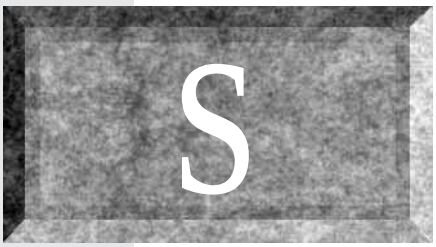
SECRETS OF THE BORLAND® C++ MASTERS



SECRETS OF THE BORLAND® C++ MASTERS

Ed Mitchell

SAMS PUBLISHING

A Division of Prentice Hall Computer Publishing
201 West 103rd Street, Indianapolis, Indiana 46290

COPYRIGHT © 1992 BY SAMS PUBLISHING

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. For information, address Sams Publishing, 201 W. 103rd St., Indianapolis, IN 46290.

International Standard Book Number: 0-672-30137-7

Library of Congress Catalog Card Number: 92-73966

95 94 93 4 3 2

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit number, the number of the book's printing. For example, a printing code of 92-1 shows that the first printing of the book occurred in 1992.

Composed in Goudy and MCPdigital by Prentice Hall Computer Publishing.

Printed in the United States of America

TRADEMARKS

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark. Borland is a registered trademark of Borland International, Inc.

PUBLISHER
Richard K. Swadley

ACQUISITIONS MANAGER
Jordan Gold

MANAGING EDITOR
Neweleen A. Trebnik

ACQUISITIONS EDITOR
Gregory Croy

DEVELOPMENT EDITOR
Stacy Hiquet

PRODUCTION EDITORS
Howard Peirce
Tad Ringo

COPY EDITORS
Gayle Johnson
Melba Hopper
Sandy Doell
Lori Cates

EDITORIAL COORDINATORS
Rebecca S. Freeman
Bill Whitmer

EDITORIAL ASSISTANTS
Rosemarie Graham
Lori Kelley

TECHNICAL EDITOR
Greg Guntle

COVER DESIGNER
Tim Amrhein

COVER ILLUSTRATOR
Ron Troxell

DIRECTOR OF PRODUCTION AND MANUFACTURING
Jeff Valler

PRODUCTION MANAGER
Corinne Walls

IMPRINT DIRECTOR
Matthew Morrill

BOOK DESIGNER
Michele Laseau

PRODUCTION ANALYST
Mary Beth Wakefield

PROOFREADING/ INDEXING COORDINATOR
Joelynn Gifford

GRAPHICS IMAGE SPECIALISTS
Dennis Sheehen
Jerry Ellis

PRODUCTION
Katy Bodenmiller
Julie Brown
Lisa Daugherty
Terri Edwards
Carla Hall-Batton
John Kane
R. Sean Medlock
Roger Morgan
Juli Pavey
Angela Pozdol
Linda Quigley
Michelle Self
Susan Shepard
Greg Simsic
Angie Trzepacz
Alyssa Yesh

INDEXER

Sue VandeWalle

OVERVIEW

	INTRODUCTION	xxvii
1	OPTIMIZING YOUR SYSTEM FOR BEST PERFORMANCE	1
2	POWER FEATURES OF THE IDE AND BORLAND C++	27
3	USING PROGRAMMING UTILITIES	59
4	VERSION CONTROL SYSTEMS	101
5	MANAGING MEMORY	121
6	USING LIBRARY ROUTINES	167
7	WRITING ROBUST AND REUSABLE CLASSES	213
8	VIEWPOINT GRAPHICS IN C++	233
9	GRAPHICS PROGRAMMING IN BORLAND C++	265
10	AUDIO OUTPUT AND SOUND SUPPORT UNDER DOS	337
11	DEBUGGING TECHNIQUES	369
12	PROGRAM OPTIMIZATION AND TURBO PROFILER	421
13	USING BORLAND C++ WITH OTHER PRODUCTS	455
14	CREATING SOFTWARE FOR THE INTERNATIONAL MARKETPLACE	489
15	HOW TO WRITE A TSR	523
16	HIGH-SPEED SERIAL COMMUNICATIONS	597
17	TEMPLATES, PARSING, AND MATH	639
	APPENDIX: SOURCES FOR SOFTWARE TOOLS, UTILITIES, AND LIBRARIES	683
	INDEX	689

CONTENTS

1 OPTIMIZING YOUR SYSTEM FOR BEST PERFORMANCE 1

Hardware Enhancements 2
Setting the Interleave Factor 3
CPU Selection 4
Memory Configuration 4
Using Extended or Expanded Memory 5
Software Enhancements 7
Making the Best Use of System Memory 7
Configuring MS-DOS 8
Configuring DR DOS 10
Other Ways of Reducing DOS and DR DOS Memory Requirements 11
Using FASTOPEN 13
Using Memory Managers 13
Setting Up a Windows Swap File 14
Loading Borland C++ Faster 15
Using Disk Caching Software 16
Using Cached Write Buffers 16
Using RAM Disks 18
Setting Up a RAM Disk 18
Setting Borland C++ to Use a RAM Disk 20
DOS Command-Line Features 20
Using Microsoft DOSKEY Macros 21
Setting the Keystroke Repeat Rate 23
Whole Disk Data Compression 24
Using STACKER 2.0 With Borland C++ 25

2 POWER FEATURES OF THE IDE AND BORLAND C++ 27
Using the Transfer Options in the IDE ... 28
Changing the Transfer Menu ... 29
Adding a New Transfer Item ... 30
Editing an Existing Transfer Item ... 30
Deleting a Transfer Item ... 30
Using Macro Commands with Transfer Programs ... 31
Using Third-Party Text Editors ... 35
Customizing the IDE Editor ... 36
Using the Turbo Editor Macro Compiler ... 36
About the Turbo Editor Macro Language (TEML) ... 38
Customizing the Mouse Buttons ... 40
Configuring the Mouse for Left-Handed Usage ... 41
Achieving the Fastest Compiles ... 41
Some Simple Ideas ... 42
Using */x*, */e*, and */r* IDE Command-Line Switches ... 43
BCC Versus the IDE ... 44
Disabling Optimizations ... 45
Setting the */s* Switch ... 45
Using Precompiled Headers ... 46
Selecting the Precompiled Headers Option ... 46
Using Precompiled Headers Efficiently ... 47
Using *#pragma hdrstop* ... 48
Generating the Fastest and the Smallest Code ... 48
Version 2.0 Users Only: Using Protected-Mode BCX ... 57
Problems with Borland C++ 2.0 and DOS 5.0 ... 57

3 USING PROGRAMMING UTILITIES 59
CPP ... 60
Using CPP ... 61
MAKE ... 65
Sample Use of MAKE ... 65
Explicit Rules ... 69
List All Files ... 70
Command Lines ... 71

Implicit Rules 71
Directives 72
Using BUILTINS.MAK 76
Batching 77
Macros 77
MAKE Command-Line Options 81
PRJ2MAK 83
TOUCH 84
File Searching Utilities 84
Using GREP 85
Third-Party Search and Replace Tools 88
Using whereis 89
Using Turbo Search and Replace 90
OBJXREF 92
PRJCFG 93
PRJCNVT 94
THELP 94
TRANCOPY 95
TRIGRAPH 95
Other Utilities 97
4PRINT Printing Utility 97
LZEXE EXE Compressor 98
HEXEDIT and DUMP 99

4 VERSION CONTROL SYSTEMS 101

Do You Need a Version Control System? 102
Controlling File Ownership 103
Using Version Control for Documentation 105
Tracking Software Revisions 105
Using ATTIC 106
Adding Files to a Library 107
Checking Out Individual Files 108
Checking Out Multiple Files 108
Other Features 109
Introduction to PVCS Version Manager 110
Overview 110

Setting Up PVCS 111
Adding Files to an Archive 113
Checking Files Out 115
Accessing Older Revisions 116
Using Version Labels 116
Maintaining Source Revision Histories 117
Overriding a Locked Revision 119

5 MANAGING MEMORY 121

Choosing a Memory Model 122
The 80x86 CPU Registers 122
Memory Addressing 124
Near and Far Memory References 126
Memory Models 127
Memory Model Restrictions 129
Selecting a Memory Model 130
Special Points About Pointers 131
Huge Pointers 131
Segment Pointers 133
Creating Pointers to Specific Locations 133
Mixed Model Programming and Pointer Modifiers 134
Using the *near* Modifier 137
Creating a .com Program 138
Storing Data 139
Using Dynamically Allocated Memory 141
The Heap 142
malloc() and Related Routines 143
Common Problems Using *malloc()* and *free()* 146
Using *calloc()* 147
Using *realloc()* 148
Using *alloca()* 149
DOS Memory Allocations 149
farmalloc() and Related Routines 150
Using C++ *new/delete* for Simple Data Types 159
Trapping Allocation Errors Using *set_new_handler* 160
Pointer Problems and Memory Trashers 162

6 USING LIBRARY ROUTINES 167

Working with Filenames ... 168
Parsing Filenames ... 168
_fullpath() ... 172
Creating Temporary Files ... 173
tmpfile() and rmtmp() ... 174
tmpnam() ... 174
tempnam() ... 175
mktemp() ... 176
creattemp() ... 177
Using File Attributes ... 178
Creating and Deleting Subdirectories ... 180
mkdir(), *rmdir()*, and *chdir()* ... 180
Drive Selection Functions ... 181
Reading and Searching Directories ... 182
Three Ways to Obtain a Directory Listing ... 182
Accessing a Directory as a File ... 183
Directory Searching ... 185
Using the *ff_fdate* and *ff_ftime* Fields ... 189
Using *_searchenv()* and *searchpath()* ... 190
Accessing Command-Line Parameters ... 193
Using Environment Variables ... 194
Intercepting Ctrl-Break ... 196
Using *TFileDialog* in Turbo Vision ... 197
Using *TFileDialog* in ObjectWindows ... 201
The Container Class Libraries ... 202
Understanding the Container Libraries ... 203
Using the Container Libraries ... 204
The *ForEach* Iterator ... 208
Compiling This Example ... 209
Using the *SortedArray* Container ... 209

7 WRITING ROBUST AND REUSABLE CLASSES 213

Get the Interface Right ... 214
Behavior Defines Classes ... 215
Make It Complete ... 217

Keep It Concise 220
Names are Important 220
Use Standard Idioms 222
Don't Get Carried Away with Operator Overloading 222
Document Carefully 223
The Public and Protected Interfaces 224
Design for Strength 224
Hiding the Implementation Details 225
A Note on Access Restrictions in General 226
Internal Checking 227
Test Thoroughly 229
Just Because It Works, Don't Assume That It's Right 229
Test Early, Test Often 231
Write Meaningful Tests 231
Keep Your Customer Satisfied! 232

8 VIEWPOINT GRAPHICS IN C++ 233

Introduction to ViewPoint 234
World Coordinates 236
How It Works 236
Implementation 237
Class *scaler* 238
Drawing with Scaling 238
Modular Design 239
A Tour of Features 240
Lines 241
Filled Polygons 242
Ellipses 242
Flood Fills 243
Bitmaps 244
Fonts 244
Mice 245
Color 245
Color Palettes 245
True Color 246
Gamma Correction 247
A Simple Business Graphics Class 248

Using the Mouse 252
The Mouse Event Queue 253
The Pie Chart Revisited 254
Displaying Graphics Files 255
PCX Files 258

9 GRAPHICS PROGRAMMING IN BORLAND C++ 265

Introduction to Borland C++ Graphics 266
The Graphics Coordinate System 271
Drawing Circles 272
Displaying Text 273
Selecting Fonts and Character Sizes 274
Setting Text Justification 276
Viewports 278
The Current Pointer 280
Selecting Colors 281
Choosing Colors from the Color Palette 283
Available Colors 283
Using *setrgbpalette()* 284
Selecting Interior Colors and Patterns for Objects 284
Fixing Aspect Ratio Problems 292
Using *drawpoly()* and *fillpoly()* 295
Charting 296
The Pie Chart 299
The Bar Chart 306
The Line Chart 318
Graphics Drivers and Font Files 328
Linking Device Drivers and Font Files 329
Converting .bgi and .chr Files into .obj Files 330
Modifying Your Program to Reference the Linked .bgi and .chr Files 331

10 AUDIO OUTPUT AND SOUND SUPPORT UNDER DOS 337

Bored of the Beep 337
The Physics of Sound 338
Computers and Sound: Principles of Digital Audio 339

Sound Output Without Special Hardware Under DOS 344
The PC Speaker: "What Makes It Beep?" 344
Direct Speaker Manipulation 345
High-Level Speaker Manipulation 349
Beyond the Beep 350
Producing Music 352
Producing Polyphonic Music 360
The Sample Application: DASS.EXE 365
Sound Output from PC Audio Cards Under DOS 366
Sound Card Features 366
Recording and Playing Sound Card PCM Files 367

11 DEBUGGING TECHNIQUES 369

Program Testing Strategies 370
Catching Software Defects Before They Happen 372
Isolating Programming Defects 374
Logic Errors 374
Uninitialized Variables 375
Uninitialized or Erroneous Pointer Values 376
Changes to Global Variables 377
Failure to Free Up Dynamically Allocated Memory 377
Typographical Errors 378
Off-by-1 Errors 379
Clobbering Memory and Out-of-Range Errors 380
Ignoring Scoping Rules 381
Undefined Functions 382
Expression Errors 384
Check All Returned Error Codes 384
Boundary Conditions 384
Debugging Techniques 385
The IDE Debugger 385
Compiling for the IDE Debugger 386
Using the Integrated Debugger 387
Debugger Windows 388
The Watch Window 389
Changing the Value of Variables 390

Using Breakpoints 391
Other Breakpoint Features 394
Debugging the Old-Fashioned Way 394
Using Turbo Debugger 396
Compiling for Turbo Debugger Compatibility 396
Compiling with the Command-Line Compiler 397
Starting Turbo Debugger 397
The Watch Window 399
Inspector Windows 399
Evaluate/Modify 401
Viewing All Variables 401
The View | Hierarchy Command 402
Controlling Program Execution 402
Breakpoints 403
Setting Breakpoint Options 403
Inserting Executable Expressions 405
Changed Memory Global... 406
Expression True Global... 406
Viewing Breakpoints 406
Turbo Debugger and Assembly Language Programs 406
Protected-Mode Debugging on the 80286 407
Virtual Debugging on the 80386 407
Starting the Virtual Debugger 409
Using Turbo Debugger Macros 409
Debugging TSRs 411
Debugging Turbo Vision Applications 412
Turbo Debugger for Windows 413
Using Turbo Debugger for Windows 414
Watching Messages 414
Using Winsight 416
Turbo Debugger for Windows Command-Line Options .. 417
Other Debugging Features 418

12 PROGRAM OPTIMIZATION AND TURBO PROFILER 421

Program Optimization 422
Using the Turbo Profiler 423
Compiling for Turbo Profiler Compatibility 424

Selecting Program Areas to Profile 425
Improving the Program 428
Statistics Provided by Turbo Profiler 432
Turbo Profiler Output Options 433
Active Versus Passive Profiling 434
Optimization Tricks 436
Cleaning Up Loop Statements 437
Test for the Most Likely Outcomes First 438
Set Compiler Options for Most Efficient Execution 439
Replace Function Calls with Lookup Tables 439
Don't Be Afraid of *Goto* 440
Use Better Algorithms 441
Use Pass-by-Address Parameters Instead of Value Parameters 447
Consider Assembly Language 447
Use Fixed Point Arithmetic in Place of *float* Data Types 448
Increase File I/O Buffers 452
Memory Reduction 453
Use Local and Dynamic Variables 453
Recycle Memory 454

13 USING BORLAND C++ WITH OTHER PRODUCTS 455

Exporting Routines to Turbo Pascal 456
Writing Portable C and C++ Code 460
C and C++ Language Issues 461
General Guidelines 461
Data Types 462
Library Functions 464
Special Issues Concerning Microsoft C/C++ and Borland C++ 466
Header Files 467
Using Assembly Language 467
A Very Brief 80x86 Processor Instruction Set Overview 469
The Built-In Assembler 470
Using the Built-In Assembler 471
How Procedures and Functions Are Called 472

Accessing Global Variables 473
Distinguishing between Values and Addresses 474
The Difference between Constants and Variables 476
Local Variables in Functions 476
Accessing Pass-by-Value and Pass-by-Reference Parameters 477
Accessing Structures 478
Statement Labels 479
Jump Instructions 479
Assembler Expressions 480
Turbo Assembler 482

14 CREATING SOFTWARE FOR THE INTERNATIONAL MARKETPLACE 489

Localization Versus Translation 490
More Than Just Changing the Language 490
Localization Is Country Specific 491
System Differences 492
Code Pages and Character Sets 492
Character Sets and Fonts 495
Keyboards and Keyboard Drivers 496
How to Enter Characters 496
How to Change the Keyboard Driver and Code Page 497
Other Hardware Considerations—Printers, Screen Displays, and Graphics Hardware 499
Localized Versions of DOS and Windows 499
Using Resources for Translation 500
Windows Versus DOS 500
Windows: Using Resource Files to Improve Localizability 501
Bitmaps and Icons 502
Speed Keys and Accelerators 502
Help Text 503
DOS 503
Formatting Data 505
Numeric Formats 507
Supporting Multiple Currency Symbols 507
Date Formats 509

Time Formats 510
List separators 510
Input and Output Considerations 511
File I/O 511
File Formats 512
Keyboard Input 513
Mouse Input 513
Output 513
Display 514
Character Support, Sorting, and Searching 514
Borland Libraries and the Windows API 515
Character Identification 515
Searching and String Comparisons 516
Collation Sequences 517
Windows Control Panel 518
Quality Considerations 520

15 HOW TO WRITE A TSR 523

Device Drivers, TSRs, Interrupts, IRQs, and InLine Assembler 525
Device Drivers 525
Load-on-Demand Drivers 526
Ports and IRQs 526
Interrupts 526
Information Exchange 534
Multitasking 535
MS Windows Caveats 535
Useful Vectors 536
Floating Point Emulation 537
General Constraints and TSR Programming Practices 538
Chaining Versus Hooking Interrupts 538
Issuing the TSR Function Request 539
InDos Flag 540
MS-DOS Idle Loop Interrupt—Int 28h 542
When and How to Activate Your TSR 542
Use of CLI 543

Stack Usage 544
Input and Output 545
Dealing with Other TSRs—the TesSeRact Standard 552
Doing Useful Things with Your TSR 557
Safe File I/O 557
Saving Screen Information 563
Windows and Other Gotchas 583
Gotchas of Windows in General 584
Gotchas of Windows Standard Mode 584
Gotchas of Windows 386 Enhanced Mode 584
Detecting the Presence of Windows 585
EOIs 586
Unloading and Cleaning Up 587
Releasing Interrupt Vectors 587
Releasing Memory 588
Memory Hole Issues 589
Environment Segment 592
Detecting other TSRs 593
Advanced Topics 593
Walking the Device Driver Chain 593

16 HIGH-SPEED SERIAL COMMUNICATIONS 597

A Brief History of IBM PCs and UARTs 597
Life, the UART, and Everything 600
Determining the Base I/O Address 600
How the UART Works 601
The Registers 602
Processing Incoming Data 603
Sharing IRQs with Other Devices 608
Reducing Interrupt Latency Problems 609
A High-Speed Serial Interface Class Library 610
For Further Reference 635

17 TEMPLATES, PARSING, AND MATH 639

Class Templates 640
Special Rules Regarding Class Templates 647

Function Templates 648
Parsing 649
Using *sscanf()* 650
Using *strtok()* 651
Constructing a Formal Parser 653
The Sample Language 654
Lexical Analysis 655
Syntax Analysis 666
Borland C++ Math Options 675
bcd Data Type 678
Special Situations Using *bcd* Arithmetic 680
complex Data Type 681

A SOURCES FOR SOFTWARE TOOLS, UTILITIES, AND LIBRARIES 683

Magazine Sources 684
Mail-Order Sources 685
Shareware and Freeware 685
Mail-Order Shareware and Freeware Disk Distributors 687
CD-ROM Disc Distributors 688
Selected Online Services and Libraries 688

INDEX 689

ACKNOWLEDGMENTS

This book could not have been completed without the help of numerous individuals. Each bit of assistance, from merely answering a simple question to providing a Beta release or product in a timely fashion, was critical to the success of this project. Their contribution reflects well on their respective companies' attention to customer service. I wish to thank Nan Borreson, Karen Giles, Greg Meyer, and Bob Arnson of Borland International; Rose Kearsley of Novell Press; Jonn Tracy of Intersolv, Inc.; and Pam Teal of Genus Microprogramming. I wish to thank Greg Guntle for his help with this book. Thanks also go to Acquistions Editor Greg Croy, Development Editor Stacy Hiquet, Editors Howard Peirce, Gayle Johnson, and Lori Cates, and the many others at Sams Publishing who helped to bring this book into the form that you see here.

I am indebted to the contributing authors who lent their expertise and talent to the creation of several critical chapters. Their respective employers—Macro-Media, Inc., Software Publishing Corporation, Interactive Home Systems, Inc., Borland International, Inc, and Traveling Software, Inc.—all deserve thanks for providing an environment where their employees can contribute their professional skills for the good of the industry.

Lastly, my wife Kim is a tremendous source of encouragement when the work load increases and the days seem to get shorter and shorter. I would never have made it this far without the backup support that she provides.

Ed Mitchell
principal author
Secrets of the Borland C++ Masters

About the Authors

(in alphabetical order)

Pete Becker

Pete Becker has worked at Borland International for four years. He started out in Quality Assurance, working on Turbo C 2.0 and Turbo C++ 1.0. He is now in Languages Research and Development, where he works on class libraries (especially Turbo Vision and the Container libraries) and, occasionally, linkers.

John Dlugosz

John Dlugosz has been programming in both C and C++ for many years, and was a reviewer of the C++ 2.0 language specification document from AT&T. He has written several dozen magazine articles for such publications as *Dr Dobb's Journal*, *Embedded Systems Programming*, *Computer Language*, *Byte*, and others. He is currently managing a project at Tobias Associates. John graduated cum laude in Computer Science from the University of Texas at Dallas.

Cynthia Finnell-Fruth

Cynthia Finnell-Fruth is a Software Quality Assurance Engineer specializing in international versions of PC software products for the Windows, DOS, and OS/2 environments. She is currently a member of the Borland International staff responsible for quality assurance for all international versions of the Quattro Pro product. Before joining Borland, Cynthia spent several years at Software Publishing Corporation. Her accomplishments at SPC included major contributions to the Harvard Graphics for Windows, InfoAlliance, Draw Partner, PFS:Professional Write, and PFS:First Choice products. Cynthia studied at the University of Kansas and the University of California, Berkeley.

Gordon Free

Gordon Free earned an MS in Computer Science from the University of Illinois in 1984. He is currently a Principal Engineer at Traveling Software where he heads the Blackbird project, the ultra-high-speed communications library used in LapLink Pro, WinConnect, LapLink XL, and LapLink for PenPoint. Gordon has been programming IBM PCs since 1982 and is often

seen prying CPUs from their sockets to attach logic analyzers. When he's not tearing apart computers, Gordon enjoys photography, woodworking, and spending time with his wife Laurasue and daughter Nikkole.

Robert Fruth

Robert C. Fruth has over ten years of experience in the PC software industry, nearly all of it at Software Publishing Corporation. Currently Manager of the Productivity Services Group, Robert has also served as Project Manager and Software Engineer. His contributions include the Harvard Graphics for Windows, Harvard Graphics, PFS:First Choice, PFS:Professional Plan, IBM Graphing Assistant and PFS:Graph products. Robert studied Economics and Computer Science at the University of California, Berkeley.

Brian D. Herring

Brian Herring is a software engineer at Macro-Media, Inc., Redwood City, Califorinia. Prior to employment at Macromedia, he worked as an engineer on the best-selling Harvard Graphics for Windows and PFS:First Choice. He is now working on future editions of Authorware Professional for Windows, Macromedia's multiplatform authoring tool for interactive learning.

Ed Mitchell

Ed Mitchell is formerly a project manager at Software Publishing Corporation where he was creator and coauthor of the award-winning, best-selling PFS:First Choice integrated software package. At SPC, he was also coauthor of one of the first word processors for the IBM PC, PFS:Write (now known as Professional Write). He now writes computer books full time and is principal author of *Secrets of the Borland C++ Masters*, coauthor of *Using Microsoft C/C++ 7.0* (Que Books), and author of *Borland Pascal Developer's Guide* (Que Books, 1992), plus other books and magazine articles. You may contact Ed Mitchell via electronic mail at CompuServe 73317,2513 or at EdMitch @ao1.com.

Karl Schulmeisters

Karl Schulmeisters is Systems Project Leader at Interactive Home Systems in Redmond, Washington. Prior to working at IHS, Karl worked at Traveling Software and at Microsoft. At Traveling Software, Karl created and participated in the development of the WinConnect file access utility. At Microsoft, Karl was a member of the Lan Manager 1.0 and MS-DOS 3.2 development teams, and was a group leader in the OS/2 1.0 development project. He has written numerous device drivers and TSRs in C and assembly in a variety of operating environments.

INTRODUCTION

Welcome to *Secrets of the Borland C++ Masters*. We've worked hard to bring you tips, tricks, and in-depth technical solutions to complex programming problems. These solutions come from experts in the field of PC software development—people who have written the software that you know and use. As insiders in the industry, we've seen what works and what doesn't work, and when things can go wrong. We've created program examples and text that highlight the correct way to get a job done. Occasionally, we point out a few of our amusing mistakes, which is especially useful in helping you avoid running into the same problems yourself.

Our goal is to increase your productivity through improved programming techniques, system optimization, and the use of libraries to ease your development efforts. This book includes detailed instruction about specialized topics such as TSR construction, high-speed serial communications, incorporation of audio output support into your programs, and design techniques to ease the translation of your software into the international marketplace. The text also covers system configuration strategies for improving compiler performance, graphics handling, debugging, profiling strategies, program optimization, assembly language, and much more.

By using the secrets of experienced PC programming experts, various third-party tools, libraries, and utilities—many of which are described in this book—you can put together complex programs and products more rapidly than if you had to write your own code from scratch.

IS THIS BOOK FOR YOU?

We assume that you already know how to program in C or C++ at an experienced novice or intermediate level or better. You do not need to be an expert, but you should be comfortable reading through C source listings. Program examples, depending on the specific point being illustrated, are

presented in either C or C++. You do not need to be a C++ expert to read and derive value from the C++ listings. All the examples have been tested in Borland C++ and should work also in Turbo C++.

This book is intended for both professional and nonprofessional programmers whose skills range from intermediate to advanced C programming levels. If you are a professional programmer, you will gain insight to specialized topics such as TSR construction and use, creating software for the international marketplace, implementing high-speed (115.2 kbaud) serial communications, and other features. On the other hand, if you are a professional (but not necessarily a professional programmer—meteorologists, microbiologists, foresters, economists, research scientists, consultants, teachers, civil engineers, mechanical engineers, and other professionals program PCs in their daily work), *Secrets of the Borland C++ Masters* gives you insight to the tools and techniques used by professional PC programmers.

In summary, if you'd like to learn how to increase your productivity; to debug advanced software; to produce software with greater reliability; to manage large projects; to use advanced features such as TSRs, audio output and sound boards, and 256-color graphics; to port your C code to other compilers; or to create commercial grade software for the international marketplace, you will find this book to be an indispensable reference. If you are new to the C and C++ programming languages, you should first refer to a C language introductory text such as *Using Borland C++ 3*, 2nd Edition, by Lee Atkinson and Mark Atkinson, published by Que Corporation.

WHERE TO START

Chapter 1, "Optimizing Your System for Best Performance," and Chapter 2, "Power Features of the IDE and Borland C++," highlight a number of system and Borland product features that you may not yet be using. Chapter 3, "Using Programming Utilities," describes all the Borland C++ programming utilities (Borland provides about a dozen programming utilities that are independent of the Integrated Development Environment, the command-line compiler, and the major tools such as Turbo Debugger). The Borland C++ development environment is so large that you may not have even realized some of these tools were hidden in the Borland subdirectories.

If you are an advanced programmer, you may want to skim Chapters 1 through 4 and jump in at about Chapter 5, "Managing Memory," or Chapter 6, "Using Library Routines." Chapter 4, "Version Control Systems," describes version control software systems and why you should be using one for individual or team-based software development. In Chapter 5, "Managing Memory," you'll learn about memory models and model choices, the use of dynamic memory versus static memory, solving common problems involving memory allocations and the use of pointers, and other topics such as implementing a dynamically discardable memory management system. Chapter 6, "Using Library Routines," includes a description of many standard library routines that seem to cause confusion, plus an overview of the container libraries, which includes several program examples that illustrate how to put the container classes to use.

Chapter 7, "Writing Robust and Reusable Classes," is written by Pete Becker, the Borland International engineer who is responsible for the container libraries and Turbo Vision. In this chapter, Pete brings you his insights into designing classes and class libraries.

Chapter 8, "ViewPoint Graphics in C++," introduces the ViewPoint C++ Graphics class library. This chapter is written by John Dlugosz, author of the ViewPoint C++ graphics library. Chapter 9, "Graphics Programming in Borland C++," is an introduction to the Borland Graphics Interface.

Chapter 10, "Audio Output and Sound Support Under DOS," shows you how to generate high-tech sound effects using the PC speaker and pulse-code-modulation techniques. You also learn how to create polyphonic sound, voice synthesis, and audio output using sound boards such as Sound Blaster. Chapter 10 is written by Brian Herring, a multimedia software engineer at Macro-Media, Inc.

Chapter 11, "Debugging Techniques," covers the use of the internal IDE debugger and the external (and vastly more powerful) Turbo Debugger. The chapter includes many suggestions to help you prevent program defects and isolate defects when they do occur. Chapter 11 also offers suggestions for debugging event-driven applications (such as Turbo Vision or Windows) and TSRs.

Chapter 12, "Program Optimization and Turbo Profiler," explains how to optimize your programs to achieve faster program execution. Turbo Profiler is used to identify the best program locations to create speed improvements that

have the greatest impact. This chapter describes several tricks, such as instruction cycle counting, to help you speed up your code.

