figure 24-4 Syntax Diagram of Record Constant

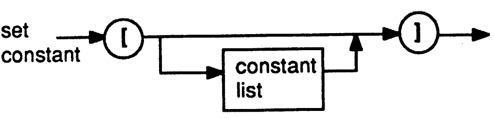


Figure 24-5 Syntax Diagram of Set Constant

Each kind of structured constant (array, record, and set) merits detailed explanation. Let's begin with the simplest, the *array constant.*

ARRAY CONSTANTS

An array constant consists of a constant for each element of an array, separated by semicolons. These constants may be of any type, even structured types, except a *file* or *pointer type* (since there is no way to describe constants of these types in Turbo). For instance, suppose you were to declare

```
type
  LaundryType : (Socks, Shirts, Blouses, Pants, Skirts,
                 Ties, Jackets);
```

You could then specify a constant describing a laundry list by writing

```
const
  LaundryList : array [LaundryType] of
    integer = (3, 2, 4, 1, 8, 1, 1);
```

The preceding constant would then describe a laundry list containing 3 socks (oops, guess an odd one got in there somehow!), 2 shirts, 4 blouses, 1 pair of pants, 8 skirts, 1 tie, and 1 jacket. In the special case of arrays of characters, either the previous notation or a string notation can be used. Thus, the two definitions that follow are equivalent

```
const
  Digits : array [0..9] of char =
                ('0','1','2','3','4','5','6','7','8','9');
const
  Digits : array [0..9] of char = '0123456789';
```

RECORD CONSTANTS

A *record constant* contains constants giving values for each of the fields of a record. As with the elements of an array constant, these fields may have values of any type except file or pointer types.

Here is an example of a record constant. If you were to define

```
type
  Point : record
            X, Y, Z : integer;
          end;
```

you could then define a typed constant as

```
const
  Here : Point = (X : 0; Y : 14; Z : 23);
```

Note that in a record constant definition you must specify in order the names of the fields in which the constant values are to be placed. This allows you to specify the fields of a variant record, if necessary. Remember if your variant record type contains a tag field, a value must be specified for that field.

SET CONSTANTS

A *set constant* has a syntax similar to that of a set constructor (described in Chapter 16), except that it can contain only constants. A set constant can only contain constants of the base type of the set.

Here are some examples of set constants:

```
type
  CharSet : set of char;
  DaySet  : set of (Monday, Tuesday, Wednesday, Thursday,
                    Friday, Saturday, Sunday);
```

```
const
  Vowels : CharSet = ['A', 'E', 'I', 'O', 'U', 'Y',
                      'a', 'e', 'i', 'o', 'u', 'y'];
  Days   : DaySet  = [Monday, Wednesday, Friday..Sunday];
```

Note that subrange notation is allowed in set constants; in the previous example, we have used it to save typing.

SPECIAL PROPERTIES OF TYPED CONSTANTS

We've already discussed the most common and obvious use of typed constants: to create constants of structured types. When used for this purpose, typed constants can be treated exactly as ordinary constants. Unlike ordinary constants, however, Turbo Pascal typed constants have a number of unique properties that make them even more useful.

Mutability

The most important (and perhaps surprising) property of typed constants is that *the value of a typed constant may be changed during execution.* When a program is first loaded into memory and begins running, the typed constants in that program have the values given in the typed constant definition. However, if a typed constant is changed (either by assignment or by a procedure or function called with it), it will assume the new value *as if it were a variable.*

While this property of typed constants may at first seem odd or even dangerous, it is extremely useful, as we shall see in later sections.

Lifetime

Another important property of typed constants is that their lifetime (as defined in Chapter 12) is equal to the total time the program runs, even if they are defined in subprograms. In other words, if a typed constant is defined in the declaration part of a procedure or function, and that subprogram stores a value in the typed constant, the same value will be there the next time the procedure is called. This contrasts sharply with the behavior of local variables, which "lose" their values when the subprogram in which they are defined is exited.

An object whose lifetime is the entire length of the program is said to be *static.* Typed constants, like variables declared in the declaration part of a main program, are static objects.

Scope

The third important property of typed constants (which only shows when they are defined within subprograms) is that recursion does not

cause new instances of them to be created as it does with local variables. Consider, for example, the following program:

```
program TCDemo;

procedure Recurse;
const
  A : integer = 0;
var
  B : integer;
begin
  B := A;
  A := Succ(A);  { We can assign to A as if it were an
                   integer variable }
  if A < 10 then Recurse;
  Writeln (A:3, B:3);
end;  { Recurse }

begin { TCDemo }
  Recurse;
end.  { TCDemo }
```

When run, this program will call the procedure *Recurse*, which will recurse 10 times. Can you tell what will be printed when this program is run, before going on to the next paragraph?

The correct sequence output by the program is

```
10  9
10  8
10  7
  .
  .
  .
10  1
10  0
```

Why does *A* retain its value after each recursive call, while *B* doesn't? The answer is that a new *B* is created each time the procedure *Recurse* is called, and the old one is saved until after the call is finished. This is not the case for typed constants, however—all references to *A* refer to the same *A*.

TYPED CONSTANTS AS STATIC VARIABLES

As stated earlier, all variables declared within a procedure or function "lose" their values when the procedure or function is exited. In standard Pascal, which does not have typed constants, this "feature" makes it impossible to make a procedure or function "pick up where it left off" from a previous call—unless, of course, global variables are declared specifically for use by that subprogram.

Such global variables can make a program harder to understand, since they must be declared far from where the subprogram is declared

(usually at the beginning of the program). Typed constants, defined within the subprogram where they are used, avoid this confusion and let you write procedures like the *Counter* procedure that follows:

```
program CounterDemo;
var
  I : integer;

procedure Counter;
const
  Count : integer = 0;
begin
  Writeln (Count);
  Count := Succ(Count);
end;

begin { CounterDemo }
  for I := 1 to 10 do
  Counter;
end. { CounterDemo }
```

In this program, the procedure *Counter* is able to remember where it was in its counting sequence because it stores the current count in a typed constant.

TYPED CONSTANTS AS INITIALIZED VARIABLES

In a Pascal program, all of the variables are undefined from the time a program starts running until they are assigned a value. Of course, most of the time we want at least some of our variables (counters, for instance) to start out with known values. For this reason, many Pascal programs start out with an initialization procedure containing large numbers of assignment statements to set up all of these values.

Unfortunately, this mass of assignment statements, confined to a separate procedure far from where the variables are used, can be very confusing. (Where is the declaration of the initialized variable? How can you tell if a variable has been left out?) Because Turbo Pascal typed constants let you specify the initial value next to the definition itself, this "scattering" of information is minimized. And, because no assignment statements need be executed (the value of the typed constant is read directly from disk), the program is smaller and faster as well.

MANIPULATING COMPONENTS OF CONSTANTS

One idiosyncrasy of standard Pascal is that it treats constants as indivisible objects, even if they are naturally divisible into smaller pieces. For

instance, if you try to use an array index to pull the third character out of a string constant, the compiler will complain. As a demonstration, suppose we define a constant string:

```
const
  Twister = 'Peter Piper picked a peck of pickled peppers.';
```

and wish to send it, character by character, to a modem. Suppose also that we have a procedure called *Modemout*, which takes a character as an argument and sends it to the modem. A seemingly "obvious" way to program such a task might be:

```
var
  I: integer;
      :
      :
  for I := 1 to Length(Twister) do
  Modemout(Twister [i]);
```

If you were to compile this example with Turbo, however, the compiler would stop at the last line shown, with the message:

```
Error 5: ')' expected. Press <ESC>
```

Upon pressing the Esc key, you would return to the editor with the cursor on the [. Why? Well, the Pascal language in general doesn't allow you to manipulate the components of constants. In this case, a subscripting operation is being used to pick out a single element of a constant string. But since ordinary constants are treated by Pascal as whole and indivisible objects, this is not allowed.

The answer to this problem, as you may have already guessed, is to replace the "ordinary" constant with a typed constant. If the constant definition given earlier is changed to

```
const
  test : string [45] = 'Peter Piper... etc.
```

then the program will work as expected.

SAVING CONSTANT SPACE

In standard Pascal, constants frequently waste space. Whenever they are used, they are simply inserted as a whole directly into the compiled code. Thus, if you were to make the declaration:

```
const
  Hello = 'Hello, world, my name is Joe';
```

and then wrote, as part of your program:

```
      :
      :
Writeln(Hello);
      :
Writeln(Hello);
```

```
       :
Writeln(Hello);
       :
       :
```

a new copy of the whole string, "Hello, world, my name is Joe" would be inserted into the compiled program as part of each one of these statements. In a large program, with many messages, much valuable memory can be wasted this way. Fortunately, typed constants again come to the rescue. If the first declaration given is replaced with

```
const
  Hello : string[30] = 'Hello, world, my name is Joe';
```

then each of the *Writeln* statements will not generate a new copy of the string; instead, the one instance of the typed constant is used for all the calls to *Writeln*.

PASSING CONSTANTS AS VAR PARAMETERS

Another use of typed constants is to pass a "constant" value to a procedure as a **var** parameter. Turbo Pascal, like all versions of Pascal, cannot permit an object that cannot be changed (that is, an ordinary constant) to be passed as a **var** parameter to a procedure or function, since the subprogram needs to be able to change such a parameter if needed. Because typed constants can be changed, they can be passed as **var** parameters. This feature can save both space and time when large structures (such as long strings) are passed as parameters to a subprogram.

As mentioned in Chapter 12, when a non-**var** parameter is passed to a subprogram, a working copy is made of that parameter before the subprogram starts to execute. This copying process can use large amounts of memory (for the copy) and large amounts of time (to make it), and can be avoided entirely if the parameter is a **var** parameter. Using typed constants, rather than ordinary constants, allows this option.

HOW TYPED CONSTANTS ARE STORED IN MEMORY

Turbo Pascal typed constants work the way they do because they are stored in the same part of memory as the compiled code of your program. When your program is loaded in from the disk (or processed by the compiler if you are running the program from memory), the proper values are put into these locations before the program even starts running. By contrast, your program's global variables are not accessed until the program starts running, and the place where a local variable is stored is not used until the subprogram in which it is defined is called.

Thus, typed constants are "ready to use" from the moment the program is loaded into memory and need not be initialized at all. Also, they occupy well-defined memory locations (unlike ordinary constants, which can't be said to have an address at all), so they can be passed as **var** parameters or altered, if necessary.

A FINAL CAVEAT: TYPED CONSTANTS AND EXECUTION FROM MEMORY

Because of the storage scheme previously mentioned, it is important to take care when running programs that change typed constants from memory (instead of from a .COM or .CMD file). Why? Well, as we said before, typed constants are set to their initial values when the program is loaded in from the disk, or when it is compiled into memory. They are not, however, reset to their initial values when a program is *re-run* from memory. Thus, if you compile a program to memory, run it once (altering a few typed constants), then run it a second time from memory without recompiling, the typed constants *will not be re-initialized.* Rather, they will start out with the values they had at the end of the previous run of the program, and may produce erratic results.

For this reason, we recommend *not* running programs that alter the values of typed constants from memory, unless you remember to always recompile between runs.

REVIEW

Turbo Pascal's typed constants are a powerful extension to the Pascal language, allowing you to define constants of *any* type. They are also useful as pre-initialized static variables, and also as a way of saving space in a Turbo Pascal program. Typed constants can be passed as **var** parameters to a subprogram, and allow full access to their components. Because they are stored in the section of memory normally reserved for program code, typed constants are only initialized when a program is loaded or compiled into memory. Care must be taken when running a program that alters its typed constants from memory rather than from disk.

25 The Goto Statement

In this chapter, we'll take a brief look at a statement type that is rarely used in Pascal programs—the **goto** statement. A **goto** statement simply says a program should continue executing with another statement somewhere else in the program text, a place marked by a special construct called a *label.*

SYNTAX OF THE GOTO STATEMENT

The syntax of the **goto** statement is simple: It consists of the reserved word **goto**, followed by a number or identifier that has been declared as a label in the declaration part of the currently executing block (program or subprogram). **Goto**s cannot be used to jump outside of procedure bounds. Its syntax diagram is shown in Figure 25-1.

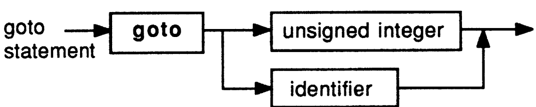


Figure 25-1 Syntax Diagram of Goto Statement

A label marks the destination of a **goto** statement and must be declared in a label declaration part, which is in turn included in the declaration part. The syntax of a label declaration part in Turbo Pascal is depicted in Figure 25-2.

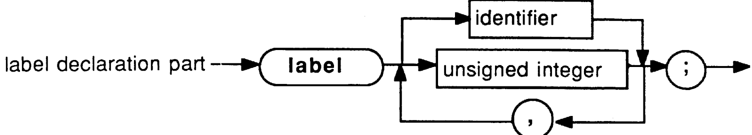


Figure 25-2 Syntax Diagram of Label Declaration Part

In standard Pascal, labels cannot be identifiers; they must be unsigned integers only. Turbo allows labels to be identifiers, to describe the places in the program to which they refer.

HOW TO USE THE GOTO STATEMENT— AND WHY NOT TO

In early programming languages, such as BASIC and FORTRAN, there were far fewer "structured" statements (such as **if**, **while**, **repeat**...**until**, and **for**) than are available in Pascal. For instance, the original FORTRAN language had nothing similar to the **else** clause in its version of the **if** statement. Here are two versions of a program fragment, one written in Pascal, the other in FORTRAN. See if you can figure out what the FORTRAN version means:

Pascal

```
if A < 27 then
  A := A + 2
else
  A := A + 1;
```

FORTRAN

```
  IF A .LT. 27 THEN GOTO 1
  A = A + 1
  GOTO 2
1 A = A + 2
2 CONTINUE
```

Since we don't expect you to understand FORTRAN, we'll run through the FORTRAN example briefly. The first line is FORTRAN's version of the **if** statement, which does something only if a condition is TRUE. In this case, if the value of the variable *A* is less than 27 (.LT. is FORTRAN's way of saying "less than"), execution continues on the line with the number 1 (a label) before it; otherwise, the statement $A = A + 1$ is executed. (FORTRAN uses = for assignment; Pascal uses :=.) Then the computer executes the statement GOTO 2, which jumps to the statement with the word CONTINUE in it. (The CONTINUE statement in FORTRAN does nothing; it is merely a placeholder for a label.)

Note how much clearer the Pascal version is. Instead of thinking in terms of labels and jumping to them, you can think in terms of **if**... **then**...**else** conditions in the program. Because of the descriptive power of Pascal's structured statements, you have no need to use **gotos** and labels.

However, there are a few places where using a **goto** is helpful in Pascal. One instance is when you must exit from the middle of a loop, such as a **for** or a **while**. For instance, suppose we had the loop:

```
for I := 1 to 10 do { Get 10 numbers }
begin
  Write('Enter a number ( < 0 to stop ): ');
  Readln(Number[I]);
  if Number < 0 then
    goto NoMore;
end;
NoMore:
```

This loop normally gets 10 numbers from the user, but can be terminated at once if the user enters a negative number. Even in this case, it is better programming style to place the loop in a procedure and use *Exit* to leave the procedure (as well as the loop) if you need to terminate early.

Here's another example of how confusing just a few **goto** statements can be. The following program contains only 4 labels and 4 **gotos**, but can you tell what it writes to the screen?*

```
program Spaghetti;
label
  One, Two, Three, Four;
var
  A : Integer;
begin
  A := 0;
  One: if A > 3 then
    goto Three;
  Two: A := A + 5;
    goto Four;
  Three: A := A + 3;
    goto Two;
  Four: if A mod 3 <> 0 then
    goto One;
  Writeln(A);
end.
```

As Niklaus Wirth and Kathleen Jensen said in *The Pascal User Manual and Report*: "The presence of **gotos** in a Pascal program is often an indication that the programmer has not yet learned 'to think' in Pascal (as this is a necessary construct in other programming languages)."

We agree, and encourage you to build your programs without using the **goto** statement.

* *Answer:* 21

REVIEW

Though **goto**s are rarely used in Pascal, there are a few instances where they prove useful. Overall, though, you are better off discovering the ins and outs of Pascal rather than resorting to the use of **goto**s.

26 Absolute Variables and Untyped Parameters

Standard Pascal was originally intended as a teaching language and therefore omitted facilities students would be unlikely to need. Turbo Pascal, however, was designed as a language for serious programmers as well as for students, and includes special features to make certain programming tasks easier. Two of the most important of these are ***absolute** variables*, which reside at any address you specify in memory, and *untyped parameters*, formal parameters that allow a subprogram to accept a variable of any type as an actual parameter.

ABSOLUTE VARIABLES

Normally, when you compile and run a Turbo Pascal program, the compiler and the operating system of your computer make decisions about where data will be stored in memory. (This is generally not true in assembly language programming, where the programmer must specify where everything goes.) While this automatic allocation is exactly what you need 99.9 percent of the time, there are a few instances where you might want to tell the compiler to place a variable in a specific spot, such as in the same place as another variable, or at a fixed place in memory.

Turbo allows you to do both by providing you with **absolute** variables, variables whose "absolute" address is specified explictly in the variable declaration. Using **absolute** variables, you can exchange information directly with your computer's operating system and make quick use of untyped parameters (described in the next section).

How do you declare an **absolute** variable? The syntax is the same as the declaration of an ordinary variable, except that you must add the reserved word **absolute** and a numeric address (or an identifier) after the variable's type. If you specify a numeric address, then the variable is considered to be at that location. If you specify an identifier (which

must be the name of a variable or a typed constant), then the variable you are declaring will reside at the same address as the object specified.

The syntax diagram for the variable declaration part (shown in Figure 26-1) includes the syntax for declaring an absolute variable. (The syntax diagram of an address is shown in Figure 26-2.)

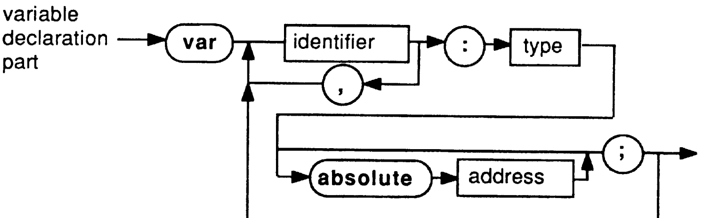


Figure 26-1 Syntax Diagram of Variable Declaration Part

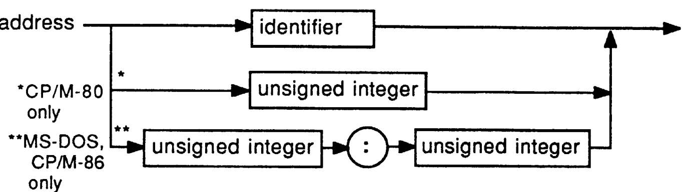


Figure 26-2 Syntax Diagram of Address

(Note that for 16-bit (MS-DOS and CP/M-86) processors, an address consists of two unsigned integers; on 8-bit (CP/M-80) processors an address is just one unsigned integer.)

When you use **absolute** variables, you will most often use them to make two variables occupy the same space (we'll demonstrate this in the "Untyped Parameters" section). The following two examples of **absolute** variables are used to access system functions.

In CP/M-80 systems, the operating system resides in the high part of memory, and your program starts near the bottom. Many programs (including Turbo Pascal itself) want to use as much memory as possible. They must therefore find where CP/M's lowest memory location is and avoid writing above it.

You can look at that address by defining the variable

```
var
  HiMem : Integer absolute 6;
  { Address of first byte of CP/M appears at
    absolute location 6 in memory }
```

On an IBM PC system running PC-DOS, many locations in the lowest part of memory contain useful information. Two of these are storage locations where the computer remembers what time it is.

```
var
  ClockLow : Integer absolute $0040:$006C;
             { Low word of tick count  }
  ClockHi  : Integer absolute $0040:$006E;
             { High word of tick count }
```

Together, these locations form a 32-bit counter that changes 18.2 times a second. Turbo Pascal uses this counter in the built-in procedure *Delay(ms)* to measure delay times; your programs can use it for this purpose as well. Special areas of memory like these can contain a wealth of useful data on current activity in your system. For more information on the specific addresses used by your computer, see the documentation for your hardware and operating system.

UNTYPED PARAMETERS

When declaring a formal parameter for a procedure or a function in standard Pascal, you *must* declare a type for that parameter. By doing so, you give the compiler important information about how the parameter can be used within the subprogram.

Sometimes, however, you may want to write a subprogram to take an actual parameter of *any* type, such as Turbo's *FillChar*, which fills any variable with a number of bytes all of the same value.

Or you may want to write a "universal" string function, one that operates on any string (regardless of its length) without requiring the user to invoke the {$V-} directive. (This directive, described at length in Chapter 14, "relaxes" type checking on strings passed as **var** parameters.)

These and other operations are made possible by Turbo's untyped parameters. To create an untyped parameter, you can declare a **var** parameter to a subprogram, omitting the colon and the type specification that normally follow, like so:

```
procedure TrimTrailing(var Str);
```

Of course, you need not limit yourself to one parameter, or one untyped parameter. A procedure can have any number of untyped parameters, and they can be mixed in with ordinary typed parameters:

```
procedure SwapVars(var Var1, Var2; Size : Integer);
```

Note that both *Var1* and *Var2* are untyped variable (**var**) parameters, while *Size* is a value parameter of type integer. An untyped parameter cannot be a value parameter. The syntax diagram for a formal parameter list (shown in Figure 26-3) includes a path for untyped parameters. It is the shaded path that "detours" around the colon and type of a formal parameter.

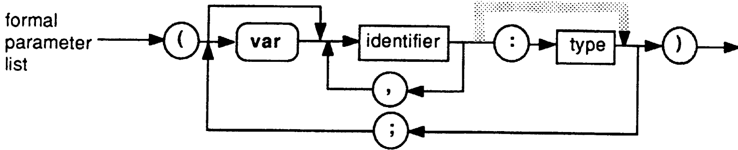


Figure 26-3 Syntax Diagram of Formal Parameter List

Using Untyped Parameters

Untyped parameters are visible within the subprogram, just like any other parameter. However, since they have no type, they are incompatible with anything that does. In fact, they cannot even be coerced to a scalar type through the mechanism discussed in Chapter 18. Nonetheless, their addresses are available through the *Addr* function and can be passed to Turbo procedures and functions that work on variables of any type (such as *BlockRead*, *BlockWrite*, *FillChar*, and *Move*). Also, because the address of an untyped parameter is passed to the subprogram at the time of the call, **absolute** variables can be made to reside at the same memory location as an untyped parameter. This allows the untyped parameter to be manipulated as if it were of any type at all.

Here are two examples that demonstrate the usefulness of untyped parameters. The first is a simple procedure that removes or "trims" any blanks present at the end of a string; the second, also a procedure, swaps the contents of any two variables regardless of size or type.

```
procedure TrimTrailing(var AnyString);
var
  Len : byte absolute AnyString;
        { The variable Len overlays the length
          byte of AnyString }
  St  : string[255] absolute AnyString;
        { The variable St overlays all of AnyString, and
          perhaps other things. This is OK, since only the
          length byte is changed. }
begin { TrimTrailing }
  while (Len > 0) and (St[Len] = ' ') do
     { If St ends with ' ' }
    Len := Pred(Len);              { Decrement its length }
end;  { TrimTrailing }
```

In this procedure, we define a **var** parameter *AnyString*, which we expect to be a string when the procedure is called. (If this doesn't occur, neither the compiler nor the program will detect the error.) We don't know the maximum size of the string to be passed, however, we *do* know that its length will be present in its first byte. We will only make

the string shorter—never longer—and will do this by changing the length byte. Because every string, regardless of size, has a length byte, it is safe to define an absolute variable of the type **string**[255] (the largest possible type of string) to overlay *AnyString*, as long as the other bytes of *AnyString* do not change. When called, the procedure simply reduces the length of the string until the last character is not blank (or the string is empty).

SwapVars, the second procedure, swaps the contents of two variables of a given size (up to Maxint).

```
procedure SwapVars(var Var1, Var2; Size : Integer);
type
  BigArray = array [1..Maxint] of Byte;
var
  V1    : BigArray absolute Var1;
          { Treat Var1 and Var2 like large }
  V2    : BigArray absolute Var2;
          { arrays of bytes for this move  }
  Count : Integer;
          { Count of bytes moved }
  Tmp   : Byte;
          { Temporary place to keep byte }
begin { SwapVars }
  for Count := 1 to Size do
  begin { for }
    Tmp := V1[Count];           {Save original byte from V1  }
    V2[Count] := V1[Count];     {Move value from V2 to V1    }
    V1[Count] := Tmp;           {Store original from V1 in V2}
  end;  { for }
end;  { SwapVars }
```

Here, as before, we have declared **absolute** variables at the same addresses as the untyped parameters. We then swap their bytes, one at a time, without any regard for what those bytes really mean. (It doesn't matter, since for our purposes they are just bytes.) Because *Size* is not a **var** parameter, the caller can use the built-in *Sizeof* function (which gives the size of a data object in bytes) as the actual parameter. Therefore a call to *SwapVars* might look like this:

```
SwapVars(A, B, Sizeof(A));
```

Untyped parameters don't allow *total* freedom in the specification of parameters. They won't let you write subprograms with varying numbers of arguments (like *Readln* and *Writeln*), and cannot provide your subprogram with a definite indication of the actual parameter type used. However, if used with care, untyped parameters can help you avoid duplicating code when the same subprogram can be used with more than one type of data.

REVIEW

Here, we've provided you with a brief look at some special features of Turbo Pascal: **absolute** variables and untyped parameters. Both of these features can make programming in Turbo Pascal easier. In the next chapter we'll examine computer numbering systems and boolean operations.

27 Computer Numbering Systems and Boolean Operations on Integers

In our discussion of operators in Chapter 9 we mentioned that it was possible to use the operators normally reserved for values of the type boolean with integers and bytes as well. What does it mean to talk about "3 **and** 25," or "254 **xor** 12?" In this chapter, we will explore the meanings and uses of such expressions.

INTEGERS AS BITS AND BYTES: HOW INTEGERS ARE REPRESENTED IN MEMORY

In order to fully understand how boolean operators work on integers and bytes, you must first understand the way in which these data objects are stored in your computer's memory. As you may recall from earlier discussions, all data is represented in your computer as 1s and 0s, that is, as bits. How does a group of bits "get together" to describe the value of an integer? The "code" is similar to what we normally use to represent integers. We will examine the base 10, or decimal, system we use daily and see how we can make it work if there are only two possible values for a digit: 0 and 1.

Place Value

When we count to 10, we begin counting by single digits, like so:

```
0   9 digits left
1   8  "     "
2   7  "     "
3   6  "     "
4   5  "     "
5   4  "     "
```

```
6    3  "    "
7    2  "    "
8    1  "    "
9    no digits left
```

After counting to 9, virtually all of us will automatically write 10—a 1 followed by a 0. What are we really doing here? We are "re-using" symbols to avoid inventing new ones. The 1 in the number 10 is the symbol for 1, but because of its place in the number it represents a value of *ten* (the 0, a placeholder, represents a value of *zero*). Similarly, the 2 in the number 20 represents a value of *twenty* and the 3 in 35 represents a value of *thirty* (with the 5 still representing a value of *five*). The idea of having a digit represent a different value depending on its location in the number is called *place value*, and is the root of all modern numbering systems (including binary and hexadecimal).

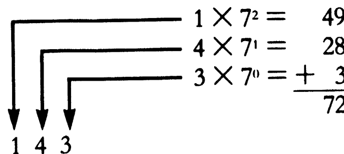
The second digit from the right in a decimal number is said to be in the "tens" place; similarly, the 5 in the number 543 is said to be in the "hundreds" place. Each place to the left has a value of 10 times its predecessor, ad infinitum (or until you get tired of writing digits). It is no accident that the multiplying factor between places is 10.

If suddenly there were no longer the digits 8 and 9, we would have no way of representing them with a single digit; but we could make the symbol 10 stand for the value *eight*, 11 for *nine*, and so on. In this case, we'd be counting in *octal*, or base 8.

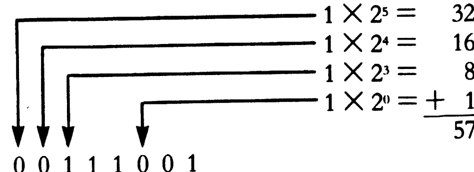
If we continued our exercise by eliminating all the digits down to 7, 6, or 5, then the number we could count to without adding more places to our number would decrease—and the multiplying factor for the numbers in successively higher places would decrease as well. In base 2, there are two digits, and a multiplying factor of two between places in the number. Here's a table to show what happens:

Base	Available Digits	Place Values	Value of Symbol '10'
Base 10	0,1,2,3,4,5,6,7,8,9	1,10,100,1000,...	Ten
	...		
Base 8	0,1,2,3,4,5,6,7	1,8,64,256,...	Eight
Base 7	0,1,2,3,4,5,6	1,7,49,343,...	Seven
	....		
Base 3	0,1,2	1,3,9,27,81,...	Three
Base 2	0,1	1,2,4,8,16,32,64...	Two

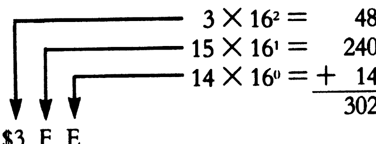
Using these place values, we can understand how to read a number in any base. For instance, in the number 143_7 (where the subscript 7 indicates that the number is to be read in base 7), we can look at the places as follows:



How about the number 00111001_2? It would read



Of course, it is possible to have a numbering system with *more* than 10 digits. In fact, hexadecimal (called "hex" for short) has 16 digits—0 through 9 plus *A* through *F*. The digit *A* has the value *ten*, *B* is *eleven*, and so on up to *F*, which has the value *fifteen*. Here's an example of how to read hex:



Hexadecimal is commonly used to represent numbers in certain computing tasks. Since few computer screens have subscripts and superscripts, however, hexadecimal numbers are usually marked by a dollar sign, or (in some systems) with a letter *H* at the end. Turbo Pascal uses the dollar-sign convention to indicate that a number is expressed in hex. Note that the values of the places in hex go up by a factor of 16 each time you move to the left. Thus, it is possible to write large numbers in only a few hex digits. (The number 100000_{16} is equal to 1,048,576 in decimal.) In general, a numbering system with greater possible digits (that is, with a higher base) can always express a number more compactly than one with fewer possible digits (a lower base). For instance, the number 4095 is FFF_{16}, 4095_9, and 111111111111_2.

Exercises Now, try translating some numbers to base 10 on your own. (The answers are given in Appendix B.) Here are a few to start with:

1. 1212_3
2. 8435_9
3. 111111_2
4. 3051_7
5. $3FF_{16}$

It is also useful to understand how to reverse the process, to convert numbers from decimal to an arbitrary base. To do this, consider the values of the places in the base you are converting *to*, and find the largest number that is less than the one you are converting *from*. Then, divide the number you are converting by the value of that place, and place the quotient in that place. Next, take the remainder from the division and repeat the process, filling in unused places with 0s as necessary.

To show this technique in action, let's convert the decimal number 37 to binary. In the preceding chart, we can see that the places in binary have the values 1,2,4,8,16,32,64, and so on, the powers of two from 0 on up. Since 37 is less than 64, the left-most digit (also called the most- significant digit) of the binary representation of 37 must be in the 32's place:

```
                              32's  16's   8's   4's   2's   1's
                              ------------------------------------
37/32 = 1, remainder = 5       1     ?     ?     ?     ?     ?
```

Now, begin again with the remainder 5. Since 5 is smaller than 16 or 8, those places are filled in with 0s; however, since 5 is greater than 4, we do another division operation:

```
5/4   = 1, remainder = 1       1     0     0     1     ?     ?
```

Finally, we only have a 1 remaining. We therefore put a 0 in the 2's place, and the 1 fits into the 1's place with a remainder of 0:

```
1/1   = 1, remainder = 0       1     0     0     1     0     1
```

Thus, the binary representation of 37_{10} is 100101_2. Before we give you some examples to try, let's run through another example, converting the number 50085_{10} to hex.

```
                                        4096's  256's  16's  1's
                                        -------------------------
50085/4096 = 12 = $C, remainder=933        C      ?      ?    ?
933/256 = 3, remainder=165                 C      3      ?    ?
165/16 = 10 = $A, remainder=5              C      3      A    5
```

In this example we converted all quotients produced by division to single digits of the base we were converting to. Thus, the quotient of 12 decimal from the first division was converted to the single digit *C*. Two divisions later, we converted 10 decimal to the value *A*. In *all* cases, the result must be able to be converted to a single digit, or you have made a mistake.

Exercises Convert the following numbers to the bases indicated (see Appendix B for solutions):

6673_{10} to base 5

65533_{10} to hex

45_{10} to binary

3262_{10} to base 7.

THE SPECIAL RELATIONSHIP BETWEEN BINARY AND HEX

Because 16 is a power of two, you might expect that a relationship exists between binary and hexadecimal numbers. In fact, this turns out to be the case, and the resulting relationship makes it easy to manipulate binary numbers using hex digits.

Let's look at the number 151 expressed in both binary and hex. The binary for 151 is:

```
1 0 0 1  0 1 1 1
```

while in hex it is:

```
9        7
```

Note that if you divide the digits in the binary number into groups of four and convert those groups to hex digits, then you get the equivalent number in hex. Similarly, you can convert hex to binary. The hex number $AF can be converted to binary by remembering that $A = $10_{10} = 1010_2$, and $F = $15_{10} = 1111_2$. Thus,

```
A        F
```

equals

```
1 0 1 0  1 1 1 1
```

This "trick" is used by programmers when they want to keep track of values of the individual bits of a number, but do not want to write every number out as a collection of 1s and 0s.

TWO'S COMPLEMENT NOTATION: REPRESENTING NEGATIVE NUMBERS

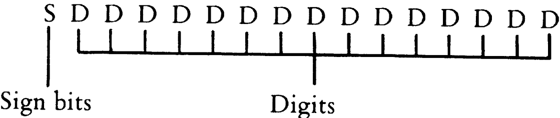
So far we've shown you how to represent positive numbers in binary, hex, and other bases, but we haven't talked about representing negative numbers. In Turbo, an integer can range from −32768 to 32767; so, how does Turbo know whether a number is positive or negative?

Turbo Pascal, like virtually all computers and computer languages, makes use of a notation known as *two's complement* to represent integers. Here we'll introduce this notation, and show the way to determine how any integer is stored in memory.

As you may recall, an integer in Turbo is stored in 16 bits, or 2 bytes, of memory. If we only wanted to represent positive numbers using these 16 bits, the largest number we could represent would be 1111111111111111_2 (that is, 16 ones, or 65535 decimal), and the smallest would be 0—a total of 65536 combinations. In order to represent negative numbers, two's complement notation divides the 16-bit num-

bers into two equal groups: 32768 positive numbers (0 to 32767) and 32768 negative ones (−1 to −32768).

To tell immediately whether a number is positive or negative, the left-most bit of the binary number is used as the *sign bit.* If the left-most bit is a 0, the number is positive; if it is a 1, the number is negative.



If the number is positive, the digits are encoded the same as any binary number. However, if the number is negative, they are encoded so that it's easy to do arithmetic with them—as the two's complement of the corresponding positive number.

The two's complement of a binary number is the result of changing all 1s to 0s and all 0s to 1s, plus 1. (For 16-bit numbers, you can also subtract the number from 65536—the value of the next place after the left-most place of the number—and get the two's complement and the sign bit.)

Why are negative numbers represented this way, rather than as ordinary positive numbers plus a sign? One reason is to avoid having two representations for the number 0 (+0 and −0). Since storage space in a computer can be a valuable commodity, it would be wasteful to use two combinations of bits to represent the same number. It would also make the circuitry of the computer more complex, as it would have to recognize both "kinds" of 0. The second reason is that logic circuits that work with two's complement numbers are easy to design (and simple to build) and are included in virtually every microprocessor on the market today. Turbo Pascal uses these circuits to do integer math as quickly and efficiently as possible.

HOW BYTE VALUES ARE STORED IN MEMORY

The type byte (discussed in Chapter 7) is actually a subrange of the type integer: 0..255. Turbo Pascal saves memory when storing bytes by using only 8 bits (instead of 16), treating bytes exactly as if they were the lower 8 bits of an integer (with the upper 8 bits always 0). Because of this convention bytes can never have a negative value, since the implicit sign bit is always 0.

When Turbo Pascal works on a byte value, it converts it to an integer by appending a byte of all 0s to it. When a byte result is stored in memory once again, the top byte of the integer's intermediate value is removed, and the lower byte is written to memory. Turbo does not

check to see whether the result of a calculation done with byte values is larger or smaller than the value held by a byte, thus it is important that your byte calculations do not overflow.

BOOLEAN OPERATIONS ON INTEGERS AND BYTES

Now that we understand a bit more about how integers and bytes are stored in memory, we are ready to take a look at what it means to use the boolean operators **and**, **or**, **xor**, and the special operations **shl** and **shr**, on objects of these types.

When the **and** operation works on operands of the type boolean (as mentioned in Chapter 11), it produces a value of TRUE if and only if both of its operands are TRUE. Since it doesn't make sense to talk about an integer as being "true" or "false," what does it mean to use **and** on them?

The answer lies in the way integers are stored as bits in memory. Each bit of each integer can be thought of as a tiny boolean variable—holding a 1 to represent TRUE, or a 0 to represent FALSE. When we do an **and** of two integers, we are actually performing the **and** operation on the pairs of bits in corresponding places in the two integers, as follows:

```
5 and 97 =
      0000000000000101 (=  5 decimal)
  and 0000000001100001 (= 97 decimal)
      ----------------
      0000000000000001
```

The same is true for the **or** and **xor** operations. Here are some examples of how these operations work on integers:

```
      0000000001100011 (=   99 decimal)
  xor 0000001111111111 (= 1023 decimal)
      ----------------
      0000001110011100 (= 924 decimal)

      0000000001100011 (=   99 decimal)
   or 0000001111111111 (= 1023 decimal)
      ----------------
      0000001111111111 (= 1023 decimal)
```

The **not** operator also works on integers and bytes. When used on these operands, **not** *inverts* every bit, changing all 1s to 0s and all 0s to 1s. Thus,

```
not  1010111001010001 = 0101000110101110
```

Under most conditions, you will probably not use the boolean operators in this manner. However, some operations are much more efficient using boolean operators than more conventional ones. For instance,

one common application of the **and** operator is to unconditionally clear a particular bit of a byte or word. One of the situations in which this function is useful is in programs that read files produced by word processors such as WordStar.

When WordStar creates files on the disk, it sometimes sets the high bits of bytes, which represent the characters of your text. These bits remind WordStar that these characters are at the ends of words, and that these words can be spaced out evenly along a line by filling the gaps between them with spaces. While this format works well for WordStar, it often makes WordStar files unreadable by other programs; thus, these programs must "strip" (clear) the high bit of every byte in the file as it is read.

The **and** operator can be used to do this:

```
     11000001  ($CA = ASCII "A" ($4A) with high bit set)
and  01111111  ($7F = byte with all bits but high bit set)
     --------
     01000001  ($4A = normal ASCII "A")
```

By performing an **and** operation on every byte of the file, we can render all of the characters in the file readable. These operations will also prove useful if your program needs to pass information directly to the operating system (through the MSDos call in MS- DOS or PC-DOS Turbo, or through the BIOS and BDos calls in CP/M versions). For more information on these calls and what they can do, see the reference guide for your operating system and computer.

THE SHIFTING OPERATORS: SHL AND SHR

Turbo's **shl** and **shr** operators provide you with direct access to two powerful operations that move the bits within a byte or integer to the left or the right. These have no direct counterparts in standard Pascal.

Suppose we had an integer *I* with the binary representation 0000000011111111. Here are the results of applying the **shl** and **shr** operations to *I* and an integer from 1 to 16:

```
I shr  1 = 0000000001111111     I shl  1 = 0000000111111110
I shr  2 = 0000000000111111     I shl  2 = 0000001111111100
I shr  3 = 0000000000011111     I shl  3 = 0000011111111000
I shr  4 = 0000000000001111     I shl  4 = 0000111111110000
I shr  5 = 0000000000000111     I shl  5 = 0001111111100000
I shr  6 = 0000000000000011     I shl  6 = 0011111111000000
I shr  7 = 0000000000000001     I shl  7 = 0111111110000000
I shr  8 = 0000000000000000     I shl  8 = 1111111100000000
I shr  9 = 0000000000000000     I shl  9 = 1111111000000000
I shr 10 = 0000000000000000     I shl 10 = 1111110000000000
I shr 11 = 0000000000000000     I shl 11 = 1111100000000000
I shr 12 = 0000000000000000     I shl 12 = 1111000000000000
I shr 13 = 0000000000000000     I shl 13 = 1110000000000000
```

```
I shr 14 = 0000000000000000   I shl 14 = 1100000000000000
I shr 15 = 0000000000000000   I shl 15 = 1000000000000000
I shr 16 = 0000000000000000   I shl 16 = 0000000000000000
```

As you can see, the **shr** operator works by shifting all of the bits of the first operand to the right by the number of places indicated in the second operand, and then adds 0s to the left. Similarly, the **shl** operator shifts its operand to the left and adds 0s to the right.

How are these operations useful? From our previous discussion of place value, you may recall that the values of the places in a binary number increase by a factor of 2 as you move to the left. Thus, *I* **shl** 1 is equivalent to *I**2 (unless, of course, a 1 is shifted into the sign bit). Similarly, *I* **shr** 1 is equivalent to *I* **div** 2 for all positive numbers.

Because a multiplication or division operation is much more complicated than a shift operator (and takes about 40 times as long), you can frequently make your programs run faster by using shift operations in well-chosen places. As with other operations on bytes and integers, Turbo will not provide a warning if overflow occurs, so use these operations with care.

REVIEW

In this chapter, we provided an in-depth look at how computer numbering systems work, how integers and bytes are represented in memory, and how boolean and shifting operations can be used on integers and bytes. Armed with this information, you can use Turbo Pascal to perform many sophisticated machine-level operations in your programs.

28 Using 8088/8086 Assembly Language with Turbo Pascal

This chapter shows experienced assembly language programmers how to include their assembly routines in their Turbo Pascal programs. If you are unfamiliar with assembly language, do not try to use this chapter as a tutorial.

Turbo Pascal provides two methods of incorporating assembly language (or, more precisely, the machine codes produced once it has been assembled) into your Turbo program. The first technique, the ***external subprogram***, allows you to incorporate the machine language output of an assembler or debugger into your program without modifications. The second, the ***inline statement***, requires you to edit the hexadecimal output from one of these programs into your Turbo Pascal code files, while making access to Turbo variables and constants very straightforward.

EXTERNAL SUBPROGRAMS

An external subprogram is a procedure or function that has been assembled by an assembler or debugger to run as a stand-alone program and then be called by Turbo at runtime. For example, an assembly language subprogram for Turbo on the IBM PC (or another MS-DOS or CP/M-86 machine) might look like this:

```
CODE    SEGMENT
        ASSUME  NOTHING; Can make no assumptions about
                       ; location!
MYPROC  PROC    NEAR   ; External routines must
                       ; NEAR procedures
        PUSH    BP     ; Save the value of BP
        MOV     BP,SP  ; Set up stack to access parameters

        . . .          ; Code for MyProc goes here
```

```
        MOV     SP,BP   ; Restore SP
        POP     BP      ; Restore BP
        RET     PARAMETERBYTES      ; Leave routine and
MYPROC  ENDP                        ; restore the stack
CODE    ENDS
        END
```

How can such a subprogram be called by a Turbo Pascal program? The first step in making this possible is to create a file with the extension .BIN (or .CMD for CP/M-86)—a process that can be performed differently according to the set of tools you have available on your computer. Under PC-DOS or MS-DOS, the most commonly used tools are the programs MASM.EXE (the Microsoft Macro Assembler), LINK.EXE (the Microsoft Linker), and EXE2BIN.EXE (a utility that converts the .EXE files produced by the linker to .BIN files). If the code shown previously were contained in a file called MYPROC.ASM, then the commands to create the .BIN file might include these:

```
ASM MYPROC;
LINK MYPROC;
EXE2BIN MYPROC.EXE
```

After MYPROC.BIN has been created, the next step is to let Turbo know that it exists and what it contains. This is done by declaring *MyProc* as an ***external*** *procedure* (it could also be a function) as follows:

```
procedure MyProc(var X : integer; Y : integer); external
'MYPROC.BIN';
```

The reserved word **external** tells the Turbo compiler that the machine code for the procedure *MyProc* is contained in the file MYPROC.BIN. When this declaration is compiled, the machine code in MYPROC. BIN is read from the disk and incorporated directly into the code of your program at the point of declaration. Note that since this happens at compile time not at runtime, you *must* recompile every time the .BIN file is changed in order to load in the new version.

Because of the way in which Turbo incorporates the machine code into the compiled program, certain rules must be observed. First, since there is no easy way to determine the address at which the code will ultimately reside, the assembly language code must be "relocatable"; that is, it must never rely on being loaded into memory at a specific address. Also, since all Turbo subprograms are activated with a *Near* call, the subprogram must do the same and terminate with a *Near* return. Finally, the routine may change any of the processor's registers, but must restore Turbo's SS (Stack Segment), SP (Stack Pointer), BP (Base Pointer), CS (Code Segment), and DS (Data Segment) registers before returning.

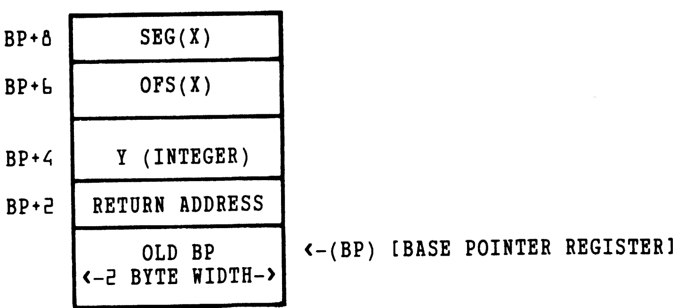
ACCESSING PARAMETERS FROM EXTERNAL SUBPROGRAMS

Most useful subprograms have parameters. When a subprogram is called, the parameters (or their addresses, if they are **var** parameters) are pushed onto the runtime stack in the same order in which they are declared. The following illustrates how to access these parameters:

```
CODE    SEGMENT
ASSUME  NOTHING
; procedure Simple(var X : integer; Y : integer);
;
; Simply adds X to Y and returns the sum in X.
; Note that X is a var parameter (address passed on
; the stack) and Y is a value parameter (value passed
; on the stack).

SIMPLE  PROC    NEAR
        PUSH    BP     ; Save old BP, and load in SP so that
        MOV     BP,SP  ; BP can be used to address parameters
        LES     DI,DWORD PTR [BP+6] ; Move X's ADDRESS
                                    ; into ES:DI
        MOV     AX,[BP+4] ; Move Y's VALUE into AX
        ADD     WORD PTR[DI],AX ; Add Y's value to X's
        POP     BP     ; Restore BP
        RET     6      ; Return, popping 6 bytes of
                       ; parameters bytes of parameters from
                       ; stack: 4 for x (a var parameter) and
                       ; 2 for y (an integer value parameter)
SIMPLE  ENDP
CODE    ENDS
        END
```

To understand how parameters are accessed, let's look at a "picture" of the stack. After executing the first two instructions of SIMPLE (PUSH BP and MOV BP,SP), the stack will look like this:



The stack grows downward from SS:SP to SS:00, which means that it starts at high-memory addresses and moves toward low-memory addresses. Everything is pushed onto the stack in 2-byte quantities.

The parameters are pushed onto the stack immediately before the call to *Simple*, and the call itself causes the return address to be pushed as well. To access the parameters, the subprogram can use the BP register. Once BP has been loaded with the previous value of SP, it points the stack segment at a location relative to the parameters.

Turbo pushes the parameters in the order specified by the external procedure's parameter list. Since the stack grows downward, the last parameter, *Y*, is at the lowest address—BP + 4. *Y* is a value parameter, which means that its actual value is pushed onto the stack. Since *Y* is an integer, it occupies 2 bytes.

X, on the other hand, is a **var** parameter. This means that its long address—segment and offset—is pushed onto the stack, consuming 4 bytes. Note that all addresses take 4 bytes of stack, regardless of the size of the object they reference. In keeping with 8088 conventions, the segment address is pushed first, followed by the offset address. Thus, *X*'s segment and offset are located at BP + 8 and BP + 6, respectively, and can be accessed as a "double word" (by the LES and LDS instructions) at BP + 6. (These two instructions are the fastest way of obtaining the location of a **var** parameter, and have the added advantage of preparing the processor for block and string operations. Remember, however, to save and restore the DS register when using LDS.)

When a subprogram is completed, it returns control to the calling routine with a *Near* RET instruction. If the subprogram is called with parameters, they must be popped from the stack when returning; in addition, if the subprogram is a function, a *function-result variable* may need to be popped as well. The RET instruction allows this task to be performed automatically during a return (use the instruction RET N to pop *N* bytes of parameters). In procedure *Simple*, we used a RET 6 instruction to deallocate the parameters from the stack—2 bytes for *Y* and 4 for *X*. If the procedure has no parameters, *N* need not be specified.

If the subprogram is a function, a function result (of the size required by the type of the result) is allocated on the stack before the first parameter is pushed. This area can be used as temporary storage by the function during execution. The method by which function results are returned depends on the result type. Scalar function results are returned in the AX register, and boolean functions set the *Z* flag as well (1 for FALSE, 0 for TRUE). Scalar functions should always remove the one-word function result from the stack when returning.

Functions that return pointers do so by placing the pointer in DX:AX and popping the two-word function result when returning.

Functions that return real results do so by leaving the result on the stack instead of popping it off when returning.