Chapter 13, "Using Borland C++ with Other Products," shows you how to export C functions to be linked with Turbo Pascal programs, and how to use the built-in assembler and Turbo Assembler. Chapter 13 also covers issues related to converting source code between Borland C++ and Microsoft C/C++.

Is your software headed overseas? Translating software into an international market is far more complex than merely translating a few character strings from English to another language. Even the alphabets of the major languages are different than the English alphabet. As a consequence, even a simple `upcase()` function to convert lowercase letters into uppercase letters won't work when your program is translated for an international market. Sorted lists won't sort correctly, and string comparisons will fail. You can learn about these issues in Chapter 14, "Creating Software for the International Marketplace," which is coauthored by Cynthia Finnel-Fruth, a Software Quality Assurance Engineer at Borland International who is working on international versions of Quattro Pro, and Bob Fruth, formerly Project Manager for Harvard Graphics for Windows at Software Publishing Corporation.

Ready for pop-up TSR applications? Learn the gory details of low-level DOS programming in Chapter 15, "How to Write a TSR," which is written by Karl Schulmeisters. Schulmeisters was an engineer on DOS 3.2 and LanManager at Microsoft. He is also formerly of Traveling Software, Inc., where he was the lead engineer on the data communications TSR application called WinConnect.

Officially, the IBM serial port operates at a maximum rate of 19,200 bps. Unofficially, programmers have been running the serial port at speeds up to 115,200 bps using a variety of programming tricks. These are described in Chapter 16, "High-Speed Serial Communications," which is written by Gordon Free, the principal engineer and inventor of the high-speed serial and parallel communications routines used inside Laplink Pro.

Chapter 17, "Templates, Parsing, and Math" covers templates; parsing techniques for processing user input, expressions, or any other type of command language you might invent; BCD (binary-coded decimal) math; and the use of the floating-point math coprocessor. The appendix, "Sources for Software Tools, Utilities, and Libraries," provides sources of shareware and freeware, a list of technical magazines geared to the PC programmer, CD-ROM suppliers, and other information.

CONVENTIONS USED IN THIS BOOK

The text in this book uses certain stylistic conventions to point out features, highlight items, distinguish between program keywords and ordinary text, and so on.

Specialized information is presented in *tables*, *figures*, and *bulleted lists*. Tables usually contain columnar information. Figures are drawings or *screen snapshots* (illustrations taken directly from the screen).

Bulleted lists look like the following:

- Each item in a bulleted list is preceded by the bullet character.
- The items in the list often provide detailed explanations.
- Generally, the order of the items in the bulleted list is unimportant. The information could appear in any order.

Program keywords and all program listings appear in `monospaced` type. *`Monospaced italic`* type identifies variable parameters in function declarations. For instance, in the declaration

```
int abs( int x ) {...}
```

the parameter variable *`x`* is displayed in *`monospaced italic`* type to indicate that it is a placeholder for a value that you will substitute later.

Italicized type is used to introduce new terms and occasionally to emphasize important words or phrases.

Most program listings appear in `monospaced` format, with line numbers shown along the left side of the listing. These line numbers are not part of the program—they are provided so that the text can refer to individual lines in the listing and to help you keep track of your location when you read a program listing. If you type a listing into the computer, do not include these line numbers! All program listings that display line numbers will compile. (Keep in mind, however, that some listings, such as header files, are intended for use in conjunction with other listings and source files.)

Occasionally, short *code fragments* are inserted directly into the text or between paragraphs. Code fragments may not compile if you enter them as is. Code fragments illustrate a concept; in most instances, the code fragments are extracted from working code and would be expected to compile if you placed them within a suitable source program.

ABOUT THE SAMPLE PROGRAMS

Secrets of the Borland C++ Masters is intended to be of value to a wide range of intermediate to advanced C and C++ programmers. We recognize that many programmers are skilled in C but are new to C++ and may be uncomfortable with the features of C++, especially object-oriented programming and the new language overhead that this entails. For this reason, some of the sample programs have been written in C, and some have been written in C++.

A number of the examples are written using what I call *minimal C++*. C++ offers a number of improvements over standard C. These include improved type checking, true pass-by-reference function parameters, the single-line comment, C++ stream I/O, and overloaded functions. You don't have to make the big leap from C into object-oriented C++ to take advantage of these features. You can make a small step merely by using the .cpp extension. When your source files use .cpp instead of .c, the Borland C++ compiler automatically switches into C++ language mode. Once C++ is activated, you can continue to compile most of your existing C code, but you can begin to creep up on the enhanced features of C++ in a fairly painless manner. For this reason, you will see several fairly standard C programs that have the .cpp extension.

Some of the *maximal C++* examples (those using true object-oriented programming) are written using Turbo Vision or ObjectWindows class libraries. Even if you do not own these supplemental libraries, the source code examples are still useful to learn how advanced techniques are implemented. Turbo Vision and ObjectWindows are excellent user interface development tools. If you want to create DOS programs that use overlapping windows, pull-down menus, and dialog boxes, and can be operated with a mouse, you should seriously investigate Turbo Vision. Turbo Vision provides a powerful and easy-to-use class library (it *is* somewhat difficult to learn; once you master it, however, it is a quick way to write software) for the creation of modern user interfaces.

ObjectWindows provides a class interface to the Microsoft Windows API. If you have used the Windows API, you know that you, as a programmer, are required to create a significant amount of obfuscated code to support the original API. For instance, using the Windows API, you must manually create parameter records and perform other error-prone bit twiddling that is best left to the automation of a compiler. ObjectWindows hides the unnecessary

complexity of the Windows API and lets you concentrate on the details of your applications. Whether you are an experienced Windows programmer or you want to learn how to develop Windows applications, you should definitely use ObjectWindows. If you are familiar with the C++ class concept, you will find that learning ObjectWindows is an easy way to learn Windows programming. Using ObjectWindows enables you to create Windows applications much faster and with far fewer errors than if you use the Windows API in standard C.

Both Turbo Vision and ObjectWindows are included in the *Borland C++ & Application Frameworks for Windows and DOS* product. Both components can also be purchased separately from Borland International.

A DISCLAIMER

The sample programs presented in this book are intended for educational purposes. Reasonable effort has been made to ensure the accuracy of these programs; however, they are not intended to be used "as is" in production-quality programs. In particular, they have not undergone the rigorous testing regimen of a professional quality assurance organization. In many instances, in the interest of brevity and to provide clean examples to illustrate specific techniques, internal error checking may be omitted or may be less extensive than is typical of a production program. No warranties of any type regarding the sample programs are implied. When you use these routines in your own programs, be sure to test the routines, and, in particular, to test the routines as they are integrated into your software.

CHAPTER 1

OPTIMIZING YOUR SYSTEM FOR BEST PERFORMANCE

Borland's high-performance products execute best when given a proper hardware and software environment in which to run. Although Borland's products are already among the fastest in the marketplace, you can obtain even better performance by tweaking your system. Suggestions for enhancing your system configuration to achieve peak development productivity are the subject of this chapter.

Performance optimization falls into the following categories:

- Hardware enhancements, such as adding more memory or faster hard drives.
- Software enhancements, such as using and properly configuring disk caching and disk data compression.

Hardware enhancements

Software enhancements

Configuring DR DOS

DOS command-line features

Whole disk data comparison

- Software settings, such as enabling the proper options in the compiler and selecting appropriate features of DOS.

Many of the software options are included within the DOS or DR DOS operating systems and need only to be installed to improve your system performance. Other utilities are available at low cost. Hardware enhancements include additional memory or faster and larger hard disks. In some cases, you can substitute software (such as disk data compression utilities) for hardware and achieve nearly the same results, saving yourself some money in the process. These enhancements are discussed throughout this chapter. Even if you are a systems guru, you may want to give this chapter a quick look. You might find a few new ideas hidden inside this section.

HARDWARE ENHANCEMENTS

If you install all the options of the newest version of the Borland C++ compiler, they will consume up to nearly 50 megabytes of disk space. Although you do not need to install all of these options, there certainly will be times when you wish you had enough space to do so. Therefore, an 80M hard drive is recommended as a minimum configuration, with 100M or larger typically used for development. If you are just purchasing a new computer, buy a 200M to 300M hard disk if you can afford it. You will be surprised at how fast your hard drive will fill up with new software.

Also be sure to buy a caching drive controller. Software development can be particularly disk-intensive. By purchasing a large and fast disk drive and a fast caching controller, you can significantly improve your overall productivity. You can boost performance even more by adding a software-based disk cache (see the section “Using Disk Caching Software”).

If you find yourself rapidly running out of disk space, the problem might also be due to software quirks or features. Borland C++ includes an option that uses precompiled header files to speed up program compilation. This option stores the symbol tables defined by the header files to an internal format file that can be read at high speed during compilation. This reduces the time needed to recompile lengthy header files. For each project that uses precompiled header

files, Borland C++ creates a special .sym file. These files, if left unchecked, can consume enormous amounts of disk space. Delete them when you no longer need them. See "Using Precompiled Headers" in Chapter 2, "Power Features of the IDE and Borland C++," for more information.

In other cases, particularly if you must reboot your system after your program causes it to hang, you may leave portions of the file structure unaccounted for. When this occurs, you should run the DOS CHKDSK utility with the /F switch to clean up lost file clusters. If CHKDSK finds any lost file allocations, you have the option of converting them to files. If you choose to convert them to files, CHKDSK creates files in the root directory with a filename format such as file*nnnn*.chk, where *nnnn* is a sequence number. After running CHKDSK, delete any such files that might have been created if they don't contain anything you need.

Many applications, including Borland's, install many files that are never used. These files include special hardware and printer device drivers, as well as product components that you seldom use. If you really need to free some disk space, you can experimentally remove a number of files. For example, if you have Windows installed, you can delete WRITE.EXE and other application files if you have no need for them.

Within the Borland C++ product, install only the options you will need. If you need only the large memory model library, install only that library. If you do not need ObjectWindows or Turbo Vision, do not install those components. You can always install them later. You also might be able to delete all the examples, documentation (named docs), or source directories.

SETTING THE INTERLEAVE FACTOR

When you low-level format your hard disk (this is different from the high-level format provided by the DOS FORMAT command), you must specify an appropriate *interleave factor*. The interleave factor sets how each sector or data block is written to the disk. On many disks, it is not efficient to write the blocks in sequential order, that is, block 1, block 2, block 3, and so on around the disk. Instead, the blocks may be organized in an offset pattern, for example, block 1, block 40, block 2, block 41, block 3, and so on. The reason for this is that some drives need a moment to process the data between block reads. In that brief moment of processing, the next block whirls into view, but the disk controller

is not ready to begin reading the data. In order to read the next block, the controller must wait for the disk surface to make a complete revolution before it can again access the data. If you stagger the data blocks across the disk's surface so that the sequential blocks are placed at every other block position (or some other ratio), the controller has enough time to do its job before the next logical sequential block revolves into view under the read/write head of the drive. Depending on the drive and the controller, typical interleave ratios vary from 1:1 up to 6:1 or so.

Your drive's interleave is probably set correctly already. Utility programs are available that can automatically test your system configuration and determine the best interleave ratio. Some programs require a low-level reformat to reset the interleave, while others can realign the interleave without destroying the existing data on your disk. Central Point Software's PC Tools (among other products) includes a disk optimization utility to help you correctly adjust the interleave ratio without damaging existing data. Be aware, though, that some of the newer disks cannot be low-level formatted. If that is the case, the interleave ratio has already been set to its optimum setting.

CPU SELECTION

For serious software development, a fast 80386- or 80486-based CPU is recommended, preferably with built-in caching support. The 80386 and 80486 processors are not only fast; they also provide *virtual 8086* support. Using a Borland-supplied device driver and a feature of the Turbo Debugger (see Chapter 11, "Debugging Techniques"), you can relocate the Turbo Debugger in extended or expanded memory, giving your application a full 640K DOS memory space for execution. You can use special debug features of the 80386 that enable the software-based Turbo Debugger to perform functions that were previously available only on hardware-based debugging tools.

MEMORY CONFIGURATION

For best performance, you should have at least 4M of total RAM, with the memory above the DOS area configured as extended memory. By increasing your RAM to 8M, you can increase the size of the disk cache to enhance performance. If you are developing Windows applications, 8M is recommended.

Additional extended (XMS) or expanded (EMS) memory can improve your system performance in several ways. First, you can use the additional memory for *disk caching* software. Disk caching software automatically keeps a portion of your hard disk in fast, random-access memory (RAM). Accessing RAM buffers instead of the disk can substantially improve the performance of all disk-intensive applications and usually results in much faster loading and program start-up. Second, many new applications, including newer versions of the Borland C++ package, can use the extra memory. For instance, Turbo C++ for Windows, Borland C++ for Windows, and Borland C++ 3.1 can now use up to all of the available extended or expanded memory. On my system configuration, the IDE's Compile | Information... dialog box typically displays five to six megabytes of memory available for source code, project files, and compiled programs. Keeping all of these in RAM reduces and often eliminates disk swapping.

Windows and most of its applications can quickly consume extremely large amounts of memory. On limited memory systems, Windows applications are slow and tedious to use. Adding memory reduces the disk swapping that otherwise occurs.

As a rule of thumb, a typical development system should have a minimum of 4M of RAM—8M is becoming the standard configuration. Although you might not need as much as 8M of RAM directly, you can certainly use the additional memory for disk buffering or caching, improving overall system performance for all DOS programs, whether they use EMS or XMS memory directly or not.

USING EXTENDED OR EXPANDED MEMORY

In order for the Borland compiler, IDE, and other tools to use extended memory, the compiler must run using the DOS Protect Mode Interface (DPMI). (If you are using Version 2.0, that version handles extended or expanded memory much differently than Version 3.x. See the section "Version 2.0 Users Only: Using Protected-Mode BCX" in Chapter 2, "Power Features of the IDE and Borland C++.") During installation of the Borland C++ package, the INSTALL program attempts to identify your machine's characteristics. If INSTALL does not recognize your machine, you must run DPMIINST. This utility program, provided in the Borland package,

automatically attempts to configure the Borland software to work with your machine. If the automatic procedure fails, you might want to remove some of your TSRs or device drivers. Some of these applications might interfere with the use of upper memory and DPMIINST's capability to determine your machine's configuration.

Borland C++ attempts to allocate (but not necessarily use) all available extended and expanded memory for its own use. Any expanded memory that Borland C++ finds is reserved as a swapping area. To restrict the amount of memory used by Borland C++, set the `dpmimem` environment variable at the DOS command line or in your autoexec.bat file like this:

```
set dpmimem=maxmem nnnn
```

where *nnnn* is the desired maximum memory, specified in one-kilobyte blocks. For example, to limit the maximum memory to 4M, write:

```
set dpmimem=maxmem 4000
```

When you launch the DOS-based Borland C++ from Windows, do not rely on the `dpmimem` variable to set the memory resource requirements. Instead, edit the values specified in the \borlandc\bin\bc.pif file. Then, use bc.pif to launch Borland C++ from within Windows.

As noted earlier, the IDE allocates but does not necessarily use all of the memory. By providing a large allocation, you have space to launch other protected-mode applications from within the IDE. You can limit the IDE's demand on XMS and EMS either by setting a dialog box option or by using the `/x` or `/e` command-line options when starting the IDE.

By default, all available EMS is allocated for use as a swapping device. To disable all use of EMS, add `/e-` to the command line. To limit the amount of EMS to be used, add `/e=`*nnnn*, where *nnnn* is the number of 16K-sized pages to use for swapping. To disable extended memory, add `/x-`. You can limit the total memory by adding `/x=`*nnnn*, where *nnnn* is the amount in kilobytes of XMS to use.

Instead of using command-line options, you can set these values in the Startup Options dialog box. Choose Options | Environment | Startup... and enter the desired memory limits in the Use Extended Memory and Use EMS Memory fields. If you are already using disk caching software, you might not need to allocate any EMS memory—the high-speed disk cache provides nearly equivalent functionality.

SOFTWARE ENHANCEMENTS

A number of software products provide support for features that significantly enhance your system. These include memory managers that optimize the use of memory and enlarge the DOS applications' memory space, RAM disks, and disk caching software that reduces the time needed to read or write data to the hard drives.

MAKING THE BEST USE OF SYSTEM MEMORY

The compiler and related utilities run best if given the maximum amount of memory. The original DOS operating system reserves the *low memory area* from 0 to 640K for application programming and the *upper memory area* from 640K to 1M for other purposes, such as video memory. On early PCs this was fine, because most PCs had 640K or less of RAM. Everything—your application, DOS, device drivers, and TSRs—had to fit into the lower 640K of RAM. This resulted in a problem known as "RAM cram": by the time you loaded DOS and your network software, you might have only 400K to 450K of RAM left for your applications.

Today's computers now come with 1M or more of memory as standard. Consequently, both DR DOS and DOS have added features to load drivers and utility programs into high memory, moving these programs out of the DOS application memory area. By loading network software, mouse drivers, and other code and data into high memory, you increase the amount of DOS memory available for your applications.

You can configure DOS or DR DOS to make use of this feature. But some of the upper memory area between 640K and 1M that normally is reserved for device control often is unused on your PC. This unused memory can be reclaimed for use by system software. Trying to figure out which high memory is free and which is not is a feat best left to experts. Fortunately, several memory management utility programs automatically search through your system configuration to locate and optimize the allocation of memory. Some of these utilities might find additional unused memory space—perhaps 100K or more of extra memory that ordinarily would be unavailable to your programs.

CONFIGURING MS-DOS

If you want to manage memory using DOS or DR DOS without the benefit of a sophisticated memory manager, you certainly can do so. Each operating system contains a device driver to manage the upper memory (see the section "Configuring DR DOS" to learn more about how DR DOS is configured). For detailed instructions on configuring your system, always refer to the instruction manual that comes with your operating system software.

In MS-DOS, the driver is named himem.sys, and it manages the extended memory on your system. (Note that himem does not manage expanded memory.) To make extended memory available to your applications, you must install himem.sys (or one of the memory managers described in the section "Using Memory Managers"). As a bonus, if you are running an 80386 or an 80486 processor, himem also can make available certain areas within the 640K to 1M range that normally are not accessible for program use.

To install himem.sys, add a statement such as

```
device=c:\dos\himem.sys
```

to your config.sys file. Place this statement *before* any other devices that use extended memory. Remember that changes to config.sys do not take effect until you reboot your system.

If you are using MS-DOS, you can load most of the DOS operating system itself into high memory. After the himem.sys device statement in config.sys, add the following statement:

```
dos=high
```

If you have an 80386- or an 80486-based computer, other device drivers may be placed into high memory with the `devicehigh` configuration command. Setting up your system to use `devicehigh` can become a time-consuming process requiring you to edit and reboot several times. For this reason, third-party products such as QEMM and 386MAX have been created to produce an optimized configuration for your system automatically. These products also do a better job than the limited tools available from DOS. If you want, you can configure your own `devicehigh` statements using the instructions provided here.

To use the `devicehigh` command, you must tell DOS that it needs to maintain link information between upper memory and conventional memory. Therefore, before you use the `devicehigh` command, you should insert the following command into config.sys:

```
dos=umb
```

`umb` is an abbreviation for *upper memory block*. You might also be able to use the emm386.sys device driver in place of the `umb` handler.

For example, you may load the smartdrv.sys disk caching device driver into high memory by using the `devicehigh` command in place of `device`. For example:

```
devicehigh=C:\dos\smartdrv.sys 1024 512
```

USE CAUTION WHEN MODIFYING CONFIG.SYS

The config.sys file is read and processed during the system boot process. If you make changes to config.sys that cannot be successfully processed—or which can hang your system during boot—you will not be able to reboot your system. To protect yourself, *always* keep handy a DOS boot disk that contains a bootable copy of MS-DOS.

Take this advice seriously! Due to a simple typographical error, I once caused my config.sys file to load an incorrect driver that hung my system. I could not boot the system at all. I inserted my DOS boot disk, but it had a damaged sector—I had to go to a computer store to format a new bootable disk. Today, I keep several boot disks in my disk rack just in case. I learned a lesson about the dangers of making even simple modifications to config.sys without having a backup available.

A major problem with the `devicehigh` command is that it does not work for devices that allocate additional memory after start-up. For such devices, you need to install them first using the `device` command, and then run the DOS utility MEM from the command line, like this:

```
mem /C ¦ more
```

This produces a display showing the memory size of each driver. You need to find each device that will be loaded into high memory and identify the amount

of memory that each requires. On the `devicehigh` statement, you must specify the `size=` parameter followed by the size value in hex:

```
devicehigh size=5830 C:\dos\smartdrv.sys 1024 512
```

Here, 5830 is the hexidecimal value of the size required by smartdrv.sys.

You can also place TSR utility programs into high memory by using the `loadhigh` command in your autoexec.bat file. Wherever you launch a TSR, insert `loadhigh` before the TSR program name. For instance, I have a TSR named ASCIITBL that displays a table of ASCII values. To load this TSR into high memory, you would write

```
loadhigh C:\tools\asciitbl.exe
```

When loaded high, some TSRs do not execute or might not fit into an upper memory block. If this occurs, they should be run in conventional memory.

Another alternative for starting a TSR is to use the `install` command in your config.sys file. The `install` command may be used to load the DOS FASTOPEN, KEYB, NLSFUNC, and SHARE TSR utilities. For example, to load FASTOPEN, you should put this statement into your config.sys file:

```
install=c:\dos\fastopen c: = 50
```

Using `install` is roughly equivalent to starting a TSR from the command line. The difference is that when `install` loads a TSR, it does not allocate a program environment prefix for the running program. Any TSR that requires an environment to be set up in advance should not be loaded with `install`. (This includes TSRs that reference environment variables and certain shortcut keys or TSRs that do not trap critical errors.)

CONFIGURING DR DOS

In DR DOS, you should install the special device driver named hidos.sys or emm386.sys or emmxma.sys. (To determine which driver is right for your system, see "Memory Management Overview" in the DR DOS 6.0 *Optimization and Configuration Tips* booklet that is part of the DR DOS 6.0 package.) For example, place the following statement in your config.sys file:

```
device=hidos.sys
```

As soon as you have selected the appropriate memory support driver, you might be able to load DR DOS into the extended memory area using this statement in your config.sys file:

```
hidos=on
```

You can load device drivers into upper memory using the `hidevice` configuration command as illustrated:

```
hidevice=ansi.sys
```

Within config.sys you also can launch TSR applications using the `hiinstall` command. The `hiinstall` command tries to place your TSR into the upper memory area. If there is insufficient upper memory available, the TSR is then loaded into the lower conventional memory area. Use `hiinstall` like this:

```
hiinstall=C:\tools\asciitbl.exe
```

From the DR DOS command line, you can install TSRs into high memory by using the `hiload` command. You can place `hiload` commands into your autoexec.bat file for automatic installation of TSR programs. An example `hiload` command looks like this:

```
hiload C:\tools\asciitbl.exe
```

OTHER WAYS OF REDUCING DOS AND DR DOS MEMORY REQUIREMENTS

Five other configuration options can be used to reduce system memory requirements. When running applications such as the compiler, you need to permit many files to be open simultaneously. For this reason, you probably have a statement in your config.sys file that specifies the maximum number of open files:

```
files=30
```

This example tells DOS to allocate sufficient internal space to manage as many as 30 files simultaneously. You might be able to adjust this value downward slightly, perhaps to as few as 15 to 20 (Borland recommends at least 20). If your compiler or other applications start to misbehave, it is likely that you have adjusted this value too low. Be aware that many applications return a spurious error message if there are insufficient file handles allocated for the system. Some applications just lock up, and others give a seemingly irrelevant error message.

The `buffers` statement is used in config.sys to allocate internal disk buffering space. Having a sufficient number of buffers can improve access speed to your system. Your config.sys file may contain a statement such as

```
buffers=30
```

which allocates 30 buffers, each 512 bytes in size. If you use disk caching software (see the section "Using Disk Caching Software"), the disk cache performs roughly the same function as these internal DOS buffers. Therefore, when a disk cache is active, set `buffers` to a much smaller value, such as 5 or 10. DR DOS users may use the `hibuffers` command, which operates the same as `buffers` but places the buffer allocation into high memory.

If your config.sys file contains an `fcbs` statement, you might be able to delete it. The `fcbs` command allocates space for file control blocks. Programs written during the past several years no longer use file control blocks, so you might be able to delete the `fcbs` statement altogether.

Use the `lastdrive` configuration statement to specify a range of drive letters available to applications on your system. For example:

```
lastdrive=m
```

configures your system for drives lettered A to M. You should set `lastdrive` to the minimum number of drives likely to be used on your system. By default, `lastdrive` is set to the letter following the last drive installed on your system. If you have only a C drive, then the default value of `lastdrive` is D. If you are connected to a network, it is likely that you have `lastdrive` set to a high value, such as Z. This wastes memory space by allocating extra space for drive management. Set `lastdrive` to the lowest value that makes sense for your system and applications.

For MS-DOS users only, the `stacks` statement allocates space for stacks that are used when processing hardware interrupts. The `stacks` statement has the form

```
stacks=8, 256
```

where `8` is the number of stacks to allocate and `256` is the number of bytes to be allocated to each stack. Some computer systems do not need to allocate any stacks by this command. You can experiment to see if setting `stacks = 0, 0` works for you. If it works, this will save a small amount of memory.

USING FASTOPEN

FASTOPEN is a utility program that comes with DOS 5.0 and DR DOS 6.0 to improve the speed of access to frequently used files. FASTOPEN comes in two different versions. Use the device driver fastopen.sys for automatic installation in config.sys or use fastopen.exe for launching from the DOS command line or from within your autoexec.bat file. Each time a file is opened, FASTOPEN stores information about the file's disk location in a RAM-based table. When a previously opened file is opened again during the same session, DOS is able to retrieve the file's location from RAM, eliminating a disk access.

You can launch FASTOPEN from the command line by typing

```
C:>FASTOPEN x: = n
```

where *x:* is one of your local hard drive volumes (do not use this over networks) and *n* is the number of files you want FASTOPEN to track. *n* can range from 10 to 99. If *n* is unspecified, FASTOPEN will track as many as 48 files. Each file that is tracked uses less than 50 bytes of memory per file in FASTOPEN's internal table. If you want to have the internal table stored in expanded memory (and EMS is available), append /x to the command line. You can track files on more than one disk volume by appending additional drive letters to the command line, as in this example:

```
C:>FASTOPEN C:=40 D:=50 E:=50
```

For unknown reasons, some software is incompatible with the use of FASTOPEN. I have seen several instances of cross-linked file sectors that were traced to the use of FASTOPEN, and eliminating the use of FASTOPEN cured the problem. If you experience erratic behavior such as this, stop using FASTOPEN.

USING MEMORY MANAGERS

Third-party software developers have produced alternative memory managers that can substitute for MS-DOS's himem.sys, EMM386, or DR DOS's hidos.sys. Furthermore, these products have their own *automatic* installation procedures that search through your system memory, creating an optimum memory management configuration for your system. DR DOS 6.0 includes its own

MEMMAX utility to help you map and optimize memory usage on your system. However, MEMMAX does not substitute for the features available in QEMM/386 or 386MAX.

The two most popular memory managers probably are 386MAX (or BlueMax for PS/2 systems) and QEMM/386. I can't give you precise installation instructions because it is reasonable to expect that newer versions of these products will appear during the useful life of this book. I can tell you, however, that these products install themselves and search out hidden memory (in the 640K to 1M range); then insert appropriate statements into your config.sys or autoexec.bat file to maximize available memory. I highly recommend using these products.

The default installation of QEMM/386 version 6.0 with the Optimize option works well with Borland C++ 3.0. If you set the memory manager's switches to allocate no EMS, Borland C++ 3.0 will have problems. Borland recommends that if you use a memory manager, allocate at least some EMS, with 750K to 1024K of RAM recommended. Version 6.01 of 386MAX also works well with Borland C++ 3.0 and 3.1, although older versions of 386MAX encounter difficulties with certain system configurations. These problems can be fixed by experimenting with the switch settings for the 386MAX.SYS memory manager.

SETTING UP A WINDOWS SWAP FILE

When you first run Microsoft Windows in Enhanced mode (which requires 80386 or better), Windows creates a temporary swap file on disk. By making the temporary swap file into a permanent file, you reduce the time needed by Windows to open and access the file. This enables Windows both to start and to exit quicker than when using the temporary file.

FOR WINDOWS 3.1 USERS

To set up a permanent swap file from within Windows 3.1, launch the Control Panel application by double-clicking its icon in Program Manager's Main program group. Then, double-click the 386 Enhanced desktop icon. From the resulting dialog box, select the Virtual Memory... button. This displays a dialog box, shown in Figure 1.1, displaying information about your current configuration. To change the configuration, select the Change button. This makes the

dialog box larger to display additional choices for changing the configuration. At the Type combo box, select Permanent. In the New Size edit field, enter the value shown to the right of Recommended Size.

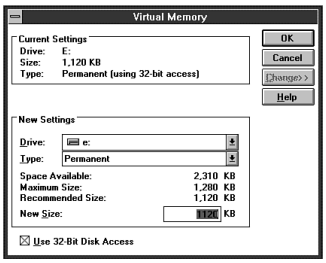


Figure 1.1. The Windows Dialog box used to set a permanent swap file.

FOR WINDOWS 3.0 USERS

If you are using Windows 3.0, you can make the swap file permanent by running swapfile.exe. in Windows real mode. You can access real mode by starting Windows with the `/R` switch, like this:

```
C:>win /R
```

From the Program Manager, select File | Run. Enter the program name `swapfile`, and then follow the prompts to set up your swap file.

LOADING BORLAND C++ FASTER

When you type a command at the keyboard such as

```
C:>BC
```

DOS looks for the desired program file by first checking the current directory and then examining each subdirectory specified in the PATH statement (see the section "Using FASTOPEN"). If you can, move the subdirectory containing the Borland executable files nearer to the beginning of the path list. This reduces the number of directories that must be searched, speeding up the launching of the various Borland tools.