Functions that return other structures are heavily dependent on internal Turbo Pascal runtime routines to perform function returns (their implementation as externals is not recommended). If you must implement such functions, we suggest that you use the **inline** statement or **var** parameters instead.

The sample assembly routines discussed in this chapter demonstrate the use of both value and **var** parameters, as well as techniques for returning a function result. The *Turbo Pascal Reference Manual* contains more information on the internal data formats of parameters.

ALLOCATING LOCAL VARIABLE SPACE

If your **external** subprogram needs a local work space in which to perform calculations, it may allocate room for them on the stack. This can be done by saving the stack pointer before performing the MOV BP,SP (shown previously), as follows:

```
PUSH   BP               ; Always save BP
SUB    SP,LOCALS        ; Allocate space for locals
MOV    BP,SP            ; Save BP (used to access locals
                        ; AND parameters)
ADD    SP,LOCALS        ; Restore stack pointer to where BP
                        ; was saved
POP    BP               ; Restore BP
RET    PARMS            ; Return, deallocating parameters (if
                        ; any)
```

When this technique is used, the local variables may be accessed in the same way as parameters. It often helps to draw a picture of the stack to aid in calculating the offsets of both.

Note that this method of allocating local storage provides temporary storage only; the locals are destroyed upon exiting from the routine.

ALLOCATING STATIC STORAGE

It is also possible (though not straightforward) to allocate the equivalent of a local typed constant; that is, local storage that retains its value between calls. Such data objects may be allocated by the assembler directly in the code segment; however, since their location within the code segment is not known, it must be determined at runtime.

Since the IP register of the processor is not directly accessible by the program, the best way to obtain it is to perform a call to a known

location, then pop the return address off of the stack instead of returning. If the call is made immediately before the beginning of the local storage area, the proper address can be calculated as shown here:

```
        ORG     00H
CODE    SEGMENT
        ASSUME  CS:CODE     ; Use to get offsets relative
                            ; to start of routine
TRICKY  PROC NEAR
        CALL    RESUME      ; First instruction is a call
TCONST1 DW 4 DUP(0)         ; An array of 4 zeroes
TCONST2 DB 5                ; A string of length 5
        DB 'Hello'

        ; More static storage here

RESUME: POP BX              ; Pop the return address
        SUB     BX,3        ; Subtract 3 to get location of
                            ; routine (short call=3 bytes);
                            ; leave in BX
        PUSH    BP          ; Usual setup to access
                            ; parameters/locals
        SUB     SP,LOCALS   ; Optional Allocate space
                            ; for locals
        MOV     BP,SP       ; Set up BP to access
                            ; parameters and/or locals

        ; Some examples of how to access static storage:

        MOV     AX,CS:[BX + OFFSET TCONST1]
                            ; Load value into AX from
                            ; static area
        MOV     CL,BYTE PTR CS:[BX + OFFSET TCONST2]
                            ; CL gets string length

        ADD     SP,LOCALS   ; Restore stack pointer to
                            ; where BP was saved
        POP     BP          ; Restore BP
        RET     PARMS       ; Return, deallocating
                            ; parameters (if any)
TRICKY  ENDP
CODE    ENDS
        END
```

This method is not recommended unless you have a good grasp of the subtleties of 8088/8086 programming.

LIBRARIES (PC-DOS/MS-DOS ONLY)

Often, you may want to create an assembly language file containing a number of related **external** routines. This may be strictly for ease of development (only one file to assemble, link, convert, and read in), or it may be to allow a group of externals to be distributed to other Turbo Pascal users as a single file.

Turbo Pascal allows you to do this by declaring **external** subprograms to have a known offset relative to the beginning of a .BIN file. As we have already mentioned, the declaration

```
procedure MyProc(var X : integer; Y : integer);
  external 'MYPROC.BIN';
```

indicates that the procedure *MyProc* starts at location 0 of the file MYPROC.BIN. After this declaration is made, however, you can also declare other subprograms at other offsets relative to the beginning of the file, as follows:

```
function  MagicNumber : Integer; external MyProc[50];
```

This declaration tells Turbo Pascal that the function *MagicNumber* begins at byte number 50 (counting from 0) of the same file as *MyProc.*

As you develop a library of **external** subprograms, the relative offsets of the routines in the file are likely to change. To avoid having to modify the offsets in your Turbo code every time you change your assembly language routines, we strongly recommend the use of a structure called a *jump table.* A jump table is merely a group of jump instructions at the beginning of the file, each of which jumps to a routine in the library. Since the offsets of these jump instructions (which are always 3 bytes long) do not change when the routines are moved within the library, the Turbo program can be recompiled with no changes.

Let's describe a library of assembly language routines that access and modify blocks of data. The structure of the jump table for the three library routines is as follows:

```
ChrPos  PROC    NEAR
        JMP     ChrPosCode
          ; ChrPos has been called, bypass other jumps
        JMP     GetStr
          ; GetStr has been called  bypass last jump
        JMP     PutStr
          ; PutStr has been called
ChrPosCode:
    ... actual code for ChrPos
GetStr  PROC    NEAR
    ... code for GetStr
PutStr  PROC    NEAR
    ... code for PutStr
    ...
```

The corresponding declarations of externals in a .PAS file reflect the positions of the entries in the jump table:

```
function ChrPos(var Block; Search: char;
  Start,NmBytes: integer) : integer; external 'TEST.BIN';
procedure GetStr(var Block; var Retrieve : Str255;
  Start : integer; StrLen : byte); external ChrPos[3];
```

```
procedure PutStr(InStr : Str255;
  var Block; Index : integer); external ChrPos[6];
```

As you can see, the declaration of the first **external** subprogram in a file specifies the name of the file, and the routine begins (or, in this case, has a jump to it) at the beginning of the file. Other subprograms in the same file do not redundantly specify the machine code file name; instead, the offset from the start of the first procedure (that is, the start of the file) is specified. Because a short, intrasegment jump takes up 3 bytes, *GetStr* and *PutStr* can be reached by entering the jump table at offsets 3 and 6, respectively. The jumps in the table then direct execution to the actual code for these procedures.

THE INLINE STATEMENT

Syntax and Semantics of the Inline Statement

Turbo's **inline** statement allows you to include virtually *any* sequence of bytes in the executable code of your Turbo program, without (necessarily) using a separate assembler. The syntax of the **inline** statement is shown in Figure 28-1. The diagrams of the code element and the data element are shown in Figure 28-2 and Figure 28-3, respectively.

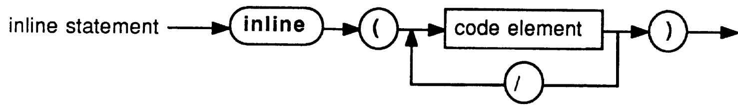


Figure 28-1 Syntax Diagram of Inline Statement

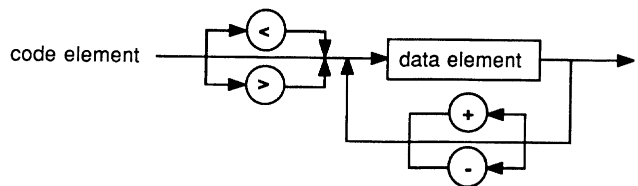


Figure 28-2 Syntax Diagram of Code Element

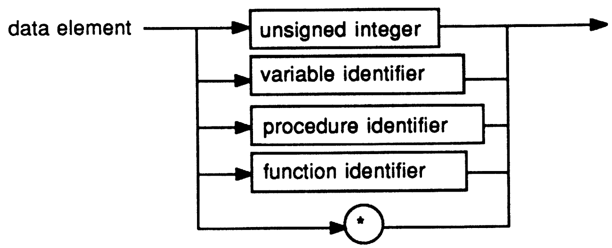


Figure 28-3 Syntax Diagram of Data Element

The **inline** statement consists of a list of code elements, which are in turn made up of one or more data elements separated by the addition (+) and subtraction (−) operators. A data element may be any legal byte or integer constant, or a Turbo identifier. Unlike **external** subprograms that are assembled completely independently of Turbo, **inline** statements may contain Turbo identifiers. These identifiers are converted to data bytes or words, as follows:

Identifier Type	Value
Scalar Constant	Byte or word value of constant
Local Variable	Offset of variable relative to BP
Global Variable	Offset of variable relative to DS
Typed Constant	Offset of constant relative to CS
Procedure or function	Offset of routine relative to CS
Parameter of current procedure or function	Offset of parameter value (address or function for var parameters) relative to BP
Non-local, non-global variables, parameters	DO NOT USE * *

Note that a function identifer does *not* give the location of the function result. However, this location can be calculated using the location and size of the first parameter. Also, a result may be placed in a local variable and transferred to the function result by an assignment following the **inline** statement. The data element * refers to the location at which the current data element is about to be stored (that is, the "current" address in the code segment).

Each code element in the **inline** statement is stored as a byte if its value is less than 256; otherwise, it is stored as a word. This automatic sizing process can be overridden with the 〈 and 〉 operators. The operator 〈 will cause only the least-significant byte of the code element to be stored, regardless of magnitude; the 〉 operator will force a word to be stored.

The following sample procedure uses an **inline** statement:

```
procedure VInLine(var Value : integer);
{ A simple use of inline code. Note that some constants
  have been defined and used, while other values are left
  as literal hexadecimal constants. This is done just for
  illustration. The following routine example divides an
  integer by 2 and discards the remainder.}
const
  CLC = $F8;
  INC_DI = $47;
begin
inline ($C4/$BE/VALUE/ { LES DI,VALUE[BP] }
        CLC/ { CLC }
        $26/$D0/$1D/ { RCR ES:WORD PTR [DI] }
end; { VInLine }
```

The hexadecimal codes for an **inline** statement may be "hand-assembled," or (for sequences of any size) an assembler or debugger may be used. In either case, care should be taken to ensure that the correct operand lengths for offsets and immediate values are observed, since the 8088 allows most of these to be either a byte or a word (depending on the addressing field of the instruction). If possible, use representative "dummy" values in the code processed by the assembler, then replace them with the appropriate Turbo symbols in the **inline**.

SPEEDING UP TURBO PROGRAMS WITH INLINE STATEMENTS

Some routines are time-critical enough that you may want to optimize them for speed (video output, for example). **Inline** code is an ideal way to optimize small routines in your Turbo program.

How can you examine the code generated by Turbo to determine if it needs to be optimized? One good way to locate specific object code is to surround it with a string of No-Ops (machine code instructions that do nothing). Then, once the program has been compiled, you can use your favorite debugging program to search for these strings and disassemble the object code between them:

```
inline($90/$90/$90/$90/$90/$90/$90); { front marker }
{... Brief Pascal routine to be optimized..}
inline($90/$90/$90/$90/$90/$90/$90); { back marker }
```

If you see that the Turbo code can be improved upon, you can then replace it with an **inline** statement or an **external** routine.

Interrupt Handling

Turbo Pascal provides all of the necessary resources for writing an *interrupt service routine* (ISR), such as the one invoked with each tick of the system clock. An interrupt handler can be written almost entirely

in Pascal source code, with a few bits of **inline** code to handle the entrance to and exit from the ISR.

The following is a sample of a Turbo Pascal ISR:

```
Program InterruptHandler;
const
  DataSave : integer = 0;
    { a typed constant that will hold the value of DS.
      It is located in the code segment so it is
      accessible by the ISR }

procedure TurboISR;
begin
  { ISR Entry code:
    The interrupt service routine must save the contents of
    all registers that may be used in the compiled code of
                                      the body of the ISR }
inline($50/         { PUSH AX }
       $53/         { PUSH BX }
       $51/         { PUSH CX }
       $52/         { PUSH DX }
       $57/         { PUSH DI }
       $56/         { PUSH SI }
       $1E/         { PUSH DS }
       $06/         { PUSH ES }
       $FB/         { STI  Enable other interrupts }
         { The following instructions are only necessary if
                the ISR needs to access global variables }
     $2E/
     $A1/
     DataSave/   { MOV AX,CS:[DataSave] this instruction
                  moves the value of DS for this Turbo
                  program accessed by a typed constant
                  (since a code segment variable is
                  always accessible) }
     $8E/$D8);   { MOV DS,AX Now DS has the segment value
                                         for this program }
                 { ... Pascal source body of the ISR ... }

 ISR Exit code:
 The previous contents of the register are restored as
 well as the restore of the SP and BP that were automatically
 pushed by Turbo at the start of the procedure.
 inline($07/        { POP ES }
       $1F/         { POP DS }
       $5E/         { POP SI }
       $5F/         { POP DI }
       $5A/         { POP DX }
       $59/         { POP CX }
       $5B/         { POP BX }
       $58/         { POP AX }
       $8B/$ED/     { MOV SP,BP }
       $5D/         { POP BP }
       $CF);        { IRET }
                { Return to the interrupted program }
 end;
```

```
  { ... other declarations...  }

begin { Main }
  DataSave := DSeg;
  { Stores the value of DS into DataSave, which
    will make the value accessible to the
    interrupt service routine }
  { ... other statements ... }
end.
```

The Interrupt Service Routine is set up by pointing the vector for the interrupt you are servicing to the address TurboISR. To obtain TurboISR's segment and offset addresses, use CSeg and Ofs(TurboISR), respectively. As interrupt handlers are often quite difficult to debug (a programming error usually causes the machine to lock up), you should understand the intricacies involved in the interrupt system for your machine before attempting to write one.

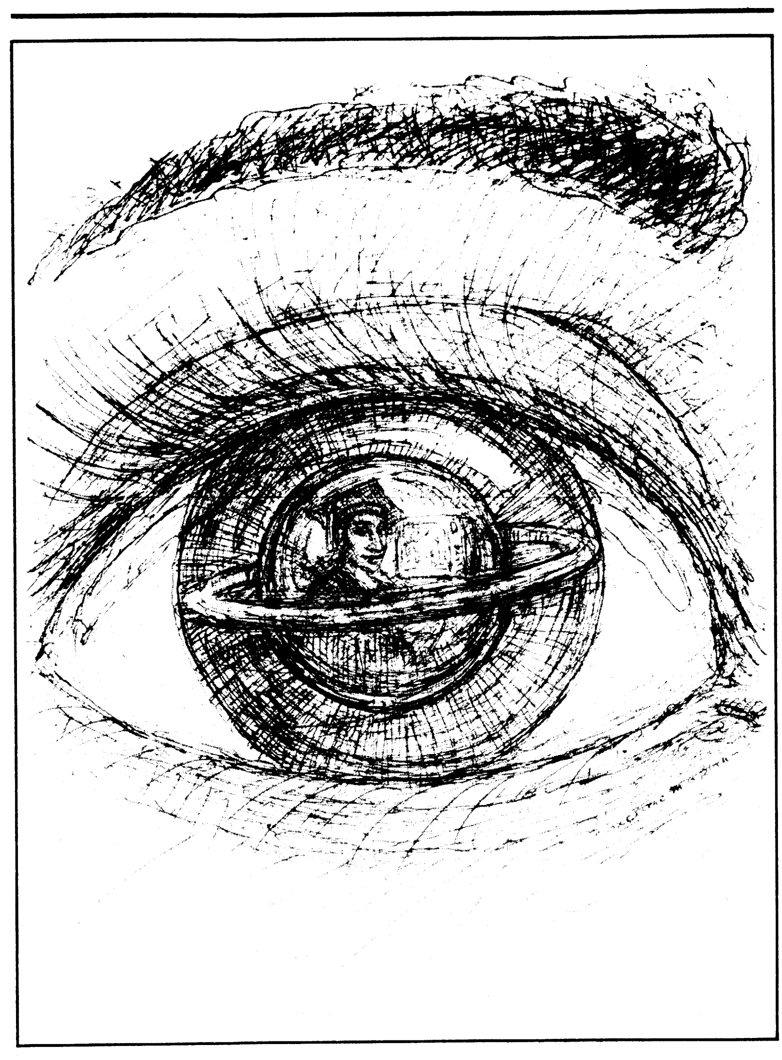
REVIEW

This chapter has provided details of how to access and use assembly language routines within a Turbo Pascal program.

This chapter also marks the end of the advanced section—you've come a long way. The remainder of this manual is devoted to appendices that encompass such information as CP/M-80/86 operation with Turbo Pascal, a complete set of syntax diagrams, a summary of the standard procedures and functions in Turbo Pascal, and more. Make use of what you need currently and refer back to the other materials as necessary.

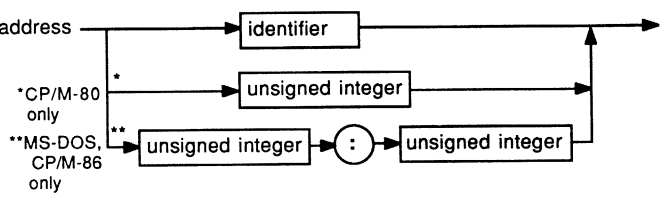
PART IV

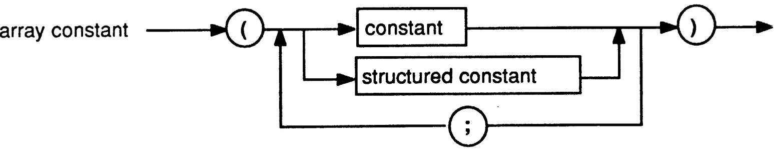
Appendices

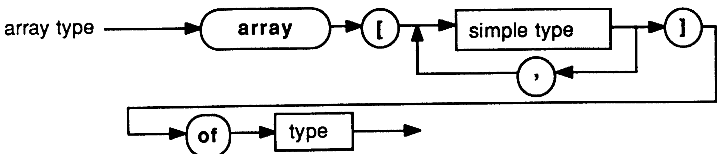


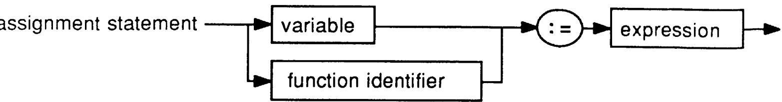
A Syntax Diagrams for Turbo Pascal

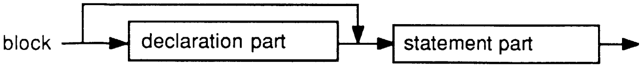
We have provided several syntax diagrams throughout this manual, with specific instructions on how to read them in Chapter 7. The following serves as a complete alphabetical reference guide to all of the Turbo Pascal syntax diagrams, displaying those we have yet to mention as well as the ones previously discussed.

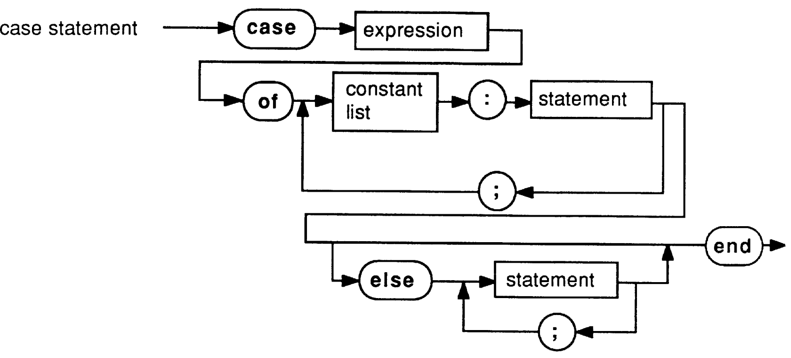




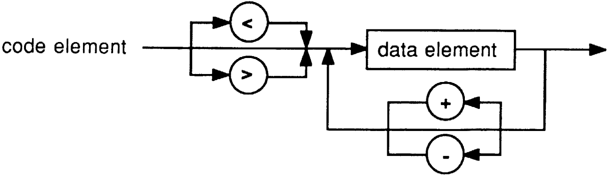


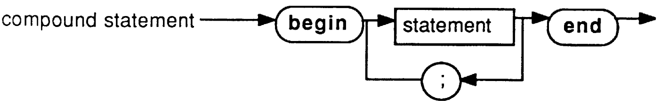


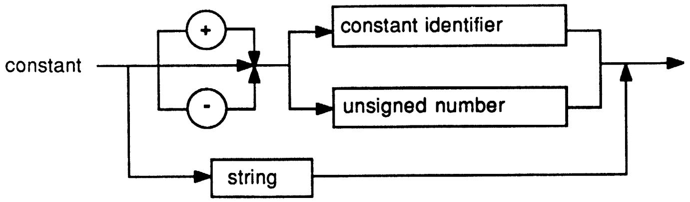


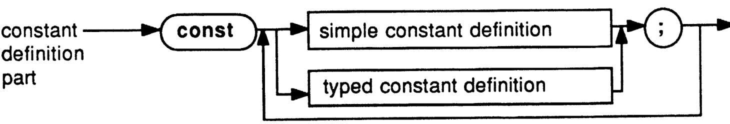


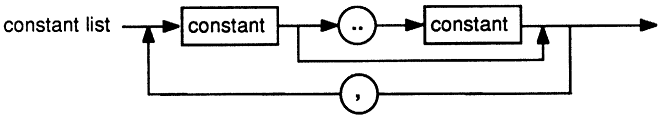
character → < machine-dependent >

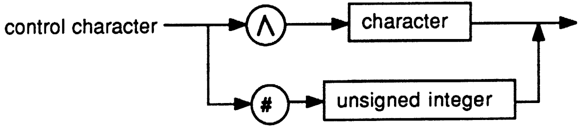


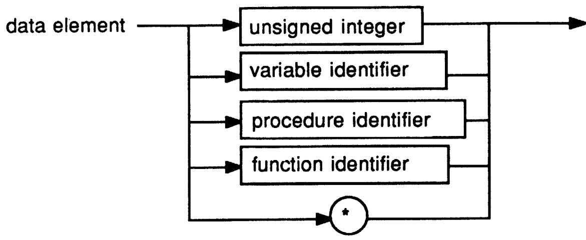


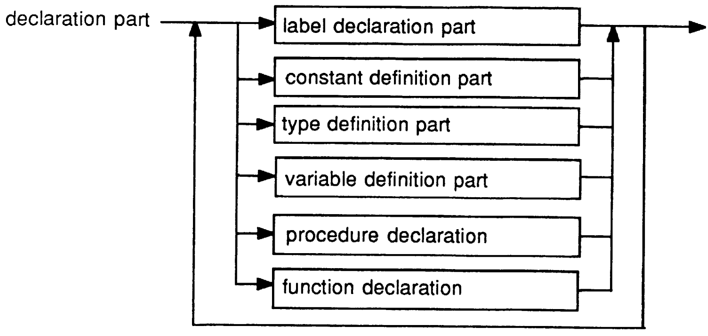

constant
+
-
constant identifier
unsigned number
string

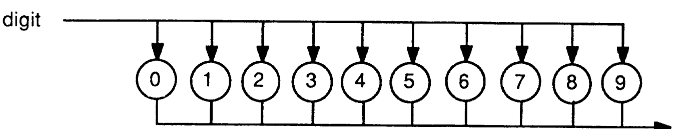

constant
definition
part
const
simple constant definition
typed constant definition
;

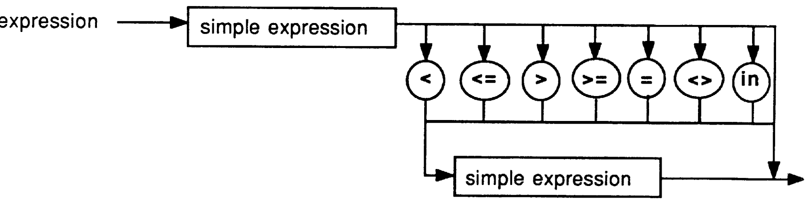

constant list
constant
..
constant
,

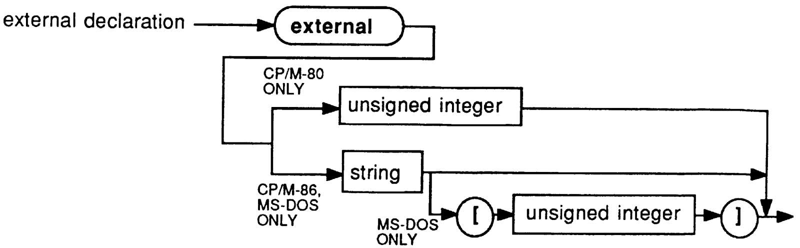

control character
^
character
#
unsigned integer

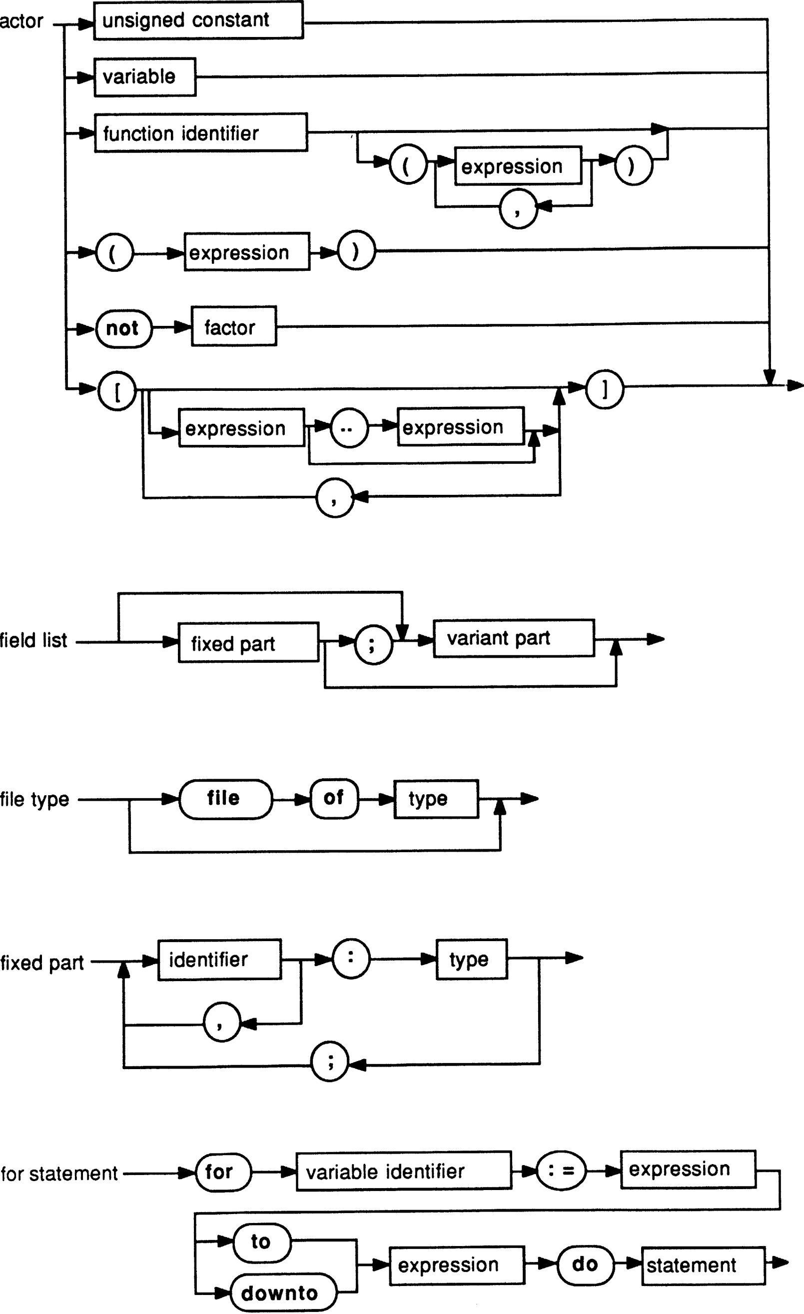

data element
unsigned integer
variable identifier
procedure identifier
function identifier
*

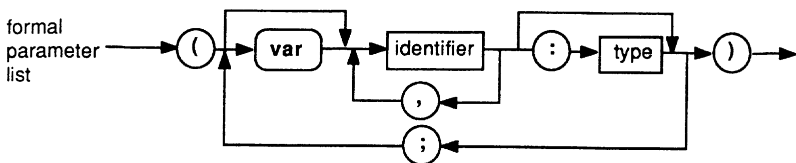

declaration part
label declaration part
constant definition part
type definition part
variable definition part
procedure declaration
function declaration

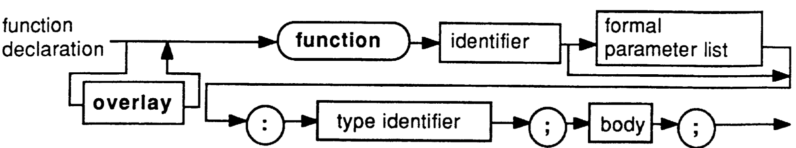

digit
0
1
2
3
4
5
6
7
8
9

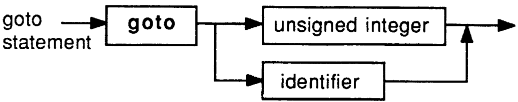

expression
simple expression
<
<=
>
>=
=
<>
in
simple expression

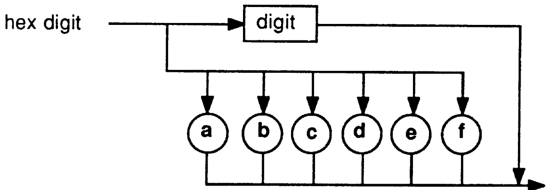

external declaration
external
CP/M-80 ONLY
unsigned integer
string
CP/M-86, MS-DOS ONLY
MS-DOS ONLY
[
unsigned integer
]

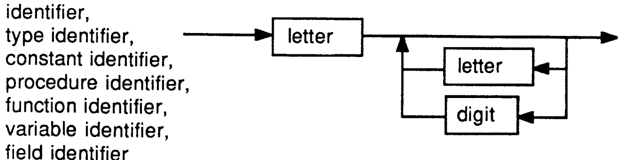

factor
unsigned constant
variable
function identifier
(
expression
,
)
(
expression
)
not
factor
[
expression
..
expression
,
]
field list
fixed part
;
variant part
file type
file
of
type
fixed part
identifier
,
:
type
;
for statement
for
variable identifier
: =
expression
to
downto
expression
do
statement

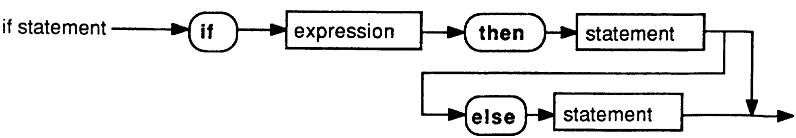

formal
parameter
list
(
var
identifier
,
:
type
)
;

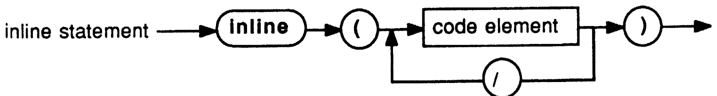

function
declaration
function
identifier
formal
parameter list
overlay
:
type identifier
;
body
;

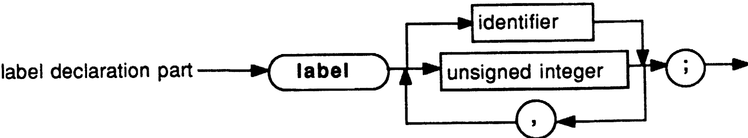

goto
statement
goto
unsigned integer
identifier

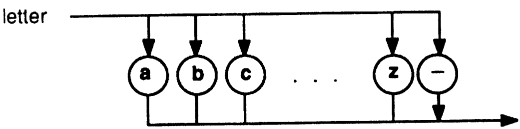

hex digit
digit
a
b
c
d
e
f


identifier,
type identifier,
constant identifier,
procedure identifier,
function identifier,
variable identifier,
field identifier
letter
letter
digit

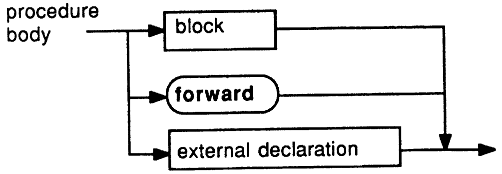

if statement
if
expression
then
statement
else
statement

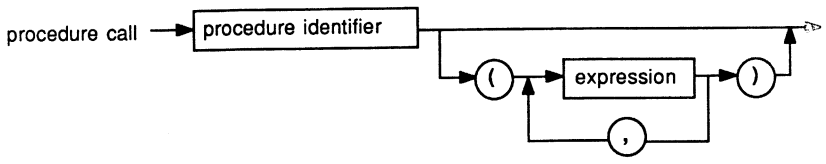

inline statement
inline
(
code element
/
)

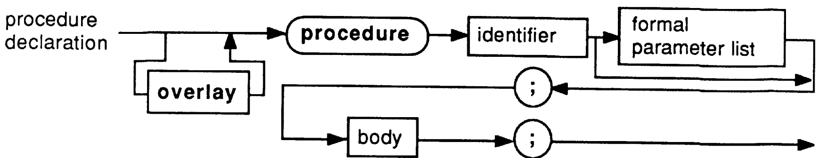

label declaration part
label
identifier
unsigned integer
,
;

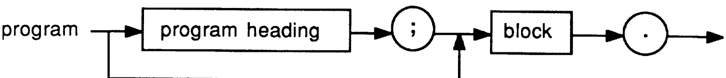

letter
a
b
c
. . .
z
_

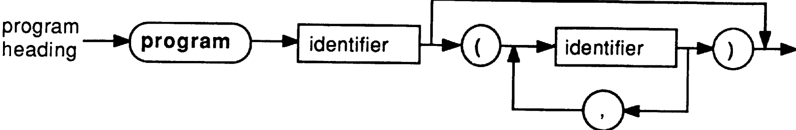
pointer type
^
type identifier

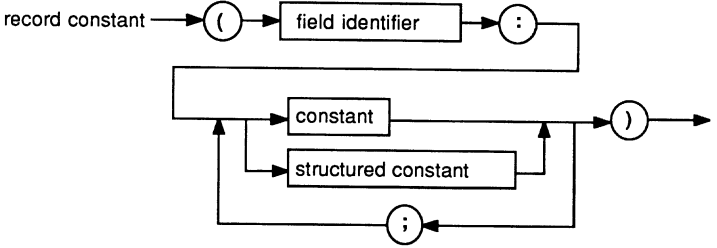

procedure body
block
forward
external declaration


procedure call
procedure identifier
(
expression
,
)

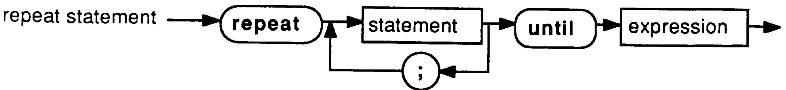

procedure declaration
overlay
procedure
identifier
formal parameter list
;
body
;

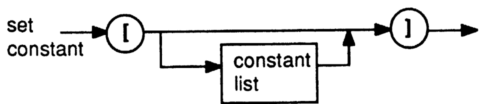

program
program heading
;
block
.

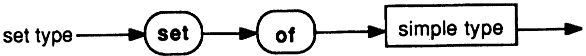

program heading
program
identifier
(
identifier
)
,


record constant
(
field identifier
:
constant
structured constant
)
;

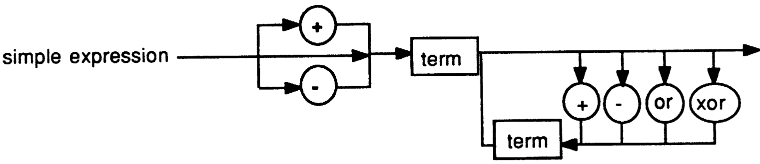
record type
record
field list
end

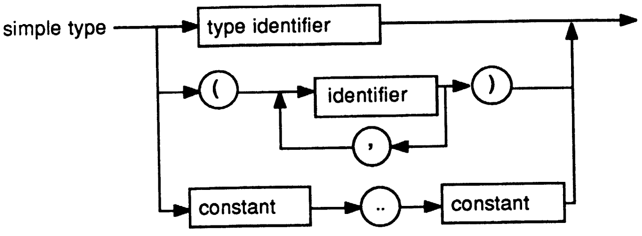

repeat statement
repeat
statement
;
until
expression

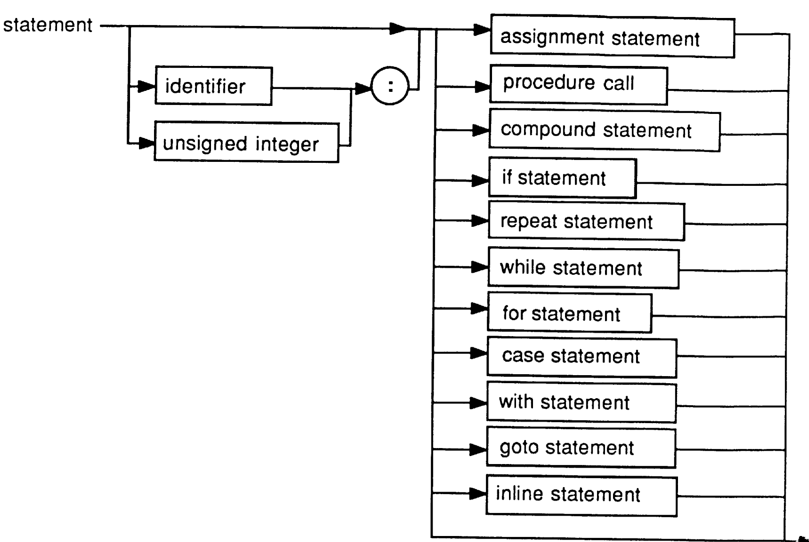

set constant
[
constant list
]

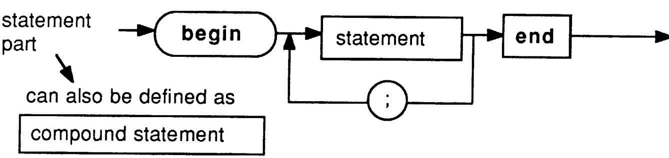

set type
set
of
simple type

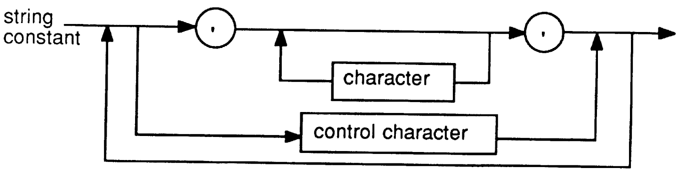
simple constant definition
identifier
=
constant

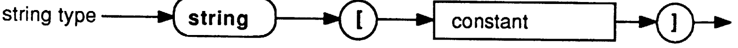

simple expression
+
-
term
+
-
or
xor
term

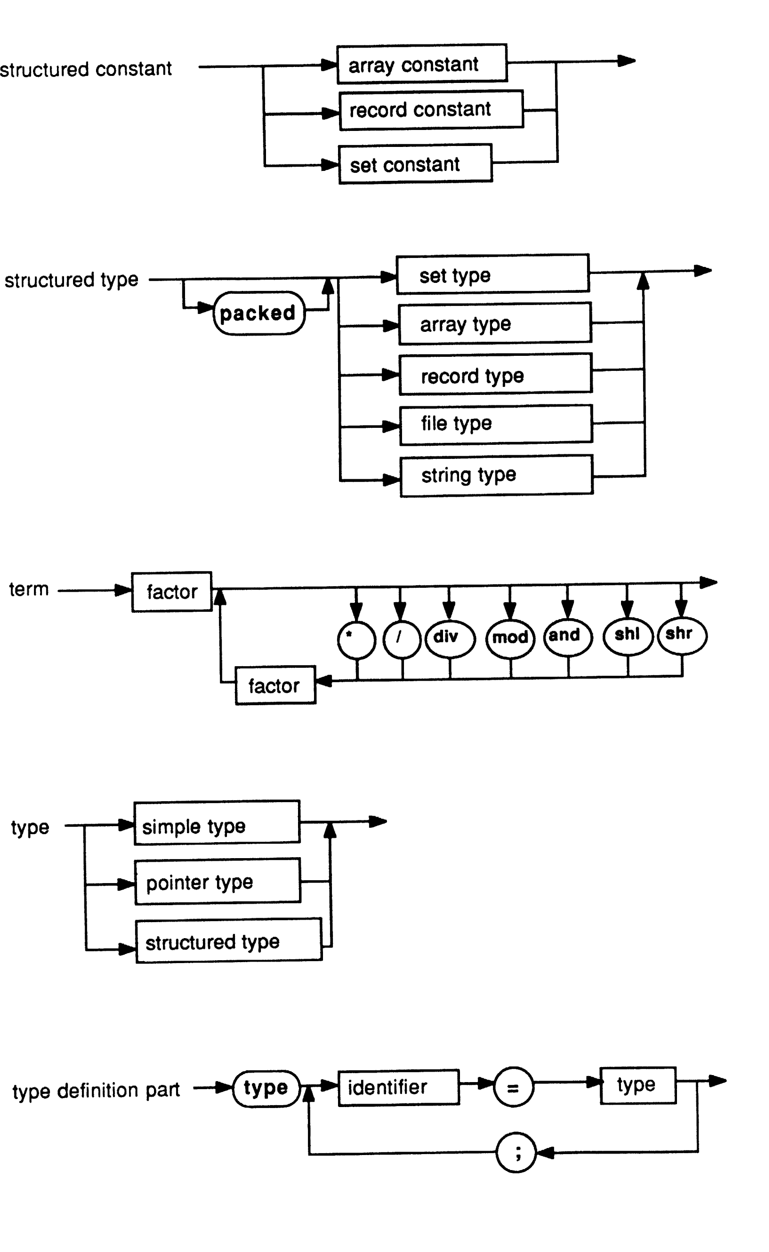

simple type
type identifier
(
identifier
)
,
constant
..
constant

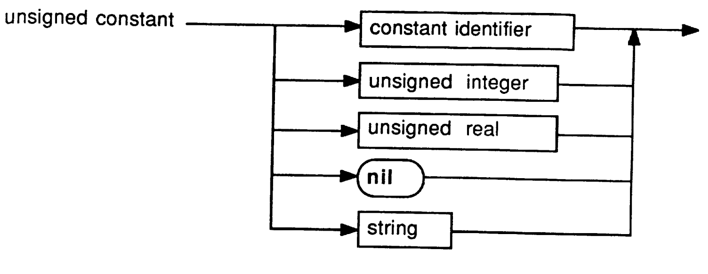

statement
identifier
unsigned integer
:
assignment statement
procedure call
compound statement
if statement
repeat statement
while statement
for statement
case statement
with statement
goto statement
inline statement

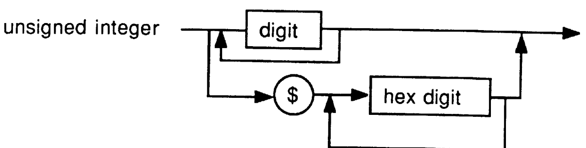

statement
part
can also be defined as
compound statement
begin
statement
;
end

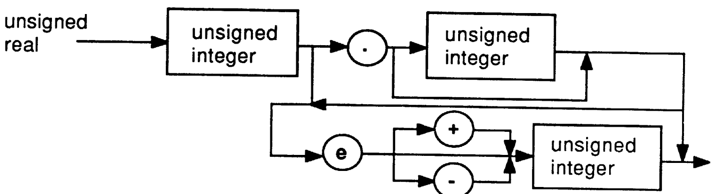

string
constant
'
character
control character
'

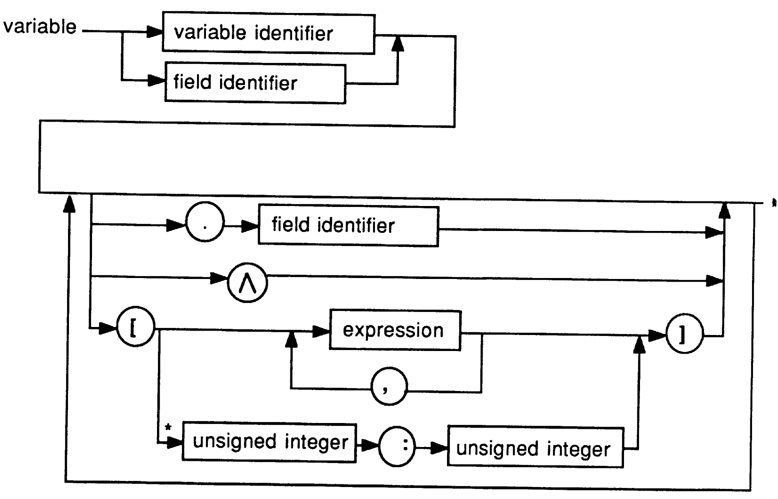

string type
string
[
constant
]


structured constant
array constant
record constant
set constant
structured type
packed
set type
array type
record type
file type
string type
term
factor
*
/
div
mod
and
shl
shr
factor
type
simple type
pointer type
structured type
type definition part
type
identifier
=
type
;
typed
constant
definition
identifier
:
type
=
constant
structured constant

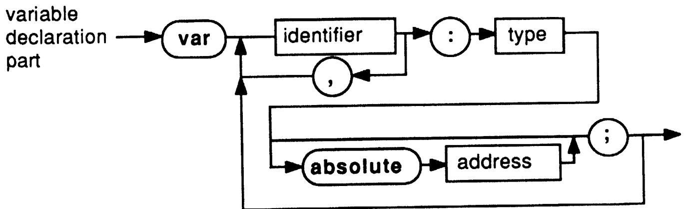


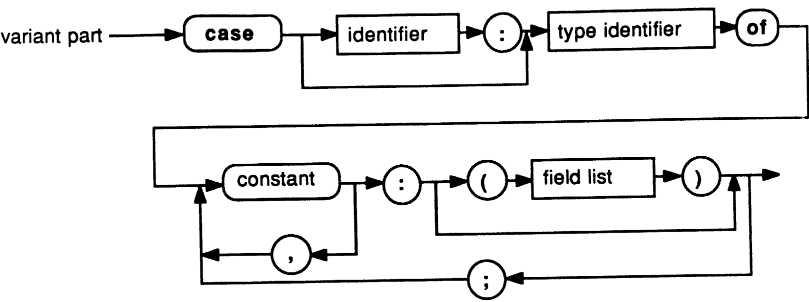




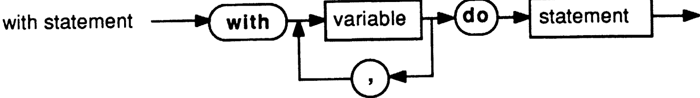


* MS - Dos, CP/M-86 only


variable
declaration
part
var
identifier
:
type
,
;
absolute
address


variant part
case
identifier
:
type identifier
of
constant
:
(
field list
)
,
;

while statement
while
expression
do
statement


with statement
with
variable
do
statement
,

B Exercise Solutions

CHAPTER 7

1. 15
2. 15 (again)
3. 24
4. 15
5. 24
6. 4

Answers to Sample Program Questions

1. Identifiers:
   1 user-defined constants (*YourName*)
   3 user-defined variables (*A*,*B*,*C*)
   3 predeclared identifiers (*integer*, *Readln*, *Writeln*)
4. Change assisgnment statement to
   a. C := 2 * (A−B);
   b. C := A − 2 * B;
   c. C := 5 * A − 3 * B;
   d. C := A * B;
   e. C := A *mod* B;
5. *A* and *B* are of the type integer, and cannot have values of the type real; trying to enter one causes a runtime error.
6. *A* and *B* are now of the type real and can therefore have real values.

CHAPTER 9

First Set

1. No (too big)
2. No (contains comma)
3. Yes (hexadecimal $b = 11 decimal)
4. Yes (predefined in Turbo)
5. Yes (smallest possible number)
6. No ("H" is not a legal hex digit)
7. No (the decimal point makes it a real number)
8. Yes

Second Set

1. 2.0E4 (or 2E4, 20E3,...)
2. −2.5E−5 (or −0.25E−4, −0.025E−3,...)
3. 4.277E1 (or 0.4277E2, etc.)
4. −5.300005E5 (and others)

Third Set

1. .00000000015
2. −5545454000000.0
3. 2.0

CHAPTER 10

1. One way to set *Yesterday* to the previous day of the week reliably would be like this:

```
if DayOfWeek = Monday then
  Yesterday:= Sunday
else
  Yesterday := Pred (DayOfWeek);
```

CHAPTER 27

First Set

1. 50
2. 6188
3. 63
4. 1065

Second Set

1. 203138_5
2. $FFD
3. 101101_2
4. 1235_7

C Summary of Standard Procedures and Functions

This appendix lists all standard procedures and functions available in Turbo Pascal and describes their use, syntax, parameters, and type. The following symbols are used to denote elements of various types:

string	Any string type
type	Any type
file	Any file type
scalar	Any scalar type
pointer	Any pointer type

When a parameter-type specification is not present, it means that the procedure or function accepts variable parameters of any type.