Another tip is to keep your hard drive's file structure from becoming overly fragmented. When files are stored on disk, they are not always stored contiguously. Instead, large files often are split into chunks and written to several spots

on the hard disk. Accessing a file that is located in several areas takes longer than accessing a file that is located in consecutive disk sectors. All file structures become fragmented over time. You can eliminate this fragmentation by using a defragmentation utility such as the COMPRESS utility available in the PC Tools set from Central Point Software.

Alternatively, if you are really desperate, you can back up your files to floppy disk, reformat your hard drive, and then restore the backed-up files. Reformatting produces a clean, unfragmented file structure, but it's a pretty drastic measure that is likely to take more time than an unfragmented disk will save.

USING DISK CACHING SOFTWARE

Both Microsoft's DOS and Digital Research's DR DOS provide disk caching software utilities. By using disk caching software, a portion of the system, EMS, or XMS memory is set aside for buffering data to and from disk. A typical Windows configuration, for instance, sets aside from 256K to 2M as a disk cache. Each time an application reads data from the hard disk, the disk caching software checks the buffers. If the data is already present in the buffers, it can be copied from RAM at high speed, eliminating the need for a comparatively time-consuming disk operation.

On my system, with 8M of RAM, I set aside 2M for use as a disk cache. Depending on your applications, the performance improvement when increasing a disk cache from, for example, 1M to 2M may be negligible.

USING CACHED WRITE BUFFERS

The newer disk caching software can buffer both disk reads and disk writes. In the latter case, this means that the data is temporarily stored in the RAM buffer and is not immediately written to disk. As with disk reads, buffered disk writes can reduce the amount of inefficient disk I/O.

A problem arises when you are caching disk writes, especially when you are working with typically buggy software that is under development. Because the disk writes are not immediately copied to disk, a software crash can hang your system, requiring a system reboot. After you reboot, the data that was written to the cache might never have reached the hard disk—it's lost forever. This can be particularly troublesome if that data includes modified source or project files.

Typically, about 90 percent of disk operations are reads, not writes. Therefore, buffering disk writes typically improves only 10 percent of the disk I/O operations. That small gain in disk performance must be weighed against the real possibility of losing your data. For finished and tested applications, the likelihood of a system crash might be small, but during development, unexpected problems can bring your application to a halt at any time. For such a small gain in performance, combined with the potential for genuine data loss, I do not recommend using disk write caching.

For software that supports disk write caching, you can optionally turn the feature on or off at the time the disk cache is installed. If you are using the Windows 3.1 version of smartdrv.exe, you enable write caching by appending a + symbol to the name of the drive to be cached. You install smartdrv by typing a command resembling the following:

```
C:>SMARTDRV D+ 2048 1024
```

In this example, D is the disk volume to cache, the + symbol enables disk write caching, 2048 is the desired size of the cache in kilobytes, and 1024 is the desired size of the cache when Windows is running. This second value gives Windows more flexibility in memory management by permitting Windows to ask smartdrv to reduce the cache to 1024K. To disable write caching, use the - symbol after the volume letter.

Central Point Software's PC Tools 7.0/7.1 includes the PC-CACHE utility. When you install PC-CACHE, you can enable disk write caching by adding the /WRITE=ON command-line option; to disable write caching, add /WRITE=OFF.

DR DOS 6.0 provides Super PC-Kwik. This disk caching utility is installed and configured automatically, at your option, by the the DR DOS INSTALL and SETUP programs. PC-Kwik is a full-featured disk caching utility. See "Super PC-Kwik Disk Accelerator" in Chapter 13 of the *Novell DR DOS 6.0 User Guide* for complete information on setting and using the features available in PC-Kwik.

386MAX 6.01 includes the QCACHE utility. QCACHE does not buffer disk writes. Many other disk caching utilities are available in addition to those mentioned here. Not all disk cache utilities are alike—and most third-party disk cache routines work better than smartdrv. Indeed, many users encounter compatibility problems when trying to work with smartdrv. If you use smartdrv and discover problems, consider using one of the other disk cache utilities.

USING RAM DISKS

A RAM disk is an area of memory configured to act like a disk drive. Because the RAM disk is located in random-access memory, access to the RAM disk is extremely fast. When your computer is turned off, however, the content of the RAM disk is destroyed. RAM disks normally are used for swap files and temporary data that does not need to be stored permanently.

With today's high-speed CPUs, fast hard drives, and disk caching software, the use of RAM disks is not nearly as important as it was in the past. You might achieve better overall system performance by allocating the memory used by a RAM disk to a disk cache utility instead. Disk caching improves access to all disk files for all applications, whereas the RAM disk is of value only to programs that reference the RAM disk directly. Consequently, I recommend that you first try using a disk cache before installing a RAM disk. If this does not give you the performance you need, consider using the RAM disk.

You can easily install a RAM disk utility that reserves an area of memory for a simulated disk drive. As soon as you have installed the appropriate RAM disk software, you access the disk drive as any other drive by using the drive's designated letter. Because the RAM disk looks and works just like any other disk drive, all your software can work with the RAM disk. You can even use automatic disk compression software such as STACKER (described later in this chapter) to compress the data stored in the RAM disk. This can effectively double the useful size of the RAM disk, if needed.

SETTING UP A RAM DISK

DOS 5.0 (and some earlier versions of DOS) provide a ramdrive.sys device driver to implement memory-based disk drives. You use the `device` or `devicehigh` commands in config.sys to load the ramdrive.sys driver. The device statement

must specify the size of the RAM drive in kilobytes and may optionally add `/e` to place the RAM drive in extended memory or `/a` to place the RAM drive in expanded memory. A typical RAM drive installation looks like this:

```
device = ramdrive.sys 1024 /e
```

In this example, `1024` is the size of the RAM drive in kilobytes (which works out to 1M in this example) and `/e` requests that the RAM drive be placed in extended memory. You must have a high-memory manager such as himem.sys loaded *before* you install ramdrive.sys. As soon as it is installed, the RAMDrive is assigned to the drive letter immediately following the last physical drive on your system. For instance, on my system, I have two hard disk volumes, C: and D:. After I install ramdrive.sys, my RAM disk volume becomes E:.

You can set up a RAM disk in DR DOS using the vdisk.sys device driver. This driver must be loaded in your config.sys file after emm386.sys but before other devices that use extended memory. You can add vdisk.sys to your config.sys file using a statement like this:

```
device=c:\drdos\vdisk.sys 1024
```

This example statement creates a virtual disk or a RAM disk 1024K in size. You can set additional options, including sector size, the maximum number of files, and switches to select the use of extended or expanded memory. Consult the DR DOS user guide for details.

To use a RAM disk with Borland C++, follow the instructions given in the next section. Many other programs can use the RAM disk for their temporary files if they include an option to select their temporary file storage location or if they detect the presence of the DOS environment variables `TEMP` or `TMP`. Many software packages check the DOS environment strings for `TEMP` (older software might look for `TMP`). You can use the DOS `SET` command to set `TEMP` and `TMP` equal to the drive and subdirectory where the temporary files should be stored. To configure `TEMP` so that it references my RAM disk, I type

```
SET TEMP=E:\
```

For your convenience, you might want to place this statement into your autoexec.bat file.

SETTING BORLAND C++ TO USE A RAM DISK

When the Borland C++ compiler runs out of memory, it attempts to swap data to disk. If you have set up a RAM disk, you should specify the name of the swap drive using the Options | Environment | Startup menu selection. This dialog box displays an option called Swap File Drive. At this field, enter the drive letter of the RAM disk. Thereafter, any data that Borland C++ must swap to disk instead is swapped to the RAM disk. Optionally, you can indicate the desired swap disk volume by adding `/rx` to the BC command line. Set *x* to the letter of the drive that corresponds to a RAM disk.

DOS COMMAND-LINE FEATURES

Whenever you use the operating system command line to enter commands, you are directly accessing DOS. DOS has some simple features that can improve your productivity. For instance, the DOSKEY program keeps track of previous DOS commands you have typed. When DOSKEY is installed, you can use the up and down arrow keys to scroll through previous commands you have entered. DOSKEY is especially useful when you are using your own editor and the stand-alone BCC command-line compiler. Because you tend to go through several cycles of editing, compiling, and debugging the same set of files, you can use DOSKEY to quickly repeat the command sequence. Consider the edit-compile-debug sequence of a program file cw.cpp. During the course of development, I repeatedly type the following commands:

```
edit stredit.c
bcc cw.cpp
td cw.exe
```

When I have finished the debug step, I can manually retype `edit cw.cpp`, or I can press the up arrow a few times to scroll back through the list of previously typed commands. Using DOSKEY can speed up your keyboard command entries and reduce the number of command-line keystroke errors by reusing previously executed commands.

To install DOSKEY (it should be in your DOS files directory), run `doskey` from the command line or from within your autoexec.bat file. This establishes a

default command buffer of 512 bytes. You can establish a different buffer size by starting `doskey` with the `/bufsize=` switch, like this:

```
doskey /bufsize=1024
```

DR DOS users have similar functionality available by inserting the `history=on` statement into the config.sys file. You can set the keystroke buffer size by using `history=on,`*`nnnn`*, where *`nnnn`* is the number of bytes to allocate for the history buffer.

As soon as it is installed, use the up and down arrow keys to scroll through the history list. Press PageUp to see the oldest command in the buffers, or PageDown to see the most recent command. You also may edit individual command lines by using the left and right arrow keys to move the cursor left or right (use Ctrl-left arrow to move left one word and Ctrl-right arrow to move right one word). Use the Delete key to delete individual characters, and press Insert to toggle back and forth between insert and overstrike modes. DOSKEY (and DR DOS's history feature) provides many more editing commands than these, but in general I've found that these are more than enough. Anyway, they're the only ones I usually can remember.

USING MICROSOFT DOSKEY MACROS

DOSKEY provides macros, or special text-substitution features, that you can execute on the command line. Using a macro, you can equate a symbol to a sequence of one or more commands. For instance, to define a macro named ed to quickly edit the cw.cpp file, you might type

```
doskey ed=edit cw.cpp
```

To use or run the macro, type `ed` at the command line:

```
C:>ed
```

You may combine multiple commands into a single macro. For example, you might create a macro named `ecd` for edit-compile-debug. Each command in the macro is separated by placing a `$T` or a `$t` symbol between the commands:

```
doskey ecd=edit cw.cpp$Tbcc cw.cpp$Ttd cw.exe
```

A number of special symbols are available to use in your macros. These symbols are shown in Table 1.1.

TABLE 1.1. LIST OF **DOSKEY** MACRO SYMBOLS.

Symbol	*Purpose*
`$G` or `$g`	When placed in a macro, this symbol is equivalent to the `>` redirection operator for sending output to a disk file. The `>` symbol by itself is not recognized inside a macro. Example: `doskey d=dir \source\*.cpp$Goutput.txt`
`$G$G` or `$g$g`	Similar to `$G`, this symbol is equivalent to the double `>>` redirection operator. The `>>` symbol outputs data to the end of an existing file.
`$L` or `$l`	This symbol is equivalent to the `<` input redirection symbol and causes command data to be read from a file instead of the keyboard.
`$B` or `$b`	When placed in a macro, `$B` generates output to be used as input to another command. In this form, `$B` is equivalent to the DOS pipe command character ¦.
`$$`	`$` is a special symbol to DOSKEY. If you want to use a `$` character in a macro, you must place two dollar sign symbols together (`$$`).
`$1` through `$9`	To provide variable parameter values to your macros, use these symbols to substitute for each parameter. `$1` corresponds to the first parameter, `$2` to the second, and so on. To understand how these work, consider the creation of a macro to edit, compile, and debug any source file. Its definition might look like this: `doskey edc=edit $1.cpp$Tbcc $1.cpp$Ttd $1.exe`

Symbol	*Purpose*
	When you type `edc myprog`, this expands into the following: `doskey ecd=edit myprog.cpp$Tbcc myprog.cpp$Ttd myprog.exe`
`$*`	This symbol is a special form of `$1` through `$9`. The `$1` through `$9` symbols match individual tokens on the command line, where each token is any piece of text separated by more than one blank. `$*` does not extract individual tokens. Instead, `$*` is equivalent to all the text on the command line after the macro name.

You can see a list of all your defined macros by typing

```
doskey /macros
```

If you want to, you may redirect the list of macros to a file using the `>` redirection operator:

```
doskey /macros >commands.bat
```

To delete a specific macro, such as `edc`, type a command like this:

```
doskey ecd=
```

If you assign nothing to the symbol, DOSKEY deletes the symbol from memory.

SETTING THE KEYSTROKE REPEAT RATE

After you press and hold down a keyboard key for a moment, the keystroke begins to repeat automatically. This automatic repeat feature is especially useful when you are working in word processors or similar applications where you can quickly jump across a line by holding down the right arrow key. As you become a proficient user of software, a faster repeat rate lets you move around at greater speed and get more work done in less time. You can set the repeat rate using DOS 5.0's cryptic multifunction `MODE` command (also available in DR DOS 6.0). To set the repeat rate, use the command format

```
mode con: rate=r delay=d
```

where *r* ranges from 1 to 32, corresponding to approximately 2 to 30 characters generated per second, and *d* specifies the length of time you must hold down the key before the keystroke begins to repeat. *d* ranges from 1 to 4, with 1 equal to ¼-second, 2 equal to ½-second, 3 equal to ¾-second, and 4 equal to one-second delays.

WHOLE DISK DATA COMPRESSION

Several software and hardware products are now available to perform real-time data compression of the data written to or read from your hard drive. I have been using STACKER 2.0 from Stac Electronics for quite some time, and I am extremely pleased with its performance. My 100M hard drive, after I installed STACKER, provides approximately 185M of useful data space. A software tool such as STACKER is certainly the least expensive way to upgrade a hard drive's capacity. STACKER averages a 2:1 compression ratio for most systems, although the actual ratio depends on the type of data that you keep on your hard disk. STACKER is not the only data compression product available, but it has been consistently rated by the personal computer magazines and newsweeklies as one of the best performers. Other products include SuperStor and ExpanzPlus. A version of SuperStor is included in DR DOS 6.0. Consult "Using SuperStor Disk Compression" in the DR DOS 6.0 *Optimization and Configuration Tips* booklet.

Data compression technologies work by taking advantage of redundancy in typical data files and by recognizing that the American Standard Code for Information Interchange (ASCII) codes are not always the most efficient internal codes for storing information. By inserting a compression and decompression step into each disk write and read, respectively, the data stored on disk can be compressed into a smaller format. Typically, an average hard drive can hold twice as much information when the data on that hard drive is compressed. A 100M drive, when using automatic compression software, might hold as much as 200M of data.

Data compression software also uses another trick to yield increased storage capacity. When data is written to a disk, it normally is stored in sectors, each of which may be 512 bytes per sector. If your data is not an even multiple of 512 bytes (and whose is?), the last sector of a file contains unused space. Data compression software manages the sectors so that no space goes unused.

Data compression software is installed as a device driver that intercepts all disk input and output operations in real-time. Transparent to your applications, the compression software manages the decompression and compression as each block is read or written to the disk. Obviously, the extra compression or decompression step adds a small amount of processing time to each block. On 80386 CPUs or better, the time is negligible, perhaps on the order of a 10 percent increase in disk I/O time. For the 80286 CPU, which runs slower than the 80386, you might find the additional wait time to be objectionable. For this reason, data compression products are available in both software-only and hardware-assisted models.

You might think that for top-of-the-line performance you should buy the hardware-assisted compression systems. For fast CPUs in the 80386 or better class, however, there is no significant difference between hardware-assisted and software-only solutions. Therefore, if you are using a high-performance CPU, I recommend that you choose a software-only compression product.

USING STACKER 2.0 WITH BORLAND C++

All normal operations in Borland C++ work just fine in conjunction with STACKER. STACKER transparently provides data compression services behind the scenes. There is one feature that you might want to tweak to improve the Borland C++ compiler performance when using a compressed hard drive.

During compilation of large projects, the compiler can run short of memory. When this occurs, the compiler might swap some of its internal data to disk. If you are using data compression software, you can improve the swapping speed by relocating the swap file to either a RAM disk or an uncompressed disk volume. This is especially true when you use data compression because the swap file typically is accessed very frequently. Each time the swap file is read or

written to, the data must be decompressed or compressed, respectively. Even though a product such as STACKER 2.0 is extremely fast, you will notice the performance degradation. The easiest way to fix this is to use the Options | Environment | Startup dialog box to specify a Swap File Drive letter other than the compressed disk volume.

For a typical STACKER installation where drive C: is your main compressed disk volume, drive D: is the uncompressed volume. Before doing this, be sure that your uncompressed disk volume has sufficient space for the creation and operation of a swap file. If there is not enough space, you can shrink the STACKER compressed disk volume using sdefrag, although this shrinks the total amount of space available in the compressed disk volume. Consult your STACKER manual for details.

CHAPTER 2

POWER FEATURES OF THE IDE AND BORLAND C++

The Borland Integrated Development Environment (IDE) adds a great deal of flexibility to your programming. The IDE is structured so that it can become your base of operations for all software development, including editing, compiling, linking, project building, assembly language programming, access to other editors and utilities, use of the Turbo Debugger, and use of the Turbo Profiler. You also can customize the IDE interface to resemble other popular word processors, such as BRIEF. A number of options help you to tailor the compiler's code generation to produce the fastest programs or the smallest programs, or to minimize the time spent compiling. This chapter walks through these features and provides suggestions for speeding up the compile-link-run development cycle.

Using the Transfer options in the IDE

Customizing the IDE editor and the mouse buttons

Achieving faster compiles

Generating faster, smaller code

USING THE TRANSFER OPTIONS IN THE IDE

Hidden beneath the system menu (the small box containing three horizontal lines just to the left of the File menu) is the Transfer menu. This menu displays a list of programs that you can call directly from within the IDE. Figure 2.1 shows the default Transfer menu. You may use the Transfer menu to directly launch Turbo Debugger, Turbo Assembler, or other applications. When you use the menu to launch Turbo Assembler, Turbo Assembler attempts to assemble the contents of the top edit window. Using the Transfer menu in this manner enables you to stay within the IDE for all your programming. (Be sure to see the description of the TRANCOPY utility in Chapter 3, "Using Programming Utilities." TRANCOPY enables you to copy transfer menu items between projects.)

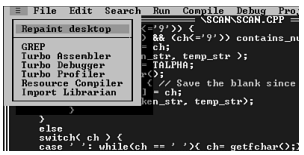


Figure 2.1. The IDE's Transfer menu.

You can drop down the Transfer menu by clicking the system menu icon or by pressing the Alt key and space bar simultaneously. Select an item from the menu just as you would select from any other IDE menu.

If you launch the IDE from within Microsoft Windows, you cannot use the Windows Alt-space bar key combination to access the window's System menu (the small rectangle at the upper-left corner of each application window). Borland C++ intercepts the Alt-space bar key-stroke combination so instead you must use the mouse to click on the System menu icon.

When running a full-screen DOS application (in Windows Enhanced mode only), you normally can press Alt-space bar to simultaneously resize the full-screen application to fit within a window on the screen and to display the System window. But again, Borland C++ intercepts the Alt-space bar combination, so this will not work. Instead, use Alt-Enter to toggle the DOS application back and forth between full-screen mode and windowed execution mode. If executing a full-screen application, press Alt-Enter to resize the application to fit inside a window. If the application is already running inside a window, press Alt-Enter to cause the application to switch to full-screen execution mode.

CHANGING THE TRANSFER MENU

The Transfer menu initially is configured to run GREP, Turbo Assembler, Turbo Debugger, Turbo Profiler, the Resource Compiler, and the Import Librarian. You also can add your own programs to this menu. To add or modify the existing Transfer menu program list, select Options | Transfer.... This displays the Transfer dialog box (see Figure 2.2) showing a list box containing all current Transfer applications. To modify an entry, highlight the desired application in the list box and choose Edit. To add a new entry, move the highlight bar past the last item on the list and choose Edit.

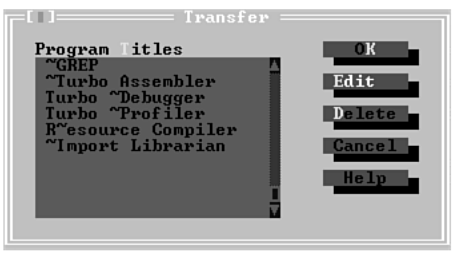


Figure 2.2. The dialog box used to add, modify, and delete items on the Transfer menu.

ADDING A NEW TRANSFER ITEM

To add a new entry, move the highlight bar to the end of the list of programs shown in the dialog box's list box. Then press the Edit button to display the Modify/New Transfer Item dialog box. Use the fields of this dialog box to describe the program you are adding to the Transfer menu. Use Program Title to enter a textual description of the program; this is the text that will appear on the Transfer menu. To create a shortcut for the menu item, place the tilde character (~) before the letter you want to highlight and activate as the shortcut key. For example, to add a text editor program, you might enter

```
~Editor
```

in to the Program Title field. The letter *E* becomes the shortcut key for this menu item. If you omit a shortcut key designation, no shortcut is defined for this selection.

At the Program Path field, type the path and the program name needed to access the executable file. Use the Command Line field to enter macro commands (see the section "Using Macro Commands with Transfer Programs"). You can optionally assign one of the available hot keys (such as Shift-F2) to this program for quick activation while editing and doing other operations in the IDE.

As soon as you have finished entering the program data, select the New button to add this program to the Transfer menu.

EDITING AN EXISTING TRANSFER ITEM

To edit an existing menu item, highlight the desired program in the Transfer items list box and choose Edit. Using the Modify/New Transfer Item dialog box, you can edit or delete any of the information described in the previous section. When you have finished entering the changes, select the Modify button.

DELETING A TRANSFER ITEM

To delete an existing Transfer item, highlight its name in the Transfer dialog box's list box and choose the Delete button. The delete operation takes effect immediately and the highlighted program name is removed from the list of available programs.

USING MACRO COMMANDS WITH TRANSFER PROGRAMS

In the Command Line field of the New/Modify Transfer Item dialog box, you can enter a sequence of macro commands to pass information to the called program and to set up the environment within which the program will execute (such as how many kilobytes of memory will be reserved for the program's execution). For example, suppose I'm installing my favorite text editor program, named EDIT. In the Command Line field, I enter

```
$EDNAME
```

When EDIT is invoked, the $EDNAME macro is expanded to the name of the file currently loaded in the IDE's edit window and is appended to the command line used to start the editor. For example, editing scan.cpp in the IDE by launching the EDIT program using its Transfer item is equivalent to typing at the DOS command line

```
EDIT scan.cpp
```

When you prepare your own macros, you might have some difficulty achieving the correct macro expansion. To help debug your transfer setup, place the $PROMPT macro at the beginning of your command-line definition. When you do this, the IDE displays a dialog box showing the completely expanded command line. If you want to, you can edit the resulting expansion or cancel the operation. Use $PROMPT when you are designing your transfer applications to ensure that all your parameters are written correctly.

Borland provides a large set of macros for use with Transfer applications to provide support to specific Borland application requirements (such as Turbo Assembler) as well as for use by your applications. Borland divides the set of macros into three groups: *state macros*, *filename macros*, and *instruction macros*. The state macros provide information about current IDE settings and options. The filename macros provide filename information or process filenames for use in constructing command lines. The instruction macros cause the IDE to take a particular action or to adjust a particular setting.

Depending on the macro, some macros return a value and others act like a function, processing the value returned by one macro and then expanding or truncating the result into a new value. Table 2.1 outlines the state macros, Table 2.2 outlines the filename macros, and Table 2.3 outlines the instruction macros.

TABLE 2.1. THE STATE MACROS.

Macro	*Description*
$COL	Is translated to the column number where the cursor is located in the active edit window. If the active window is not an editor window, this returns 0.
$CONFIG	This macro is primarily used by Borland products. $CONFIG returns the name of the configuration file so that launched applications can access or modify values in the configuration file.
$DEF	On the Options	Compiler	Code Generation dialog box is a field labeled Defines. The $DEF macro returns the text value stored in the Defines field.
$ERRCOL	If the file specified by $ERRNAME has any compile errors, $ERRCOL contains the column number of the position in the file where the error was detected. If there are no errors, $ERRCOL is a null string.
$ERRLINE	If the file specified by $ERRNAME has any compile or other errors, $ERRLINE contains the line number of the position in the file where the error was detected. If there are no errors, $ERRLINE is a null string.
$ERRNAME	When the Message window contains messages related to particular files, the $ERRNAME variable contains the name of the file that is referenced in the Message window.
$INC	On the Options	Directories dialog box is a field labeled Include Directories. $INC returns the value of the edit field associated with this label.
$LIB	On the Options	Directories dialog box is a field labeled Library Directories. $LIB returns the value of the associated edit field.
$LINE	If the active window is an editor, $LINE returns the current line number within the edit window. Otherwise, if the active window is not an edit window, $LINE returns 0.

Macro	*Description*
$PRJNAME	If a project file is defined, $PRJNAME returns the name of the project file. If no project file is in use, $PRJNAME returns null.

TABLE 2.2. THE FILENAME MACROS.

Macro	*Description*
$DIR	Provides the full directory pathname of the file that is currently being edited.
$DRIVE()	Extracts the drive letter from the directory path specified as its parameter. For example, $DRIVE($EDNAME) returns C: if $EDNAME returns C:\SOURCE\SCAN.CPP.
$EDNAME	If the active window is an editor window, $EDNAME expands to the complete filename of the file being edited. Otherwise, if an edit window is not active, $EDNAME returns a null string.
$EXENAME	Returns the filename of the executable file that would be produced by compiling the file in the edit window. If a project is opened, $EXENAME returns the name of the file that is produced by the project. If the IDE is configured to produce a .DLL, $EXENAME returns the name of the .DLL file.
$EXT()	Returns the filename extension of its parameter. For example, $EXT($EDNAME) probably would return .C or .CPP.
$NAME()	Returns the filename part of its parameter. For example, $NAME($EDNAME) returns SCAN when $EDNAME is SCAN.CPP.
$OUTNAME	Returns the content of the Output Path field in the Project | Local Options dialog box.

TABLE 2.3. THE INSTRUCTION MACROS.

Macro	*Description*
`$CAP EDIT`	When the outside application runs, this macro causes the outside application to redirect its program output to a disk file. This works only when the transfer program normally sends its output to the DOS stdout file. As soon as the application has finished, the IDE opens a new edit window and loads the content of the output file into the window.
`$CAP MSG(`*`filter`*`)`	`$CAP MSG()` is similar to `$CAP EDIT`, except that the program's output is redirected into the Message window within the IDE. The filter is the name of a program that converts the program output into special Message window format. Borland provides GREP2MSG, IMPL2MSG, RC2MSG, and TASM2MSG to process output from GREP, IMPLIB, RC, and TASM, respectively. If you include source files when you install Borland C++, the source code for these programs will be included and can be used as a model for developing your own filters.
`$DEP()`	`$DEP()` is not used by the Transfer programs, but it is used by the Project Manager. `$DEP()` is short for "depends on" and may be used to indicate dependencies between files. You use `$DEP()` with a list of files.
`$IMPLIB`	Executes the IMPLIB program.
`$MEM(`*`kbytes`*`)`	Use `$MEM(`*`kbytes`*`)` to reserve *`kbytes`* of RAM for the transfer program. The IDE tries to reserve *`kbytes`*, but if there is insufficient memory, the IDE provides as much memory as it can.
`$NOSWAP`	When you are using `$CAP` to copy a transfer program's output to a window, it is not always necessary to display the transfer program when that program executes. By referencing the `$NOSWAP` macro, the IDE will not display the transfer program's user screen.

Macro	*Description*
$PROMPT	When you write your own macros, it can be confusing trying to construct the proper arguments and parameter sequences. When you add $PROMPT to your command line, the IDE displays the fully expanded command line in a dialog box (beginning at the position of $PROMPT). Using the prompt dialog box, you may edit or change the expanded string before it is passed to the transfer program.
$RC	$RC is used when calling the Resource Compiler.
$SAVE ALL	Referencing $SAVE ALL causes all modified files in all edit windows to automatically be saved to disk without prompting.
$SAVE CUR	If the active edit window contains a modified file, $SAVE CUR causes that file to be saved to disk without prompting.
$SAVE PROMPT	Using $SAVE PROMPT causes the IDE to warn you that you have unsaved files in at least one of the edit windows.
$TASM	Used to call the Turbo Assembler.
$WRITEMSG(*filename*)	Outputs the content of the Message window to *filename* in ASCII format.

USING THIRD-PARTY TEXT EDITORS

Most programmers develop a strong affinity for their text editor. That is not surprising, considering that programmers probably spend more time editing source text than any other function. As soon as you've mastered all the ins and outs of your favorite editor, it is difficult to use any other program. You can use the Transfer menu to easily access your own editor program. See the preceding section, "Using Macro Commands with Transfer Programs," for an example of how to install your favorite editor in the Transfer menu.

The companion diskettes to this book include the MR_ED shareware programming editor. This is an excellent programmer's editor, providing editing of multiple files, a pull-down menu interface, and the capability to edit extremely large files. (I've used it to edit text files up to nearly one megabyte in size.) When you jump into your own editor from within the IDE, you still can access Borland's online help system. See the description of the THELP program in Chapter 3, "Using Program Utilities," for information on installing the THELP online help TSR program.

If your favorite editor is BRIEF, Epsilon Programmer's Editor, or the MS-DOS full-screen editor, you might not have to install your favorite editor into the Transfer menu. The IDE's editor is chameleonlike: you can reconfigure the editor's behavior to mimic one of these three editors. Reconfiguring the editor's behavior is described in the next section.

CUSTOMIZING THE IDE EDITOR

The IDE editor is implemented through a special macro language called the Turbo Editor Macro Language (TEML). *This macro language has no relationship to the macros defined for use in the Transfer program options*. Borland provides a compiler for translating TEML source programs into an internal format that may be used for adapting the IDE's editor. This section takes a look at how the macro language is used and compiled using the Turbo Editor Macro Compiler, and how macros may be used to implement the characteristics of other editors.

USING THE TURBO EDITOR MACRO COMPILER

Borland provides a set of TEML source files that you can use to emulate the keystrokes of BRIEF, the MS-DOS Editor, or the Epsilon Programmer's Editor. As soon as the editor is installed, the regular IDE editing keystrokes no longer apply. Instead, the editor mimics the commands of the desired editor. The pull-down menus, however, remain unchanged. This facility is intended for those of you who like the IDE but can't get your fingers to land on the correct keys. Only the keystrokes are changed.

The editor (and much of the IDE's) configuration is stored in the tcconfig.tc file in the \BORLANDC\BIN directory. If you want to play with the Turbo Editor Macro language or to select one of the standard editor emulators, first

make a backup copy of the original tcconfig.tc. If you don't like the result of your fiddling, you can restore tcconfig.tc from your backup.

To run the Turbo Editor Macro Compiler, type

```
temc inputfile outputfile
```

where *inputfile* is the name of the TEML source file and *outputfile* is the name of the configuration file (such as tcconfig.tc). Borland provides a set of predefined macro files ending in the .tem extension. These files are located in the \borlandc\doc directory and include the following:

Filename	*Usage*
brief.tem	Implements BRIEF keystrokes
dosedit.tem	Implements MS-DOS Editor keystrokes
epsilon.tem	Implements Epsilon Editor keystrokes
defaults.tem	Implements default IDE editor keystrokes
cmacros.tem	A set of useful macros

To operate the editor using BRIEF keystrokes, you need to compile brief.tem like this:

```
temc brief.tem tcconfig.tc /c
```

`temc` accepts two command-line switches, `/c` (or `-c`) and `/u` (or `-u`). When you add `/c` to the command line, any existing keystroke and command definitions in tcconfig.tc are deleted before adding the definitions provided in the new input file. If you do not use `/c`, temc merges the new keystrokes with the existing keystroke table stored in tcconfig.tc. I recommend that you use `/c`.

The IDE's editor has three user interface flavors: Native, Alternate, and CUA (which is short for IBM's Common User Access user interface specification). Native is the default mode of operation. When you run temc, the new keystroke definitions are added to the Alternate command set. If you add the `/u` command-line switch, the new definitions are copied over the CUA keystrokes.

When running temc, make sure that you copy the brief.tem file to \borlandc\bin prior to running `tem`, or that you run `tem` in the \borlandc\doc directory and copy the resulting tcconfig.tc file back to \borlandc\bin. Use the Options | Environment | Preferences... dialog box to select the Editor's Alternate command set or the CUA command set. If you want to create your

own macro language scripts, you probably should start with one of the existing .tem files and add to or modify the files to achieve the desired result.

ABOUT THE TURBO EDITOR MACRO LANGUAGE (TEML)

The Turbo Editor Macro Language is easy to use, especially if you use TEML for modifying existing editor macro files. The language consists of three components: keywords, such as `WordLeft`, representing basic editor functions; macro definitions that invoke multiple editor functions; and keystroke binding tables.

A typical macro language program consists of optional macro definitions (similar to subroutines in a high-level language) and keystroke bindings. For instance, if I want to add a delete word function to the IDE, I can define my own function to respond to the Alt-W keystroke. You can define functions for any keystroke you want to use, including single characters, Ctrl- or Alt- characters, and some special combinations that are recognized by the IDE. You also can intercept key combinations. To intercept the Ctrl-W keystroke followed by the capital letter *A* followed by the capital letter *B,* you separate each key with a plus (+) symbol, such as Ctrl-W+A+B. Note that in this example, A and B both are case-sensitive; you need to define Ctrl-W+a+b to catch lowercase *a* and *b*.

For example, to define a delete word function, you can edit the defaults.tem file to add the following code:

```
alt-w : BEGIN WordRight;WordLeft;DeleteWord;
END;
```

This statement attaches a sequence of built-in editor functions to the Alt-W keystroke. The built-in `DeleteWord` function deletes from the current cursor location to the end of the current word. Making this function into a delete-this-entire-word function requires that the cursor be positioned to the beginning of the word. An easy way to do this is to find the end of the word that the cursor is positioned on and then reposition to the beginning of that word. As soon as the cursor is on the first character of the word, `DeleteWord` deletes from that first character to the end of the word.

`WordRight` is inserted before `WordLeft` because in the event that the cursor is on a blank between two words, `WordRight` moves to the end of the next word. This results in Alt-W deleting the next word rather than the previous word in the line.

You can group editing functions into macro definitions that may be used

throughout the macro language program. The DeleteWord function might be coded in a macro definition like this:

```
MACRO MacDeleteWord
    WordRight;WordLeft;DeleteWord;
END;
```

and the keystroke binding written as

```
alt-W : MacDeleteWord;
```

You may also use the macro language to create automatic text templates. Suppose that you add a lot of comments to your source code using the `/*` and `*/` comment delimiters. It is easy to create a macro that automatically inserts `/*` and `*/` and then repositions the cursor to appear between the comment delimiters. Here's an example:

```
MACRO CommentBlock
 InsertText("\n/*\n\n");
 InsertText( "*/" );
 CursorUp;
END;
alt-c : CommentBlock;
```

With this definition in place, pressing Alt-C moves to the start of the next line, inserts `/*`, moves down two lines and places `*/`, then moves back up to the blank line between the `/*` and the `*/`. You can create definitions like this to insert arbitrary text such as function headers or class definitions. Borland provides a file named cmacros.tem containing a number of useful macros that you can use to add custom features to the IDE. These macros are summarized in Table 2.4.

TABLE 2.4: THE MACRO DEFINITIONS PROVIDED IN CMACROS.TEM.

Macro	*Description*
Alt-I	Inserts `#include <stdio.h>` followed by an `int main(void)` program stub.
Alt-K	Use this macro to insert a comment block. Inserts `/**********` across the line, followed by a blank line, followed by `**********/`.

continues

TABLE 2.4. CONTINUED

Macro	*Description*
Alt-M	Inserts an `int main(void)` function and a `return` statement into the editor.
Alt-N	Inserts `#include <iostream.h>` followed by a stubbed `int main(void)` function.
Alt-T	Inserts a function header comment block, including a beginning line containing `/*` followed by a line of asterisks, a text field labeled Description, and a trailing line containing asterisks followed by `*/`.
Alt-Z	After you enter a function name, pressing Alt-Z inserts `void` before the function name and `(void){};` after, creating a stubbed function declaration.

The Turbo Editor Macro Language is easy to use, especially if you confine yourself to modifying the existing .tem files provided by Borland. If you'd like to add a new feature to the editor, do not hesitate to give this facility a try. You can find a complete list of all available editor functions in the Borland-provided \borlandc\doc\util.doc file. For examples, refer to the sample files such as defaults.tem or cmacros.tem.

CUSTOMIZING THE MOUSE BUTTONS

The mouse pointing device typically has two buttons, commonly called the *left mouse button* and the *right mouse button*. The left mouse button is used to click, double-click, or perform the click-and-drag function. The right mouse button initially is configured to display context-sensitive help. When editing a source file in the IDE, you can position the mouse over a keyword and press the right button; the IDE displays help about the keyword. You also can use this to display information about all standard library functions. Simply click the right mouse button when the mouse pointer is located over a function name.

The IDE provides a mouse customization feature that lets you redefine the use of the right button, vary the time duration for double-clicking, and reverse the use of the left and right mouse buttons. To use the customization feature, select Options | Environment | Mouse.... Use the radio buttons beneath the Right Mouse Button heading to select a different function for the right mouse button. You may optionally ignore right button clicks, access the search, search again or replace editor functions, or perform various debugging functions.

To vary the mouse double-click rate, drag the thumb located in the slider control bar beneath the Mouse Double Click heading. The slider varies from a Fast setting to a Slow setting. At the fastest settings, the IDE requires a fast double-click, and at the slow rate you can issue double-clicks with a greater amount of time between the clicks.

CONFIGURING THE MOUSE FOR LEFT-HANDED USAGE

If you are left-handed you may find it easier to use the mouse with your left hand by making the right mouse button the primary selection button (rather than the left button). The IDE lets you reverse the left and right buttons. This way, when using the mouse with your left hand, you can conveniently use your left hand's index finger to click the right mouse button.

ACHIEVING THE FASTEST COMPILES

By selecting appropriate options in the Borland C++ compiler, you can obtain significant improvement in the time required to compile large projects. This section does not detail all possible speed improvements because many compiler options have little significant impact on compiler speed. For example, if you know that your program will run on an 80386, you can select 80386 code generation because this can result in fewer machine instructions, especially for handling double word (long) values. Because this produces fewer instructions, the compiler may run imperceptibly faster. The performance improvement is so small, though, that it is not noticeable. For this reason, this section mentions only option settings that have a significant potential for speeding up your compiles.

SOME SIMPLE IDEAS

Whenever possible, use the built-in MAKE to intelligently recompile only files that have changed. Using Build All when it is not necessary merely adds a lot of time to the compilation process. Both the Make and Build All options are located on the Compile menu.

Extraneous warning messages may be suppressed by using the Options | Compiler | Messages options. The compiler issues warnings when it finds suspicious programming techniques or coding methods that are known to cause problems in certain situations. These problems are not errors; indeed, the compiler will compile and link your program regardless of warnings.

Because warnings usually are benign, it is easy to get in the habit of ignoring them. Although some of the warnings may be safely ignored (such as warnings about using features of Borland C++ that might not be portable to other compilers), I do not recommend that you ignore warnings unless you are completely certain that you understand why the warning was given. For all but the most innocuous problems, you should strive to clean up your code to eliminate warning messages. This is merely good programming practice.

When you are developing new code, you can optionally disable all or some of the warning messages. To disable the display of all warnings, use the Options | Compiler | Messages | Display... selection and choose None beneath the Display Warnings heading. To selectively disable warnings, choose Options | Compiler | Messages, and from the menu that appears, choose one of the selections shown in Table 2.5.

TABLE 2.5. CATEGORIES OF WARNING MESSAGES.

Category	*Description*
Portability	This group of warnings indicates when you are using a feature that might not be portable outside of Borland C++. If you have no need to move your source to another C or C++ programming environment, then you may safely disable these warnings.
ANSI violations	The Borland C++ compiler provides exten sions to the C language that go beyond the

Category	*Description*
	American National Standards Institute (ANSI) definition of C. This group of warnings lets you know when you are using a feature outside the scope of the ANSI definition.
C++ warnings	This indicates when you are using certain obsolete portions of C++. The C++ language is a new language that is continuing to evolve. Over time, some of its features have been replaced with newer solutions. Consequently, although the compiler will continue to recognize the older forms, it will warn you when you use C++ features that have been superseded.
Frequent errors	Frequent errors are those that are considered quite common. For example, programmers sometimes forget to specify a return value for a function. When the function is used as a procedure, this really is not a program error, just sloppy programming. Today's standards recommend declaring a `void` return type for functions that are called as procedures.
Less frequent errors	Like the Frequent errors category, Less frequent errors include such items as defining an identifier that is never used.

USING /*X*, /*E*, AND /*R* IDE COMMAND-LINE SWITCHES

If you have extended memory available, use the `/x` option to instruct the IDE to use extended memory for its internal heap memory space. You may specify a size for the heap by adding `/x=`*n* where *n* is the number of kilobytes to reserve for the IDE. When `/x` is used by itself, the IDE reserves all available extended memory for its own use. The `/x`, `/e`, and `/r` options are all specified on the command line used to invoke the IDE. Here is an example:

```
C>BC /x
```

By default, the IDE uses any available expanded memory as a swapping area. You can set the size of the swap area by typing `/e=n`, where *n* is the number of 16K pages to reserve.

If your system is configured with a RAM disk, use `/rx` to tell the IDE where the RAM disk is located. Substitute the drive letter for *x*.

BCC VERSUS THE IDE

Depending on your system configuration, you might find that using the command-line compiler BCC (see Chapter 3, "Using Programming Utilities") compiles and builds your applications more quickly than using the IDE. Considering the number of times that you will recompile your source code over the course of a program's development, I recommend that you compare the speed of using BCC versus using the IDE. Make some timing tests of your program compilations, using both the IDE and the BCC. Run your timing tests more than once, particularly if you have a large software disk cache. I have found better than a two-to-one difference in compile and link times between the first run and a second run. Also, ensure that all your test runs consistently enable or disable precompiled headers (see the section "Using Precompiled Headers").

When you need to recompile multiple modules, specify the additional modules on the same BCC command line. Rather than writing

```
C:>BCC module1.c
C:>BCC module2.c
C:>BCC module3.c
```

use this alternative form to compile these modules:

```
C:>BCC module1.c module2.c module3.c
```

Because the BCC compiler does not need to be loaded from disk (BCC is sufficiently large that loading from disk takes quite a while), this form is much faster than compiling each module separately. You may also use wildcard characters in each filename. You might use `module?.c` to compile all files beginning with `module` and having any character in the space occupied by the `?`, followed by the `.c` extension. When you construct MAKE files that call the compiler, keep this alternative form in mind (see Chapter 3, "Using Programming Utilities").

DISABLING OPTIMIZATIONS

For overall fastest compiling, set up your system to use a large (at least 2M) disk cache. Enable precompiled headers and disable all possible optimizations. If you want to, you can disable most compiler optimizations. Producing optimized code adds an extra burden to the compiler. During the optimization process, the compiler analyzes your statements and the resulting compiled code to discover the best instruction sequence. By disabling optimizations, you can improve compiler speed by 20 to 50 percent.

When optimizations are in effect, the resulting machine code sometimes may bear little resemblance to what you might expect. The compiler might eliminate common subexpressions and assign them to a temporary variable. Array indices that are constant during a loop, such as a reference to `x[j]` where `j` remains unchanged, may be converted to a temporary variable. In some instances, the compiler might even rearrange your statements.

If you must debug sections of code at the machine level, this difference in generated code might prove bothersome because the generated code might not have a one-to-one correspondence with your source statements. Additionally, if you are in the habit of generating code sometimes with optimizations toggled on and sometimes off, the resulting code will be very different when toggled on versus toggled off. This could add a level of confusion to your debugging efforts. Fortunately, the Turbo Debugger can always identify which line source lines are being executed when debugging through machine code.

To set the optimization level of the compiler, select Options | Compiler... | Optimizations.... In the Optimizations dialog box, you can disable all check box items beneath the Optimizations heading. Beneath Register Variables, select None, and beneath Common Subexpressions, select No Optimizations. You can turn appropriate options back on by selecting either the Fastest Code or the Smallest Code buttons.

SETTING THE /*S* SWITCH

During compilation, the IDE uses most of its memory allocation for internal compiler data structures. In some instances, this might cause the compiler to make numerous swaps to and from the hard disk because little memory space is left for other functions. The result is that the system begins to *thrash*, or

rapidly read and write swappable blocks in and out of memory. This can quickly lead to lethargic system performance. To eliminate this occurrence, you can add the `/s-` switch to the BC command line when starting the IDE. In this configuration, the IDE limits its usage of memory, and may result in an "out of memory" error condition.

USING PRECOMPILED HEADERS

For very large projects that have many header files, such as Windows or ObjectWindows applications or Turbo Vision applications that must include many header files, the time required to compile just the header files can easily be 50 percent of the total compile time. Borland C++ introduced the *precompiled* headers feature to eliminate most of the time needed to parse a header file. When you precompile each header file and save the internal symbol table information in a special file, subsequent compilations can access the precompiled symbol table at high speed. A typical precompiled header is processed approximately ten times faster than a header source file.

The only drawback to using a precompiled header is that the symbol table file, named either tcdef.sym or *projectname*.sym, where *projectname* is the name of your project file, can use up enormous amounts of disk space. If you are working on multiple projects, you might need to delete the .sym files from the projects that you are not currently editing and compiling. From time to time, you should scan through your development directories looking for .sym files. Delete any unnecessary files; the precompiled header data can always be re-created at a later time.

SELECTING THE PRECOMPILED HEADERS OPTION

The precompiled headers option is available in both the IDE and when using the command-line compiler. In the IDE, choose Options | Compiler | Code Generation to display the dialog box shown in Figure 2.3. In this dialog box, select the Pre-compiled headers check box to use precompiled headers. Leave the check box clear if you do not want to use precompiled headers.

The symbol table information normally is written to a file created by combining the project filename with the .sym extension. If your project is named mailordr.prj, then the default symbol file is mailordr.sym. You can select a different filename through the use of the `#pragma hdrfile` directive. Place at the top of your source file a statement such as

```
#pragma hdrfile "newsymbs.sym"
```

to specify that newsymbs.sym should contain the symbol table data. This `#pragma` is ignored if you have elected not to use precompiled headers.

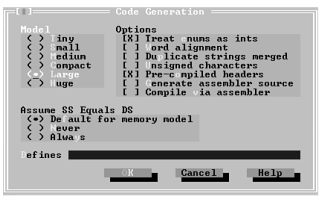


Figure 2.3. Use the Code Generation dialog box to use precompiled headers.

To enable precompiled headers in the command-line compiler, add the `-H` option to the command line. By default, precompiled headers are disabled. The `-H` option causes the compiler to both create and use precompiled header information. If you want to have the compiler only use, and not create, symbol table files, use `-Hu`. By default, the symbol table is kept in tcdef.sym. You can specify a new symbol table filename using `-H=`*`filename.ext`*.

USING PRECOMPILED HEADERS EFFICIENTLY

To optimize the use of precompiled headers and to reduce the size of the tcdef.sym file, Borland recommends the following guidelines:

- In each of our your .c or .cpp source modules, arrange the list of `#include` statements in the same order. In other words, keep the same sequence of your `#include` statements across each of your .c or .cpp source files.
- Always put the largest header files at the beginning of your `#include` statements. For example, files such as windows.h should appear before shorter header files.

- Use the `#pragma hdrstop` directive to limit the header files that will be precompiled.

If the guidelines cause a conflict, follow the first guideline that applies. For instance, in order to keep your `#include` statements in sequence across several source files, you might need to put larger header files after the initial group of sequenced files. This strategy is correct because it applies the first rule before applying the second.

Internally, the symbol table file stores the date and time stamp of the header file at the time it was incorporated into the symbol table file. If, during later compilations, the compiler finds that these header files are newer, it recompiles all the header information. The compiler also ensures that the various code generation options in effect at the time of the initial compilation are still in effect. For example, if you change the memory model configuration between compilations, the symbol table information will no longer be accurate.

USING *#PRAGMA HDRSTOP*

You can optionally disable inclusion of all of your `#include` header files. You might want to do this if your disk space is limited (and whose isn't?) or if the header files involved are small. To optionally include header file information in the precompiled symbol table, place the `#include` statements that you want to have precompiled at the beginning of your group of `#include` statements. Then, before the `#include` statements that you do not want precompiled, place the statement

```
#pragma hdrstop
```

Subsequent `#include` statements will not be included in the precompiled symbol table information.

GENERATING THE FASTEST AND THE SMALLEST CODE

The Borland C++ compiler has an internal optimizer that can produce code that is optimized for fastest execution or code that is optimized to use the fewest instructions. Optionally, you may disable all optimizations. To produce

optimized code, the compiler employs several strategies. Using the Options | Compiler | Optimizations... dialog box, you can choose which of these strategies the compiler should employ (see Figure 2.4). Table 2.6 shows the corresponding command-line compiler switches. These switches can be given individually, such as `-Oa -Oc`, or as a group, such as `-Oac`. To disable an option, prefix the option letter with a `-` symbol, such `As -O-a`.

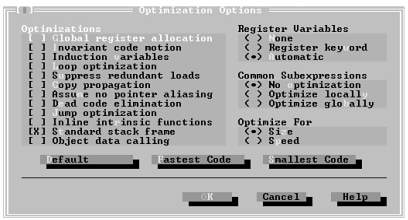


Figure 2.4. The Optimization Options dialog box.

TABLE 2.6. COMMAND-LINE COMPILER SWITCHES TO SELECT OPTIMIZATION STRATEGIES.

Switch	*Usage*
`-O2`	When the compiler has a choice, this option instructs the compiler to optimize for speed.
`-Ox`	Generates the fastest possible code.
`-O1`	Generates the smallest possible code.
`-O`	Removes redundant jumps, code that can never be executed, and unnecessary jumps.
`-Oa`	Assumes that pointers are not aliased. An aliased pointer is one that may point to a memory area owned or managed by other code or pointers. Hence, the pointer is an alias for that location. By assuming that pointers are not aliased, the compiler can produce better optimizations.
`-Ob`	Removes writes to variables that are never used and evaluates expressions during compilation that cannot be changed during program execution. Use `-Ob` when you use `-Oe`.

continues

TABLE 2.6. CONTINUED

Switch	*Usage*
`-Oc`	Eliminates common subexpressions within local blocks only. See `-Og`.
`-Od`	Disables all optimizations.
`-Oe`	Optimizes global register allocations and performs code analysis that is used by other optimization strategies. The `-Ob` optimizer requires that `-Oe` be selected to perform code analysis.
`-Og`	Eliminates common subexpressions within entire functions. See `-Oc`.
`-Oi`	Causes various string and memory library functions to become inlined (actual code is inserted rather than a call to the library routine).
`-Ol`	Optimizes loops that are used for initialization sequences (such as `for(i=0;i<50;i++){x[i]=0};`) so that they use the REP prefix CPU instructions to perform block moves.
`-Om`	Moves code whose results are unaffected by a loop to an area outside the loop. This can cause tremendous performance improvements in loop execution speed.
`-Op`	Causes the compiler to track the values of previously calculated subexpressions and reuses the previous result if possible.
`-Os`	When the compiler has a choice, the compiler will optimize the generated code for smallest size rather than speed.
`-Ot`	When the compiler has a choice, the compiler will optimize the generated code for fastest execution speed.
`-Ov`	Uses a special technique named induction-variable analysis that is helpful in optimizing loops. It is recommended that you use this option for all loops.
`-Z`	Suppresses reloading values into registers if they are already located in a register.

Switch	*Usage*
`-r`	In Borland C++ 2.0 and Turbo C++ 1.x, this option enabled the use of register variables. This option is considered obsolete but still is recognized in Borland C++ 3.0 and 3.1.
`-r-`	Setting this obsolete option (see `-r`) causes the compiler to suppress the use of register variables. Using this option is discouraged.
`-rd`	This obsolete option restricts the compiler's use of registers for variables only to variables that have been declared with the register keyword.

For most of your programs, you probably will want to generate the fastest code (select the Fastest Code button on the Optimizations dialog box). For programs in which memory limitations are a problem, select the smallest code option. When you want your code compiled as fast as possible during the development phase, disable all optimizations.

Other compiler options should be chosen appropriate to the application requirements. For instance, if you know that the application will run on an 80386 CPU, use the Options | Compiler | Advanced Code Generation... dialog box to select the 80386 as the target CPU. In some instances, this can result in programs that are both smaller and faster.

Select the proper memory model for your requirements using the Options | Compiler | Code Generation dialog box. If your application is a small one with little code and data, use the small memory model. Memory models that support larger code or data sections require twice as many addressing bytes for memory references. See Chapter 5, "Memory Management," for further details on selecting an appropriate memory model.

Table 2.7 summarizes compiler options other than those on the Optimizations dialog box that you should consider when optimizing your program. All these options are available in the command-line compiler too. To enable or disable an option in the command-line compiler, use the options shown in Table 2.7. These options (except where noted) are available in the Options | Compiler | Code Generation and Advanced Code Generation dialog boxes.

TABLE 2.7. OTHER COMPILER OPTIONS YOU SHOULD CONSIDER WHEN OPTIMIZING PROGRAMS

Option	*Description*
Memory model	Use the Options | Compiler | Code Generation dialog box in the IDE to select the model that is appropriate for your application. The larger memory models require somewhat more memory and run somewhat slower than the smaller models. Command-line compiler switches: `-mc` compact memory model `-mt` tiny memory model `-ms` small memory model `-mm` medium memory model `-ml` large memory model `-mh` huge memory model
Options/Treat enums as Ints	Leaving this option unchecked causes the compiler to generate single bytes for enums in the range of 0..255 or –128..127. When this option is checked, enums are always allocated a full word. Use this option to vary the memory requirement for enums. Command-line compiler switches: `-b` Default setting allocates a whole word. `-b-` Allocates enum values in bytes.
Options/Word alignment	By default (option unchecked), the compiler aligns noncharacter values in structures and unions on byte

Option	*Description*
	boundaries. When this option is checked, the compiler puts all values on word boundaries. Although this might waste memory bytes, the 80186, 80286, 80386, and 80486 CPUs can access word values on word boundaries much more quickly. Command-line compiler switch: `-a` Causes larger items to align on word boundaries.
Options/Duplicate strings merged	If you have a large number of constant string values, the compiler can detect duplicate strings and store one copy of each duplicated string resulting in a memory savings. Make absolutely certain that you do not modify your *constant* strings or you might encounter surprising results. Command-line compiler switch: `-d` When set, causes duplicates to be merged.
Floating Point	If you do not use any floating-point operations, select the None option. When you select None, the linker does not attempt to link floating-point library routines, thereby speeding up the link step. If you can do so, choose one of the floating-point processor selections, either 8087 or 80287/387. These options provide the fastest calculations with the least amount of overhead code. If the sys-

continues

tem that runs your application does not have a math coprocessor or if you

Option	*Description*
	do not know whether it will have a math coprocessor, select the Emulation option. The math emulator will simulate the math coprocessor in software. If it detects the presence of a math coprocessor, it uses the math processor, providing your application with the capability to run with or without a math coprocessor. Command-line compiler switches: `-f-` No floating-point routines are needed. `-f` Uses the floating-point emulation library. `-f87` Generates code to use the 8087. `-f287` Generates code to use the 80287 or the 80387.
Instruction Set	Use the Options | Compiler | Advanced Code Generation dialog box to indicate on which type of CPU your program will run. If you know that the target machine will always be an 80386 or better, choose the 80386 CPU option. If you know that the target machine will be an 80286 or better, select the 80286 CPU. Matching the generation code to the target

Option	*Description*
	CPU can reduce the number of machine instructions and provide a modest performance boost.
	Command-line compiler switches: `-1-` Default setting for 8088- com patible code `-1` Generates 80126 instructions `-2` Generates 80286 instructions `-3` Generates 80386 instructions
Options/Fast floating point	The Fast floating point option enables the compiler to ignore certain issues regarding the conversion of one floating-point type to another (such as a float to a double). This can result in the loss of precision in certain circumstances, but for most programming this is not a problem. Command-line compiler switches: `-ff` Enables fast floating point. `-ff-` Disables fast floating point.
Options/Fast huge pointers	When using huge pointers, setting this option can dramatically improve the computation of expressions using huge pointers, provided that huge array elements do not cross segment boundaries. In the latter case, this option will generate code that does not work. Command-line compiler switch:

continues

	`-h` Selects fast huge pointers (default is off).

Option	*Description*
Calling Conventions	On the Options | Compiler | Entry/Exit Code Generation dialog box, you may select C, Pascal, or of the Register calling conventions. Use of the Register calling convention enables the compiler to pass parameters to functions using the registers rather than the stack. Up to a maximum of three parameters placed into registers. Use of the Register calling convention can significantly increase the speed of access to some functions. You can enable this option on a per-function basis by placing the `fastcall` keyword before a function declaration. Command-line compiler switch: `-pr` Enables the use of the `fastcall` convention.
Smart C++ Virtual Tables	Use the Options | Compiler | C++ Options dialog box to select Smart C++ Virtual Tables. This option results in the smallest and most efficient code generation for C++ virtual functions. Command-line compiler switch:

-v Enables smart C++ virtual tables.

VERSION 2.0 USERS ONLY: USING PROTECTED-MODE BCX

This section does not apply to users of Borland C++ 3.0 or newer compilers. If you are still using Borland C++ 2.0, you should know about the protected-mode version of BC named BCX that comes with Version 2.0 (but not 3.0 or newer). (There is also a companion command-line version named BCCX and a protected-mode linker named tlinkx.) If you have an 80286, 80386, or 80486 microprocessor with 640K of system RAM, plus at least 576K of extended or expanded RAM, you can use the protected-mode BCX compiler. The major advantage to using BCX is to permit the compilation of much larger programs with little or no disk swapping. The BCX interface is identical to BC. You run BCX by typing the following:

```
BCX
```

You need to ensure that the files bcx.ovy and tkernel.exe are located in a directory specified in your DOS PATH statement, or in the same directory where bcx.exe is kept. tkernel.exe and bcx.ovy are loaded into memory automatically when you invoke bcx.exe.

tkernel is a special protected-mode interface required by the BCX program. You can speed up program loading and reduce the amount of conventional memory required by tkernel by manually loading tkernel yourself. To load tkernel, type the following at the DOS command-line before running one of the protected-mode programs:

```
tkernel hi=yes
```

The command-line option causes tkernel to load itself into extended memory, which reduces the demands on conventional memory.

PROBLEMS WITH BORLAND C++ 2.0 AND DOS 5.0

If you use Borland C++ and DOS 5.0, you might need to make some configuration changes, depending on your environment. Borland recommends that when you install emm386 in your config.sys file (assuming you use emm386 and that emm386 is loaded after himem.sys), you should use the following form for the `device` command:

```
device = emm386.exe ram nnnn
```

where *nnnn* is the amount in kilobytes of EMS to be allocated. Borland recommends that you use a value greater than or equal to 800K.

If you have a version of TD386 (the 80386 debugging kernel) that is earlier than Version 2.51, you might not be able to use it with DOS 5.0. If you have Turbo Debugger 2.0 or a more recent version, you can download an updated TD386 file from most of the online services or direct from Borland's own BBS at (408) 439-9181. Look for the file named TD386.ZIP. After downloading it, you must unzip it using either the unzip.exe utility in the \borlandc\bin directory or by using the popular decompression utility pkunzip.exe.

If you have a version of Turbo Debugger older than Version 2.0, then you must contact Borland for an upgrade. If you encounter other problems, particularly with TD386 2.51, Borland recommends configuring your system so that you do not use `dos=high` or emm386 in your config.sys.

If you encounter difficulties using the MAKE utility, be sure to delete the `.SWAP` directive in your MAKE command files. This appears to keep MAKE from hanging under DOS 5.0.