INPUT/OUTPUT PROCEDURES AND FUNCTIONS

The following procedures use a non-standard syntax in their parameter lists:

```
Read;
Read(var V: type);
Read(var F: file of type; var V: type);
Read(var F: text; var I: integer);
Read(var F: text; var R: real);
Read(var F: text; var C: char);
Read(var F: text; var S: string);
Readln;
Readln(var V: type);
Readln(var F: text);
Readln(var F: text; var I: integer);
Readln(var F: text; var R: real);
Readln(var F: text; var C: char);
```

```
Readln(var F: text; var S: string);
Write(var V: type);
Write(var F: file of type; var V: type);
Write(var F: Text; I: integer);
Write(var F: Text; R: real);
Write(var F: Text; B: boolean);
Write(var F: Text; C: char);
Write(var F: Text; S: string);
Writeln;
Writeln(var V: type);
Writeln(var F: Text);
Writeln(var F: Text; I: integer);
Writeln(var F: Text; R: real);
Writeln(var F: Text; B: boolean);
Writeln(var F: Text; C: char);
Writeln(var F: Text; S: string);
```

ARITHMETIC FUNCTIONS

```
Abs(I: integer): integer;
Abs(R: real): real;
ArcTan(R: real): real;
Cos(R: real): real;
Exp(R: real): real;
Frac(R: real): real;
Int(R: real): real;
Ln(R: real): real;
Sin(R: real): real;
Sqr(I: integer): integer;
Sqr(R: real): real;
Sqrt(R: real): real;
```

FILE-HANDLING ROUTINES

Procedures

```
Append(var F: text; Name: string); {PC/MS-DOS, CP/M-86}
Assign(var F: file; Name: string);
BlockRead(var F: file; var Dest: Type; Num: integer);
  {untyped files}
BlockRead(var F: file; var Dest: Type; Num: integer;
          var RecsRead: integer);  {untyped files PC/MS-DOS}
BlockWrite(var F: file; var Dest: Type; Num: integer);
  {untyped files}
BlockWrite(var F: file; var Dest: Type; Num: integer;
           var RecsWritten: integer);
           {untyped files PC/MS-DOS}
Chain(var F: file);
Close(var F: file);
Erase(var F: file);
Execute(var F: file);
Rename(var F: file; Name: string);
Reset(var F: file);
```

```
Reset(var F: file; BlockSize : integer);
  {untyped files PC/MS-DOS}
Rewrite(var F: file);
Rewrite(F: file; BlockSize : integer);
  {untyped files PC/MS-DOS}
Seek(var F: file Pos: integer); {except text files}
LongSeek(var F: file; Pos: real);
  {except text files, PC/MS-DOS only}
```

Functions

```
Eof(var F: file): boolean;
Eoln(var F: Text): boolean;
FilePos(var F: file of type): integer;
FilePos(var F: file): integer;
LongFilePos(var F: file): real;
  {except text files, PC/MS-DOS only}
FileSize(var F: file): integer; {except text files}
LongFileSize(var F: file): real;
  {except text files, PC/MS-DOS only}
SeekEof(var F: file): boolean;
SeekEoln(var F: Text): boolean;
```

HEAP CONTROL PROCEDURES AND FUNCTIONS

Procedures

```
Dispose(var P: pointer);
FreeMem(var P: pointer, I: integer);
GetMem(var P: pointer; I: integer);
Mark(var P: pointer);
New(var P: pointer);
Release(var P: pointer);
```

Functions

```
MaxAvail: integer;
MemAvail: integer;
Ord(P: pointer): integer; {CP/M-80}
Ptr(segment, offset: integer): Pointer; {PC/MS-DOS, CP/M-86}
```

MISCELLANEOUS PROCEDURES AND FUNCTIONS

Procedures

```
Bdos(Func {,Param }: integer); {CP/M-80}
Bdos(Func: integer; Param: record); {CP/M-86}
Bios(Func {,Param }: integer); {CP/M}
Delay(MS: integer);
Exit;
FillChar(var Dest, Length: integer; Data: char);
FillChar(var Dest, Length: integer; Data: byte);
```

```
Halt;
Intr(Func : integer; Param : record); {PC/MS-DOS}
MsDos(Func: integer; Param: record);  {PC/MS-DOS}
Move(var Source,Dest; Length: integer);
Randomize;
```

Functions

```
Addr(var Variable): Pointer; {PC/MS-DOS, CP/M-86}
Addr(var Variable): integer; {CP/M-80}
Addr(<function identifier>): integer; {CP/M-80}
Addr(<procedure identifier>): integer; (CP/M-80}
Bdos(Func, Param: integer): byte;  {CP/M-80}
BdosHL(Func, Param: integer): integer; {CP/M-80}
Bios(Func, Param: integer): byte; {CP/M}
BiosHL(Func, Param: integer): integer; {CP/M}
Hi(I: integer): byte;
IOresult: integer;
KeyPressed : boolean;
Lo(I: integer): byte;
Ofs(var Variable): integer; {PC/MS-DOS, CP/M-86}
Ofs(<function identifier>): integer; {PC/MS-DOS, CP/M-86}
Ofs(<procedure identifier>): integer; {PC/MS-DOS, CP/M-86}
ParamCount: integer;
ParamStr(N: integer): string;
Random(Range: integer): integer;
Random : real;
Seg(var Variable): integer; {PC/MS-DOS, CP/M-86}
SizeOf(var Variable): integer;
SizeOf(<type identifier>): integer;
Swap(I: integer): integer;
UpCase(Ch: char): char;
```

SCALAR FUNCTIONS

Functions

```
Odd(I: integer): boolean;
Pred(X: scalar): scalar;
Succ(X: scalar): scalar;
```

DIRECTORY-RELATED PROCEDURES (PC/MS-DOS)

Procedures

```
ChDir(Path: string);
GetDir(Drv: integer; var Path: string);
MkDir(Path: string);
RmDir(Path: string);
```

SCREEN-RELATED PROCEDURES AND FUNCTIONS

Procedures

```
CrtExit;
CrtInit;
ClrEol;
ClrScr;
DelLine;
GotoXY(X, Y: integer);
InsLine;
LowVideo;
HighVideo;
NormVideo;
```

STRING PROCEDURES AND FUNCTIONS

The *Str* procedure uses a non-standard syntax for its numeric parameter.

Procedures

```
Delete(var S: string; Pos, Len: integer);
Insert(S: string; var D: string; Pos: integer);
Str(I: integer; var S: string);
Str(R: real; var S: string);
Val(S: string; var R: real; var p: integer);
Val(S: string; var I, p: integer);
```

Functions

```
Concat(S1,S2,...,Sn: string): string;
Copy(S: string; Pos, Len: integer): string;
Length(S: string): integer;
Pos(Pattern, Source: string): integer;
```

TRANSFER FUNCTIONS

```
Chr(I: integer): char;
Ord(X: scalar): integer;
Round(R: real): integer;
Trunc(R: real): integer;
```

IBM PC PROCEDURES AND FUNCTIONS

The following procedures and functions apply to IBM implementations only.

Basic Graphics, Windows, and Sound

Procedures

```
Draw(X1,Y1,X2,Y2,Color: integer);
GraphBackground(Color: integer);
GraphColorMode;
GraphMode;
GraphWindow(X1,Y1,X2,Y2: integer);
HiRes;
HiResColor(Color: integer);
NoSound;
Palette(Color: integer);
Plot(X,Y,Color: integer);
Sound(I: integer);
TextBackground(Color: integer);
TextColor(Color: integer);
TextMode(Color: integer);
Window(X1,Y1,X2,Y2: integer);
```

Functions

```
WhereX: integer;
WhereY: integer;
```

Constants

```
BW40: integer;              =  0
C40: integer;               =  1
BW80: integer;              =  2
C80: integer;               =  3
Black: integer;             =  0
Blue: integer;              =  1
Green: integer;             =  2
Cyan: integer;              =  3
Red: integer;               =  4
Magenta: integer;           =  5
Brown: integer;             =  6
LightGray: integer;         =  7
DarkGray: integer;          =  8
LightBlue: integer;         =  9
LightGreen: integer;        = 10
LightCyan: integer;         = 11
LightRed: integer;          = 12
LightMagenta: integer;      = 13
Yellow: integer;            = 14
White: integer;             = 15
Blink: integer;             = 16
```

Extended Graphics

Procedures

```
Arc(X,Y,Angle,Radius,Color: integer);
Circle(X,Y,Radius,Color: integer);
ColorTable(C1,C2,C3,C4: integer);
FillScreen(Color: integer);
FillShape(X,Y,FillColor,BorderColor: integer);
FillPattern(X1,Y1,X2,Y2,Color: integer);
GetPic(var Buffer: AnyType; X1,Y1,X2,Y2: integer);
Pattern(P: Array[0..7] of Byte);
PutPic(var Buffer: type; X,Y: integer);
function GetDotColor(X,Y: integer): integer;
```

Turtlegraphics

Procedures

```
Back(Dist: integer);
ClearScreen;
Forward(Dist: integer);
HideTurtle;
Home;
NoWrap;
PenDown;
PenUp;
SetHeading(Angle: integer);
SetPenColor(Color: integer);
SetPosition(X,Y: integer);
ShowTurtle;
TurnLeft(Angle: integer);
TurnRight(Angle: integer);
TurtleDelay(Ms: integer);
TurtleWindow(X,Y,W,H: integer);
Wrap;
```

Functions

```
Heading: integer;
Xcor: integer;
Ycor: integer;
TurtleThere: boolean;
```

Constants

```
North = 0;
East = 90;
South = 180;
West = 270;
```

D Using Turbo Pascal with Other Borland Products

If you have an IBM PC or compatible, you can enhance the power of Turbo Pascal and many of your other programs with Borland's SideKick, SuperKey, and Turbo Lightning.

If you do not currently own a copy of these products, check with any local computer dealer. If you own one or all of these products, or are simply curious about how they can enhance your programming capabilities in Turbo Pascal, read on.

SIDEKICK

The Notepad

If you know how to use Turbo Pascal's editor, then you already know how to use SideKick's Notepad—all the text-editing keys are the same. Notepad, however, doesn't auto-indent program lines the way Turbo Pascal does, so you'll have to do the indentation yourself if you're using Notepad for editing. By and large you will find the Notepad editor to be perfect for small, programming-related editing tasks.

Reminders

By this time in the tutorial, you have doubtless been warned many times that all identifiers must be declared before they are used. SideKick's Notepad provides a convenient way to avoid violating this rule: With SideKick installed, press Ctrl Alt, then N (or F2), and up comes a Notepad window to hold comments, reminders, or even the actual declarations you want to enter into your program. You can then return to your work, confident that you will be able to pick up all the loose ends later.

Documentation

Documenting your program is always a good idea, whether it's to benefit the next user or programmer, or yourself. Most programmers comment their code; but even the best of us sometimes put off writing the instructions for the user until the last minute, and the documentation suffers as a result.

Notepad lets you jot down instructions *as you write the program*, instead of afterward. Every time you add a new feature or "trick" to your program, pop up the Notepad window and add a line or two to your program notes. Then, when you're finished writing your program, the documentation will be nearly complete.

Include Files

When writing big or complex Turbo Pascal programs, you may want to take advantage of Turbo Pascal's ability to include a number of smaller files in a program. (You can do this by using the *I* compiler directive described in Chapter 17 of the *Turbo Pascal Reference Manual.*) Using Include files is a good way to organize your program, but can sometimes make editing more difficult.

For example, if you are in the middle of writing a program and you need to know what is in a different Include file, you must:

- Save the changes to the file you are currently working on.
- Make the Include file the current work file.
- Find the specified data.
- Go back to your original work file.
- Find the place where you left off in the original file (since Turbo Pascal always places the cursor at the top of the file when the editor is re-entered).

There is an easier way. Notepad can handle any Include file up to 4K in length (longer, if you use the program SKINST to increase the Notepad size). Once the file has been loaded into the Notepad, you can view it (and make changes to it) without leaving the Turbo Pascal editor. All the declarations, comments, and code in the file are available to you without having to reload it every time it's needed. (If you position the Notepad window at the right place on the screen, you should be able to see the code you are writing alongside the code in the Include file.) When you leave Notepad, SideKick puts you back in Turbo Pascal exactly where you left off.

Interruptions

Because good programming requires intense concentration, you will probably want to minimize interruptions while you work. For most of us, however, some interruptions are inevitable—and SideKick can help you deal with them gracefully.

If, for example, an idea that you just can't lose pops into your head (perhaps, you just figured out what to do with that big account at work), you don't have to quit Turbo Pascal to save it. With SideKick, you needn't leave your chair or Turbo Pascal; just jot the information down in Notepad and come back to it later.

The Calculator

The SideKick Calculator was designed with the programmer in mind. It can do calculations in decimal, hex, or binary notation, and can convert between the three at the touch of a key (great for those hackers using the **inline** feature to insert assembly language into a program).

Once a result has been computed with the calculator, it's easy to bring it into the text of your paragraph using the *P* ("program") command. Simply press P while you're in the calculator, followed by whatever key you choose to insert the number on the screen into the text. Then return to Turbo Pascal, press the key you've selected, and the number will appear in your program.

The ASCII Table

Have you ever tried to draw a box on your screen using the graphics characters in the IBM PC character set? Or have you ever wanted to print out (or test for) a character that has some meaning to the Turbo Pascal editor? If you want the codes for these and other special characters at your fingertips, you'll make good use of the SideKick ASCII table.

To bring the table up, enter SideKick by pressing Ctrl Alt, then press A. All characters in the table will be displayed exactly as they appear on the screen. Use the arrow keys to scroll through the pages of the table to find the character(s) you want.

SUPERKEY

SuperKey's macro key capability can improve the already simple and powerful user interface of Turbo Pascal. This section describes the macros included on the SuperKey disk, as well as other macros you can add or customize yourself.

Using the Predefined Macros in TURBO.MAC

To load the standard set of Turbo Pascal macros, press Alt / to activate SuperKey and select Load from the Macros menu. When prompted for the name of a macro file to load, type TURBO.MAC. (Note: If this file is not in SuperKey's current directory, you may need to add directory information to the file name or change disks). When the file is successfully loaded, a help window will appear. To bring this window up any time, press Ctrl F1.

Fast Entry to Turbo Pascal

When starting Turbo Pascal, did you ever wish you could skip the initial "Include error messages?" prompt and get right down to business? The Alt T and Alt W macros let you do just that by supplying a *Y* in response to that prompt. (There's hardly ever a reason not to include the error messages, since they take up only 3K and make programming a lot easier.)

When you press Alt T at the DOS command prompt, SuperKey invokes Turbo Pascal, answers the question with a *Y*, and leaves you at the Turbo> prompt. The Alt W keys, which are used to start a new work file, let you enter the name of the new file. Then this macro jumps into the editor and inserts an {R+} compiler directive. (This directive causes Turbo Pascal to catch range errors in your program, which is quite useful when debugging.) The macro then leaves you in the editor so that you may begin writing your new program.

Turbo Pascal Templates

The five macro keys in Turbo Pascal—Alt F, Alt C, Alt B, and Alt R—let you create the "skeleton" for an entire block of Pascal code with a single keystroke.

Alt P helps you create a procedure by typing the word "procedure," then waits for you to supply the procedure name. When you press ↵, it finishes the job by creating the main **begin/end** block, and helps to document your code by commenting the **end** for you. Alt F helps you define a function in a similar way, stopping to wait for you to type both the function name and the return type. Alt C and Alt R accomplish the same thing using **case** and **repeat...until** statements. Alt B prints out a simple **begin/end** block.

Of course, every programmer has his/her own style, and these macros may not produce code in the exact format you require. Fortunately, all of these macros can be edited with SuperKey's macro editor, enabling you to write code to your own specifications.

Saving your Work

The macros, Alt S, Alt X, Alt M, and Alt N, help you save your work (in case of power failures, random crashes, and so on) and perform other, related functions with a minimum of fuss and bother.

Alt S, when used from within the Turbo Pascal editor, saves your program to disk and returns you to your editing session. Alt X also saves the program, but then exits Turbo Pascal and returns you to the DOS command level. Alt M saves your file and then compiles it (but doesn't start it running), allowing you to check your program for syntax errors. Finally, Alt N saves the file and prepares Turbo Pascal to accept the name of a new file for editing. (Note: This definition of Alt N overrides the use of this key combination to bring up the SideKick Notepad. If you have SideKick and use the notepad, you may wish to define another key for this macro.)

Other Macros

Non-SuperKey macros may also prove useful when you are using Turbo Pascal. One of the most common key combinations a Pascal programmer must type is :=, an awkward sequence requiring you to test your digital dexterity. One way to make this sequence easier (and avoid typographical errors) is to program a single key, such as Ctrl :, to produce the := all at once. To program this new macro into SuperKey, type:

Alt = Ctrl : : = Alt -

Because the Turbo editor only defines two of the IBM PC function keys (F7 and F8), the remaining keys are available for use in defining other macros for the editor. For instance, the key sequences for "find" (Ctrl Q F) and "find and replace" (Ctrl Q A) can be programmed into the function keys F1 and F2 like this:

Find: Alt = F1 Ctrl Q F Alt -

Find and Replace: Alt = F2 Ctrl Q A Alt -

After this is done, you still have six function keys to define.

Other SuperKey Functions

SuperKey has so many diverse and useful functions that we couldn't possibly describe all of them in detail here. Some of the most useful ones for Pascal programmers include:

- *Privacy* to prevent visitors from looking over your shoulder when there's confidential information on your screen.
- *Keyboard lock* to prevent access to your programs when you're away from your desk.

- *Cut and paste* allows you to cut and paste material from other programs to Turbo Pascal (and vice versa).
- *Encryption* to protect your confidential files, and/or convert them to a form easily transmittable by modem.

TURBO LIGHTNING

Turbo Pascal programmers should be sure to turn the Control-Break {$C-} option OFF when writing programs to be used with Turbo Lightning. Otherwise, when you run the program, the keyboard will be flushed and the *Auto-Proof* function can cause a character to be dropped if you type quickly.

For a complete description of any of the features and functions of these products, refer to their respective manuals.

E Programming Style

"Programming style" refers to many aspects of programming—how you format programming commands, the length and capitalization of declared identifiers, the types of programming structures used or omitted, and more. And just as there is no universal style for writing English, there is no universal style for writing programs; however, there are some general rules. Read these suggestions, evaluate them carefully, and choose the ones you find most appealing.

CONSISTENCY: A MATTER OF STYLE

Using good programming style promotes the writing of efficient, readable programs that are easy to maintain and modify.

Don't let yourself fall into the trap of quickly typing in your program, with good intentions to edit it "later on." Instead, type your programs using good programming style the first time through, commenting the code as you go. If you are eager to test an idea or a routine, first create a test file, perfect the routine, and then comment the code before merging it with your main program.

The following conventions—or others like them—will make it easier for you to modify your programs, or to cannibalize routines for use in other applications.

INDENTATION

Always indent nested statements two or three spaces. If you use more than this, any long statements will run off the screen; if you use less, it will be more difficult to visually discern the program blocks:

```
if condition then
begin
  Stmt1;
    ..
  StmtN;
end; {if}
```

```
case i of
  0 :  Write('None found');
  1 :  Write('Found it!');
  else;
end; {case}
```

Take up the habit of aligning associated **begin/end** statements. This will save you hours of thumbing through printouts trying to find a missing **begin**.

SPELLING

To avoid trying to figure out why you abbreviated something and what it stands for, spell things out! For example, use an identifier called *NetPay* instead of *NP*; use *Balance* instead of *ba*. Save those short cryptic names for short simple procedures where long names may get in the way.

Keep in mind that any important identifier should be well documented; consider the following approaches and pick the best one for you:

Terse Identifier Names

```
var                         { global declarations }
  B, D, W : real;

function Nb(b, d, w : real);
begin
  Nb := b + d - w;
end;
```

Descriptive Identifier Names

```
var                         { global declarations }
  Balance
  Deposits,
  WithDrawals : real;

function NewBalance(bal, dep, wid : real);
begin
  NewBalance := bal + dep - wid;
end;
```

Of course, we stacked the deck to prove a point: The first example would require a comment on virtually every line to make the code readable. In the long run, by choosing descriptive identifier names you will actually cut down on excess typing.

Note that the parameter identifiers are shorter than the global ones. That's because most of the work of a "real" program is performed by trusted procedures and functions. Many people use abbreviated lower-case identifiers to designate local variables and parameters. Then, when they are pouring through the code at half-past midnight, they know to look "locally" to trace a particular variable.

Save those one-letter identifiers for scratch variables of little importance, or for more traditional uses like the following:

```
x,y    Represent screen coordinates
i,j    Represent integer indexes into an array
n      Traditional integer loop variable
```

Consider the following (and note the difference in readability):

```
for n := 1 to 7 do              { seems reasonable }
  Read(SomeArray[n]);

for day := 1 to 7 do            { obviously superior }
  Read(SomeArray[day]);
```

PARAMETERS

Use parameters and functions whenever possible. If your programs have scores of global variables and few procedures or functions with parameters, then you have not yet mastered the fundamentals of Pascal. For example, you could type this:

```
var
  OK, Ok1 : boolean;
  FileName  : string[66];

procedure OpenFile;
begin
 ..  { opens file, sets OK to indicate success/failure }
end;

begin
  FileName := 'Test1';
  OpenFile;
  Ok1 := OK;
  FileName := 'Test2';
  OpenFile;
  if Ok1 and OK then ...
end;
```

But then why not try this instead:

```
type
  NameType = string[66];

function OpenFile(FileName : NameType) : boolean;
begin
 ..  { opens file, returns true/false }
end;

begin
  if OpenFile('Test1') and OpenFile('Test2') then ...
end;
```

Ah, that's much better. We eliminated three global variables used inside *OpenFile*. Of course, sometimes you want to use globals, but not nearly as often as you feel compelled to type them.

STRUCTURES

Pascal provides many powerful, high-level structures and constructs (sets, user-defined scalar types, case statements, records, and so on)—use them. Beginners often take some poor overworked structure and use it unnaturally; for example:

```
var
  Answer : char;
begin
  repeat
    Read(KBD,Answer);
    Answer := UpCase(Answer);     { convert to capital }
  until (Answer = 'Y') or                      { yes    }
        (Answer = 'N') or                      { no     }
        (Answer = 'M');                        { maybe }
end;
```

That works, but now look at this:

```
var
  Answer : char;
begin
  repeat
    Read(KBD,Answer);
  until UpCase(Answer) in ['Y','N','M'];  { yes,no,maybe }
end;
```

No big deal, you say. Well, what if there were 20 legal answers instead of only 3? In general, use sets and records to collect "like" objects. Don't leave a pile of related globals lying around haphazardly; tuck them out of sight by putting them into a record, like so:

```
var
  x, y,
  Ht, Wid,
  ForeG, BackG : byte;
  Title : string[20];

procedure DrawWindow;
begin
  { has to "remember" all the globals, can only
    work on one window definition unless you want
    to pass all those guys as parameters separately }
end;
```

Compare the previous one with the following:

```
type
  WindowRec = record
                x, y,
                Ht, Wid,
                ForeG, BackG : byte;
                Title : string[20];
              end;
```

```
var
  MainWindow : WindowRec;

procedure DrawWindow(w : WindowRec);
begin
  with w do
  begin
    { ... draw the window ... }
  end;
end;
```

That's much cleaner—now you can pass different windows to the routine. Similar objects are placed in a well-labeled container and handed over to the *DrawWindow* procedure.

Become familiar with user-defined scalars:

```
var
  Month : 1..12;
case Month of
  12..2 : Writeln('Winter');
  3..5  : Writeln('Spring');
  6..8  : Writeln('Summer');
  9..11 : Writeln('Autumn');
end;
```

That's fine, but Pascal let's you write English with no additional overhead. For example:

```
type
  Months = (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
var
  Month : Months;

case Month of
  Dec..Feb : Writeln('Winter');
  Mar..May : Writeln('Spring');
  Jun..Jul : Writeln('Summer');
  Aug..Nov : Writeln('Autumn');
end;
```

The following scheme really comes in handy for menu routines:

```
repeat
  Selection := GetChoice([Reports..Quit]); { legal choices }
  case Selection of
    Reports : PrintReports;
    Edit    : EditFile;
    Setup   : ChangeSetup;
  end; { case }
until Selection = Quit;
```

CONSTANTS

Develop the habit of using constants wherever possible. For example:

```
const
  MenuRow   = 10;
  MenuCol   = 10;
  MenuItems =  5;
procedure GetMenuItem(var item : integer);
begin
  repeat
    GoToXY(MenuCol, MenuRow);
    ClrEOL;
    Read(item);
  until item in [1..MenuItems];
end;
```

The advantage here is clear: If you need to add more menu items, you don't have to search for every occurence of the number 5 in your program. In addition, using constants makes it much easier to use the same routine in a different program (or another part of the same program).

COMMENTS

We've already discussed the basics of comments, but here are some new ideas.

There are different kinds of comments, each with an appropriate place. For instance, always place a long, explanatory comment at the beginning of your program:

```
{ Written 01/01/86 by Ace Coder.
Last modified 01/02/86 by More AlertCoder.
O/S:      PC DOS 2.0 or later, CP/M 2.2 or later
          MS DOS 2.0 or later, CP/M 86 1.1
Systems:  All machines that run Turbo Pascal
Memory:   64K or greater
This program creates a feeling of euphoria in
the user by ...
}
```

You can use other comments to date-stamp modifications, both at the top of the file (like in the previous one) and in the code:

```
i := Pred(i);   { Modified 01/06 by FB }
```

You can also place comments next to each global variable to explain what it does and which routines use or modify it:

```
var
  Error : boolean;  { set by IOerror if an
                      I/O error occurs.  Error msg
                      is displayed by PaintMsg }
```

Then again you can place a comment beneath every procedure and function to explain what each one does, which global variables it uses (if any), and when it was last modified (if ever):

```
procedure Abort(msg : StringType);
{ Displays the message, then halts program execution }
```

And if you port the routine from another program or extract it from somewhere else, mention it in a comment so you'll know who to praise later.

SUMMING IT UP

We could go on, but then we'd probably just keep trying to convince you to use *our* style conventions. Instead, be reasonable—find a style that is easy to maintain and readable to your fellow programmers (and also to yourself six months down the road). Don't believe anyone if they tell you their style is the only way; instead, remember that your objective is to write well-behaved, well-documented code. It is our experience that the two generally accompany each other.