USING PROGRAMMING UTILITIES

Years ago, all you needed to write a PC program was a text editor, an assembler, and a link program. (You still can write PC software that way if you want to.) But then along came compilers and high-level programming languages. With high-level programming came additional features and new tools to manage those new features. With Borland C++, you get three high-level programming languages: assembler language (through either Turbo Assembler or built-in assembly language), C, and C++. You also get the enhanced features of those languages, such as macros and `#include` files, project and configuration files, and the need to manage ever-growing software projects. Consequently, a wide variety of utility programs have been created to help you create more powerful projects.

Using CPP to show macro definitions converted to C code

Managing the compilation and linking of large projects with MAKE

Locating specific files with GREP and other text- and file-searching utilities

This chapter looks at the utility programs provided by Borland and a few freeware and software utilities that can help make your programming time a bit more productive. Here are the three most important utilities:

- CPP is the C preprocessor that shows exactly how your macro definitions look as soon as they are converted to C code.
- MAKE is an essential tool for managing the compilation and linking of large projects. MAKE is similar to the IDE's built-in project manager but is designed for use with the BCC command-line compiler and other tools, such as Turbo Assembler and the RC resource compiler.
- GREP and other text- and filename-searching utilities help you locate specific files quickly. I have more than 4,000 files in about 200 separate directories. I lose my files all the time, and it helps to use one of these utilities to find them fast.

Other utility programs help you insert international characters into source code when you are missing the appropriate keys on your PC keyboard, perform various conversions on project and configuration files, produce reports of the contents of object and library files, and perform other useful operations. Freeware and shareware utilities also are introduced in this chapter, including the LZEXE.EXE executable file compressor, 4PRINT (a terrific shareware print utility), and several others.

CPP

CPP is a C (or C++) preprocessor that enables you to see the effects of expanded macro definitions and include files. You use CPP just like you would use BCC, but instead of producing a compiled object module or program, CPP produces a text file containing the *preprocessed* program source. CPP is provided in the Borland C++ package and is located in the \borlandc\bin directory.

When the C language was first defined, most compilers operated by making multiple passes over the program source code, each time gathering information that would be used in converting the source into machine code. Near the end of this multistep process, the compiler would issue the completed object module. As a consequence of the way these early compilers operated, the C inventors envisioned a first-pass step known as the C preprocessor (hence the

name CPP), whose primary purpose was to expand `#include` references so that the main source file would then incorporate the text brought in from outside files, and to process `#define` macros, conditional statements, and `#pragma` compiler directives. At the end of this preprocessing step, the program would consist of pure C statements with no `#include`, `#define`, or conditional statements. All that would be left was the actual code to be compiled.

The Borland compilers are single-pass compilers, which means that they do not have a separate preprocessing step like traditional compilers. To provide a preprocessed source listing, Borland wrote the CPP utility. CPP scans your source, fully expanding each `#include` and macro reference. You can use the resulting output file to better understand how the compiler has interpreted your directives. In particular, there are many times when writing macro definitions that the macro expansion produces code different from what you were expecting. CPP is especially useful in helping you uncover and resolve these types of problems.

USING CPP

You operate CPP the same as the BCC command-line compiler. You enter the same command-line switches and filename specifications. The default mode of operation processes the input files and produces an output text file where each line is prefaced with both the source filename and a line number. The output is written to a file whose name is constructed from the module or project name and which ends in an .i extension. Preprocessing a program named begin.c produces a file named begin.i.

Optionally, you may disable the source and line number information by adding the `-P` command-line switch when you invoke CPP. The resulting output text file then will contain only preprocessed program source. If you want to, you can run the resulting text file (only when `-P` is used to strip the line numbers) through the BCC compiler to produce an executable program. When you look through the output file, you will notice that there are no comments, because they are removed by the preprocessor. What might look especially unusual is the presence of many blank lines. These blank lines occur when large block comments are deleted or macro definitions and conditional statements are evaluated and removed.

Listing 3.1 shows a sample program named democpp.c. Listing 3.2 shows democpp.i, the output file produced after preprocessing democpp.c using CPP. Note the expansion of the `#include` statement, and note also the elimination of the manifest constant macros, the macro `sumit`, and the conditional compilation directives. In Listing 3.2, large blocks of blank lines and portions of the stdio.h expansion have been removed in order to save space.

LISTING 3.1. DEMOCPP.C BEFORE PREPROCESSING WITH CPP.

```
 1 /* DEMOCPP.C
 2    Demonstrates the usage of the CPP preprocessor.
 3 */
 4 #include <stdio.h>
 5
 6 #define prompt1 "Good, Morning"
 7 #define prompt2 "Good, Afternoon"
 8
 9 /* Use this for conditional compilation */
10 #define useprompt1 1
11
12 /* Illustrate macro expansion */
13 #define sumit(a,b,c) ( (a) + (b) + (c) )
14
15 void main( void )
16 {
17 #ifdef useprompt1
18   printf("%s\n", prompt1);
19 #else
20   printf("%s\n", prompt2);
21 #endif
22
23   printf("The sum of 7 + 9 + 11 is: %d\n", sumit(7,9,11) );
24
25 }
```

LISTING 3.2. DEMOCPP.I SHOWING THE EFFECTS OF PREPROCESSING DEMOCPP.C USING CPP.

```
democpp.c 1:
democpp.c 2:
democpp.c 3:
democpp.c 4:
C:\BC3\INCLUDE\stdio.h 1:
C:\BC3\INCLUDE\stdio.h 2:
```

[Blank lines deleted for clarity.]

```
C:\BC3\INCLUDE\_defs.h 1:
C:\BC3\INCLUDE\_defs.h 2:
C:\BC3\INCLUDE\_defs.h 3:
```

[Blank lines deleted for clarity.]

```
C:\BC3\INCLUDE\_defs.h 105:
C:\BC3\INCLUDE\_defs.h 106:
C:\BC3\INCLUDE\stdio.h 14:
C:\BC3\INCLUDE\stdio.h 15:
C:\BC3\INCLUDE\stdio.h 16:
C:\BC3\INCLUDE\stdio.h 17:
C:\BC3\INCLUDE\_nfile.h 1:
C:\BC3\INCLUDE\_nfile.h 2:
```

[Blank lines deleted for clarity.]

```
C:\BC3\INCLUDE\_nfile.h 16:
C:\BC3\INCLUDE\stdio.h 18:
C:\BC3\INCLUDE\stdio.h 19:
C:\BC3\INCLUDE\stdio.h 20:
C:\BC3\INCLUDE\stdio.h 21:
C:\BC3\INCLUDE\_null.h 1:
C:\BC3\INCLUDE\_null.h 2:
```

[Blank lines deleted for clarity.]

```
C:\BC3\INCLUDE\_null.h 15:
C:\BC3\INCLUDE\_null.h 16:
C:\BC3\INCLUDE\_null.h 17:
C:\BC3\INCLUDE\stdio.h 22:
C:\BC3\INCLUDE\stdio.h 23:
C:\BC3\INCLUDE\stdio.h 24:
C:\BC3\INCLUDE\stdio.h 25:
C:\BC3\INCLUDE\stdio.h 26: typedef unsigned size_t;
C:\BC3\INCLUDE\stdio.h 27:
C:\BC3\INCLUDE\stdio.h 28:
C:\BC3\INCLUDE\stdio.h 29:
C:\BC3\INCLUDE\stdio.h 30:
C:\BC3\INCLUDE\stdio.h 31: typedef long    fpos_t;
C:\BC3\INCLUDE\stdio.h 32:
C:\BC3\INCLUDE\stdio.h 33:
C:\BC3\INCLUDE\stdio.h 34:
C:\BC3\INCLUDE\stdio.h 35:
C:\BC3\INCLUDE\stdio.h 36: typedef struct  {
C:\BC3\INCLUDE\stdio.h 37: int             level;
C:\BC3\INCLUDE\stdio.h 38: unsigned        flags;
C:\BC3\INCLUDE\stdio.h 39: char            fd;
C:\BC3\INCLUDE\stdio.h 40: unsigned char   hold;
```

continues

LISTING 3.2. CONTINUED

```
C:\BC3\INCLUDE\stdio.h 41: int             bsize;
C:\BC3\INCLUDE\stdio.h 42: unsigned char    *buffer;
C:\BC3\INCLUDE\stdio.h 43: unsigned char    *curp;
C:\BC3\INCLUDE\stdio.h 44: unsigned        istemp;
C:\BC3\INCLUDE\stdio.h 45: short           token;
C:\BC3\INCLUDE\stdio.h 46: }       FILE;
```

[Blank lines deleted for clarity.]

```
C:\BC3\INCLUDE\stdio.h 47:
C:\BC3\INCLUDE\stdio.h 48:
```

[Blank lines deleted for clarity.]

```
C:\BC3\INCLUDE\stdio.h 106:
C:\BC3\INCLUDE\stdio.h 107: extern  FILE    __cdecl _streams[];
C:\BC3\INCLUDE\stdio.h 108: extern  unsigned    __cdecl _nfile;
C:\BC3\INCLUDE\stdio.h 109:
C:\BC3\INCLUDE\stdio.h 110:
```

[Approximately 240 lines deleted here, including nested `includeS` and other definitions brought in by stdio.h.]

```
C:\BC3\INCLUDE\stdio.h 249:
C:\BC3\INCLUDE\stdio.h 250:
democpp.c 5:
democpp.c 6:
democpp.c 7:
democpp.c 8:
democpp.c 9:
democpp.c 10:
democpp.c 11:
democpp.c 12:
democpp.c 13:
democpp.c 14:
democpp.c 15: void main( void )
democpp.c 16: {
democpp.c 17:
democpp.c 18: printf("%s\n", "Good, Morning");
democpp.c 19:
democpp.c 20:
democpp.c 21:
democpp.c 22:
democpp.c 23: printf("The sum of 7 + 9 + 11 is: %d\n", ( (7) + (9) + (11) ) );
democpp.c 24:
democpp.c 25: }
democpp.c 26:
```

MAKE

The IDE includes a built-in project make facility that automates the task of keeping track of which modules need recompiling. But when you use the stand-alone command-line compiler BCC, you must use MAKE to access the project make facilities. MAKE is a stand-alone utility for which you must prepare a separate *make file* describing the dependencies between the files that cause them to be recompiled or reassembled. MAKE is provided in the Borland C++ package and installed in the \borlandc\bin directory. MAKE is a protected-mode application. Borland also provides MAKER.EXE, which is identical to MAKE except that it runs in real mode only. If you can, use the protected-mode version of MAKE, because it can process much larger projects than MAKER.

As soon as a make file is created to describe the requirements of your application, MAKE uses the make file to check and compare the date and time stamps assigned by DOS to each file. If any source file is newer than its corresponding object file, MAKE ensures that each object file is recompiled or reassembled to incorporate the latest changes. The make file describes the dependencies between the source files and the object modules, and specifies what commands should be issued in order to recompile or reassemble the source or to access the resource compiler.

If you are using BCC and currently are recompiling all your source files each time you make simple changes, you can save yourself a great deal of time by learning how to use MAKE. MAKE is not difficult to use, especially for building fairly straightforward projects. MAKE does, however, include a large variety of options and capabilities. Fortunately, you can ignore most of these options as you prepare to create your first make files. As you'll see in this section, learning to use MAKE's basic functionality is quite straightforward—only the advanced features get a bit messy. If you want to, you can convert project files created within the IDE into make command files (see the section "PRJ2MAK" later in this chapter).

SAMPLE USE OF MAKE

The make file consists of a sequence of specially formatted commands stored in a command file. You can prepare a make file using your own editor or the IDE editor. Save the make file information to a file having a .mak extension. By

default, when MAKE is executed, it looks for a file called makefile.mak. Optionally, you may specify a particular make file using the `-f` command-line option. For example,

```
MAKE -Fshell.mak
```

or

```
MAKE -Fshell
```

The latter example works because MAKE uses the default file extension of .mak. Additional command-line options are available; see the "MAKE Command-Line Options" section in this chapter.

The basic structure of the MAKE file consists of *dependency statements*, followed by commands. Each dependency statement specifies a target or destination file, a colon character (:), and a list of source files (or object files) from which the destination file is constructed. MAKE checks each of the source files, and if any of the source files are newer than an existing copy of the destination file, MAKE executes the subsequent command lines to bring the destination up-to-date.

Listing 3.3 shows a sample MAKE file for creating shell.exe, a simple C program that is dependent on four modules named shell, utility, dirlist, and menuunit.

LISTING 3.3. A SAMPLE SHELL.MAK FILE.

```
# SHELL program MAKE file
#
# SHELL is dependent on four object modules
#

shell.exe: shell.obj \
     utility.obj \
     dirlist.obj \
     menuunit.obj
  bcc shell.obj utility.obj dirlist.obj menuunit.obj
# Note use of bcc to do the linking; it's much easier than using tlink directly

shell.obj: shell.c utility.h dirlist.h menuunit.h
  bcc -c -ms shell.c

utility.obj: utility.c utility.h
  bcc -c -ms utility.c
```

```
dirlist.obj: dirlist.c utility.h
  bcc -c -ms dirlist.c

menuunit.obj: menuunit.c utility.h
  bcc -c -ms menuunit.c
```

The first few statements, preceded by the # symbol, are comments. Comments can appear anywhere in the file. Lines beginning in the first column, other than comments, specify dependent relationships. For example, the statement

```
shell.exe: shell.obj \
     utility.obj \
     dirlist.obj \
     menuunit.obj
```

says that shell.exe is dependent on the four .obj files—shell.obj, utility.obj, dirlist.obj, and menuunit.obj. Wildcard filenames are not permitted in the make file (see the section "Implicit Rules").

Each dependency statement is a conditional test, instructing MAKE to compare the date and time stamps of the source or input files to the destination file. If changes have occurred to the source files, then and only then does MAKE rebuild the destination file using the command lines that follow the dependency statement. In this example, if any of the module files are newer than the existing shell.exe file, shell.exe is rebuilt to incorporate the latest changes by executing

```
bcc shell.obj utility.obj dirlist.obj menuunit.obj
```

The backslash character (\) at the end of a line is a line continuation marker. Lines ending in a backslash are continued to the next statement in the make file. In this particular instance, the filenames could easily fit on one line, so the use of the backslash here is for illustration only. If a line must end in a backslash, you can override its use as a line continuation symbol by placing two backslashes together, as in this macro definition (using macro is explained later):

```
source=C:\TP\TV\SRC\\
```

Following the dependent condition, on a line beginning with one or more spaces or tab characters is a command that MAKE should execute if the preceding conditional test requires the destination file to be rebuilt. In the MAKE command line

```
bcc shell.obj utility.obj dirlist.obj menuunit.obj
```

the compiler is invoked to link the respective modules. (Using BCC to call tlink is easier than attempting to correctly specify all the tlink parameters yourself). Any number of commands may be executed as a consequence of a dependent test, provided that each command is indented by at least one space or tab. The next statement starting in the first column of the make file is interpreted as a new dependency condition.

When using BCC to compile individual modules, note the use of the `-c` compiler command-line switch:

```
dirlist.obj: dirlist.c utility.h

  bcc -c -ms dirlist.c
```

The `-c` switch tells the compiler to compile and create an .obj object module only. Without the `-c` switch, the compiler will create the .obj and then call the linker. The link likely will fail because the other needed .obj modules have not yet been created.

Command lines may invoke any .com, .exe, or .bat file and may also use any DOS command. As such, the make file can do much more than merely invoke compilers and assemblers. MAKE becomes a general-purpose automation utility that can perform functions such as copying source code to back up subdirectories when changes are made to the program.

Because most MAKE files contain a large number of dependencies, MAKE initially scans through the entire MAKE file, identifying which files are dependent on other source files. If any of the source files are themselves listed as destination files in other dependency relationships shown elsewhere in the file, MAKE ensures that those files are brought up-to-date first. In the sample shell.mak file shown earlier, because each .obj file is listed in its own dependency statement, MAKE ensures that the necessary commands to bring the object modules up-to-date are executed before linking shell.exe.

A COMMON PROBLEM

A common problem encountered by all users of MAKE, sooner or later, occurs when the system clock is changed, intentionally or unintentionally. MAKE compares the date and time of the destination file to the source files, so if the clock has changed so that an incorrect date or time is associated with a source file, strange problems can crop up. Clock changes can occur intentionally, as when you manually set the system time. Changes also can occur unintentionally, for example, when you run a program that fiddles with the clock, when the clock's battery runs low, or as a consequence of serious software errors.

If you are using MAKE to build a large application with a large number of modules, it is easy not to notice if an incorrect time stamp has been placed on a file. As a result, you might make a project and find that no matter how hard you try, your latest changes do not show up in the resulting .exe file. Thinking that your code is wrong, you'll probably keep rewriting the errant section over and over. Then, suddenly, you'll notice that the system clock has been reset, the file's time stamp is incorrect, and none of your changes made it into the .exe file.

If your changes do not appear to be included in a successful MAKE, be sure to carefully examine the time and date stamps on the files. If needed, use the TOUCH utility, described later in this chapter, to update the time and date stamps.

EXPLICIT RULES

MAKE provides two kinds of dependent relationship statements: *explicit rules* and *implicit rules*.

An explicit rule lists the file to be created (also known as a *target file*) and the source files that it depends on, in the format

```
destination file: source file 1 source file 2 . . .
   command
. . .
```

For example,

```
utility.obj: utility.c utility.h
  bcc -c utility.c
```

Here, utility.obj is the resulting file, and it is dependent on utility.c and utility.h. If any of the dependent files have changed since the last time utility.obj was built, MAKE issues the following command:

```
bcc -c utility.c
```

to rebuild utility.obj.

Any number of commands may follow the dependency relationship, provided that each is indented by at least one space or tab character. At least one of the commands should construct a new destination file.

If a dependent relationship contains a destination file only, and no source files, as in

```
shell.exe:
  bcc shell.c
```

the command line is directly executed.

LIST ALL FILES

When you build the make file, it is important to ensure that every dependency for each file is completely specified. For example, consider this dependency relationship:

```
shell.exe: shell.obj \
     utility.obj \
     dirlist.obj \
     menuunit.obj
  bcc shell.obj utility.obj dirlist.obj menuunit.obj
```

If utility.obj had not been specified elsewhere in a dependency relationship and utility.obj had not been recompiled, the linkage of shell.exe would have failed.

For this reason, you should check the dependent relationship statements carefully to ensure that the MAKE file can successfully build the project. To test this, delete all .OBJ files and then run MAKE. If any dependencies are missing, MAKE should stop with a compile or link error. If the dependencies are listed correctly, modules that are required by other modules will be compiled or assembled before compiling the module that uses them is compiled.

COMMAND LINES

Each command line or group of command lines follows a dependency statement and is indented by at least one blank or tab character. Normally, MAKE displays each command as it is executed. If the command is prefixed with the `@` symbol, however, MAKE does not display the command when it is executed.

If a command executed from MAKE returns a nonzero exit code, MAKE normally aborts the make file. You can restrict the abort process by prefacing each command line with a hyphen (`-`) followed by an optional number. If no number is specified, MAKE ignores all exit codes. If a number follows the hyphen, then only when the exit code equals the specified number does MAKE abort the make file. For example, this command line aborts only when the exit code equals 1:

```
-1 bcc shell.c
```

IMPLICIT RULES

Implicit rules provide a way of specifying wildcards for filenames that need compiling or assembling. The syntax for an implicit rule references the file extensions rather than complete target and source filenames. The implicit rule

```
.c.obj:
  bcc -c $<
```

means that all .obj files are created from .c files having the same filename. For example, utility.obj is created from utility.c. By placing this implicit rule into the make file, you do not need to specify each of the units as separate dependencies. Here's the resulting make file for shell.exe:

```
shell.exe: shell.obj \
     utility.obj \
     dirlist.obj \
     menuunit.obj
  bcc shell.obj utility.obj dirlist.obj menuunit.obj

.c.obj:
  bcc -c $<
```

The symbol `$<` is a special macro symbol that is defined in the section "Macros" later in this chapter.

DIRECTIVES

Borland's MAKE program includes a large number of directives that control the processing of the make file. Table 3.1 shows the conditional directives. You use *conditional directives* the same way you use C's conditional compilation instructions (such as `#if`). Additional directives (called *dot directives*) control the execution of the MAKE program. These are shown in Table 3.2. You can use constant symbols in expressions. Table 3.3 shows the arithmetic operators that may be used in expressions.

TABLE 3.1. MAKE'S CONDITIONAL DIRECTIVES.

Directive	*Description*

`!if`	A conditional statement adding flexibility to the make file's design. `!if` has the following syntax:
	`!if expression`
	`  make file lines`
	`!endif`
	When an `else` part is added, `!if` takes the form
	`!if expression`
	`  make file lines`
	`!else`
	`  make file lines`
	`!endif`
	The `else` part may be extended indefinitely into an `if-then-else-if` statement type by using the `!elif` keyword:
	`!if expression`
	`  make file lines`
	`!elif expression`
	`  make file lines`
	`!endif`

Directive	*Description*
	The `expression` may reference macro symbols and symbols defined on the MAKE command line using the `-D` option (see Table 3.5), as well as constant values and basic arithmetic operators (see Table 3.5). Constants may be written in decimal, octal, or hexadecimal. Any value beginning with a 0 is treated as octal, unless it begins with `0x`, in which case it is treated as a hexadecimal constant.
`!error text`	Outputs the value of `text` to the display and halts the make file.
`!undef symbol`	Causes the definition for `symbol` to go away. Example: `!undef UNITDIRECTORY`
`!include filename`	Incorporates the contents of `filename` into the current make file, where `filename` is a string surrounded by angle brackets or double quotes (`<DIR.MAK>` or `"DIR.MAK"`). Included files may be nested if you avoid recursive includes.

TABLE 3.2. DOT DIRECTIVES USED TO CONTROL THE EXECUTION OF MAKE. DOT DIRECTIVES OVERRIDE ANY SETTINGS PLACED ON THE MAKE COMMAND LINE.

Dot Directive	*Description*
`.autodepend/.noautodepend`	Use `.autodepend` to turn on autodependency checking and `.noautodepend` to turn it off. When it is on, MAKE can automatically determine whether the included files (.h files) used to build an .obj have changed since the time of the .obj's compilation. The Borland compilers store inside the .obj file the names

continues

TABLE 3.2. CONTINUED

Directive	*Description*
	of the files that were used to create the .obj (including nested `#includes`). Autodependency checking is quite impressive. Consider the shell.mak file shown in Listing 3.2. Assume that dirlist.h has been updated. With autodependency checking on, MAKE automatically determines that all files that include dirlist.h must be recompiled.
`.ignore/.noignore`	When `.ignore` is encountered, MAKE ignores the return values from commands that have been executed. To enable recognition of return values, use `.noignore`.
`.path.ext`	Used to specify the search path for files having the extension `.ext`. For example, `.path.c = c:\source` is an instruction to look in the c:\source directory for all files ending in `.c`. You can set up multiple `.path` statements for each extension that is used in your make file.
`.precious`	Used as a prefix to an explicit rule, like this: `.precious shell.exe: shell.obj dirlist.obj ...` `.precious` tells MAKE that the target file (here shown as shell.exe) is *precious* and should not be deleted if an error occurs when creating the target. This is an especially important dot directive if the target is an existing .lib file to which you want to add a modified .obj file.
`.silent/.nosilent`	With `.nosilent` in effect, MAKE prints each command immediately before executing the command. When `.silent` is in effect, MAKE does not display each command.

Directive	*Description*
`.swap/.noswap`	With `.swap` in effect, MAKE swaps most of itself out of memory when executing commands. With `.noswap` in effect, MAKE remains resident.

`.suffixes`	Use `.suffixes` to sort out which file should be used when one filename extension appears in multiple implicit rules. Remember that an implicit rule matches a source file extension to a destination file. This example causes all .obj files to be manufactured from .c files:
	`.c.obj:`
	`bcc -c $<`
	When you have multiple source file extensions that all produce the same output file (such as .asm, .c, and .cpp, which all compile to .obj files), use `.suffixes` to specify a priority list to the matching scheme. `.suffixes` has a form as illustrated in this example:
	`.suffixes: .c .cpp .asm`
	This means that for a specified filename, such as srcfile, it should first look for srcfile.c before looking for srcfile.cpp and srcfile.asm.

TABLE 3.3. MAKE EXPRESSION OPERATORS.

Operator	*Description*
-	Unary negation
~	Unary bitwise complement
!	Unary logical not
+	Addition
-	Subtraction
*	Multiplication

continues

TABLE 3.3. CONTINUED

Operator	*Description*
`/`	Division
`%`	Remainder
`>>`	Right shift
`<<`	Left shift
`&`	Bitwise AND
`¦`	Bitwise OR
`^`	Bitwise exclusive OR
`&&`	Logical AND
`¦¦`	Logical OR
`>`	Greater than
`=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`==`	Exactly equal
`!=`	Not equal
`( )`	Parentheses may group expression elements.
`? x : y`	If the value before `?` is zero, the result is `y`; otherwise, the result is `x`.

USING BUILTINS.MAK

If you use a variety of make files and each uses the same symbols or dependent rule definitions, you can store all the common items in a file called builtins.mak. Each time MAKE is run, it will attempt to open and read this file before executing your make file. Consequently, you can use builtins.mak to automatically share common symbol and relationship statements across several make files.

BATCHING

As mentioned in Chapter 2, "Power Programming Using the IDE", it is more efficient to use a single BCC command line to compile several source files than to call BCC repeatedly for each source file. When you write a macro command statement, you can batch your commands together by enclosing the parameters to the command within braces, like this:

```
BCC {source1.c}
BCC {source2.c}
BCC {source3.c}
```

Upon encountering a sequence of statements like this, MAKE will recognize that the all three source files can be combined into a single BCC command line, producing

```
BCC source1.c source2.c source3.c
```

This feature is especially useful when you are using implicit rules, because an implicit rule may translate into multiple commands. Rather than writing the implicit rule like this:

```
.c.obj:
  bcc -c $<
```

use this format:

```
.c.obj:
  bcc -c {$<}
```

MACROS

Macros provide text substitution by inserting a special symbol into the make file that is translated, when used, into actual text parameters. Macro symbols are assigned at the beginning of the make file by writing the symbol name, followed by an equal sign (=), followed by the substitution text. For example,

```
source=C:\TP\TV
```

Consider the situation of sharing a single make file among a team of software developers. Each team member may want to store some of the source and object files in different directories than those used by other team members. Without macro symbols, each team member needs to edit the make file and change each subdirectory name to his or her subdirectory. With macro symbols, the problem

is much easier to resolve. Instead, the macro file uses the first few lines to define macro symbols that are set equal to the subdirectory names. Each user changes only the symbol definitions, not the entire file. For example, in the following make file, the two symbols `source` and `objects` specify their respective subdirectories:

```
source=C:\TP\TV\SRC\\
objects=C:\TP\TV\OBJS\\
$(objects)shell.exe: $(objects)shell.obj \
     $(objects)utility.obj \
     $(objects)dirlist.obj \
     $(objects)menuunit.obj
  bcc $(objects)shell.obj\
      $(objects)utility.obj $(objects)dirlist.obj\
      $(objects)menuunit.obj

$(objects)shell.obj: $(source)shell.c \
     $(source)utility.h $(source)dirlist.h $(source)menuunit.h
  bcc -c -ms shell.c
```

Macro symbols may be defined or redefined anywhere in the make file. When a symbol is redefined, the old definition is thrown out. A set of predefined macros, shown in Table 3.4, is available. The macros that return a filename usually return the dependent filename when they are used in an implicit rule, or the target filename when they are used in an explicit rule. Be aware that you can use MAKE without having to use all these funny macro symbols and expressions. These extra features are provided to create remarkably powerful file updating routines, but you probably can manage to use MAKE without worrying about so many details.

TABLE 3.4. MAKE'S CONDITIONAL DIRECTIVES.

Macro symbol	*Purpose*	*Example*
`$d`	Defined?	`$d` determines whether a macro symbol is defined or undefined. For example, if you type `$d(units)`, `$d` returns 1 if the symbol `units` is defined or 0 if `units` is undefined. As such, `$d` may be used in conditional `!if` macro statements.

Macro symbol	*Purpose*	*Example*

`$*`	Returns full filename, including path but no extension	`$*` returns the filename part of a `filename.ext` and often is used in both explicit and implicit rules. For example:
		`.c.obj:`
		`bcc -c $*`
		where the filenames are dirlist.obj, utility.obj, and menuunit.obj, translates into
		`bcc -c dirlist.c`
		`bcc -c utilit.y`
		`bcc -c menunit.c`

`$<`	Returns full filename, including path and extension	`$<` returns the full filename, including extension and leading drive and directory information.
`$:`	Returns path only	`$:` returns the path name minus the filename. For example, given the filename `c:\tp\tv\src\dirlist.c`, `$:` returns `c:\tp\tv\src`.
`$.`	Returns filename and extension only	Returns the filename and extension, minus the leading drive and directory information.
`$&`	Filename only, without path or extension	Translates `c:\tp\tv\src\dirlist.c` to `dirlist`.
`$@`	Returns full target filename with path	Returns the complete filename including path.

continues

TABLE 3.4. CONTINUED

Macro symbol	*Purpose*	*Example*
`$**`	Returns full dependent filename including path	In explicit rules, this macro returns a string containing all the dependent files (to the right of the `:` in the rule definition).
`$?`	Returns full dependent filename including path	In explicit rules, this macro returns a string containing all the dependent files that are out-of-date.
`$(macroD or F or B or R)`		Use this special macro modifier to split any filename returned by any of the other macros. Use `D` to extract the drive and directory name; `F` to extract the `filename.extension`; `B` to return the filename only; or `R` to extract the drive, directory, and filename (without an extension).
`__MSDOS__`	Predefined constant	Returns 1 if MAKE is currently running on MS-DOS. You can only guess that it will return other values as soon as MAKE works on OS/2, UNIX, or other operating systems.
`__MAKE__`	Predefined constant	Returns the version number of the MAKE program that is executing, as a hexadecimal string.
`MAKE`	Predefined constant	Contains the name of the MAKE program (usually make.exe).
`MAKEFLAGS`	Predefined constant	Returns the options that were specified on the make command line.
`MAKEDIR`	Predefined constant	The subdirectory name where make was executed from.

MAKE COMMAND-LINE OPTIONS

You run MAKE by typing MAKE at the DOS command line, followed by options and target filenames. MAKE has the following command syntax:

```
MAKE options target_filenames
```

If you type MAKE by itself (without parameters), MAKE looks for a make file having the name makefile.mak and uses it if it is found. To specify a different make file, use the `-f` option (see Table 3.5). The default extension for a make filename is .mak.

Options are specified at the start of the parameter list, prefaced with a single hyphen character:

```
MAKE -Dsource=C:\TP\TV -fshell
```

Multiple options may be specified by placing them next to one another, such as

```
MAKE -BSFshell
```

Table 3.5 shows the various MAKE options. The option letters are case-sensitive; use the appropriate case as shown in the table. For example, `-B` is a valid command but `-b` is not. Each option letter may be followed by either a `+` to enable the option or a `-` to disable the option. Normally, you do not need to specify the `+` symbol, which is the default value, unless the option has been disabled and stored as a new default value.

TABLE 3.5. MAKE COMMAND-LINE SWITCH OPTIONS.

Option	*Description*
`-a`	Automatically checks .obj files against their `#include` files. See `.autodepend` in Table 3.2.
`-B`	Builds the entire application by executing the commands for every target destination file. Use `-B` to ensure that all files are recompiled or reassembled, regardless of the last edit date and time.
`-ddirectory`	When `-S` is selected, *`directory`* is where MAKE outputs its swap file.
`-Dsymbol`	Defines *`symbol`*.

continues

TABLE 3.5. CONTINUED

Option	*Description*
`-Dsymbol=string`	Defines *`symbol`* and equates it to *`string`*.
`-e`	If a macro symbol is already defined as a DOS environment variable, MAKE uses the environment variable instead of the macro symbol. Any new definitions for that symbol inside the make file are ignored.
`-ffilename`	Specifies that *`filename`*`.MAK` should be used as the make command file.
`-i`	Causes MAKE to ignore all errors returned by executed command lines.
`-Idirectoryname`	Sets *`directoryname`* as the default directory for make `!include` files.
`-K`	Causes all temporary files created by MAKE to be kept rather than deleted by default.
`-m`	With `-m` selected, the date and time of each file accessed by MAKE is displayed.
`-n`	Displays the commands that normally would be executed, but does not actually execute the commands.
`-N`	Use `-N` only when you must attempt to use a make file that originally was created for use with Microsoft. When `-N` is in effect, MAKE interprets the make file text slightly differently to provide a degree of compatibility with Microsoft make files.
`-r`	Ignores all rules that may be defined in builtins.mak.
`-S`	Swaps MAKE out of memory, providing additional memory for command execution.
`-s`	Causes MAKE to run in “silent” mode, providing no display of commands as they are executed.

Option	*Description*
`-U`*`symbol`*	Undefines *`symbol`*.
`-W`	Sets the current options to become the new defaults for MAKE. The options are stored inside the make.exe file.
`-?` or `-h`	Displays a help message showing a list of command-line options.

PRJ2MAK

If you want to use the MAKE program but find the MAKE command language daunting, you can instead create project files using the IDE and then convert them to make-file format using the PRJ2MAK utility program. PRJ2MAK also is helpful if you decide to switch from using the IDE to using the BCC command-line compiler. The Borland C++ package includes PRJ2MAK, and normally it is installed in the \borlandc\bin directory.

Project files are created using the Project menu in the IDE. Use the Project menu to create new projects or to edit existing project files. When you are ready to convert a project file into make file format, issue the command

```
PRJ2MAK project.prj makefile.mak config.cfg
```

Substitute the name of your project file for *`project.prj`* and the name of the desired make file for *`makefile.mak`*. If no make file is specified, the default of *`project.mak`* is used, where `project` is the filename portion of the *`project.prj`* filename. The extensions .prj and .mak are assumed if they are omitted from the filenames. When no configuration filename is specified, a default name created from *`project.cfg`* is used instead. The configuration file is used by BCC to establish default command-line switch settings. When you invoke BCC, you can request a specific configuration file by using the `+` switch, as in this example:

```
BCC +settings.cfg file1.c
```

The created .mak file will automatically reference the configuration file created by PRJ2MAK. Otherwise, the compiler's normal mode of operation is to look for a default turboc.cfg in the current directory. If no turboc.cfg file is found, BCC then looks in the directory where BCC is located.

TOUCH

TOUCH updates the date and time stamp of a file, resetting to the current date and time. For example,

```
touch file.exe
```

updates the date and time associated with the file. The filename may be replaced with a wildcard specification for updating a group of files all at once.

TOUCH is most often used in conjunction with the MAKE utility. MAKE checks the date and time of each file used to build an application. MAKE ensures that the newest version of each file is incorporated into the final application, compiling or assembling the source files if needed. When source files are touched, their date and time stamps are made newer than any existing .obj files, forcing the source files to be recompiled or reassembled.

FILE SEARCHING UTILITIES

File searching comes in two flavors: searching for a file having a certain name, or searching for files that contain a certain text pattern inside the file. In the first case, third-party freeware and shareware command-line utilities are available to scan through your directory structure, locating all filenames that match your filename search pattern. Because I have nearly 4,000 files on my hard drive, I find file searching utilities indispensable.

In the second case, Borland C++ includes a pattern searching utility named GREP (the utility and its unusual name originated in the UNIX world), and MS-DOS 5.0 and DR DOS 6.0 include their FIND utilities. The DR DOS 6.0 FIND is a full-featured search utility, and the MS-DOS 5.0 FIND is, quite frankly, almost useless for the purposes described in this section. A number of third-party freeware and shareware utilities also provide high-speed search capabilities, including both search and replace functions. These utilities enable you to rapidly scan through a large number of source files and perform functions such as renaming a variable in all locations where it is referenced. See the section, “Third-Party Search and Replace Tools.”

USING GREP

GREP is a search utility to scan though disk files looking for strings that match a specified search pattern. GREP is run from the DOS command line, specifying the search pattern, search options, and the files on which the search should be conducted.

The format for a GREP command is

```
GREP options search_string file_specification
```

In its simplest and most common usage, GREP is used to search for a specific string in a group of files. A typical search request might be

```
GREP "ThisLevel" *.pas
```

In response, GREP scans all files in the current subdirectory matching the *.pas file specification, producing output showing the names of the files that contain the string, followed by each line containing a match. Here is an example of output when GREP is used to search through a set of Turbo Pascal source files:

```
File SHELL.PAS:
     ThisLevel : Integer;
     ThisLevel := CursorEntry^.Level;
       (ThisLevel <= CursorEntry^.Level) do
       SubLevel := Remove_ThisLevel( EntryAddr );
File TVSHELL8.PAS:
     function  Remove_ThisLevel ( StartEntry : Integer ) : Integer;
     ThisLevel : Integer;
     ThisLevel := AnEntry^.Level;
```

By default, the output is displayed on the screen. But if you use DOS redirection, the output may be sent to a file or to a printer. For example, to send the output to a file named patterns.txt, issue the command

```
GREP "ThisLevel" *.c >patterns.txt
```

To abort a search, press Ctrl-Break.

The options, if used, select output formats or customize the search. Options are placed before the search string, prefaced with a single hyphen character:

```
GREP -c "ThisLevel" *.c
```

When the `c` option is selected, GREP counts the number of matches found, displaying the result for each file scanned. Multiple options are placed next to one another, for example:

```
GREP -ci "ThisLevel" *.c
```

Here, the letter `i` means to ignore case when making comparisons. The option letters, shown in Table 3.6, may be followed by either a `+` to enable the option or a `-` to disable the option. Normally, you do not need to specify the `+` symbol, because `+` is the standard default value.

TABLE 3.6. GREP COMMAND-LINE OPTIONS.

Option	*Description*
`-c`	GREP counts the number of pattern matches found and displays the total for each file.
`-d`	When `d` is selected, GREP also searches all subdirectories in the current path.
`-i`	Normally, GREP matches strings exactly, requiring the same case in the search string as in the matched text. By selecting `i`, GREP ignores case and makes comparisons as if the search pattern and the matched text are the same case.
`-l`	Displays the names of the files containing the pattern, and nothing else, unless selected with other options.
`-n`	Displays line numbers next to each line containing a matched item.
`-o`	Selects UNIX-compatible output format, which displays the filename before each matching text line.
`-r`	Specifies that the search string contains a *regular expression*. Regular expressions provide a detailed specification of the search pattern by embedding special symbols into the pattern. The special symbols are described in Table 3.7.
`-u`	GREP's defaults are set to the current options and stored in the GREP.COM file.

Option	*Description*
`-v`	Displays all nonmatching lines.
`-w`	Restricts pattern matching to whole words. For instance, `GREP -w "Size" *.c` will match "`Size`" but not "`GetBufSize`".
`-z`	Selects verbose output mode, printing the filename of every file searched, regardless of whether the file contains a pattern match.

TABLE 3.7. GREP REGULAR EXPRESSION SYMBOLS.

Symbol	*Description*
`^`	Placing a circumflex at the start of the string forces the search pattern to match only items that begin at the start of a line.
`$`	The dollar sign is the opposite of the circumflex. It forces the pattern to match items that fall at the end of a line.
`.`	Use a period to match any character value. This is equivalent to the DOS ? wildcard for matching any single character in the string.
`*`	The asterisk matches an entire string of characters (like the DOS * wildcard) and is used, for example, to quickly match any characters before or after a specific character set. For example, `CA*` matches Call, California, Cauliflower, and even just CA. The asterisk, unlike the `+` symbol, matches even if there are no characters following the primary search string.
`+`	Behaves like the asterisk, except that there must be at least one character for the `+` symbol. `CA+` does not match CA but does match Call and California.

continues

TABLE 3.7. CONTINUED

Symbol	*Description*
`[ ]`	Use brackets to specify a list of single characters that may appear in this position. For example, `ABC[DEF]` matches ABCD, ABCE, and ABCF, but not ABCG or any other characters not including D, E, or F in the fourth position. You also can search for a character match not appearing in the bracketed list by placing a circumflex as the first character. `ABC[^DEF]` means find all strings beginning with ABC and not ending in D, E, or F.
`-`	To match a range of character values, place a hyphen between the first and last characters in the range. `ABC[D-F]` is equivalent to `ABC[DEF]` because this hyphenated list matches all characters beginning with D and up to and including F.
`\`	When you need to search for one of the special characters—`^`, `$`, `.`, `*`, `+`, `-`, or brackets—prefix the special character with the backslash. For example, `\$` searches for the dollar sign character.

THIRD-PARTY SEARCH AND REPLACE TOOLS

GREP searches for text patterns within a group of files. When you need to both search for a text pattern and replace it when you find it, use Turbo Search and Replace provided on the *Secrets of the Borland C++ Masters* companion disks. Turbo Search and Replace (the program is named SNR) is a shareware program offered by Curtiss Little. Ordering information is contained in the TSNR directory on the companion disks. A third utility, whereis, helps you locate files by filename.

USING WHEREIS

Use whereis to search through all or portions of your disk's file directory to locate a specific file or files. The whereis primary command-line option is a file specification with optional DOS wildcard characters, as in these examples:

```
whereis myfile.c

whereis myfile.*

whereis ??file.*
```

whereis scans the directory structure, reporting the location of each occurrence of matching filenames. If you enter a filename specification only, whereis searches your entire hard disk. To search another hard disk, preface the filename specification with a drive letter, such as

```
whereis d:myfile.*
```

To restrict the search to a portion of your files, enter a directory name as part of the filename specification:

```
whereis \source\myfile.*
```

When a directory name is specified, whereis searches only in that directory and in any subdirectories beneath it.

If you use the pkarc .arc archive file format, you can use whereis to search inside your .arc files too. Add `-a` to the end of the whereis command line to request inspection of .arc files.

whereis cannot find files stored in directories whose names have an extension. This limitation is caused by the way whereis performs its file search. whereis takes a clever shortcut to provide searching capability up to twice as fast as other file search programs. By restricting whereis to examining directory names that have no extension (such as \source or \dos, rather than \source.dir or \dos.dir), whereis performs a very fast search. For most people this restriction is not a problem, because few PC users place extensions on their directory names.

USING TURBO SEARCH AND REPLACE

Turbo Search and Replace is a powerful search-and-replace shareware utility similar to GREP in its precise searching capabilities but having the power to perform complex string replacements. TS&R is a command-line-driven program named snr.exe. The basic format for an SNR command is similar to GREP:

```
snr -options -s "string" filename.ext
```

The *-options* are optional. The *string* contains a regular expression pattern and its replacement, and *filename.ext* contains a standard DOS filename or DOS wildcard characters. To see how snr might work, consider a command to replace every occurrence of `puts()` with a new routine you've just written named `put_thestring()`:

```
snr -s "puts=put_thestring" *.c
```

Note the use of the equal sign (`=`) to separate the search pattern from the replacement string. You may specify multiple search and replace patterns on the same command line, prefacing each with the `-s` option switch. If you need to use `=` inside the search or replace string, preface the `=` with a backslash, like this: `\=`.

In snr, the search pattern may contain a variety of regular expression symbols. See Table 3.7 and refer to snr's own documentation file. Use regular expressions to indicate precisely how the search pattern should match text within source files (such as at the beginning of lines or only when following a certain text). snr also performs intelligent text substitution, providing capital letters where expected. You may also use snr to scan through binary files.

snr's command-line options are shown in Table 3.8. Options are preceded with - or / and may be followed by + to enable the option or - to disable the option. The + or - is the default setting. None of these options are case-sensitive.

TABLE 3.8. THE SNR COMMAND-LINE OPTIONS.

Option	*Description*
`a+`	Operates in ASCII mode.
`b+`	Creates backup files before making changes.
`c-`	Displays a total count of the number of strings that have been replaced.
`d-`	Includes subdirectories of the parent directory when searching.
`i-`	Ignores case when searching.
`k-`	Retains the case of the found string when inserting the replacement. Performs intelligent case replacement so that when "the" matches "The," the replacement also begins with a capital letter, and so on.
`l-`	Displays or lists each line that matches one or more of the search pattern strings.
`p-`	Programming search mode. Attempts to identify language tokens.
`o-`	See snr documentation.
`r-`	When enabled, this option enables you to place regular expressions in the replacement string. See the snr documentation for details.
`v+`	The verbose switch controls the amount of output produced during the scanning and replacement process.
`w-`	Separates words from text. A word is text separated from other text by blanks and punctuation symbols.
`z-`	If a match is found, this option causes snr to prompt you before making a replacement.
`8-`	When scanning through ASCII files, use this option to strip the high or eighth bit out of the text. Some programs, such

continues

TABLE 3.8. CONTINUED

Option	*Description*
	as older versions of WordStar, often set the high bit of a character to indicate a special function to WordStar. If you eliminate the high bit setting, the character may be converted to standard ASCII.
`#-`	Displays the line number where the match was found in a file. Implies that snr is searching through an ASCII file.

OBJXREF

OBJXREF is a cross-reference utility that generates a report detailing the symbols referenced and defined within individual .obj object modules or .lib library files. For typical object files, the output files can be quite large. Therefore, sample reports are not reproduced here in text. However, OBJXREF is easy enough to use that you can give it a try on any Borland C or C++ produced object modules you have handy. OBJXREF is located in the \borlandc\bin directory.

To use OBJXREF, you specify a set of optional command-line switches followed by a list containing one or more object or library files to be cross-referenced. The command-line switches are either control switches that tell OBJXREF how to go about its business or report switches that select the type of cross-reference report. There are several control switches, only one of which is detailed here (see \borlandc\bin\util.doc for additional information). Use `/o` (note the use of a slash (`/`) character rather than the more typical `-` to precede an option) followed by a filename to direct OBJXREF to write its output to a file. For example,

```
objxref /Oreport.txt object.obj ...
```

Table 3.9 shows the different types of reports that are produced and the command-line option used to request the report. Place the command-line option before the list of object files to be cross-referenced.

TABLE 3.9. REPORTS PRODUCED BY OBJXREF.

Report	*Description*
/RC	Issues a report arranged by segment class type. Here class refers to memory segment classifications, such as CODE or DATA, and has nothing to do with C++ classes.
/RM	Displays a list showing each module name, followed by the public symbols defined within the module.
/RP	Similar to /RR, /RP produces a list of every public symbol and its defining module.
/RR	Displays a list of every public symbol (sorted by name) and the module in which it is defined.
/RS	Reports on the size of each module within each memory segment.
/RU	Produces a report of symbols that are unreferenced in other modules. These symbols include static variables in C, private definitions in assembly, and any symbol that is never referenced or used.
/RV	RV stands for *report verbose* and is a single command option to produce all the possible report types.
/RX	Issues a report showing all the external symbols referenced within each module.

PRJCFG

PRJCFG converts .prj files into the command-line compiler's .cfg configuration files. You may also convert .cfg files back into .prj file format. To convert a project file into a configuration file, type

```
prjcfg projectfile.prj configfile.cfg
```

If no configuration file is specified, turboc.cfg is used as the default. To convert a configuration file into a project file, use

```
prjcfg configfile.cfg projectfile.prj
```

PRJCNVT

If you have upgraded from Turbo C 1.0, 1.5, or 2.0, your original project files are no longer compatible with the project files used by Borland C++. Use PRJCNVT to convert old project or configuration files into new format project files. To convert an old project file into a new project file, type

```
prjcnvt oldfile.prj newfile.prj
```

where *oldfile.prj* is your original file and *newfile.prj* becomes the converted file.

To convert an old-style configuration file into a project file, type

```
prjcnvt oldconf.tc newfile.prj
```

where *oldconf.tc* is the old style configuration file.

THELP

THELP is a pop-up TSR program providing online, context-sensitive help to both the Borland C++ language and IDE features. THELP provides Borland C++ language help when you are using other text editors to prepare your source. This is particularly convenient when using a separate editor in conjunction with the BCC stand-alone compiler or when transferring into a third-party editor from within the IDE.

To make THELP active, run it from the command line by typing THELP. As soon as THELP is memory-resident, it may be popped up at any time by pressing the 5 key on the numeric keypad. Like the IDE's help system, THELP is context-sensitive. If the cursor is located on a C or C++ keyword when THELP is made active, THELP displays help that corresponds to the keyword, standard library procedure, or function within a window. You can customize the

window's location and size when you install the TSR. Use `/Wx,y,w,h` on the THELP command line to specify the column *x* and row *y* (zero relative) of the upper-left corner of the window, plus the width *w* and height *h* of the window. A complete set of THELP command-line options is available in \borlandc\bin\util.doc. Other options enable you to customize the screen colors, select a different keyboard activation key, or explicitly state the location of the help database file.

When THELP is no longer needed, it may be unloaded by typing

```
THELP /U
```

TRANCOPY

The contents of the Transfer menu (see Chapter 2, "Power Features of the IDE and Borland C++") are stored in the current project file. You can copy (or merge) the transfer menu items from one project into another project file. Use TRANCOPY to specify a source and destination project file:

```
trancopy source.prj dest.prj
```

The default operation merges the transfer items from `source.prj` into `dest.prj`. If you want the transfer items in `source.prj` to replace the transfer items in `dest.prj`, use the `-r` command-line option:

```
trancopy -r source.prj dest.prj
```

TRIGRAPH

The languages that are spoken and written in countries outside the U.S. have alphabets that range from mildly to significantly different from the American form of the English alphabet (see Chapter 14, "Creating Software for the International Marketplace"). The original IBM PC and its offspring provide support for the additional characters used by many other alphabets. Depending on what country you live in, your PC keyboard may or may not be able to type some or all of these characters. In the case of non-U.S. keyboards, some of the

keys that normally are used to type standard C language characters (such as the # and ^ characters) may be replaced with international character substitutes. As a consequence of international keyboard support, it can be difficult, if not impossible, to type a standard C program.

The only way around this difficulty, short of purchasing a separate keyboard containing the keystrokes needed for C programming, is to substitute other characters into your source file in place of the actual C characters. Then, use a utility program to scan through your source and convert the substitute characters into legitimate C characters. Borland provides TRIGRAPH in the \borlandc\bin directory to convert special three-character sequences (hence the name *TRIGRAPH*) into corresponding C-compatible characters. Table 3.10 shows C language characters and their corresponding three-character substitutes. When you need to use a character such as #, type ??= instead. When you are finished entering your text, use TRIGRAPH to scan through the source file. TRIGRAPH converts each trigraph sequence into its corresponding C text character. To run TRIGRAPH, type

```
trigraph file1.c file2.c ...
```

where *file1.c*, *file2.c*, and any other files you may specify represent the input files. TRIGRAPH renames the input files as .bak files (that is, *file1.c* becomes *file1.bak*) and scans through each file, producing new .c files containing the conversion. If you want to undo the conversion (to go from standard C characters back into trigraph sequences), insert the -u (*undo*) switch before the filenames:

```
trigraph -c file1.c file2.c ...
```

TABLE 3.10. THE SPECIAL TRIGRAPH CHARACTER SEQUENCES.

C Character	*Trigraph Sequence*
#	??=
\	??/
[	??(
]	??)
^	??'

C Character	*Trigraph Sequence*
¦	??!
{	??<
}	??>
~	??-

OTHER UTILITIES

The following sections describe some other utilities you might find useful.

4PRINT PRINTING UTILITY

If you use an HP LaserJet, DeskJet, or compatible printer, 4PRINT is a must-have shareware utility, particularly if you'd like to save paper and produce compact listings. 4PRINT is located on the companion diskettes for your evaluation.

Let's face it: programmers use a lot of paper to make listings of source code. Further, if you are using one of the Jet-type printers, a typical 8½-by-11-inch sheet of paper is not the most convenient format for looking at your source. When you need to stare at your code, you usually need to stare at more than one page at a time. 4PRINT helps by squeezing two or three traditional output pages onto each sheet of paper.

Traditional printing orients the output in *portrait mode*, placing the text horizontally across the short or 8½-inch direction of the paper. 4PRINT prints in *landscape mode*, changing the orientation so that lines are output horizontally along the long direction of the sheet (the 11-inch side). 4PRINT also switches to a finer font so that it can display two or three pages side-by-side in landscape mode. I've found that the two-page mode works best for my printer. You also can tell 4PRINT to output on both sides of the paper, helping to reduce your paper budget at the same time.

4PRINT has far too many options to detail here. But if you own a Jet-type printer, you definitely should investigate the purchase of this excellent shareware product.

LZEXE EXE COMPRESSOR

LZEXE is an ingenious executable file-compression program that shrinks an .exe file but leaves it in an executable form. Specifically, LZEXE performs data compression on the contents of the specified .exe file and appends a mere 330-byte decompressor routine to the end of the file. The result is a new .exe file that is sometimes as much as half the size of the original. I've used LZEXE on a wide variety of software—commercial, commercial shareware, freeware, and my own—with great success. If you are short on disk space, or in particular, short on diskette space, consider using LZEXE.

Programs that are compressed by LZEXE load and execute just like any other program. The only difference is that the compressed programs are loaded into RAM and then are decompressed in place in RAM. With most software, you will not notice any degradation in loading time because the decompression time is offset by the loading of fewer blocks from the disk.

> If you are using a disk compression scheme such as SuperStor or STACKER on your hard drive, you probably will gain little by compressing your .exe files. The disk compression system takes care of that automatically. A peculiarity of data compression algorithms is that most data that is already compressed actually expands if compressed again!

LZEXE could not be easier to use. Type the command

```
LZEXE filename.exe
```

where *filename.exe* is the name of the program to compress. You should immediately notice an unusual aspect of this program (especially for American readers): all the program output is in French. The distribution diskette contains

English translations of the documentation files. Don't worry about the French output—there's not much else to do but launch the program. As soon as the program is completed, the original, unmodified .exe file is copied to *filename.old*, and the new, compressed file is located in *filename.exe*.

LZEXE might run into difficulty on extremely large programs because it can run out of internal table space used to perform the compression. LZEXE should not be used on any programs that contain overlays, that modify or read data from their own .exe file, or that depend on making a check of their .exe's file size. If you receive an error message (remember, it will be in French), your best bet is to abort the program and consult the documentation files.

HEXEDIT AND DUMP

The companion diskettes contain the DUMP and HEXEDIT utility programs. DUMP displays any file, binary or text, in hexadecimal format. HEXEDIT does the same but provides an editing feature so that you can change items in the file. You may also search files to locate specific search patterns. When you start either program, it prompts you for the name of a file to read. After the first data on the screen is displayed, you may press F1 for a brief help message.

On my configuration, HEXEDIT works just fine, but DUMP will hang if launched within a Microsoft Windows 3.1 DOS box. DUMP works just fine when launched directly from DOS.

VERSION CONTROL SYSTEMS

Whether you create small programs by yourself or large applications using teams of programmers, you need a version control software system (VCS). Version control systems keep track of all source code changes as you develop your program. Using a version control system, you can backtrack at any point during development and re-create your source as it existed at some point in the past. A VCS may also maintain control over ownership of the source code. Ownership problems occur when multiple programmers need access to the same files.

Controlling file ownership

Using version control for documentation

Tracking software revisions

Using ATTIC

An introduction to PVCS Version Manager

DO YOU NEED A VERSION CONTROL SYSTEM?

Initially you may avoid using a version control system, especially if you develop programs by yourself. But think about how software is developed and how new software editions must be tracked. Every few months you might launch a new version of your software, distributing copies to coworkers, friends, the shareware market, or maybe the retail marketplace. Over a year or two, multiple versions of your software might be floating about (because not everyone updates to the latest edition). Keeping track of the source code to rebuild any particular version can become quite a chore.

To complicate the situation, you might throw in a few custom changes for a colleague. You'll then decide to improve an important algorithm or switch from one library to another. All of a sudden, you need to track several versions of your software. After a few iterations, you might discover that you've made a programming error and that you must *back out* your changes, reverting to an earlier version of the source. How can you possibly keep track of all this? One way might be to keep different versions in separate subdirectories. This works to a point: as soon as you need to make a change common to all the source files, you must carefully reedit each of the separate versions and hope like the dickens that you haven't made any mistakes.

Version control systems help you make sense out of managing multiple versions of your software and provide you with the ability to backtrack to previous versions at any time. A typical version control system operates somewhat like a library. You check files out, make your modifications, and then check them back in. When you check the files back in, the version control system automatically tracks changes by saving only the changes between the new edition and the previous edition. By keeping the first edition on file and then tracking changes between subsequent editions, the VCS uses a minimum of disk space. Later, when you want to retrieve any one of the versions, the VCS starts with the original source it has on file and applies the changes that were made as each new edition was checked in. You can re-create a previous edition at any time.

In actuality, modern VCS software operates slightly differently than just described. Because you are most often interested in retrieving recent editions of your software files (rather than very old copies of the files), a good VCS stores the entire content of the most recent update and records the changes needed

to rebuild each previous edition. This is a minor point, but it helps the VCS operate at maximum efficiency in re-creating the more recent versions from its historical database.

CONTROLLING FILE OWNERSHIP

When you develop an application using two or more programmers, a VCS helps you provide control and ownership of the files. Consider a typical C or C++ program containing perhaps 50 to 100 source files. When you have more than one programmer working on the source, how do you prevent two programmers from working on the same file at the same time?

Let's look at an example of the problems that can occur when you are not using a VCS. Suppose that one of the programmers (we'll call him Tad) links to the network file server and copies module20.c to his hard drive and begins work. A few hours later, another programmer (we'll call her Stacy) also needs to make a change to module20.c. Stacy also copies module20.c to her disk. Neither Tad nor Stacy knows that the other is making changes to the same file. At the end of the day, Tad copies his updated file back to the network file server. Stacy, who is working late, copies her modified copy of module20.c to the server a few hours later.

The next day, Tad arrives at work and decides to make a few more changes to module20.c. He copies module20.c from the network file server, but none of his changes are in the source code. "What happened?" he wonders. Obviously, due to the lack of control over file ownership, Stacy's copy of module20.c was copied over Tad's changes.

With a VCS in place, you use the check-in and check-out mechanism to control ownership of the files. Again, think of a library. When you need to edit a file, you must check it out from the file server. While the file is checked out, no other programmer can gain access to the file. When it's checked back in, the file is again made available. A VCS does not necessarily prohibit access to a file when it is checked out—it can always be marked as read-only. That way, you can copy the latest updates for your own compilations. Because the files are marked as read-only, however, you are prevented from making changes when you do not have ownership.

Version control works only when everyone plays by the rules: anyone can change a protected read-only file to a read-write file using the DOS ATTRIB command and then mess with the file to their heart's content. You might even be able to fool the system into enabling you to check in a file that you never checked out. By going around the safeguards built into the VCS, you can get yourself in serious trouble.

I once worked at a software company that was a day or so away from shipping a new product when a moderately bad defect was uncovered. The programmer who knew how to fix the problem checked out the necessary source code file and began to work on the problem. Meanwhile, another programmer found a minor problem that required a simple change to the same file. Due to the unusual time constraints, he tried to shoehorn the changes into the source code of a file for which he did not have ownership. This required breaking the rules of the VCS.

In cleverly circumventing the normal VCS process to put the files back on the network file server, the second programmer inadvertently overwrote the first programmer's changes. Without the knowledge of either programmer, the file updates collided and only one of the fixes made it back to the master source on the file server. Both programmers thought that both defects were fixed. Due to the simplicity of the changes and the pressures to roll the product out the door, many of us thought that the product could ship with only a minimal round of final testing. The Quality Assurance staff successfully argued otherwise and the product was subjected to a thorough test analysis again.

Much to our surprise, the original defect still persisted. This perplexed everyone because our paper trail of source changes (and the .exe file on one of the systems) showed that the defect was definitely fixed. After some investigation, we discovered the flaw in the file ownership and control process. The file was again checked out, the correct source code changes were carefully reinserted, and the updates were then properly checked back into the VCS.

A word to the wise: When you begin using a VCS, do not arbitrarily bypass the safeguards built into the system. Even when you think you've got all the ownership issues sorted out between you and the other programmers, let the VCS deal with the control functions. VCS software, such as the PVCS Version Manager system described later in this chapter, features the capability to merge changes in multiple copies of the same file, preventing the type of problem just described from occurring.