Happy coding.

F Using Turbo Pascal with CP/M-80 and CP/M-86 Systems

We have mentioned information applicable to CP/M-80/86 systems throughout this manual, but much of the system-specific details pertain to PC-DOS and MS-DOS systems. This chapter offers those CP/M-80/86 users some basic information about Turbo Pascal operation with their systems. For more detailed information, refer to the *Turbo Pascal Reference Manual*.

MAKING BACKUPS

CP/M-80 and CP/M-86 systems provide a utility called PIP.COM (PIP.CMD for CP/M-86) that will copy files onto a disk, one file at a time. Even if your system provides a utility that copies a disk all at once, it is a good idea to use the PIP utility for your first copy. (This is because PIP allows a single-sided disk to be copied onto a machine that has a double-sided native format.)

To make a backup of your Turbo Pascal system disk, follow these steps:

1. Place your CP/M system master in drive A, and a new disk to be formatted in drive B.
2. Format the disk in drive B by using your system's formatting utility. If there is an option for copying CP/M onto the disk at the same time, use this option and skip step 3.
3. Copy CP/M onto the disk in drive B using the operating system's copying utility. This utility is generally called SYSGEN, and can be used by typing in the utility name and the destination disk drive, like so:

```
A> Sysgen b:
```

(If your system does not have SYSGEN, consult your documentation for an equivalent utility.)

4. Now copy the program PIP.COM onto the disk in drive B:

   ```
   A> pip b:=a:pip.com
   ```

   For CP/M-86:

   ```
   A> pip b:=a:pip.cmd
   ```

5. Take the disk out of drive B and place it into drive A. Insert your Turbo Pascal or Turbo Tutor disk (whichever one you wish to copy) into drive B and reboot. If you have a reset button that reboots the machine, press it. If you don't, then turn the computer off and then on again for a complete reset.

6. At the A⟩ prompt, get a directory listing of all the files on drive B by typing:

   ```
   A> dir B:
   ```

7. After the A⟩ prompt, copy all of the files on drive B to drive A by typing:

   ```
   A> pip a:=b:*.*
   ```

   Your system will display each file name as it is copied from drive B to drive A.

Now you have a copy of the Turbo Tutor (or Turbo Pascal) distribution disk and the CP/M system, too. You can use this disk to boot your machine, as well as use the programs on it.

INSTALLATION AND BDOS ERRORS

On all CP/M 2.2 versions (and some CP/M-86 versions), when you remove a disk from one of the drives, any disk you put into that drive is automatically set to read-only status. Thus, if you try to write to that disk, you will get the error

```
BDOS ERROR ON X:R/O
```

where *X* is the name of the drive and R/O means read-only. On these systems, you must boot the system with the same disks in the same drives you plan to use throughout your current session.

Thus, if you decide to change disks in the drive where you've installed Turbo Tutor, you will get a read-only error. If this happens and you have followed the preceding steps for copying the Turbo Tutor distribution disk to a system disk using the PIP utility, then press Ctrl C at the prompt and rerun TINST. If you have not followed the steps, go back to review them and make a new disk. Then boot your system with the new disk before re-running the installation program.

COMPILING TO DISK

Be sure that your Turbo disk is correctly logged in before compiling with the .COM option, otherwise, you'll get a read-only error if you change disks in the drive running Turbo Pascal. For CP/M-86 systems, this is the only difference between compiling to disk on your system and the MS-DOS and PC-DOS systems.

For CP/M-80 systems, the method for selecting a .COM file from the compiler options menu is the same as that described in Chapter 6 (see the section entitled "Saving Your Compiled Program"). There are differences, however, in what is displayed on your screen when you enter the compiler options menu. Your screen will show:

```
compile-> Memory
          Com file
          cHn file
Find run-time error  Quit

>
```

When you press [C](for **Com** file), your screen will display:

```
          Memory
compile-> Com file
          cHn file
Start address: 20E9  (min 20E9)
End   address: F042  (max F306)

Find run-time error   Quit

>
```

The **Start** and **End** address options allow you to change the beginnning or ending address of your program. These are usually used when making room for assembly code modules, compiling files to be used with chaining, or moving .COM from one computer to another.

Now press [Q] and [↵] to return to the main menu. At the main menu, press [C] for **Compile**. Your screen should now look like this:

```
Logged drive: A
Work file : A:FIRST.PAS
Main file :

Edit      Compile     Run    Save

eXecute   Dir         Quit   compiler Options

Text:    111 bytes
Free:  27902 bytes

>

Compiling->  A:FIRST.COM
6 lines
```

```
Code:     88 bytes (20E9-2141)
Free: 52856 bytes (2142-EFBA)
Data:    135 bytes (EFBB-F042)
```

The remainder of the lesson in Chapter 6 (from "Main Menu Overview" on) is applicable to your CP/M-80 system.

THE EXECUTE OPTION

The CP/M-80 implementation of Turbo Pascal has a handy feature: It can run compile, and create a .COM file from your program. Then press [X] in the main menu of Turbo Pascal to bring up the following prompt:

```
Program:_
```

Next, type in the name of your .COM program (in Chapter 6, the name FIRST was used) and press [↵] to make it run. When the program completes, Turbo Pascal will reload the source program and return you to the system prompt:

```
Loading TURBO.MSG
Loading FIRST.PAS
>_
```

Now press [↵] to return to the main menu.

G Common Questions and Answers About Turbo Pascal

GENERAL

How much RAM do I need to run Turbo Pascal?

You'll need at least 48K on a CP/M-80 machine and 128K on a 16-bit or PC-compatible machine.

Are variables initialized automatically in Turbo Pascal?

Turbo doesn't initialize user-defined variables at runtime. The programmer must initialize a variable before it can be used.

My program runs correctly in memory, but crashes or performs differently when I run the .COM file. What's wrong?

There are several possibilities:

- You are using a variable or data structure that has not been initialized.
- You are going out of bounds on an array or a string and consequently overwriting something in memory. Set the *R* compiler directive to {$R+}.
- You are using a pointer that has not been properly allocated, which can cause a program to overwrite something in memory.
- You are using assembly language externals and Turbo Pascal version 2.0 on MS-DOS, PC-DOS, or CP/M-86. Under these conditions, external assembly code is not always transferred properly during a compile to disk. It is necessary to compile first in memory, then, without running the program, select the .COM option using the compiler options menu and recompile.

- You may be overwriting memory somehow. Suspect any code that uses

    1. MEM or MEMW arrays or pointers
    2. absolute variables
    3. INLINE, externals, or interrupt calls
    4. *FillChar* or *Move* statements

What causes the runtime error F0?

There are four possible causes: (1) a recursive routine that is overlaid; (2) a procedure that calls another procedure in the same overlay group; (3) calling for an overlay inside a read or write statement, which is not allowed; and (4) insufficient file handles when calling for an overlay, a .CHN, or an *Execute* (MS/PC-DOS only).

Why do I get an I/O error F0 when I try to Append *to a text file?*

You cannot use the *Append* procedure on an empty file. The existing file must have text in it in order to successfully append.

Why am I getting a compile-time error in my **type** *declaration for a large data structure?*

It may be over 64K in size (Turbo's upper limit for the size of a data structure).

Why do I keep getting a type mismatch with the labels I'm using in my **case** *statement?*

You may be trying to use strings as labels in your **case** statement. Pascal only allows simple types to be used as **case** statement labels. In addition, the labels must be *constants*, not variables or typed constants.

How do I assign an integer variable a value of −32768?

By assigning the integer a value of $8000.

When I change my program into a .CHN/Execute *file, the program either hangs, gives a memory allocation error, or gives erroneous results. What am I doing wrong?*

For *Chain* and *Execute* to work, you must set the minimum code and data segment to the size of the code and data of the largest program in the *Chain*ed or *Execute*d series (on CP/M-80 systems, you must adjust the end address). These settings can be changed using the compiler **O**ptions menu.

When using the functions EOF *and* EOLN *on a file, my program seems to hang. What's the cause?*

Turbo Pascal adds an extension to the functions *EOF* and *EOLN*. This extension lets you pass to the two functions a parameter specifying which file you are checking (for example, *EOF*(*FileVariable*)). If you do not specify this optional parameter, then set the *B* compiler directive.

GotoXY isn't working in my program. What am I doing wrong?

The most common mistake is reversing the row and column coordinates. They should read as:

```
GotoXy(Column,Row);
```

where 1 <= column <= 80 (on most machines)
and 1 <= row <= 25 (or 24 lines on most generic machines)

How do I get a real number printed in non-exponential notation?

You must use real formatting:

```
Writeln(R:14:3)
```

This means write the value of *R*, use a field width of 14 characters, 3 of which should be to the right of the decimal point.

When I use FillChar *on a string, the string gets messed up. Why?*

Remember that the zero'th byte of a string is used to hold the current length of the string. Immediately after using the *FillChar* routine, you must be sure to set the length byte of your string to the appropriate value.

I have a **for** *loop that writes to the string position using index (str1[i]). However, when I write out the string, it has its old length. Why?*

When updating the value of a particular index of a string, you must update the length byte. We recommend using the *Insert* procedure to change the value of a particular character in a string, since all the string manipulation routines in Turbo Pascal automatically change the length of the string.

What is the maximum length of a string in Turbo Pascal?

255 characters

How do you declare an enumerated type and use it in a **for** *loop?*

Try using this code:

```
...
type
  Num = (one,two,three);
var
  Count : Num;
begin
  for count := one to three do
    Write('.');
end.
```

How can I access command-line parameters?

Use the *ParamStr* and *ParamCount* functions described in Chapter 16 of the *Turbo Pascal Reference Manual*. These functions allow complete access to the command line from your Turbo Pascal program.

How do you access a file on another disk drive?

When assigning the drive, make your file name 'B:filename' or use Chdir('B:') (MS/PC-DOS only).

When I write my linked list data structure to a disk file, why doesn't it store properly?

Linked lists are dynamic data structures that can only be properly allocated/represented in memory. In order to store the information in a dynamic data structure to a disk file, you must write a routine that traverses the entire linked list and writes each piece of data to your file. When you read the information back from the file, you must reconstruct your linked list.

I made a file with a text editor, and now I'm trying to read it as a record file and it doesn't work. What's wrong?

Record files use a different data format from text files. You'll need to write a program to convert your data from text to record files.

How do I get typeahead in a Turbo Pascal program?

First set the *C* and *U* compiler directives to {$C-} and {$U-}. This will prevent Turbo Pascal from clearing the keyboard buffer during screen I/O. From now on, whenever you do any reads, you can read one character at a time from the logical device KBD: *Read(KBD,ch)*.

Why can't I read more than one integer/character/real on a line using a **repeat** *loop and a* read(ch)*?*

The following routine reads an input line of characters, requiring the user to press ⏎ only once.

```
var
  ch : char;

begin
  repeat
    read(TRM,ch); { read from the logical device TRM }
    write(ch);
  until Ch = #13;
end.
```

You can read from the TRM device, *read(TRM,var1)*, for version 3.0 for standard input, or use the compiler directive {$B-} for version 2.0.

Why can't I read/write from the logical device AUX?

Turbo Pascal treats the logical device AUX exactly like a text file. Because of the BIOS design, most users find that they have great difficulty trying to write serial communication routines using reads and writes from AUX. We recommend writing your own interrupt service routines to check the status of the serial port before you try doing a read or write.

16-BIT MACHINES ONLY (including IBM PC)

Why would I get the I/O error F3?

Because you are trying to use too many file handles. MS-DOS and PC-DOS limit a program to a maximum number of file handles. You can raise the number to 20 using this line in your CONFIG.SYS file:

```
FILES = 20
```

This will allow you to use up to 15 files in your program (DOS uses 5). When all handles have been used, you must close some files before opening any new ones.

How can I use more than 64K of variables?

You can expand the amount of data space available to your program by using pointer variables.

How do I get the time/date in Turbo Pascal?

The files DOSFCALL.DOC and INTRPTCL.DOC on your Turbo Pascal disk demonstrate how to get the date and time.

How can I tell if my printer is ready to print?

You can check for the printer's status by polling DOS interrupt 17.

I wrote an interrupt handler, but it doesn't work. Do you mask interrupts during I/O?

No. You are probably using global variables, but DS has the wrong value after you enter the interrupt procedure. Save the value of DS in the code segment (that is, in a typed constant) and restore it within the interrupt handler.

What interrupt is used by the MS-DOS and PC-DOS implementations of Turbo Pascal to handle the keyboard?

Turbo Pascal uses interrupt 16 to check the keyboard status.

I can't get Turbo Pascal to load on my DEC Rainbow. Why?

Make sure you are using the DEC format, not the MS-DOS format.

I'm having trouble running Turbo Pascal on Concurrent PC-DOS. Why?

We recommend using the MS-DOS generic implementation of Turbo Pascal under Concurrent PC-DOS.

How can I use Turbo Pascal to write to DOS's null device?

Use the following code:

```
var
  T : text;
begin
  Assign(T, 'Nul');
  Rewrite(T);
  Writeln(T, 'help');
  Close(T);
end.
```

IBM PC ONLY

Does Turbo Pascal 3.0 for the IBM PC use direct screen writes in its editor?

Yes.

How can I get inverse video when running Turbo Pascal?

After executing the following statements, all text will be displayed in inverse video:

```
begin
  TextColor(Black);
  TextBackground(White);
end;
```

Any idea why I can't get the sample graphics programs on the Turbo Pascal disk to run on my system?

You must have an IBM Color Graphics Adapter (CGA) card or compatible in order to use the built-in graphics abilities of Turbo Pascal.

What resolution setting does the Hires graphics mode on the IBM PC version of Turbo Pascal require?

Hires is set at (640×200).

How can I change the border color on a CGA?

The following program will let you change the border color to blue (substitute any color you wish):

```
begin
  Port[$3D9] := Blue;
end.
```

Will Turbo Pascal's graphics run on the Hercules Graphics card?

Turbo Pascal's built-in graphics will only run on the IBM Color Graphics Adapter card, or something compatible with this card. To write graphics programs using Turbo with the Hercules card, you can use the Turbo Graphix Toolbox.

Why doesn't my graphics program run on my Paradise Modular Graphics card?

The graphics routines built into Turbo Pascal and included in the extended graphics (GRAPH.P) will only run on an IBM Color Graphics Adapter card (or compatible).

How can I hide the cursor in Turbo Pascal?

The following routine turns the cursor on and off in Turbo Pascal:

```
procedure SetCursor(On:boolean);
var
  reg : record
    ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
  end;
begin
  with reg do
  begin
    if On then              { turn cursor on }
      cx := $C0B { $706 if on color monitor }
    else                   { turn cursor off }
      cx := $20;
      bx := 0;
      ax := $0100;
  end;  { with }
  intr($10,reg);
end;    { procedure SetCursor }
```

How can I find out if the Caps Lock/Num Lock *keys are on?*

The status of these keys is kept in RAM at address $40:$17.

How do I get the PrtSc *key on an IBM PC to work with a Turbo Pascal program?*

To re-enable DOS's standard I/O redirection capabilities, the *G* and *P* compiler directives must be set in your program: {$P128,G128}.

Why can't I get Turbo Pascal to run under Topview?

You must have Turbo Pascal version 3.01 to use under Topview, and you must install Topview with the following parameters:

```
Does it read directly from the keyboard? Yes
Does it access video RAM directly? Yes
```

Is the 8087 version of Turbo Pascal compatible with the 80287 co-processor?

Yes.

I have the 8087 version of the compiler. The program compiles but it doesn't run. Why not?

For version 2.0 users: Turbo Pascal does not check for the 8087 at compile time; instead, it tries to use it at runtime. If it is not there, Turbo Pascal will wait until you respond.

For version 3.0 owners: Turbo Pascal will not allow compilation on a machine without an 8087. If the program is compiled and taken to a machine without an 8087, it will crash.

How fast is the 8087?

The 8087 version of Turbo performs real-number calculations approximately 10 times faster than a non-8087 compiler.

Is there a switch in the BCD or 8087 compilers that lets you use regular real number arithmetic?

No, they are separate compilers.

Is Turbo Pascal as fast as the regular Turbo compiler?

If you are using real numbers, Turbo-BCD will run more slowly than regular Turbo Pascal. Also note that Sin, Cos, Exp, and Ln are not implemented in Turbo-BCD.

CP/M-80 MACHINES ONLY

Can I use the program I developed under CP/M-80 on my IBM PC?

Yes, you can, provided there are no machine-specific calls in your code and that you recompile the source code on an IBM PC implementation of Turbo Pascal.

What software do I need to get Turbo Pascal up and running on my Osborne executive computer?

You need the Osborne version of Turbo Pascal and BIOS revision 1.21 or greater.

How can I get Turbo Pascal 3.0 to run on my Bondwell CP/M-80 computer?

To run 3.0 on your Bondwell, you will have to contact Bondwell to get a patch to their BIOS.

H Glossary

8087 A high-speed math co-processor available for 8086-based machines.

ASCII set The American Standard Code for Information Interchange's standard set of numbers used to identify the characters and control signals used by computers.

absolute variable A variable declared to exist at a fixed location in memory, rather than letting the compiler determine its location.

actual parameter A variable, expression, or constant that is substituted for a formal parameter in a procedure or function call.

address A specific location in memory.

algorithm A set of rules that define the solution to a problem.

allocate To designate memory space for a particular purpose.

array A sequential group of identical data elements that are arranged in one data structure and are accessible by an index.

argument An alternative name for a parameter (see actual parameter).

assignment operator The symbol :=, which assigns a value to a variable or function of the same type.

assignment statement A statement that assigns a specific value to an identifier.

assembler A program that converts assembly language programs into machine language.

assembly language The first language level above machine language. Assembly language is specific to the microprocessor it is running on. The major difference between assembly language and machine language is that assembly language provides mnemonics that make it more readable.

binary A method of representing numbers using base 2 notation, where the only digits are 0 and 1.

binary coded decimal (BCD) A method of floating-point arithmetic that prevents the normal round-off error inherent in computer-based arithmetic.

bit A binary digit with a value of either 0 or 1. The smallest unit of data in a computer.

block The associated declaration and statement parts of a program or subprogram.

boolean A data type that can have a value of TRUE or FALSE.

buffer An area of memory allocated as temporary storage.

byte A sequence of adjacent bits (by convention) consisting of 8 bits.

case label A constant, or list of constants, that label a component statement in a case statement.

case selector An expression whose result is used to select which component statement of a case statement will be executed.

central processing unit (CPU) The "brain" of a computer system that interprets and executes instructions.

chaining The passing of control from one program to another.

char A type that represents a single character.

code segment The place in memory where a program's code is stored.

comment A note or explanation in the source code enclosed by the symbols (* *) or { }.

compiler A program that translates a program written in a high-level language into machine language.

compiler directive An instruction to the compiler that is embedded within the program; for example, {$R+} turns on range-checking.

compound statement A series of statements surrounded by a matching set of the reserved words **begin** and **end**.

concatenate The joining of two or more strings.

constant A fixed value in a program.

control structure A statement that manages the flow of execution of a program.

data segment The segment in memory where the static global variables of a program are stored.

data structure A structure built of many data elements that are all of the same type. It is used for convenient storage, retrieval, and manipulation of data.

debugger A special program that provides capabilities to start and stop execution of a program at will, as well as analyze values that the program is manipulating.

decimal A method of representing numbers using base 10 notation, where legal digits range from 0 to 9.

declare The act of explicitly defining the name and type of an identifier in a program.

dereferencing The act of accessing a value pointed to by a pointer variable.

definition part The part of a program where constants, labels, and structured types are defined.

delimiter A boundary marker that can be a word, a character, or a symbol.

directory A work area on a disk (an MS/PC-DOS feature); a listing of files or directories on a disk.

dynamic allocation The allocation and de-allocation of memory at runtime through the use of pointers.

dynamic variable A variable on the heap.

enumerated type A user-defined type that consists of a list of identifiers in which the order and identifier names are determined by the programmer.

expression Part of a statement that represents a value or can be used to calculate a value.

extension Any addition to the standard definition of a language.

external A file of one or more subprograms that have been written in assembly language and assembled to native executable code.

field list The field name and type definition of a record.

field width The number of place holders in an output statement.

file A collection of data that can be stored on and retrieved from a disk.

file pointer A pointer that tracks where the next object will be retrieved from within a file.

file variable An identifier in a program that represents a file.

fixed point notation The representation of real numbers without decimal points.

flag A variable, usually of type integer or boolean, that changes value to indicate that an event has taken place.

floating-point notation The representation of real numbers using decimal points.

formal parameter An identifier in a procedure or function declaration heading that represents the arguments that will be passed to the subprogram when it is called.

forward declaration The declaration of a procedure or function and its parameters in advance of the actual definition of the subroutine.

function A subroutine that computes and returns a value.

global variable A variable declared in the main program block that can be accessed from anywhere within the program.

heap The area of memory reserved for the dynamic allocation of variables.

hexadecimal A method of representing numerals using base 16 notation, where legal digits range from 0 to 9 and *A* to *F*.

high-level language A programming language that is closer to human language than to machine language.

identifier A user-defined name for a specific item.

increment To increase the value of a variable.

index A position within a list of elements.

initialize The act of giving a known initial value to a variable or data structure.

input The data a program receives from some external device.

integer A numeric variable that is a whole number in the range −32768 to 32767.

interactive A program that communicates with a user through some I/O device.

interrupt The temporary halting of a program in order to process an event of higher priority.

I/O Short for Input/Output, this is the process of receiving or sending data.

I/O error An error that occurs while trying to input or output data.

I/O redirection The ability in PC-DOS/MS-DOS to direct I/O to access devices other than the default MS-DOS devices.

interpreter A program that sequentially interprets each statement in a program into machine code, and then immediately executes it.

iteration The process of repetition or looping.

label An identifier that marks a place in the program text for a GOTO statement.

linked list A dynamic data structure that is made up of elements, each of which point to the next element in the list through a pointer variable.

local identifier An identifier declared within a procedure or a function.

local variable A variable declared within a procedure or a function.

machine language A language consisting of strings of 0s and 1s that the computer interprets as instructions.

main program The main statement part of a program from which all its subprograms are executed.

module A self-contained routine or group of routines.

nesting The placement of one unit within another.

nil pointer A pointer value that is undefined; that is, it doesn't point to anything.

node An individual element of a tree or list.

object code The output of a compiler.

offset An index within a segment.

operand An argument that is combined with one or more operands and operators to form an expression.

operating system A program that manages all operations and resources of the computer.

operator A symbol, such as +, that is used to form expressions.

operator hierarchy The rules that determine the order in which operators in an expression are evaluated.

ordinal type An ordered range of values.

overflow The condition that results when an operation produces a value that is larger or smaller than the computer can represent, given the allocated space for the value or expression.

overlay A group of subprograms stored in an external file, all of which share the same part of the code segment.

parameter A variable or value that is passed to a procedure or function.

parameter list The list of value and variable parameters declared in the heading of a procedure or function declaration.

pointer A variable that points to a specific memory location.

pop The removal of the top-most element from a stack.

port An I/O device that can be accessed through the CPU's data bus.

predefined identifier A constant, type, file, logical device, procedure, or function that is available to the programmer without having to be defined or declared.

procedure A subprogram that can be called from various parts of a larger program.

procedure call The invocation of a procedure.

push The addition of an element to the top of a stack.

queue A data structure in which the first element placed in the data structure is the first element to be removed.

random access Directly accessing an element of a data structure without sequentially searching the entire structure for the element.

random access memory (RAM) The memory device that can be read from and written to.

range-checking A Turbo Pascal feature that checks a value to make sure it is within the legal range defined.

read-only memory (ROM) The memory device from which data can be read but not written.

real number A number represented by decimal point and/or scientific notation.

record A structured data type referenced by one identifier that consists of several different fields.

recursion A subprogram or program that calls itself.

relational operator The operators, =,<>,<,>,<=,>=, and **IN**, all of which are used to form boolean expressions.

reserved word An identifier reserved by the compiler.

scalar type A Pascal data type consisting of ordered components.

scope The visibility of an identifier within a program.

segment On 8088-based machines, RAM is divided into several segments, or parts, each made up of 64 Kb of memory.

sequential access The ordered access of each element of a data structure, starting at the first element of the structure.

set An unordered group of elements, all of the same scalar type.

set operator The symbols, +,- ,*,=,<=,>=,<>, and **IN**, all of which operate on set-type operands.

simple type A predefined or user-defined scalar type.

source code The input to a compiler.

stack A data structure in which the last element stored is the first to be removed.

stack overflow An error condition that occurs when the amount of space allocated to the computer's stack is used up.

stack segment The segment in memory allocated as the program's stack.

statement The simplest unit in a program; statements are separated by semicolons.

static variable A variable with a lifetime that exists the entire length of the program. Memory for static variables is allocated in the data segment (or area).

string A sequence of characters that can be treated as a single unit.

structured type One of the predefined types (array, set, record, file, or string) that are composed of structured data elements.

subprogram A procedure or function within a program; a subroutine.

subrange A continuous range of any scalar type.

subscript An identifier used to access a particular element of an array.

syntax error An error caused by violating the rules of a programming language.

terminal An I/O device for communication between a user and a computer.

tracing Manually stepping through each statement in a program in order to understand the program's behavior; an important debugging technique.

transfer function A function that converts a value of one type to a value of another type.

tree A dynamic data structure in which a node may point to one or more other nodes.

turtlegraphics An intuitive, coordinate-based graphics system.

type definition The specification of a type based upon other types that are already defined.

typed constant A variable with a value that is defined at compile time, but can be modified at runtime. (You can think of it as a preinitialized variable.)

untyped parameter A formal parameter that allows the actual parameter to be of any type.

value parameter A procedure or function parameter that is passed by value; that is, the value of the parameter is passed and cannot be changed.

variable declaration A declaration that consists of the variable and its associated type.

variable parameter A procedure or function parameter that is passed by reference; that is, the address of the parameter is passed so that the value of the parameter can be accessed and modified.

variant record A record in which some fields share the same area in memory.

word A location in memory occupying 2 adjacent bytes.