USING VERSION CONTROL FOR DOCUMENTATION

Most version control systems can manage arbitrary data files and all types of ASCII source files. As a consequence, you can use your VCS to keep tabs on documentation files and other items that tend to be updated over time. A full-featured VCS also can track binary files such as object modules and libraries.

TRACKING SOFTWARE REVISIONS

Keeping on top of evolving software projects is a tough job. You can do some limited version management using simple techniques, such as using subdirectories to organize different versions, or through the use of conditional compile statements such as `#if`. The following section describes a shareware program named ATTIC that provides a reasonable and effective source code version control capability. Later in this chapter, the PVCS Version Manager 5.0 is introduced. It is used by most professional software developers for version and ownership control. PVCS Version Manager provides comprehensive version maintenance and a command-line interface for quick check-in and check-out of files for both local and network usage.

USING ATTIC

ATTIC is a shareware program written by Roger Hering. It is included on the companion disk to this book and is also available through many shareware distribution outlets for your evaluation prior to purchase. The purpose of this section is to give you an overview of this shareware product and to show how you might apply a straightforward version tracking system to your text files.

ATTIC is optimized to track version histories of ASCII files. You can use ATTIC to store binary files, but I don't recommend this because ATTIC does not store binary files as efficiently as it stores text files. ATTIC creates a database to store the archive history of your multiple source editions. This database is called a *library* in the terminology of ATTIC. The files you work with are called *diskfiles* when they are checked out and are named *text* when checked into the library. For each text in the library, you may have several versions, named *text versions*. ATTIC is suitable for tracking source and documentation revisions, but it does not provide a mechanism for controlling source code ownership. ATTIC is best suited to individual programmers, not teams.

ATTIC is a menu-driven program featuring the original Lotus 1-2-3 menu style. To run ATTIC, type the attic program name on the DOS command line, followed by an optional library name, as in this example that names the source library:

```
C:\SOURCE> attic_source
```

If you do not specify a library, ATTIC prompts you for the library name. (You should restrict the filename to seven characters or less because ATTIC adds an eighth character.) When you run ATTIC for the first time, ATTIC asks if you want to create the library you have specified. After opening the library, ATTIC displays its main menu, shown in Figure 4.1. A variety of features are provided in ATTIC; however, you will use the Add and Xtract menu selections the most frequently.

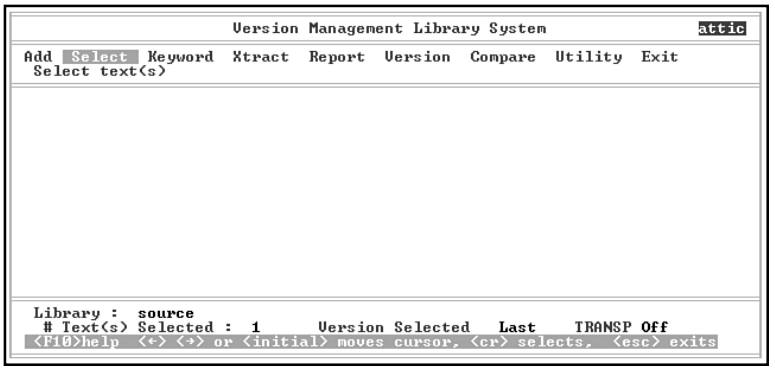


Figure 4.1. The ATTIC version management system main menu.

ADDING FILES TO A LIBRARY

To add one or more files, select the Add command. At the prompt, type

```
Read from which file(s) ?
 :
```

Enter the name of the file you want to add or enter a wildcard filename specification, such as *.c, to add a group of files. When you type the name of a single file, ATTIC prompts

```
Comment :
```

Enter a descriptive comment indicating the changes you have made to your file. After you press the Enter key, ATTIC adds your file to the library, recording the entire file if this is the first time it has been added to the library, or recording only the changes since the last time it was added. When you use wildcard characters to enter a group of files, you are given the choice of adding a descriptive comment to the entire group or to the individual files.

Each time a file is added to the library, its version number is increased by one. The first copy of the text that is placed in the library is version 1, the second is 2, and so on.

CHECKING OUT INDIVIDUAL FILES

To check out a specific file, select the Xtract menu command. When ATTIC prompts

```
Extract by Current Selection, Keyword, Text = <cr> :
```

press the Enter key. At the `Text :` prompt, enter the name of the file you want to extract. For example, if you want to check out a source file named program1.c, enter `program1.c`. When ATTIC asks for the version number, you can select the most recent version by pressing the Enter key again, or you can request a specific earlier edition of the file by typing its version number. Figure 4.2 shows the process for selecting a specific file. Note that you may extract the file to a diskfile having a different name than the one stored in the library.

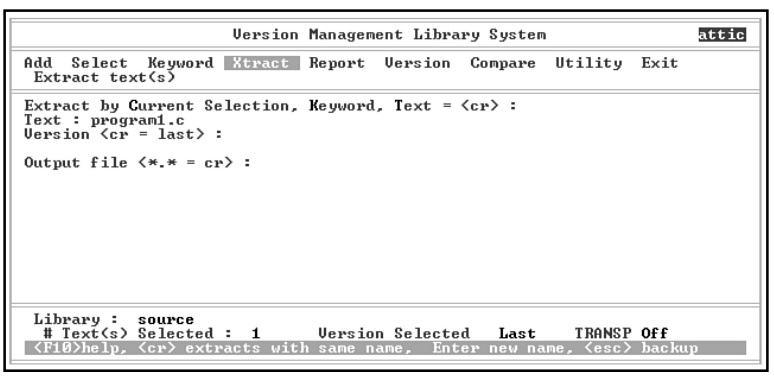


Figure 4.2. An example showing how to extract a text version from the library.

CHECKING OUT MULTIPLE FILES

To check out several files at once, or to see a list of the files stored in the library, use the Select menu command. The Select command displays a list of all files in the library, as shown in Figure 4.3. Use the numeric keypad's + key to mark text as selected, or the – key to deselect the text. The Select command creates

a *current selection* that becomes available to other commands. To check out the selected files (you can mark more than one), choose the Xtract command and then choose the Current Selection option.

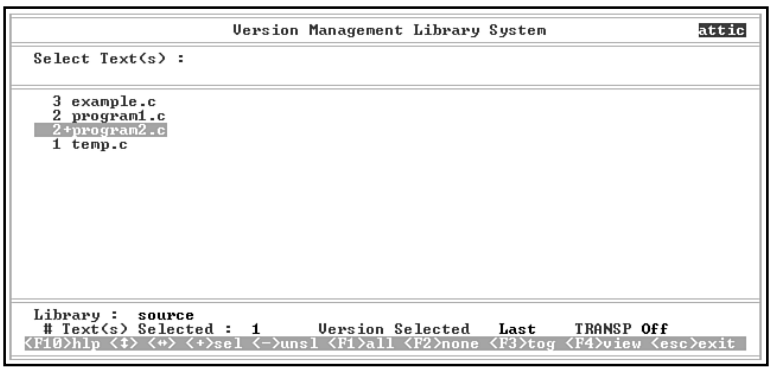


Figure 4.3. The Select command displays a list of the current library contents.

OTHER FEATURES

ATTIC includes a keyword feature for associating a keyword with particular versions of the files in the library. You use the keyword feature to mark a group of text versions with the same label. For instance, at a particular point in time you can compile your complete program and give the resulting executable a name such as firsttest. You can mark each of the text versions that is used in this compilation with the same keyword, such as `firsttest`. Later, if you need to retrieve the source that was used to build this edition of your program, you can select and extract files by reference to the `firsttest` keyword.

ATTIC's Report menu command produces a history report showing the time and date of each file version in the library, plus the comments that were added at the time each version was checked in.

INTRODUCTION TO PVCS VERSION MANAGER

PVCS Version Manager is a full-featured, commercially produced version control and management system. PVCS Version Manager provides all the features that are mentioned in the introduction to this chapter, including tracking old editions of your software, controlling access, and supporting overall project management and control duties. A companion product, PVCS Configuration Builder (which is not described in this chapter), provides an optimized project build facility. When you need to reconstruct a prior version of your software, the Configuration Builder automatically takes care of extraction, compilation, and linking as needed. PVCS Version Manager and PVCS Configuration Builder are products of INTERSOLV, Inc.

The goal of this section on PVCS Version Manager 5.0 is to introduce the capabilities that are available in a product of this nature. The instructions presented here are intended to give you a feel for the product and how it can help you improve your productivity. For complete details on all the features provided by the Version Manager (there are a great many more than are described here), refer to the PVCS documentation set.

If you want to, you can add all the various PVCS Version Manager programs, especially the get, put, and vcs programs described in this chapter, to the IDE's Transfer menu. In this way, you can access the version management system without leaving the IDE. See the section "Using the Transfer Options in the IDE" in Chapter 2, "Power Features of the IDE and Borland C++."

OVERVIEW

The PVCS Version Manager operates on the check-in and check-out model for tracking changes to source (or binary) files. Typically, when used for multiple programmer projects, PVCS Version Manager stores its history information on a network file server. You may also use the Version Manager in a stand-alone configuration. There is no difference in operation because the network support is completely transparent.

PVCS Version Manager stores your files in *archives*. Each new version or edition of a file is called a *revision*. The most recent revision in the archive is the *tip revision*. When you check a revision out from the archive, you may

optionally *lock* access to the file for your own use. A *locked revision* cannot be modified by other programmers. The copy of the file that you have checked out is called the *workfile*. Although it is not described in this chapter, the Version Manager can support restricted access to the archives by assigning different *privileges* to users or groups of users.

SETTING UP PVCS

To install PVCS Version Manager to your hard drive or network, follow the instructions detailed in the PVCS Installation Guide. The PVCS Install program automatically creates the appropriate subdirectories and copies the needed programs and files to your hard drive or network file server. Installation is a simple process that takes very little time. The default destination directory is \pvcs\dos for the DOS version of PVCS. The directory may reside on your local hard drive or on a network file server, depending on your configuration.

If you are installing a single user system, be sure to install the PVCS Version Manager tutorial option. This copies several files used in the PVCS tutorial guide to the \pvcs\vmtut subdirectory. Of particular importance, this creates a vcs.cfg configuration file in the tutorial directory. If you do not install the tutorial, the standard installation erroneously fails to create a default configuration file.

As soon as the software is installed on your hard disk or network file server, you must create a subdirectory for each project you are developing. For example, consider a geographic information system (GIS) application. Name the project directory \gis. Within the \gis directory (on your local drive), you must create three subdirectories—\gis\objects, \gis\sources, and \gis\archives—to store the necessary version management components of your project. You may also want to create a special *reference* directory to keep a read-only copy of the latest revisions of each of your files. This way you can reference these files for browsing or printing without having to manually check them out from the archive. In the \pvcs\vmtut subdirectory is a file named vcs.cfg. Use this configuration file as a sample configuration file to set up the Version Manager for your application. Copy \pvcs\vmtut\vcs.cfg to \gis (or the directory name you have chosen for your project). For the configuration

file to be found by the PVCS system, you must initialize a DOS environment variable named `vcscfg` to the directory containing vcs.cfg. You can initialize this variable at the DOS command line by typing

```
set vcscfg=c:\gis
```

For future use, you should place this initialization statement into your autoexec.bat file. You also need to run the DOS share.exe program. (The Version Manager installation guide omits this detail.) share manages file-sharing and locking and is required by the database management code in the Version Manager.

Next you need to edit the vcs.cfg file to set some configuration options. vcs.cfg is an ASCII text file that may be edited using the IDE or any text editor. The important options to set are as follows:

`VCSDir` This option tells the Version Manager where the archive directory is located, like this:

```
VCSDir=c:\gis\archives
```

`VCSID` Set this option to your name, using an underscore in place of blanks (spaces are not allowed in the ID). For example:

```
VCSID=Ed_Mitchell
```

`ReferenceDir` When you check a file into the version management system, it is removed from your working directory and copied to the archives. You can get a personal copy back from the archive by checking the file back out using the `GET` command (described in the section "Checking Files Out" in this chapter). But a simpler way is to let the Version Manager automatically maintain a directory of working source files. The PVCS Version Manager calls this the reference directory. You set up a reference directory by assigning the subdirectory name to the `ReferenceDir` option in vcs.cfg:

```
ReferenceDir=c:\gis\referdir
```

As soon as the reference directory is set up, each +time you check a file back into the archives, the Version Manager deposits a copy of it in the reference directory. For added safety and to ensure that you do not modify a file in the reference directory, you should add the `WriteProtect` keyword to the `ReferenceDir` setup statement:

```
ReferenceDir=Write Protect c:\gis\referdir
```

With the WriteProtect mode set, each file in the reference directory is read-only.

Journal Set the Journal option to a filename to keep a log of changes and updates made to the files:

```
Journal=journal.vcs
```

You may edit the journal.vcs file to see the information it contains; however, you should use the VJOURNAL program of the PVCS Versional Manager to examine the file. See "VJOURNAL command" in the PVCS Version Manager *Reference Guide*.

ADDING FILES TO AN ARCHIVE

As you create new files, they should be added to the archive using the PUT command. After you have typed in your source and are ready to check it into the version control system, type

```
put filename.ext
```

where *filename.ext* is the name of your file. For example:

```
put order.c
```

You may also use wildcard characters to check in a group of files:

```
put *.c
```

or

```
put sample??.*
```

put.exe is a program residing in the \pvcs\dos directory (or the directory where you had the Version Manager installed). Each file is stored in its own archive. When you use put for the first time, it creates the archive file in the directory specified by the vcs.cfg VCSDir configuration option. You are prompted for a descriptive comment to associate with the file. You should type a brief comment indicating the changes you have made to your file. This can prove invaluable later when you are trying to debug your source, particularly when you discover new problems that did not previously exist. This comment also can be inserted automatically into your source file to help you track changes. See the section "Maintaining Source Revision Histories" later in this chapter.

Archive files are assigned a name based on the original file name but having an extension that ends in `.?_v` where `?` is replaced by the first character of the original file. For example, when you check in the file order.c, the Version Manager creates an archive file named order.c_v. You can see the archive files by making a directory listing of the archives directory.

SUGGESTIONS ON USING *PUT*

After a file is placed in an archive, it is deleted from the source directory. You can change this action so that the Version Manager leaves a copy of the file in your working directory. In the vcs.cfg file, remove the statement that contains

```
DeleteWork
```

and replace it with this statement:

```
NoDeleteWork
```

When you make this change to the configuration file, each file that is checked into the archive leaves a copy in your working directory. The Version Manager sets the file attributes so that it is now read-only. I recommend that you use the `NoDeleteWork` option so that you can keep a copy of all your source files in one place. This makes compilations much easier because all the files stay in one place. There's no need to revise your project or make files to keep track of the subdirectory the files have been moved to.

When you are working as part of a team, you often need to check your changes into the archives so that your code will be made available to the other team members. Because you probably still want to maintain ownership (a locked revision) of the workfile, you use put followed by get

(described in the next section) to check the files back out. The Version Manager provides a simpler way to accomplish this common task. Use the L option when you put the files into the archive. The Version Manager checks in your current changes but retains your lock on the files. For example:

```
put -L modulx?.c
```

When put finishes, you still have a locked revision in your source directory.

CHECKING FILES OUT

When you want to check a file out, use the get command. To obtain a copy of a file, such as order.c, type

```
get order.c_v
```

This copies the most recent revision of the file to your source directory. Note that you should specify the name of the archive file (order.c_v in the example). get also works with wildcard filenames so that the following is permitted:

```
get files??.c_v
```

To obtain a locked copy of the file so that you prevent others from accessing the file while you are making modifications, you should use the lock option by inserting `-L` into the command line:

```
get -L order.c_v
```

A locked revision now exists in your source directory. If someone (even you!) tries to get a copy of order.c while it is locked, the Version Manager tells you that order.c is locked and provides you with the name of the user (from VCSID) who has the file. In this way, the Version Manager provides positive control over who is allowed to make source code changes at any particular moment.

ACCESSING OLDER REVISIONS

Revision numbers are assigned beginning with 1.0 by default. Each subsequent revision is numbered consecutively as 1.1, 1.2, 1.3, and so forth. You can manually override the revision numbering by using the `-R` option with the put command. Follow `-R` with the desired revision number. The only restriction is that the revision number must be greater than the highest existing revision number for the file. In other words, if you have revision 1.8 in the archives, you can assign a revision number such as 1.9 or 3.0, but you cannot assign a revision number of 1.5 because 1.5 is less than 1.8.

Use the `-R` option with get to obtain a particular revision of the file. For example, to extract revision 1.5 from the archives, type

```
get -r1.5 order.c_v
```

The Version Manager reconstructs the source as it appeared in revision 1.5, using the change history maintained in its archives. If you lock this version, make changes, and then check it back in, the Version Manager assigns revision number 1.5.1.0 to this modified, older edition of the file. In this way the Version Manager can track *branching* sources. Branching is a technique (not described here) that enables the Version Manager to track separate projects that are derived from a common source. Branching is especially valuable for handling common—but slightly modified—code such as libraries that are used in a variety of projects. The Version Manager provides a related service called *merging*, which is used to merge branching sources back into a common source module.

USING VERSION LABELS

In addition to revision numbers, you can identify your revisions using a *version label*. A version label is a textual identifier attached to a set of revisions that make up the current source code for the project. To understand the difference between revision levels and version labels, consider how one file named file1 is at revision level 1.2 and another file in the same project, file2, is at revision 1.7.

When you create an .exe file, you are assembling the program from a variety of revision levels (such as 1.2 and 1.7 in this example). This set of files and their respective revision levels is a *version*. If you need to access the source for this particular .exe file later, you can use get to fetch each revision level for each of the files involved. Of course, to do that you would need to keep track of which revision levels were used to make the executable.

With the Version Manager and version labels you can retrieve the entire collection of revisions with a single command. To do this you must assign a common version label to each of the revisions. For example, you might assign the version label ALPHA_#1 to revision 1.2 of file1 and to revision 1.7 of file2. Later, if you need to return to the source used in the ALPHA_#1 version of the software, you can get the sources by extracting all ALPHA_#1 versions. The Version Manager automatically extracts the appropriate revision from each archive. In summary, the version label marks one revision from each archive used to build a program.

There are several ways to mark the archives with a version label. The easiest way is to use the vcs command and the -V option:

```
vcs -Vversion_label
```

This assigns *version_label* to each of the tip revisions (the most current revision) in the archives. For example:

```
vcs -VALPHA#_1
```

You also assign a version label when checking files into the archive, using the -V option of the put command:

```
put -VALPHA#_1 *.*
```

To retrieve a specific version, use -V with the get command:

```
get -L -VALPHA#_1 *.c_v
```

MAINTAINING SOURCE REVISION HISTORIES

As you develop your application, you should get into the habit of maintaining a revision history as a comment in your source files. The revision history should contain the date (and optionally, the time) the file was last modified,

the name of the programmer who updated the file, and a brief description of the changes that were made. Later, when you check a file out from the archive, you will immediately know who has made changes to the file and what he or she did. You can manually insert the revision history information into each file you modify, or you can use a feature of the Version Manager to automate the maintenance of the revision histories.

If you insert special keyword symbols into your source code files, the Version Manager inserts the revision history automatically when the files are checked in. The special symbol `$Header$` expands into the name of the archive file, the date, the time, and other information. The keyword `$Log$` causes the descriptive comments you enter when adding a file to an archive to be inserted into the source text in chronological order. Listing 4.1 shows a simple program named example.c prior to its first check in. Note the placement of the `$Header$` and `$Log$` keywords. Listing 4.2 shows the same listing after it has twice been added to the archive. Note how the `$Log$` keyword keeps a running list of revision information.

LISTING 4.1. EXAMPLE SHOWING PLACEMENT OF THE $Header$ AND Log KEYWORDS.

```
/* $Header$ */
/* $Log$
*/
#include <stdio.h>

void main(void)
{
  printf("Hello, World.  Goodbye, World.\n");
}
```

LISTING 4.2. THE EXAMPLE FILE AFTER IT HAS TWICE BEEN CHECKED INTO THE ARCHIVE.

```
/* $Header:   C:/project1/archives/example.c_v   1.1   18 Jun 1992 14:18:26
                                                              Ed Mitchell  $ */
/* $Log:   C:/project1/archives/example.c_v  $
 *
 *    Rev 1.1   18 Jun 1992 14:18:26   Ed Mitchell
 * Added Goodbye, World phrase
 *
 *    Rev 1.0   18 Jun 1992 14:10:52   Ed Mitchell
 * Initial revision.
```

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

void main(void)
{
  printf("Hello, World.  Goodbye, World.\n");
}
```

As you might suspect, after a file has been checked in a number of times, the revision history at the beginning of the file gets to be quite lengthy. Each time you edit the file you must page through several screens of revision history comments. To save yourself the trouble, put the revision history at the end of the source file. Instead of writing

```
/* $Header$ */
/* $Log$
*/
```

at the top of the file, put the keywords after your last program statements.

OVERRIDING A LOCKED REVISION

In certain situations you might want to override the revision locking provided by the version control system. *You should not override the revision locking unless you have a very good reason and you know what you are doing*. One good reason to override a file lock is when you want to throw away the changes you have made to your locked copy of the file. Consider the example.c program from Listing 4.1. Perhaps I've made changes to a function, but I can't seem to get the new code to work. Instead of checking in the nonfunctioning code, I'd like to throw it away and start over. The best way to do this is to delete the working copy of example.c:

```
del example.c
```

Next, unlock the existing archive using the `-u` option of the vcs command:

```
vcs -u example.c
```

Then check out the original example.c again:

```
get -L example.c_v
```

MANAGING MEMORY

To program the PC you need to know a fair amount about the structure of the underlying CPU and memory systems. Even if you never look at assembly language code, you need to have a basic understanding of how memory is allocated in order to create the most efficient C or C++ programs. The choice of *memory model* influences the capacity, speed, and size of your program. Where you place your data storage—in global, local (or automatic variables), or dynamically allocated memory—affects your program's operations and capabilities. This chapter uncovers some of the mysteries of memory management and the decisions you must make to optimize your use of system memory. Additionally, tangential topics such as memory trashers, which occur when pointers go awry, are covered because they are part of managing memory.

Choosing a memory model

Special points about pointers

Mixed model programming and pointer modifiers

Creating a .com program

`malloc()` and related routines

CHOOSING A MEMORY MODEL

For all but small programs you must make a conscious decision about which memory model the compiler should use when compiling your program. The choice of memory model influences how much code or data you can have in your program, and it also influences both the overall size and speed of your application. Understanding how to choose a memory model requires that you first know something about the Intel 80x86 CPU family of processors.

When Intel produced the first members of the 80x86 family (the 8088 and 8086 chips), they were building an upgrade—and a big one at that—from the original 8080 microprocessor of the earliest personal computers. The 8080 was an 8-bit processor with a 16-bit addressing capability. From the standpoint of memory, that meant that all code and data combined had to fit in a 64K address space. 64K was all there was to play with in those early days.

When the 8088 and 8086 were designed, the original 64K limitation of the 8080 (as well as the 8080's original register naming convention) was incorporated into the new processors but with a distinctly new twist: program addressing was still limited to 64K *per segment*, but the computer could support up to a total of 1M of memory through the use of multiple 64K memory segments. To meet this capacity, the 8088/86 introduced *segment registers* to point to the start of memory segments. Within each segment, a 16-bit address is used to reach any part of the 64K address space. The same concept applies to the 80286, 80386, and 80486 processors, which are backward-compatible with the original 8086. An important distinction, from the standpoint of memory, is that the newer processors can address vastly increased memory spaces—up to 4 gigabytes of physical memory.

THE 80X86 CPU REGISTERS

The basic 80x86 CPU architecture provides a set of 16-bit registers and addressing of up to 1M of memory. The newer CPUs introduce 32-bit registers and can, consequently, address even greater memory spaces. To see how the memory allocation scheme is influenced by the CPU's registers, look at Table 5.1. This table presents the basic CPU registers that are provided in all of the processors from the 8088 on up.

TABLE 5.1. THE BASIC REGISTERS THAT ARE FOUND IN COMMON FROM THE 8088 TO THE 80486 CPU.

Register	*Explanation*	*Alternative 8-bit Form*
AX	Accumulator register	AH, AL—high and low bytes of AX, respectively.
BX	Base register	BH, BL—high and low bytes of BX, respectively.
CX	Count register	CH, CL—high and low bytes of CX, respectively.
DX	Data register	DH, DL—high and low bytes of DX, respectively.
BP	Base pointer	
SI	Source index	
DI	Destination index	
CS	Code segment	
DS	Data segment	
SS	Stack segment	
ES	Extra segment	
IP	Instruction pointer	
SP	Stack pointer	

In Table 5.1, the segment registers CS, DS, SS, and ES are most important to the discussion of memory addressing in this chapter. Some of the register names in the Explanation column are largely irrelevant, particularly with respect to the AX, BX, CX, and DX registers. The names and use of these registers originated with the A, B, C, and D registers of the 8080 processor. For instance, although the CX register is indeed used as a counter for some instructions, it may also be used for general arithmetic and other functions. Nevertheless, these "explanatory" names have carried over through the years.

Also shown in the table are the 8-bit registers—AH, AL, BH, BL, CH, CL, DH, and DL—which address the high and low bytes, respectively, of the AX, BX, CX, and DX registers, permitting easy byte-level operations.

On the newer 80386 processor, these original 16-bit registers have become subsets of the new processor's 32-bit registers. For example, EAX is the *extended* AX register, and AX is equivalent to the lower 16 bits of EAX. AH and AL continue to reference the high and low bytes of AX, and hence, the lowest two bytes of EAX. The 80386 also contains additional 32-bit registers, but these are not covered in this book. See an 80386 microprocessor handbook or an 80386 assembly-language programming guide for details. The 80x87 math coprocessor provides additional registers and instructions not described in this book.

MEMORY ADDRESSING

The segment registers, CS, SS, DS, and ES, are used to address memory. The index registers, DI and SI, are used in conjunction with DS and ES to assist with instructions that move or operate large byte blocks. To understand low-level memory addressing, take a look at the bit representation of registers and addresses.

When you are looking at the layout of bits within a register, the bits are numbered in ascending order from right to left, like this:

```
Bit: 15  14  13  12  11  10 9 8 7 6 5 4 3 2 1 0
```

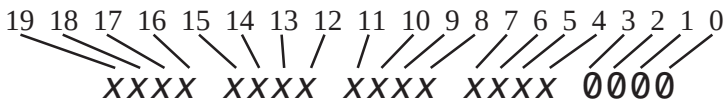
In this representation, the value of decimal 8 stored as a bit pattern is

```
0000 0000 0000 1000
```

When you are performing signed arithmetic, as for the value –8, the high bit is set and the value is stored in two's complement format as