Index

A

Absolute variables, 307–309
Addition operators, 62–63
Algorithms, bibliographic reference to, 274
Allocation (*see* Dynamic allocation)
ANIMALS.PAS, 6, 254
Animation, real-time, 234–236
Append procedure, 221
Application software, 16
Array assignment, 151–152
Array constant, 295–296
Arrays, 147–155
  base type of, 148, 150
  index type of, 148–150
  initializing, 152–153
  multidimensional, 150
  packed, 155
  range-checking and, 152
  representing in memory, 153–154
  sparse, 260–263
    mixed, 262
Art of Computer Programming, The, bibliographic reference to, 251
ASCII character set, 17, 94
ASCII table, 95
Assemblers, 16, 22
Assembly language, 22–23
  8088/8086, used with Turbo Pascal, 323–334
  allocating local variable space, 327
  allocating static storage, 327–328
  external subprograms, 323–327
  inline statement, 330–334
  interrupt handing, 332
  libraries, 328–330
Assign procedure, 213–214
Assignment statement, 66

B

Backup copies of disks, 30–31, 36
  with CP/M-80 and CP/M-86 systems, 377–378
Base type, of subrange, 108
Binary numbering system, 13–14
  compared to hexadecimal system, 317
Binary search, 271–272
Binary trees, 254–258
  inserting into, 255–256
  searching, 254–255
  traversing, 256–257

Bit, description of, 13
Boolean data type, 55, 93
Boolean expressions, 113–116
Boolean operators, 113–115
  operations on integers and bytes, 319–320
Boolean values, Turbo's extension for, 209
Byte, description of, 14
Byte data type, 55, 90
Byte values, memory storage of, 318–319

C

Case statement, 123–125
  constant list in, 125
Central Processing Unit (CPU), 12
Chain/Execute procedures, 289–290
  vs. overlays, 291
Chaining, 287–289
Char data types, 55, 94–96
Characters, 16–19
  ASCII, 17, 94–95
  control (non-printing), 17, 94, 97–98
  extended character set, 17
  printing, 16–17
ChDir procedure, 222
Children, of tree, 253
Chips, computer, 14
Circular linked lists, 241–245
Close procedure, 215
Command names, 41
Comment delimiters, 82
Comments, 66–67, 71–72, 81–84
Compiler directive, 107
Compilers, 16, 25–26
Compiling a program, 45–47
  with CP/M systems, 379
Compound statements, 80–81
  if statement, 112–113
Computer numbering systems, 313–319
  binary, 13–14
  decimal, 313–314
  hexadecimal, 88, 313–314
  place value, 313–316
Conditional control structure, 111–116
  boolean expressions, 113–116
  if statement, 111–113
Constants, 59–61, 293
  passing as var parameters, 301
  (*see also* Typed constants)
Control characters, 17, 94
  in a string, 97–98
Control structures, 111–126
  case, 123–125
  conditional, 111–116
  goto, 303–306
  iterative, 116–122
CP/M-80 and CP/M-86 systems, with Turbo Pascal, 377–380
CPU (*see* Central Processing Unit)

D

Dangling pointer, 201
Data types, 54–55
  predefined, 54–55, 87–100
    boolean, 55, 93
    byte, 55, 90
    char, 55, 94–96
    integer, 55, 87–90
    real, 55, 90–93
  simple, 87
  structured, 87
  user-defined, 55
Decimal numbering system, 314–314
Declaration part, 69, 76–79

Defined scalar types, 101–110
enumerated, 101–108
avoiding range errors in, 106
ordinal values of, 103–104
range-checking in, 106–107
type definition part, 102–103
undefined values in, 107–108
input and output, 110
memory usage of, 110
standard functions for, 104–105
subranges, 101, 108–110
Deque list type, 249
Dereferencing pointers, 194
Device I/O, 223–225
Digital data, 13–14
Directed graph, 259
Directory management procedures (MS-DOS, PC-DOS), 222–223
Directory path, 32–33, 215–216
Disk drive, 12–13
Disks, Turbo Tutor and Turbo Pascal,
backup copies, how to make, 30–31, 36
files on, 5-6, 35–36
system disk, how to make, 31–32
Dispose procedure, 200–201
Div operator, 88–89
Double-ended queue, 249
Doubly linked lists, 242
Dynamic Allocation, 193–194
dereferencing pointers, 194
heap, 199
linked links, 195–199
MaxAvail function, 199–200
New procedure, 193–194
Nil pointer, 194–195
Dynamic variables
deallocation of, 200–203
Dispose procedure, 200–201
Mark and Release procedures, 202–203

E

Editor, Turbo Pascal, 42–44, 49–50
block commands, 49
cursor movement commands, 49
customized commands, 48
insert and delete commands, 49
miscellaneous commands, 50
status line, 42–43
Endless loops, 122, 218
Eof function, 215
Eoln function, 215
Erase procedure, 222
Error messages, 40
Errors, I/O, 227, 230
.EX files, 4, 5
Examples, modifying online tutorial, 5
Execute option, on CP/M systems, 380
Executing files, 289–291
Exercise solutions, 351–353
Exit procedure, 144–146
Exponential notation, 91–93
Expressions, 63–66
Extended character set, 17
Extensions of file names, 213
External search, 273–274
External subprograms, 323–327

F

Field widths, 207
FIFO, 247 (*see also* Queues)
File I/O, 233
File names, 41–42, 213
File pointer, 214
File types, 210
Files, 205–230
  appending, 221
  closing, 215
  erasing from disk, 222
  executing, 289–291
  I/O procedures, 205–207
  on Turbo Tutor and Turbo Pascal disks, 5–6, 35–36
  random access, 215–221
  renaming, 222
  standard, 224
  text, 211–215
  truncating, 221
  untyped, 225–227
    declaring, 226
    specifying block size, 227
FILEMGR.PAS, 6
FilePos function, 220
FileSize function, 221
Firmware, 14–15
Floating-point data type (*see* Real data type)
Floating-point numbers, (*see* Two's complement notation)
Flush procedure, 222
Forest, of trees, 257
Formal parameter list, 138
Formatting
  declarations, 77–79
  real numbers, 209
  statements, 79–80
Forward declarations, 142–143
Fragmentation, of memory, 202
Free unions, 182–183
Function declaration, 139
Functions, 105, 138–141
  summary of, 355–361
  (*see also* Procedures and functions)
Fundamentals of Data Structures in Pascal, bibliographic reference to, 251

G

Games
  ANIMALS.PAS, 6,
  TYPIST.PAS, 6, 232
Get procedure, 210
GetDir procedure, 223
Goto statement, 303–306
Graphs, 259
GRAPH.BIN, 36
GRAPH.P, 36

H

Hard disk, using Turbo Tutor with, 37
Hardware, 11–14
  central processing unit (CPU), 12
  input devices, 13
  mass storage (disk drives), 12–13
  memory, 12
  output devices, 13
Hashing, 272–273
Header node, 242–243
Heap, 199–203, 280
Hexadecimal numbering system, 88
  compared to binary numbering system, 317
High-level languages, 23–24
History of programming, 21–27
How to Solve it by Computer, bibliographic reference to, 263

I

Identifier declaration, 77
Identifiers, 55–58
If statement, 66, 111–113
.INC files, 6
Include files, 280–281
Indentation, of programs, 369–370
Initialized variables, 299
Inline statement, 330–334
Input devices, 13
Input/Output, of enumerated scaler types, 110 (*see also* I/O)
Insertion sort, 265–268
Installation, 32–35
  on CP/M systems, 378
Integer data type, 55, 87–90
Integers
  and arithmetic overflow, 89
  unsigned, 87
Integrated circuits, 14
Interpreters, 16, 25–26
Interrupt handling, 332–334
I/O errors, 227–230
I/O file, 233
I/O procedures, 205–210
  Get and Put, 210
  Read, 206
  Readln, 206
  Write parameters, 207–209
Iteration, 116–122
  endless loops, 122
  for statement, 119–122
  repeat...until statement, 118
  while statement, 117–118

K

Keyboard, 13, 224–225
KeyPressed function, 225

L

Labels, 76
Large programs, writing, 275–292
Leaf, of tree, 253
.LIB files, 6
Libraries, 282
  assembly language (PC-DOS, MS-DOS), 328–330
LIFO list, 245 (*see also* Stacks)
Lightning (*see* Turbo Lightning)
Linked lists, 195–199, 241–245
  circular, 243
  deletion of, 245
  doubly linked, 242
  header node, used in, 242–243
  nodes, used in, 241
  singly linked, 241
  starting, 242–245
Lists, 250–251
LISTT.PAS, 6
Local variables, 135
Logical devices, 223–224
LongFilePos function, 220
LongFileSize function, 221
Loops, 116
  endless, 122, 218

M

Machine language, 14
Main menu, 4, 39–42
  commands, 41, 42
MANUAL.PAS, 6
Mark procedure, 202–203
MaxAvail function, 199–200
MaxInt, 88
Memory, 12
  exceeding, 275–278
  fragmentation of, 202
  heap, 199, 280
  RAM, 12
  ROM, 12

Memory management, 200–203
Menu (*see* Main menu)
Microcomputers, 24
Mixed arrays, 262
MkDir procedure, 222–223
Mod operator, 88–89
Modular programming, 281–282
Multiplication operators, 62–63

N

Negation operator, 62–63
Negative numbers, 317–318
Nested comments, 82
New procedure, 193–194
Nil pointer, 194–195
Nodes, 241–245
Null statement, 81
Numbering systems, computer, 313–319
  binary, 13–14, 317
  decimal, 313–314
  hexadecimal, 88, 317
  how byte values are stored in memory, 318–319
  negative numbers, 317–318
  place value, 313–316
  two's complement notation, 317–318

O

Online tutorial, 4
Operating systems, 15–16
Operations, order in expressions, 64–66
Operators, 62–63
Output devices, 13
Overlays, 282–287
  in menu program, 285
  location of overlay files, 286
  procedures in, 284
  restrictions, 286
  vs. chain/execute, 291

P

Parameter list, formal, 138
Parameters, 105, 135–138, 371
  untyped, 309–311
  value, 137–138
  var, 301
.PAS files, 36
Pascal, standard, 24–25
  program structure, 84–85
  reserved words, 59
  (*see also* Turbo Pascal)
Path, in directory, 32–33, 215–216
Peripherals, communicating with, 223–225
Place value, in numbering systems, 313–316
Pointers, 191–193
  dangling, 201
  dereferencing, 194
  nil pointer, 194–195
  pointer type, 192
Portability, of languages, 23
Predefined data types, 54–55, 87–100
  boolean, 55, 93
  byte, 55, 90
  char, 55, 94–96
  integer, 55, 87–90
  real, 55, 90–93
Printing characters, 16–17
Procedure body, 128, 233
Procedure call, 80
Procedure declaration, 128
Procedures and functions, 127–146, 355–361
  exit procedure, 144–146
  forward declarations, 142–143
  local variables, 135
  parameters, 135–138
    formal parameter list, 138
    value, 137–138
  scope, 130–135
    and recursion, 143–144

standard, summary of, 355–361
subprograms, 127–130
recursive, 141–142
Program heading, 69, 75–76
Program structure, 75–85, 372–373
Programming style, 369–375
Proper sets, 186
Put procedure, 210

Q

Queues, 247–249
double-ended (deque), 249
Quicksort, 268–270
Quitting Turbo Pascal, 47–48

R

RAM, 12
Random access files, 215–221
appending, 221
creating, 216–219
FilePos and LongFilePos functions, 220
FileSize and LongFileSize functions, 221
properties of, 219
Seek procedure, 220
Truncate procedure, 221
Random Access Memory (RAM), 12
Range-checking, 106–107
Read procedure, 206
with text files, 214
Readln procedure, 206
with text files, 214
Readln statement, 70–71
READ.ME, 5, 36
README.COM, 5
Read Only Memory (ROM), 12
Real data type, 55
Real numbers, formatting, 209
Record constant, 296
Records, 173–183
fields in, 174
free unions, 182–183
variant, 179–182
with statement, 176–179
Recursion, and scope, 143–144
Recursive subprograms, 141–142
Release procedure, 202–203
Rename procedure, 222
Repetitive tasks (*see* iteration)
Reserved words, 59
Reset procedure, 214, 227
Rewrite procedure 214, 227
RmDir procedure, 222–223
ROM, 12
Root, of tree, 253

S

Saving compiled programs, 46–47
Saving source programs, 45–46
Scalar types, 101, 104–105
converting from one type to another, 218
(*see also* Defined scalar types)
Scope, 130–135
and recursion 143–144
Searching, 270–274
binary search, 271–272
external search, 273–274
hashing, 272–273
sequential search, 270–271
Seek procedure, 220
SeekEof function, 215
SeekEoln function, 215
Sentinel, 266
Sequential search, 270–271
Set constant, 296–297
Set constructor, 186
Sets, 185–190
building, 186
defining a set type, 186–188
operations, 188–189
proper, 186

Shellsort, 267–268
Shifting operators (shl, shr), 320–321
Siblings, of tree, 253
SideKick, with Turbo Pascal, 363–365
Sign bit, 318
Simple data types, 87
Single drive system, using Turbo Pascal with, 37
Singly linked list, 242
Software, 14–16
  application, 16
  firmware, 14–15
  operating systems, 15–16
Solutions to Turbo Tutor exercises, 351–353
Sorting, 265–270
  insertion sort, 265–268
  quicksort, 268–270
  shellsort, 267–268
Source code examples, 6
Source program, how to save, 45–46
Sparse arrays, 260–263
  mixed, 262
  when to use, 263
Stacks, 245–247
Standard files, 224
Standard procedures and functions, summary of, 355–361
Statement part, 69–72, 79–81
Statement types, 80–81
Statements, 66
  assignment, 66
  compound, 80–81
  if, 66
  null, 81
  Readln, 70–71
  Writeln, 70
Static variables, 298–299
String assignments, 159
String comparisons, 166–167
String constants, 97
String procedures and functions, 159–164
  chr, 164
  concat, 160
  copy, 161
  delete, 162–164
  insert, 162–164
  length, 159
  pos, 161
  upcase, 164
String types, 157–158
String variables, declaring, 98
Strings, 96–99, 157–170
  as parameters, 169–170
  control characters in, 97–98
  numeric conversions of, 167–169
  representing in memory, 164–166
Structure, of Turbo Pascal programs, 75–85
Structured constant, 294–295
Structured types, 87, 96
Subprograms, 127–130
  recursive, 141–142
Subranges, 90, 101, 108–110
Subroutine libraries, 282
Subtree terminal mode, 253
Subtrees, 253
  deleting, 257–258
SuperKey, using with Turbo Pascal, 365–368
Syntax diagrams, 56–58, 337–349
  complete set, 337–349
  how to read, 57–58
System disk, how to make, 31–32

T

Tag field, omitting, 182–183
Terminal node, of tree, 253
Text files, 210–215
  Assign procedure, 213–214
  Close procedure, 215
  Eof function, 215
  Eoln function, 215
  file pointer, 214
  Read and Readln, 214
  Reset and Rewrite procedures, 214
  SeekEof function, 215
  SeekEoln function, 215
TINST (installation) files, 32–35
  on IBM or compatible system, 35
  on non-IBM system, 33–35
Trees, 253–259
  binary, 254–258
  non-binary, 258–259
Truncate procedure, 221
Turbo Lightning, used with Turbo Pascal, 368
TURBO.MSG, 35
TURBO.OVR, 35–36
Turbo Pascal
  command names, 41
  compared with standard Pascal, 26–27
  editor, 42–44, 49–50
  error messages, 40
  installation of, 32–35
  main menu, 4, 39–42
  on hard disk, 37
  on single drive system, 37
  program structure, 75–85
  quitting, 47–48
  reserved words, 59
  with CP/M-80 and CP/M-86 systems, 377–380
  with SideKick, 355–361
  with SuperKey, 365–368
  with Turbo Lightning, 368
Turbo Tutor
  before using, 29–30
  files on disk, 5–6, 35–36
  (*see also* Turbo Pascal)
Turbo Typist, 231–238
Tutorial, on-line, 4–5
TUTOR.COM, 4
TUTOR.PAS, 4–5
Two's complement notation, 317–318
Type conversion, 218
Type definition part, 102–103
Typed constants, 293–302
  array constant, 295–296
  as initialized variables, 299
  as static variables, 298–299
  defining, 294–295
  execution from memory, 302
  manipulating components of, 299–300
  passing constants as var parameters, 301
  properties of, 297–298
    mutability, 297
    lifetime, 297
    scope, 297–298
  record constant, 296
  saving constant space, 300–301
  set constant, 296–297
  stored in memory, 301–302
  structured constant,
TYPING.PAS, 6, 232

U

Unary minus operator, 62–63
Undirected graph, 259
Unsigned integers, 87
Untyped files, 225–227
  declaring, 226
  specifying block size, 227
Untyped parameters, 309–311
User-defined data types, 55
User-defined scalars,

V

Value parameters, 137–138
Var parameters, 301
Variable declarations, 61–62, 69
 Variables, 61–62
  absolute, 307–309
  dynamic, 200–203
  initialized, 299
  static, 298–299
Variant records, 179–182

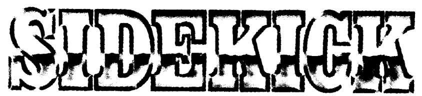
W

Weighted graph, 259
While statement, 117–118
With statement, 176–179
Write parameters, 167–169, 207–209
Write procedure, 206
Writeln procedure, 206–207
Write-protecting disks, 29

Borland Software

4585 Scotts Valley Drive Scotts Valley, CA 95066

Available at better dealers nationwide.
To order by credit card, call (800) 255-8008; CA (800) 742-1133.



Whether you're running WordStar,® Lotus,® dBASE,® or any other program, SideKick puts all these desktop accessories at your fingertips—Instantly

A full-screen WordStar-like Editor to jot down notes and edit file up to 25 pages long.

A Phone Directory for names, addresses, and telephone numbers. Finding a name or a number is a snap.

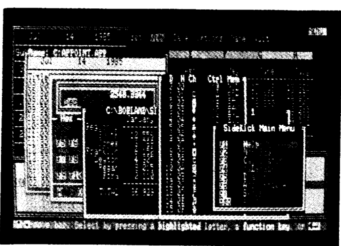
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar from 1901 through 2099.

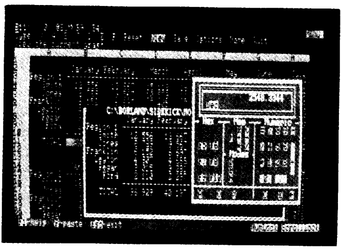
Appointment Calendar to remind you of important meetings and appointments.

A full-featured Calculator ideal for business use. It also performs decimal to hexidecimal to binary conversions.

An ASCII Table for easy reference.



All the SideKick windows stacked up over Lotus 1-2-3.® From bottom to top: SideKick's "Menu Window," ASCII Table, Notepad, Calculator, Appointment Calendar, Monthly Calendar, and Phone Dialer.



Here's SideKick running over Lotus 1-2-3. In the SideKick Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.

The Critics' Choice

"In a simple, beautiful implementation of WordStar's block copy commands, SideKick can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."
—Charles Petzold, PC MAGAZINE

"SideKick deserves a place in every PC"
—Gary Ray, PC WEEK

"SideKick is by far the best we've seen. It is also the least expensive."
—Ron Mansfield, ENTREPRENEUR

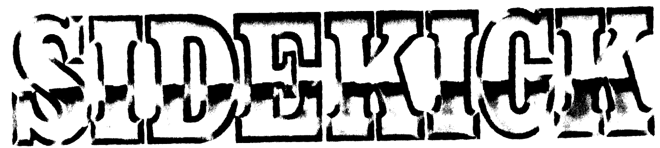
"If you use a PC, get SideKick. You'll soon become dependent on it."
—Jerry Pournelle, BYTE

Suggested Retail Price: $54.95 (copy protected); $84.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr., and 100% compatible microcomputers. The IBM PCjr. will only accept the SideKick not-copy protected versions. 128K RAM, one disk drive, and PC-DOS 2.0 or greater. A Hayes-compatible modem, IBM PCjr. internal modem, or AT&T Modem 4000 is required for the autodialer function.

SideKick is a registered trademark of Borland International, Inc. dBASE is a registered trademark of Ashton-Tate. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corp. AT&T is a registered trademark of American Telephone & Telegraph Company. Lotus and 1-2-3 are registered trademarks of Lotus Development Corp. WordStar is a registered trademark of MicroPro International Corp. Hayes is a trademark of Hayes Microcomputer Products, Inc.

BOR 0060A



SideKick, The Macintosh Office Manager,™ brings information management, desktop organization, and telecommunications to your Macintosh.™ Instantly, while running any other program

A full-screen editor/mini-word processor lets you jot down notes and create or edit files. Your files can also be used by your favorite word processing program, like MacWrite™ or Microsoft® Word.™

A complete telecommunications program sends or receives information from any on-line network or electronic bulletin board while using any of your favorite application programs.

A full-featured financial and scientific calculator sends a paper-tape output to your screen or printer and comes complete with function keys for financial modeling purposes.

A print spooler prints *any* text file while you run other programs.

A versatile calendar lets you view your appointments for a day, a week, or an entire month. You can easily print out your schedule for quick reference.

A convenient "Things-to-Do" file reminds you of important tasks.

A convenient alarm system alerts you to daily engagements.

A phone log keeps a complete record of all your telephone activities. It even computes the cost of every call. Area code look-up provides instant access to the state, region and time zone for all area codes.

An expense account file records your business and travel expenses.

A credit card file keeps track of your credit card balances and credit limits.

A report generator prints out your mailing list labels, phone directory, and weekly calendar in convenient sizes.

A convenient analog clock with a sweeping second-hand can be displayed anywhere on your screen.

On-line help is available for all of the powerful SideKick features.

Best of all, everything runs concurrently

SideKick, the software Macintosh owners have been waiting for

Suggested Retail Price: $99.95 (Includes PhoneLink—not copy protected)

Minimum system configuration: 128K RAM and one disk drive. Two disk drives are recommended if you wish to use other application programs.

SideKick is a registered trademark and PhoneLink is a trademark of Borland International, Inc. Macintosh is a trademark of McIntosh Laboratory, Inc. licensed to Apple Computer, Inc. MacWrite is trademark of Apple Computer, Inc. IBM is a trademark of International Business Machines Corp. Microsoft is a registered trademark and Word is a trademark of Microsoft Corp.

BOR 0069A

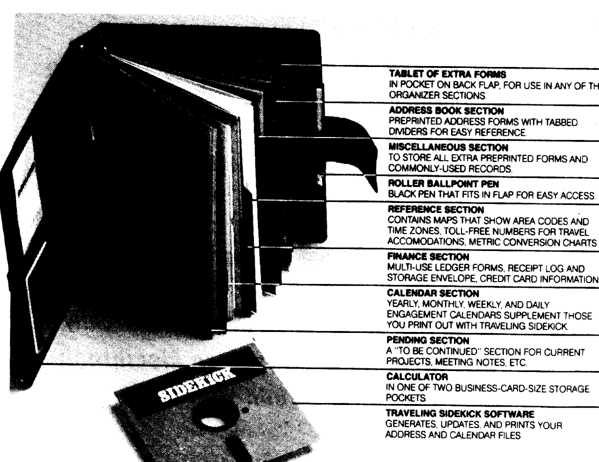
The Organizer For The Computer Age!

Traveling SideKick is *BinderWare*,™ both a binder you take with you when you travel and a software program—which includes a Report Generator—that *generates* and *prints out* all the information you'll need to take with you.

Information like your phone list, your client list, your address book, your calendar, and your appointments. The appointment or calendar files you're already using in your SideKick® are automatically used by your Traveling SideKick. You don't waste time and effort re-entering information that's already there.

One keystroke generates and prints out a form like your address book. No need to change printer paper: you simply punch three holes, fold and clip the form into your Traveling SideKick binder, and you're on your way. Because SideKick is CAD (Computer-Age Designed), you don't fool around with low-tech tools like scissors, tape, or staples. And because Traveling SideKick is electronic, it works this year, next year, and all the "next years" after that. Old-fashioned daytime organizers are history in 365 days.

What's inside Traveling SideKick



What the software program and its Report Generator do for you before you go—and when you get back

Before you go:

- Prints out your calendar, appointments, addresses, phone directory, and whatever other information you need from your data files

When you return:

- Lets you quickly and easily enter all the new names you obtained while you were away into your SideKick data files

It can also:

- Sort your address book by contact, zip code or company name
- Print mailing labels
- Print information selectively
- Search files for existing addresses or calendar engagements

**Suggested Retail Price: $69.95*

Minimum system configuration: IBM PC, XT, AT, Portable, PCjr, 3270 or true compatibles. PC-DOS (MS-DOS) 2.0 or later. 256K and SideKick software.

***Special introductory offer**

SideKick is a registered trademark and Traveling SideKick and BinderWare are trademarks of Borland International, Inc. IBM, AT, XT, and PCjr are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp.

BOR 0083

INCREASE YOUR PRODUCTIVITY BY 50% OR YOUR MONEY BACK

SuperKey turns 1,000 keystrokes into 1!

Yes, SuperKey can *record* lengthy keystroke sequences and play them back at the touch of a single key. Instantly. Like magic.

Say, for example, you want to add a column of figures in 1-2-3.® Without SuperKey, you'd have to type 7 keystrokes just to get started: ["shift-@-s-u-m-shift-("]. With SuperKey, you can turn those 7 keystrokes into 1.

SuperKey keeps your confidential files . . . CONFIDENTIAL!

Time after time you've experienced it: anyone can walk can walk up to your PC and read your confidential files (tax returns, business plans, customer lists, personal letters, etc.).

With SuperKey you can encrypt any file, even while running another program. As long as you keep the password secret, only YOU can decode your file. SuperKey implements the U.S. government Data Encryption Standard (DES).

SuperKey helps protect your capital investment

SuperKey, at your convenience, will make your screen go blank after a predetermined time of screen/keyboard inactivity. You've paid hard-earned money for your PC. SuperKey will protect your monitor's precious phosphor and your investment.

SuperKey protects your work from intruders while you take a break

Now you can lock your keyboard at any time. Prevent anyone from changing hours of work. Type in your secret password and everything comes back to life—just as you left it.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr., and 100% compatibles, 128K RAM, one disk drive and PC-DOS (MS-DOS) 2.0 or greater.

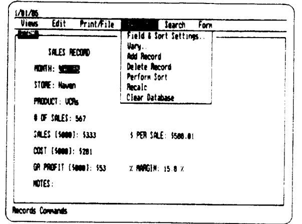
SuperKey is a registered trademark of Borland International, Inc. IBM, XT, AT, and PCjr. are registered trademarks of International Business Machines Corp. 1-2-3 is a registered trademark of Lotus Development Corp.

BOR 0062A

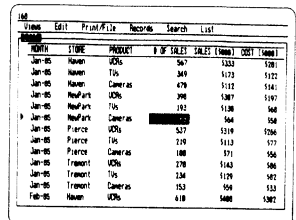
THE ANALYST

Reflex is the most amazing and easy to use database management system. And if you already use Lotus 1-2-3,® dBASE,® or PFS: File,® you need Reflex—because it's a totally new way to look at your data. It shows you patterns and interrelationships you didn't know were there, because they were hidden in data and numbers. It's also the greatest report generator for 1-2-3.

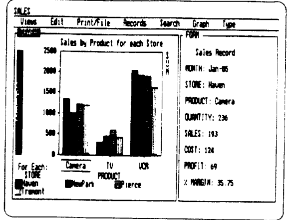
REFLEX OPENS MULTIPLE WINDOWS WITH NEW VIEWS AND GRAPHIC INSIGHTS INTO YOUR DATA.



The FORM VIEW lets you build and examine your database.

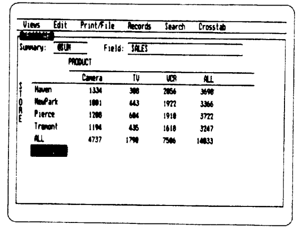


The LIST VIEW lets you put data in tabular list form just like a spreadsheet.



The GRAPH VIEW gives you instant interactive graphic representations.

The CROSSTAB VIEW gives you amazing "cross-referenced" pictures of the links and relationships hidden in your data.



The REPORT VIE\ allows you to impo and export to and fror Reflex, 1-2-3, dBASE PFS: File, and othe applications, and print out information in th formats you wan

So Reflex shows you—Instant answers. Instant pictures. Instant analysis. Instant understanding.

THE CRITIC'S CHOICE:

"The next generation of software has officially arrived."

Peter Norton, PC WEEK

"Reflex is one of the most powerful database programs on the market. Its multiple views, interactive windows and graphics, great report writer, pull-down menus, and cross tabulation make this one of the best programs we have seen in a long time . . .

The program is easy to use and not intimidating to the novice . . . Reflex not only handles the usual database functions such as sorting and searching, but also "what-if" and statistical analysis . . . it can create interactive graphics with the graphics module. The separate report module is one of the best we've ever seen."

Marc Stern, INFOWORLD

Suggested Retail Price $149.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, and true compatibles. 384K RAM minimum. IBM Color Graphics Adapter, Hercules Monochrome Graphics Card, or equivalent. PC-DOS (MS-DOS) 2.0 or greater. Hard disk and mouse optional. Lotus 1-2-3, dBASE, or PFS: File optional.

Reflex is a trademark of Borland/Analytica Inc. Lotus 1-2-3 is a registered trademark of Lotus Development Corporation. dBASE is a registered trademark of Ashton-Tate. PFS: File is a registered trademark of Software Publishing Corporation. IBM, XT, AT, and IBM Color Graphics Adapter are registered trademarks of International Business Machines Corporation. Hercules Graphics Card is a trademark of Hercules Computer Technology.

BOR 0066A

TURBO Lightning™

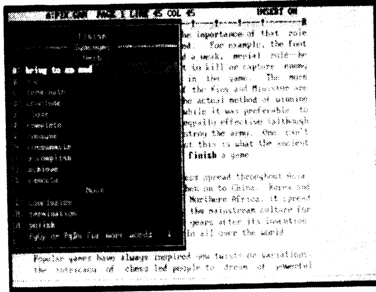
Turbo Lightning teams up with the Random House® Concise Dictionary to check your spelling as you type!

Turbo Lightning, using the 83,000-word Random House Dictionary, checks your spelling *as you type*. If you misspell a word, it alerts you with a beep. At the touch of a key, Turbo Lightning opens a window on top of your application program and suggests the correct spelling. Just press ENTER and the misspelled word is instantly replaced with the correct word. It's that easy!

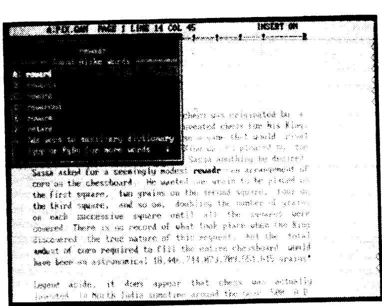
Turbo Lightning works hand-in-hand with the Random House Thesaurus to give you instant access to synonyms

Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once Turbo Lightning opens the Thesaurus window, you see a list of alternate words, organized by parts of speech. You just select the word you want, press ENTER and your new word will instantly replace the original word. Pure magic!

If you ever write a word, think a word, or say a word, you need Turbo Lightning



The Turbo Lightning Dictionary



The Turbo Lightning Thesaurus

Turbo Lightning's intelligence lets you teach it new words. The more you use Turbo Lightning, the smarter it gets

You can also *teach* your new Turbo Lightning your name, business associates' names, street names, addresses, correct capitalizations, and any specialized words you use frequently. Teach Turbo Lightning once, and it knows forever.

Turbo Lightning is the engine that powers Borland's Turbo Lightning Library™

Turbo Lightning brings electronic power to the Random House Dictionary and Random House Thesaurus. They're at your fingertips—even while you're running other programs. Turbo Lightning will also "drive" soon-to-be-released encyclopedias, extended thesauruses, specialized dictionaries, and many other popular reference works. You get a head start with this first volume in the Turbo Lightning Library.

And because Turbo Lightning is a Borland product, you know you can rely on our quality, our 60-day money-back guarantee, and our eminently fair prices.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: 256K RAM. IBM PC, XT, AT, PCjr, or true compatibles with 2 floppy disk drives and PC-DOS (MS-DOS) 2.0 or greater.

IBM, XT, AT, and PCjr are a registered trademarks of International Business Machines Corp.
Turbo Lightning and Turbo Lightning Library are trademarks of Borland International, Inc.
Random House is registered trademark of Random House Inc.

BOR 0070A

TURBO

PROLOG

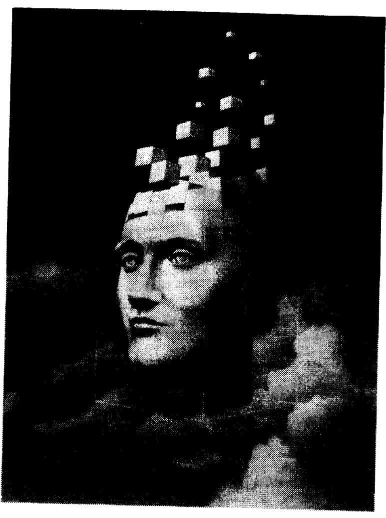
the natural language of artificial intelligence

...EP-BY-STEP
...IAL AND DEMO PROG...
WITH SOURCE CODE INCLUDE...

Turbo Prolog brings fifth-generation supercomputer power to your IBM PC!

Turbo Prolog takes programming into a new, natural, and logical environment.

With ***Turbo Prolog***, because of its natural, logical approach, both people new to programming *and* professional programmers can build powerful applications such as expert systems, customized knowledge bases, natural language interfaces, and smart information management systems.



Turbo Prolog is a *declarative* language which uses deductive reasoning to solve programming problems.

Turbo Prolog's development system includes:

- ☐ A complete Prolog incremental compiler that conforms to the Clocksin and Mellish Edinburgh standard Prolog.
- ☐ A full-screen interactive editor.
- ☐ Support for both graphic and text windows.
- ☐ All the tools that let you build your own expert sytems and **AI** applications with unprecedented ease.

Turbo Prolog provides a fully integrated programming environment like Borland's Turbo Pascal,® the *de facto* worldwide standard.

You get the complete Turbo Prolog programming system.

You get the 200-page manual you're holding, software that includes the lightning-fast ***Turbo Prolog*** incremental compiler and interactive editor, and the free GeoBase™ natural query language database, which includes commented source code on disk, ready to compile. (GeoBase is a complete database designed and developed around U.S. geography. You can modify it or use it "as is.")

MINIMUM SYSTEM REQUIREMENTS: Computer: IBM PC, XT, AT, Portable, 3270, PCjr and true-compatibles. Operating System: PC-DOS (MS-DOS®3) 2.0 or later. Minimum System Memory: 384K.

Turbo Prolog and GeoBase are trademarks and Turbo Pascal is a registered trademark of Borland International, Inc. IBM, AT, and PCjr are registered trademarks and XT is a trademark of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp.

Suggested Retail Price $99.95
(Not Copy Protected)

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066

BOR0016A

TURBO PASCAL®

VERSION 3.0

Free MicroCalc™ Spreadsheet With Commented Source Code!

FEATURES:

One-Step Compile: No hunting & fishing expeditions! Turbo finds the errors, takes you to them, lets you correct them, and instantly recompiles. You're off and running in record time.

Built-in Interactive Editor: WordStar®-like easy editing lets you debug quickly.

Automatic Overlays: Fits big programs into small amounts of memory.

MicroCalc: A sample spreadsheet on your disk with ready-to-compile source code.

IBM® PC Version: Supports Turtle Graphics, color, sound, full tree directories, window routines, input/output redirection, and much more.

THE CRITICS' CHOICE:

"Language deal of the century . . . Turbo Pascal: it introduces a new programming environment and runs like magic."

—Jeff Duntemann, PC Magazine

"Most Pascal compilers barely fit on a disk, but Turbo Pascal packs an editor, compiler, linker, and run-time library into just 39K bytes of random access memory."

—Dave Garland, Popular Computing

"What I think the computer industry is headed for: well-documented, standard, plenty of good features, and a reasonable price."

—Jerry Pournelle, BYTE

LOOK AT TURBO NOW!

- ☑ More than 500,000 users worldwide.
- ☑ Turbo Pascal is the de facto industry standard.
- ☑ Turbo Pascal wins PC MAGAZINE'S award for technical excellence.
- ☑ Turbo Pascal named "Most Significant Product of the Year" by PC WEEK.
- ☑ Turbo Pascal 3.0—the fastest Pascal development environment on the planet, period.

Suggested Retail Price: $69.95; with 8087 or BCD: $109.90; with both 8087 and BCD: $124.95

Options for 16-Bit Systems: 8087 math co-processor support for intensive calculations. Binary Coded Decimals (BCD): eliminates round-off error! A *must* for any serious business application. (No additional hardware required.)

Minimum system configuration: 128K RAM, one disk drive, Z80, 8088/86, 80186, or 80286 microprocessor running either CP/M-80 2.2 or greater, CP/M-86 1.1 or greater, MS-DOS 2.0 or greater, or PC-DOS 2.0 or greater.

Turbo Pascal is a registered trademark and MicroCalc is a trademark of Borland International, Inc. CP/M is a registered trademark of Digital Research Inc. IBM is a registered trademark of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp. Z80 is a registered trademark of Zilog Corp. WordStar is a registered trademark of MicroPro International.

BOR 0061A

TURBO PASCAL

DATABASE TOOLBOX™

Is The Perfect Complement To Turbo Pascal®

It contains a complete library of Pascal procedures that allows you to sort and search your data and build powerful database applications. It's another set of tools from Borland that will give even the beginning programmer the expert's edge.

THE TOOLS YOU NEED!

TURBO ACCESS Using B+ trees: The best way to organize and search your data. Makes it possible to access records is a file using key words instead of numbers. Now available with complete source code on disk, ready to be included in your programs.

TURBO SORT: The fastest way to sort data using the QUICKSORT algorithm—the method preferred by knowledgeable professionals. Includes source code.

GINST (General Installation Program): Gets your programs up and running on other terminals. This feature alone will save hours of work and research. Adds tremendous value to all your programs.

GET STARTED RIGHT AWAY—FREE DATABASE!

Included on every Toolbox diskette is the source code to a working database which demonstrates the power and simplicity of our Turbo Access search system. Modify it to suit your individual needs or just compile it and run. Remember, no royalties!

THE CRITICS' CHOICE!

"The tools include a B+ tree search and a sorting system. I've seen stuff like this, but not as well thought out, sell for hundreds of dollars." ***—Jerry Pournell, BYTE MAGAZINE***

"The Turbo Database Toolbox is solid enough and useful enough to come recommended."
—Jeff Duntemann, PC TECH JOURNAL

Suggested Retail Price: $54.95 (not copy protected)

Minimum system configuration: 128K RAM and one disk drive. 16-bit systems: Turbo Pascal 2.0 or greater for MS-DOS or PC-DOS 2.0 or greater. Turbo Pascal 2.1 or greater for CP/M-86 1.0 or greater. 8-bit systems: Turbo Pascal 2.0 or greater for CP/M-80 2.2 or greater.

Turbo Pascal is a registered trademark and Turbo Database Toolbox is a trademark of Borland International Inc. CP/M is registered trademark of Digital Research, Inc. MS-DOS is a registered trademark of Microsoft Corp.

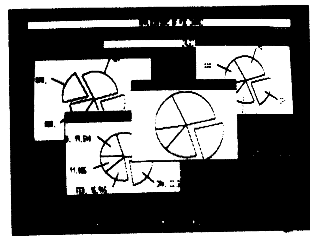
BOR 0063A

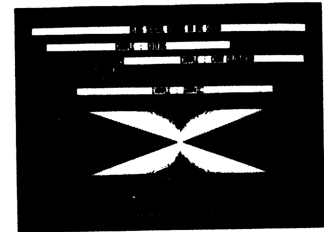
TURBO PASCAL GRAPHIX TOOLBOX™

A Library of Graphics Routines for Use with Turbo Pascal®

High-resolution graphics for your IBM® PC, AT,® XT,® PCjr® or true PC compatibles. Comes complete with graphics window management.

Even if you're new to Turbo Pascal programming, the Turbo Pascal Graphix Toolbox will get you started right away. It's a collection of tools that will get you right into the fascinating world of high-resolution business graphics, including graphics window management. You get immediate, satisfying results. And we keep Royalty out of American business because you don't pay any—even if you distribute your own compiled programs that include all or part of Turbo Pascal Graphix Toolbox procedures.





What you get includes:

- Complete commented source code on disk.
- Tools for drawing simple graphics.
- Tools for drawing complex graphics, including curves with optional smoothing.
- Routines that let you store and restore graphic images to and from disk.
- Tools allowing you to send screen images to IBM-compatible printers.
- Full graphics window management.
- Two different font styles for graphic labeling.
- Choice of line-drawing styles.
- Routines that will let you quickly plot functions and model experimental data.
- And much, much more . . .

"While most people only talk about low-cost personal computer software, Borland has been doing something about it. And Borland provides good technical support as part of the price."
John Markov & Paul Freiberger, syndicated columnists.

If you ever plan to create Turbo Pascal programs that make use of business graphics or scientific graphics, you need the Turbo Pascal Graphix Toolbox.

Suggested Retail Price: $54.95 (not copy protected)

Minimum system configuration: Turbo Pascal 2.0 or later, IBM PC, XT, AT, PCjr, or true compatibles with 192K RAM, two disk drives and an IBM Color Graphics Adapter (CGA), IBM Enhanced Graphics Adapter (EGA), Hercules Graphics Card or compatible.

Turbo Pascal is a registered trademark and Turbo Graphix Toolbox is a trademark of Borland International, Inc. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corporation. Hercules Graphics Card is a trademark of Hercules Computer Tech.

BOR 0068A

TURBO PASCAL

GAMEWORKS™

Secrets And Strategies Of The Masters Are Revealed For The First Time

Explore the world of state of the art computer games with Turbo GameWorks. Using easy-to-understand examples, Turbo GameWorks teaches you techniques to quickly create your own computer games using Turbo Pascal.® Or, for instant excitement, play the three great computer games we've included on disk—compiled and ready to run.

TURBO CHESS

Test your chess-playing skills against your computer challenger. With Turbo GameWorks, you're on your way to becoming a master chess player. Explore the complete Turbo Pascal source code and discover the secrets of Turbo Chess.

"What impressed me the most was the fact that with this program you can become a computer chess analyst. You can add new variations to the program at any time and make the program play stronger and stronger chess. There's no limit to the fun and enjoyment of playing Turbo GameWorks Chess, and most important of all, with this chess program there's no limit to how it can help you improve your game."

—George Koltanowski, Dean of American Chess, former President of the United Chess Federation, and syndicated chess columnist.

TURBO BRIDGE

Now play the world's most popular card game—bridge. Play one-on-one with your computer or against up to three other opponents. With Turbo Pascal source code, you can even program your own bidding or scoring conventions.

"There has never been a bridge program written which plays at the expert level, and the ambitious user will enjoy tackling that challenge, with the format already structured in the program. And for the inexperienced player, the bridge program provides an easy-to-follow format that allows the user to start right out playing. The user can 'play bridge' against real competition without having to gather three other people."

—Kit Woolsey, writer of several articles and books on bridge, and twice champion of the Blue Ribbon Pairs.

TURBO GO-MOKU

Prepare for battle when you challenge your computer to a game of Go-Moku—the exciting strategy game also know as "Pente.®" In this battle of wits, you and the computer take turns placing X's and O's on a grid of 19×19 squares until five pieces are lined up in a row. Vary the game if you like, using the source code available on your disk.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr, and true compatibles with 192K system memory, running PC-DOS (MS-DOS) 2.0 or later. To edit and compile the Turbo Pascal source code, you must be using Turbo Pascal 3.0 for IBM PC and compatibles.

Turbo Pascal is a registered trademark and Turbo GameWorks is a trademark of Borland International, Inc. Pente is a registered trademark of Parker Brothers. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corporation. MS-DOS is a registered trademark of Microsoft Corporation.

BOR 0065A

EDITOR TOOLBOX™

It's All You Need To Build Your Own Text Editor Or Word Processor

Build your own lightning-fast editor and incorporate it into your Turbo Pascal® programs. Turbo Editor Toolbox gives you easy-to-install modules. Now you can integrate a fast and powerful editor into your own programs. You get the source code, the manual, and the know-how.

Create your own word processor. We provide all the editing routines. You plug in the features you want. You could build a WordStar®-like editor with pull-down menus like Microsoft's® Word, and make it work as fast as WordPerfect.®

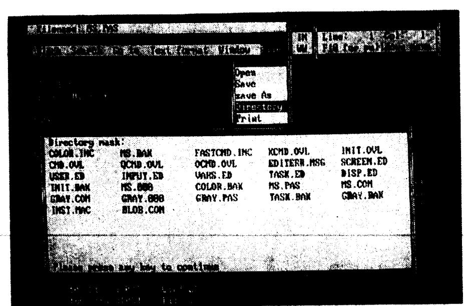
To demonstrate the tremendous power of Turbo Editor Toolbox, we give you the source code for two sample editors:

Simple Editor A complete editor ready to include in your programs. With windows, block commands, and memory-mapped screen routines.

MicroStar™ A full-blown text editor with a complete pull-down menu user interface, plus a lot more. Modify MicroStar's pull-down menu system and include it in your Turbo Pascal programs.

The Turbo Editor Toolbox gives you all the standard features you would expect to find in any word processor:

- Wordwrap
- UNDO last change
- Auto-indent
- Find and Find/Replace with options
- Set left and right margin
- Block mark, move, and copy
- Tab, insert and overstrike modes, centering, etc.



MicroStar's pull-down menus.

And Turbo Editor Toolbox has features that word processors selling for several hundred dollars can't begin to match. Just to name a few:

☑ ***RAM-based editor***. You can edit very large files and yet editing is lightning fast.

☑ ***Memory-mapped screen routines.*** Instant paging, scrolling, and text display.

☑ ***Keyboard installation.*** Change control keys from WordStar-like commands to any that you prefer.

☑ ***Multiple windows.*** See and edit up to eight documents—or up to eight parts of the same document—all at the same time.

☑ ***Multi-Tasking.*** Automatically save your text. Plug in a digital clock, an appointment alarm—see how it's done with MicroStar's "background" printing.

Best of all, **source code is included for everything in the Editor Toolbox.** Use any of the Turbo Editor Toolbox's features in your programs. And pay no royalties.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, 3270, PCjr, or true compatible with a minimum 192K RAM, running PC-DOS (MS-DOS) 2.0 or greater. You must be using Turbo Pascal 3.0 for IBM and compatibles.

Turbo Pascal is a registered trademark and Turbo Editor Toolbox and MicroStar are trademarks of Borland International, Inc. WordStar is a registered trademark of MicroPro International Corp. Word and MS-DOS are registered trademarks of Microsoft Corp. WordPerfect is a trademark of Satellite Software International. IBM, XT, AT, and PCjr. are registered trademarks of International Business Machines Corp.

BOR 0067A

How To Buy Borland Software

To Order By Credit Card, Call (800) 255-8008

In California call (800) 742-1133

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

TURBO TUTOR

Learn Pascal From The Folks Who Created The Turbo Pascal Family.

Borland International proudly presents Turbo Tutor, the perfect complement to your Turbo Pascal compiler. Turbo Tutor is really for everyone — even if you've never programmed before.

And if you're already proficient, Turbo Tutor can sharpen up the fine points. The manual and program disk focus on the whole spectrum of Turbo Pascal programming techniques.

- **For the Novice:** It gives you a concise history of Pascal, tells you how to write a simple program, and defines the basic programming terms you need to know.
- **Programmer's Guide:** The heart of Turbo Pascal. The manual covers the fine points of every aspect of Turbo Pascal programming: program structure, data types, control structures, procedures and functions, scalar types, arrays, strings, pointers, sets, files, and records.
- **Advanced Concepts:** If you're an expert, you'll love the sections detailing such topics as linked lists, trees, and graphs. You'll also find sample program examples for PC-DOS, MS-DOS and CP/M.

A Must. You'll find the source code for all the examples in the book on the accompanying disk ready to compile.

Turbo Tutor may be the only reference work about Pascal and programming you'll ever need!

***Minimum system configuration: TURBO TUTOR is available today for your computer running TURBO PASCAL for PC-DOS, MS-DOS, CP/M-86. Your computer must have at least 128K RAM, one disk drive and PC-DOS 1.0 or greater, MS-DOS 1.0 or greater, CP/M-80 2.2 or greater, or CP/M-86 1.1 or greater.**

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066

Turbo Pascal and Turbo Tutor are registered trademarks of Borland International Inc. CP/M is a registered trademark of Digital Research Inc. MS-DOS is a trademark of Microsoft Corp. PC-DOS is a registered trademark of International Business Machines Corp.

ISBN 0-87524-004-6