```
1111 1111 1111 1000
```

Certain registers—CS, SS, DS, and ES—are called *segment registers* and are used for memory addressing only. The 8086/8088 CPU's segment registers provide 1M addressing, which is a good trick because the 16 bits in each register address only 64K of memory. The secret is in how the segment registers are combined with other values to form a physical memory address. Each of the segment registers points to a 16-byte page. Effectively, the segment registers are equivalent to a 20-bit register whose lower 4 bits are always zero, like this:



Memory addresses are formed by adding the contents of a segment register, shifted 4 bits to the left, to another 16-bit register (such as BX) or a 16-bit offset value, resulting in a 20-bit memory address, like this:

	xxxx xxxx xxxx xxxx 0000	Segment register value
+	*nnnn nnnn nnnn nnnn*	Other 16-bit register or constant
=	*aaaa aaaa aaaa aaaa aaaa*	Producing a 20-bit address

The segment registers are combined in specific ways with other registers and 16-bit constant values to address memory. The CS (code segment) and IP (instruction pointer) registers are added together to point to the next machine instruction to be executed by the CPU. Because the IP register is just 16 bits wide, a single code segment is limited to a maximum of 64K of code. Because these registers are always used together, they often are written as the pair CS:IP.

The SS (stack segment) and SP (stack pointer) point to the top of the processor's stack, for recording temporary values and procedure call return addresses. (In the 80x86 family of processors, stacks grow downward in memory; hence the stack's top is actually *below* the stack's bottom.) Stacks are limited to a total of 64K of memory due to the 16-bit address capability of the SP registers. As with the CS:IP pair, the SS and SP registers often are referred to as SS:SP.

Data stored in the heap area usually is referenced as an offset from the ES (extra segment) register. If you change the value in ES, the entire heap storage area may be accessed. Depending on the specific machine instruction, the DI and SI registers may be added to the DS and ES registers to point to groups of bytes within their respective memory segments.

Memory segments do not need to be 64K. Indeed, most memory segments, particularly those that contain code, are considerably less than 64K. Each time the program begins to execute machine instructions within a segment, the CS register is set to point to the beginning of the segment and the IP register is set

to an offset within the segment. Control is transferred to other segments when the program executes a far jump or a subroutine call to a procedure located in some other segment. Similarly, segments that store data may be less than 64K. You gain access to those data values by setting one of the other segment registers, usually DS or ES, to point to a segment. Then use BX, DI, or SI as offsets from the start of the segment.

NEAR AND FAR MEMORY REFERENCES

Depending on the memory model (see the next section, "Memory Models"), the compiler can generate machine instructions that use either *near* or *far* addressing. Consider a simple C program containing three functions—`main()`, `func1()`, and `func2()`—all located within a single source module and compiled using the small memory model. When `main()` calls `func1()` or `func2()`, the CPU sets the CS and IP registers to point to the first instruction in the functions. Because all three functions are located in the same code segment pointed to by CS, only the IP register needs to be set to point to the function being called. This means that to call, for example, `func1()`, the call instruction needs only the 16-bit address of `func1()` because the value of CS is unchanged. When a function call is made entirely within a segment, only 16-bit addresses are used. This is known as a *near memory reference*.

Next, consider what happens when `func1()` is located in a different source module and the program is compiled using the large memory model. In order for `main()` to call `func1()`, the CPU is given a new value for both CS and IP. The new CS value is the address of the segment containing `func1()`. In this form, the program is making a *far memory reference*.

To summarize, a near memory reference is a 16-bit address used entirely within a segment. A far memory reference is one that is made to a separate segment and that requires two 16-bit addresses to specify both a segment and an offset. As you can see, a near memory reference requires half as many address bytes as a far memory reference. This means that using near memory references produces smaller programs. And fewer instruction bytes mean faster execution. You learn how to explicitly create near and far pointers in the section "Mixed Model Programming and Pointer Modifiers" later in this chapter.

You can perform limited arithmetic on the address in a far pointer, but it will affect only the offset portion of the address. If you add one to a far pointer, the

offset value increments by one. If the addition causes the offset to exceed 16 bits (as in hex FFFF + 1), the offset wraps back to zero. Pointers frequently are incremented and decremented using C's postfix and prefix increment and decrement operators (for example, `*p++` or `*(--p)`). This increments or decrements the offset portion of a segment:offset pair.

MEMORY MODELS

Now that you have had a glimpse of the underlying processor architecture, you can begin to understand how memory models determine the layout of your compiled programs. The memory model choice determines where and how much code and data can be allocated to your application and how the segment registers will be used to access that code and data. You must also know about memory models if your program must link in routines compiled using a different memory model. (This is covered in the section "Mixed Model Programming and Pointer Modifiers" later in this chapter.)

The simplest memory model is the *compact model*, where CS, SS, DS, and ES are all set to point to the same area of memory. This limits the total size of your program, including code, data, and stack space, to a maximum of 64K. Within this space, all functions may be reached with a simple 16-bit address. Table 5.2 describes each of the six memory models and presents information about their advantages and disadvantages.

TABLE 5.2. MEMORY MODELS SUPPORTED IN BORLAND C++.

Model	*Advantages and Disadvantages*
Tiny	The CS, SS, DS, and ES all point to the same memory address. Maximum program size is limited to a combined total of 64K. Only near pointers are permitted. Tiny model programs can be converted to .com files (see the section "Creating a .com Program"). A .com file is slightly smaller than an .exe file and generally is considered to be an obsolete executable file format. You may not use the tiny model for Windows programs.

continues

TABLE 5.2. CONTINUED

Model	*Advantages and Disadvantages*
Small	The small model often is used as a substitute for the tiny model. Small model programs are divided into two segments: code and data/stack. Each segment is limited to a maximum of 64K, and the DS and SS registers share the same 64K maximum segment. Like the tiny model, all memory references are made using simple 16-bit near pointers, providing for the smallest and fastest possible memory references.
Compact	The compact model is designed for small programs that must manipulate lots of data. There is only one code segment, and it may be up to 64K in size, but multiple data segments permit up to 1M of memory addressing.
Medium	The mirror image of the compact model is the medium model, providing only 64K of data but up to 1M of code space. Because many programs tend to have lots of code but little data, this is a popular memory model.
Large	Large model programs can accommodate up to 1M of both code and data. Far pointers are used for all code and data references. The disadvantage to using the large model, is the extra overhead of far pointers. Also, although the total data may be up to 1M, the largest single data element may be no larger than 64K.
Huge	The huge model essentially is the same as the large model but with the 64K data element size restriction removed. Huge model programs can allocate data structures that are larger than 64K, and each code module may have its own data segment up to 64K. The huge memory model is not supported for Windows programs.

The tiny, small, and compact models are known generically as small code memory models because they all limit the code space to a maximum of 64K. The

medium, large, and huge models are referred to as large code models because they provide multiple code segments with intersegment calls made using segment:offset addressing.

By choosing the correct memory model for your application, you may be able to reduce the size of your program and create faster code, especially if you can use a memory model that uses near pointers for code or data values. The medium and compact memory models are good compromises if your application must have a lot of code or a lot of data.

MEMORY MODEL RESTRICTIONS

In each memory model, the maximum code size for any module is still 64K. This means that when you compile a source file, the source file must result in no more than 64K of code. This restriction occurs because the IP (instruction pointer) register is a 16-bit register with a 64K addressing limit. Because each source file is compiled into its own code segment, it must necessarily be restricted to a maximum of 64K code bytes. If you exceed the 64K restriction, you must split your code into two or more code modules.

The amount of static data your program can have depends on the memory model you use, but generally it is 64K or less. Static data includes all variables that have file scope (variables defined outside a function are automatically treated as static), variables declared as static or extern, and all string constants. The following list summarizes data memory allocations:

- In the small and medium models, the data segment is shared with the stack, limiting the total static data to something less than 64K, depending on the memory required by the stack. Small and medium model programs can use dynamic memory allocation to manipulate more than 64K of data memory. See the section “Using Dynamically Allocated Memory” later in this chapter.
- In the large and compact models, the data segment is unshared and permits the maximum 64K of static data.
- The huge memory model is fundamentally different from the others. The huge memory model supports multiple data segments so that each code module may have its own static data segment, each up to 64K.

SELECTING A MEMORY MODEL

When you compile your object modules, you must tell the compiler which memory model to use. Generally, all object modules and libraries used in a program should use the same memory model, although in some instances it is possible to mix models (see the section "Mixed Model Programming and Pointer Modifiers"). To select a memory model in the IDE, use the Options | Compiler... | Code Generation... dialog box shown in Figure 5.1. Select Tiny, Small, Compact, Medium, Large, or Huge as appropriate for your application.

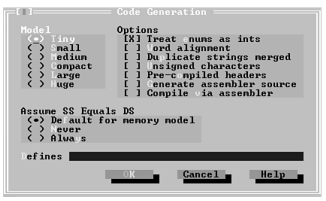


Figure 5.1. Selecting a memory model in the IDE.

To set a memory model using the command-line compiler, use the command-line switches shown in Table 5.3.

TABLE 5.3. BCC COMMAND-LINE SWITCHES FOR SELECTING A MEMORY MODEL.

Switch	*Model*
`-mt`	Tiny memory model
`-ms`	Small memory model
`-mc`	Compact memory model
`-mm`	Medium memory model
`-ml`	Large memory model
`-mh`	Huge memory model

SPECIAL POINTS ABOUT POINTERS

The choice of memory model influences how pointers are used within your software. A memory model providing near addresses uses 16-bit addresses to reach its data. A model that relies on far pointers requires 32 bits consisting of a 16-bit segment and a 16-bit offset. This doubling of the address supports greater amounts of data but at the expense of slower execution and larger code. Huge pointers enable your programs to manipulate any size data anywhere in memory, but they are burdened by the additional overhead needed to reference and use this pointer type. Finally, programs sometimes need to work with mixed memory models. That is, sometimes a program must be constructed using modules that were not created using the same memory module. For instance, a small model program might need to call routines in a large model object file. Techniques for dealing with these issues are described in this section.

HUGE POINTERS

You have already seen examples of near and far pointers. Like the far pointer, the huge pointer is also a 32-bit address, but it is manipulated quite differently than the customary 32-bit segment:offset address of a far pointer. For any specified address, multiple far pointer segment and offset pairs can point to that address. For example, the following far addresses are equivalent:

`0040:0100` and `0050:0000`
`1123:4C67` and `15E9:0007`

You might want to perform this address conversion arithmetic by hand to convince yourself that these addresses are equivalent. Shift the segment value left by 4 bits, producing a 20-bit address, and then add the offset value.

The huge pointer, though, is a unique address value: there can be only one huge pointer to a memory location. Huge pointers don't store ordinary segment:offset values. Instead, they store a quantity known as a *normalized* 32-bit pointer that can be used in arithmetic operations and compared to other huge pointers.

A normalized pointer is a far address whose segment and offset values have both been adjusted so that the offset value varies only from 0 to 15. In this way, the 16-bit segment address, shifted into a 20-bit address field to which the 4-bit offset (values 0 to 15) is added, produces a true 20-bit address.

Normalizing a far pointer is easy to do. Take a look at a couple of examples. First, examine a far pointer to the BIOS low memory area. Assume that the pointer has the value `0040:0117`. To normalize this far pointer, shift the segment register left by 4 bits, converting the segment into a 20-bit address:

```
00400
```

Next, add the offset value to this:

```
  00400
+  0117
-------
= 00517
```

To convert this 20-bit sum into a normalized pointer, let the upper 4 hex digits be the segment and let the lowest digit become the offset. This produces the normalized address:

```
0051:0007
```

Here's another example. Given the segment:offset pair `0E1F:4C67`, this is normalized as follows:

```
20-bit segment:          0E1F0
Add offset:               4C67
-------------------------------
=                        12E57
```

Moving the lower 4 bits of this 20-bit result into the offset produces the normalized address: `12E5:0007`.

The normalization process produces a unique huge pointer address for each address in memory (unlike the far pointer, where multiple segment:offset pairs can point to a specified address). Because huge pointers are unique, they may be compared to one another and used in arithmetic. When you use a postfix or prefix increment operator, for instance, the huge pointer may be incremented by the size of the operand to which it points. However, because the huge pointer's offset may be only in the range of 0x0 to 0xF, special arithmetic routines are called to perform the arithmetic and then reset the offset and segment values, if needed. This extra overhead means that the use of huge pointers is much slower than the use of conventional near and far pointers. On the other hand, this extra overhead is what enables the huge pointer to manipulate data objects that are greater than 64K.

SEGMENT POINTERS

Your program can directly access data in the various segments by using a special addressing modifier to declare a far pointer that is based in the desired segment. Borland C++ provides four keywords for defining segment pointers: `_cs`, `_ds`, `_es`, and `_ss`. You use the segment keywords like this:

```
int _cs *ptr;
```

This declares `ptr` to be a pointer that is offset from the code segment register. You may also declare a special segment pointer whose offset value is always kept at zero. To declare a segment pointer, use the `_seg` modifier:

```
int _seg *ptr;
```

In this form, `ptr` is a pointer like any other far pointer, except that only the segment portion of the pointer is used as the address. The offset value is always set to zero so that a `_seg` pointer points only to 16-byte paragraph boundaries. A `_seg` pointer is therefore useful for accessing a full 20-bit address space as long as you need only 16-byte address resolution. A number of restrictions apply to the use of `_seg` pointers. The most noticeable is that you cannot use the customary `++`, `--`, `+=`, and `-=` arithmetic operators.

CREATING POINTERS TO SPECIFIC LOCATIONS

To initialize a pointer to a specific memory address, use the `MK_FP()` macro (defined in dos.h). You might want to initialize a pointer to a specific address when accessing low-level system resources. For example, to obtain information about the state of the Caps Lock key, you need to check a byte located in the BIOS data memory area. You can check the state of certain keyboard settings, such as the Caps Lock key status, by checking bits located at 0040:0017 (hex). Accessing this byte of memory requires that you build a pointer to this location using the `MK_FP()` macro. The keycheck.c program examines the bits located at 0040:0017 to determine the state of the toggled keyboard keys.

Macros are also provided to perform the inverse of `MK_FP()`—that is, to obtain the segment and offset values when given a far pointer. Use `FP_SEG(`*`farpointer`*`)` to obtain the segment value of *`farpointer`*, and use `FP_OFF(`*`farpointer`*`)` to obtain the offset value. Here is an example:

```
int far *p;
...
segment = FP_SEG( p );
offset = FP_OFF( p );
```

A demonstration of `MK_FP()` is shown in Listing 5.1.

LISTING 5.1. DEMONSTRATION USE OF THE `MK_FP()` MACRO.

```
 1  /* KEYCHECK.C
 2     Demonstrates use of the MK_FP macro.
 3  */
 4  #include <stdio.h>
 5  #include <dos.h>
 6  #include <conio.h>
 7
 8  void main(void)
 9  {
10    unsigned char far *p;
11
12    p = MK_FP(0x0040,0x0017);
13
14    do {
15      if (*p & 128) puts("Insert Mode toggled on.");
16      else puts("Insert Mode toggled off.");
17      if (*p & 64) puts("Caps Lock toggled on.");
18      else puts("Caps Lock toggled off.");
19      if (*p & 32) puts("Num Lock is toggled on.");
20      else puts("Num Lock is toggled off.");
21      if (*p & 16) puts("Scroll Lock is toggled on.");
22      else puts("Scroll Lock is toggled off.");
23      puts("Press a key to continue; Esc to stop:  ");
24    } while ( getch() != 27 );
25  }
```

MIXED MODEL PROGRAMMING AND POINTER MODIFIERS

Occasionally, programs need to link modules that are compiled using a different memory model than that of the main program. Normally, you will not want to mix memory models, because various kinds of problems can occur.

When you need to mix code modules compiled with different memory models, however, you override the default near or far type, as appropriate, for the functions that your program calls.

Consider a small model program that must link an object module (or more typically, a module from a library) that has been compiled using the large memory model. By default, the compiler will generate near calls to all functions. A near function call will not reach the large model code; indeed, it probably will crash your program. Any data parameters that should be passed as a far pointer will be passed as near pointers, wreaking all kinds of havoc.

There is a solution to this mixed-model situation. You need to add a function prototype for the routines that will be called from the outside module. Suppose that you have a small model program that must display a complex number using a function named `put_complex()`, where `put_complex()` is defined in a large model object file named mycomp.obj. By default, the small model program issues a near call to `put_complex()`, passing near addresses as parameters. To override this default setting, either create a new prototype for the function, placing the far modifier before the function name, or better, define the header file for mycomp to explicitly use the `far` keyword on all functions and function parameters.

Listing 5.2 shows a sample mixed-model program named mixmodel.c, which was compiled using the small model. Listing 5.3 shows the header file for mycomp.h, and Listing 5.4 shows the mycomp.c source file. mycomp.c was compiled using the large model. There are two ways that the mixed memory models can be accommodated. Listing 5.3 shows the use of the `far` keyword in the header file. This way, all source files that include mycomp.h will get a function prototype that forces the compiler to generate a far call to `put_complex()`.

LISTING 5.2. THE MAIN SOURCE FILE FOR DEMONSTRATING MIXED-MODEL PROGRAMMING.

```
1  /* MIXMODEL.C
2     Demonstrates calling a far function from a small model program.
3  */
4  #include <math.h>
5  #include "mycomp.h"
6
7  void main(void)
8  {
9    struct complex c;
```

continues

LISTING 5.2. CONTINUED

```
10    c.x = 3;
11    c.y = 1;
12
13    put_complex( &c );
14
15  }
```

LISTING 5.3. HEADER FILE FOR THE LARGE MODEL MODULE.

```
 1  /* MYCOMP.H
 2  */
 3  #include <math.h>
 4
 5  void far put_complex ( struct complex far *x );
```

LISTING 5.4. THE LARGE MODEL MODULE CONTAINING THE `put_complex()` FUNCTION.

```
 1  /* MYCOMP.C
 2     Contains put_complex() compiled under the large memory model.
 3  */
 4
 5  #include <stdio.h>
 6  #include <math.h>
 7  #include "mycomp.h"
 8
 9  void far put_complex ( struct complex far *x )
10  {
11    printf("%f+%fi", x->x, x->y );
12  }
```

Another way, not shown in the sample listings, is to create your own header or function prototype for `put_complex()`. Suppose that mycomp.h had contained this definition:

```
void put_complex ( struct complex *x );
```

When the compiler sees this definition while compiling a small code model program, it will generate near function calls to `put_complex()`. You can manually fix this by creating your own prototype:

```
void far put_complex ( struct complex far *x );
```

You might insert this prototype directly into your source code, or you could copy mycomp.h to a new file, edit the header, and then include the revised header in your program.

After you've done all this, you still need to do some special work to get this program to link properly. If you try to compile and link this application using a conventional approach, you will get an abnormal program termination due to the linker's confusion when trying to link the correct library for the call to `printf()`. To understand the problem, you can compile this program by typing

```
bcc -c -ml mycomp.c
bcc -ms mixmodel.c
```

These commands will compile mycomp.c into a large model object file and mixmodel.c into a small model object file. The program will link but it won't execute correctly because the linker brings in a small model library version of `printf()`. To get this to run, you need to reverse the order of the compilation and link:

```
bcc -c -ms mixmodel.c
bcc -ml -emixmodel.exe mycomp.c mixmodel.obj
```

The second command compiles mycomp.c as a large model program and links in the previously compiled mixmodel.obj, producing the executable mixmodel.exe.

You can see that mixed-model programming must be done sometimes, but you also can see how mixing memory models can cause problems.

USING THE *NEAR* MODIFIER

When a module is compiled using a large code model, each call to a function within the module is also made as a far call. But because these functions are all located within the same code segment, it might not be necessary to use far calling conventions for all these functions. When you have functions that are

used only within the module and that are not called from outside the module, you can use the `near` keyword to change these to near functions. Here's an example showing the `near` keyword used in a function prototype:

```
unsigned int near compute_elevation( double latitude, double longitude);
```

Functions called as near procedures are more efficient than those called as far procedures. Only two bytes are used for the function's address (instead of four), so the underlying CALL machine instruction is shorter. Because the function is in the same segment, the current value of `cs` is not saved to the stack, saving two extra bytes of space on the stack and speeding up the push and pop of the return address.

CREATING A .COM PROGRAM

The only advantage a .com executable file has over its cousins, the tiny or small memory model .exe files, is that a .com file is slightly smaller than the .exe file because it contains no *relocation information*. Relocation information is used by DOS when loading an .exe file into memory. Relocation allows memory segments to be dynamically repositioned in memory. The .com file does not have a relocation table, so the file size is somewhat smaller. For this reason, if you are creating small programs, it's simplest to use the tiny or small memory models directly and not worry about creating a .com file.

If you want to create a .com file, you need to use the command-line compiler (or manually specify options to the tlink linker program). Using the command-line compiler is definitely easier. Listing 5.5 is a small program that is suitable for compilation under the tiny model. To compile and link, use the bcc command-line compiler, with these options:

```
bcc -mt -lt tiny.c
```

The `-mt` switch selects the tiny memory model (this is required for creating a .com file) and the `-l` option passes optional parameters to the linker. Here, the `-lt` option tells the linker to produce a .com output file.

LISTING 5.5. A SAMPLE PROGRAM THAT CAN BE COMPILED INTO A .COM FILE.

```
/*
  TINY.C
  Demonstrates creation of a .com file.
  Compile using bcc -mt -lt tiny.c
*/

#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  char input_char;
  FILE *input_file;

  if (( input_file = fopen( "tiny.c", "rt" )) == NULL)
  {
    printf("Problem opening tiny.c source file.\n");
    exit( 1 );
  };

  while ( (input_char = fgetc( input_file )) != EOF)
  { putchar( input_char ); };

  fclose( input_file );
}
```

STORING DATA

Where and how you define variables and data structures for your program's data influences how much memory your application will require. Each program has three basic types of data:

- Local or automatic duration variables such as function parameters and variables declared within functions. Memory space for local variables is created automatically upon entry to a function and is discarded at the function's exit. Consequently, these variables are useful for storing data that is local to a function.

- Static duration variables are those defined using the `static` keyword or those that are defined within a source file but not inside a function. Static variables occupy memory space for the entire duration of the program's execution.
- Dynamic duration variables are those that are dynamically allocated by calling memory management functions. You decide when such variables should be created and when they should be destroyed. Dynamic variables are allocated space from the heap area, which is the memory left over after allocating space for program code, static data, and the stack.

Local variables are allocated space on the program stack when a function is entered. When the function exits, it discards the excess bytes by subtracting the size of the local variables from the stack. Hence, the allocation and deallocation of space for local variables is very fast. Keep in mind that the total stack space is limited to a maximum of 64K, and it may often be less, depending on the memory model in use and the memory available during execution.

Static variables hang around for a program's entire duration. They are especially useful within functions because they can retain information between function calls (unlike local variables that go away after each call) and because you can preinitialize static variables. A major drawback of static variables, especially if you share your code, is that if each module tracks a good deal of data in static allocations, the sum of their requirements may leave little room for the rest of the program. A few years ago, when I was working on PFS: First Choice, I had to use a standardized corporate library for certain routines. This library, however, was written so that all of its data tables were kept in static variables. Before I'd written a line of code, 21K of my 64K maximum had been eaten up by this library, potentially making my life as a programmer very difficult. The solution, fortunately, was to persuade the library developer to put the static tables into dynamic allocations. This solved the problem and gave nearly all of the 64K allocation back to the First Choice application.

A SUGGESTION REGARDING LOCAL VARIABLES

For most programs, local variables are easy to use and conserve memory because their space is recycled each time a function exits. Occasionally, though, the use of local variables can bite you. If you have a series of functions that call each other in a chain—so that `f1()` calls `f2()`, `f2()` calls `f3()`, `f3()` calls `f4()` and so on, for example—and each needs to allocate space on the stack for local variables, you might find yourself in the midst of an out-of-stack-space error. This happens only if your functions allocate space for large variables such as arrays, or if recursive function calls are used. Whenever you use a recursive function, try to keep local variable storage to a minimum, because each call creates a new copy of the locals. If you encounter an out-of-stack-space error, look at the depth of your *call stack*—the sequence of function calls that gets you to a particular point in the program. In the IDE, use the Debug | Call stack menu selection to see how deep your function calls have become. Check each of the procedures in the call stack to ensure that you are not cumulatively exceeding the available stack space for local storage.

USING DYNAMICALLY ALLOCATED MEMORY

Dynamic memory allocation provides you with direct control of your program's memory requirements. A dynamic allocation can have temporary or permanent duration, depending on the application requirements. It's entirely up to you to decide when the memory should be allocated and when it should be discarded. Depending on the memory model in use, you can also exceed the 64K maximum limit of static variables.

Unlike local and global variables, dynamic allocations can be determined while your program is running. This enables your program to allocate memory space tailored exactly to the requirements of the task at hand. Sometimes it is not possible to determine in advance how much memory to allocate to a particular data structure. For instance, an array type sets aside a fixed amount of memory. If your program does not need the entire array, the excess memory

space is wasted. If your program needs more, you must recompile the program with a larger array size. Dynamic memory, as the name suggests, is allocated when your program is executing. Your program can allocate as much or as little memory as it requires for the task it is working on.

Using dynamic memory allocations demands a high degree of precision in your programming. You must use pointer types and casting operators and you must ensure proper management of your memory blocks. It is remarkably easy to encounter pointers that do not point where you think they do or to inadvertently access nonexistent memory blocks. Such wayward pointers can at least cause program errors and at worst cause system crashes and destruction of data. Allocating dynamic memory requires that you be familiar with C pointers. If you are not familiar with C pointer types, you should consult Chapter 4, "Using Pointers and Derived Types," of *Using Borland C++ 3*, Second Edition, or *Using Microsoft C/C++ 7*, both published by Que Corporation.

THE HEAP

Dynamic memory is allocated from an area of storage called the *heap*. When your program is running, its memory layout might look like that shown in Figure 5.2. The exact layout differs somewhat, depending on the memory model the program uses. (See "DOS Memory Management" in the *Borland C++ Programmer's Guide* for details on the different memory layouts.) For the large data models, the heap refers to the area of memory existing beyond the program's stack and running up to the top of available memory. The heap is essentially all the memory left over after loading your program. When you request a dynamic memory allocation, the C or C++ memory management system carves out a chunk of memory from the heap and returns a pointer to the memory block.

In the small data models, the heap is the area of memory that is shared with the stack but is not currently in use by the stack. It is addressed using a near pointer. Depending on the memory model—tiny, small, or medium—the segment containing the near heap may be shared, with both the stack and static data providing considerably less than the 64K maximum of near heap space.

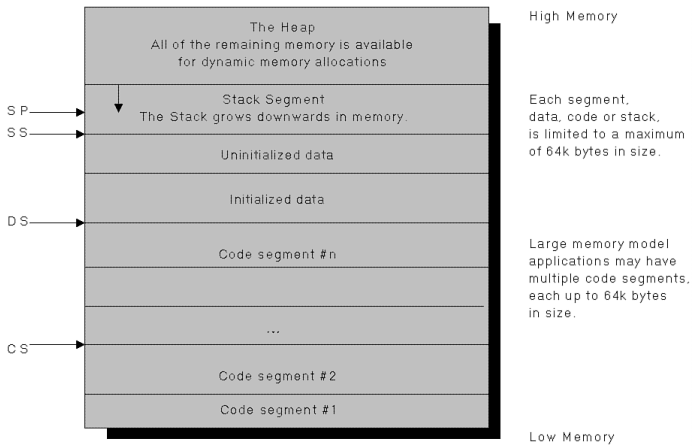


Figure 5.2. The memory layout of a large model program.

MALLOC() AND RELATED ROUTINES

The Borland C++ 3.0 Library provides several memory management routines for the dynamic allocation and deallocation of memory. Many of these routines are industry standards and may be used with compilers from different vendors and across operating systems. Most of the routines are declared in stdlib.h or alloc.h. In addition to the standard and traditional memory allocation routines, don't overlook the use of C++'s `new` and `delete` constructor and destructor. By adding the .cpp extension to your source files, you can use many of the C++ features without necessarily graduating to all that C++ has to offer. The use of `new` and `delete` is described in the section "Using C++ `new`/`delete` for Simple Data Types."

The primary dynamic allocation routines are `malloc()`, to request a memory allocation, and `free()`, to discard a dynamically allocated memory block. There are also a number of related routines including `alloca`, `allocmem`, `calloc`, `coreleft`, and `realloc`.

`malloc()` allocates a memory block of a requested size, up to a maximum of 64K, returning a pointer to the block. If you need to allocate objects larger than 64K, you should compile using the huge memory model and call `farmalloc()` instead. `farmalloc()` is described in the section "`farmalloc()` and Related Routines."

`malloc()` is defined with this prototype:

```
void * malloc( size_t size );
```

size is the number of bytes requested for the allocation. Note that `malloc()` returns a `void *` type, which is a pointer to a generic type. For the generic pointer to be usable, you must recast the result to the type of your pointer. For example, to allocate an 80-byte character string, type

```
char * str80;
...
str80 = (char *) malloc( 80 );
```

The call to `malloc( 80 )` reserves an 80-byte segment from the heap and returns a pointer to the start of those 80 bytes. If there is insufficient memory available, `malloc()` returns a null pointer. Note the use of the `(char *)` type cast. Also note that `str80` is *not* declared as

```
char * str80[80];
```

The latter definition creates an array of 80 pointers to type `char`, not a pointer to an 80-byte string.

When you are declaring a pointer type, it is a good idea, although it is not required, to preinitialize the pointer to `NULL`. Without preinitialization, the value of the pointer is random, and it's easy to mistakenly use such a pointer without realizing your error. To initialize a pointer, you may either set it to `NULL` as part of the declaration, or you can initialize it as part of the first statements in your program. In the former case, you initialize a declaration by typing

```
char * str80 = NULL;
```

Most routines either ignore null pointers or issue an error code or error message. Initializing the pointer helps you to quickly identify recalcitrant pointers.

When you no longer need a memory allocation, you should discard it by calling `free()`. Failure to discard a memory block results in wasted memory. Blocks that are discarded by calling `free()` become available for new dynamic memory allocations. To discard a block, pass the pointer to `free()`, like this:

```
free( str80 );
```

Listing 5.6 illustrates the use of `malloc()` and `free()` to create an extra-large file buffer for fast file reading. This sample program uses `malloc()` to allocate a large buffer and then calls `setvbuf()` to associate the buffer with the open file. When you increase the size of the buffer, file I/O can be performed at a much higher speed.

LISTING 5.6. A PROGRAM THAT USES `malloc()` AND `free()` TO CREATE AN EXTRA-LARGE FILE BUFFER FOR FAST FILE I/O.

```
/*
  MALLOC.C
  Demonstrates use of malloc() to allocate a large buffer
  for use with file i/o.
*/

#include <stdio.h>
#include <stdlib.h>
#include <io.h>

#define BUFSIZE 32000

void main(void)
{
  FILE *in_file;
  char * buffer;
  char textline[83];

  in_file = fopen("data.txt", "rt");

  buffer = (char *)malloc( BUFSIZE );

  if (setvbuf( in_file, buffer, _IOFBF, BUFSIZE))
    printf("Set up of file buffer failed.\n");
  else
  while( fgets( textline, 82, in_file ) != NULL )
    puts(textline);

  fclose( in_file );
};
```

COMMON PROBLEMS USING *MALLOC()* AND *FREE()*

When you discard a memory block by calling `free()`, `free` does not set the pointer to null. After the statement

```
free( str80 );
```

`str80` will still contain a pointer to where the memory block had been allocated. In many instances, you can continue to use this pointer, even though you should not attempt to do so. Until the memory is reclaimed for other uses, it might remain intact, and then suddenly—*boom!* Your program hangs due to an invalid memory reference.

When you use `malloc()` to allocate a memory block, you should always test to see that the returned pointer is not null. `malloc()` returns null to indicate that it is out of memory. Most programs that I have examined (and many that I've written) do not regularly check the return result. In the event of an out-of-memory condition, this will quickly cause your software to fail. Always check the return result; do not assume that you have enough memory.

You can get an indication of the amount of memory remaining by checking the result returned by the `coreleft()` function. `coreleft()` returns an unsigned integer value for the tiny, small, and medium models, or a long for compact, large, and huge models, indicating the number of bytes available between the current top of the heap and the top of the stack. By checking `coreleft()` first, you may be able to safely allocate a number of blocks without checking the return result from `malloc()`.

Finally, it is common to create dynamic allocations and assign them to local pointers. If you fail to discard the memory block prior to exiting the function, however, the allocated memory will be unavailable to your application for the duration of the program's execution. This occurs because the local pointer variable is itself of local duration. When the function exits, the local pointer goes away, but the memory it points to remains allocated. Without the local pointer, you have no way to use or free that dangling block of memory. See the section "Using `alloca()`" for another way to handle local pointers.

USING *CALLOC()*

`calloc()` allocates and returns a pointer to a memory block just like `malloc()`. The difference between `calloc()` and `malloc()` is that `calloc()` has two parameters and is best suited for allocating memory for use as an array:

```
void * calloc(size_t nitems, size_t size);
```

The parameter *nitems* specifies the number of items in the array, and *size* specifies the width in bytes of each item. In effect, `calloc()` is the same as calling `malloc(nitems * size)`, except that `calloc()` also initializes the allocation to all zeros. Listing 5.7 is an example of this, using `calloc()` to allocate an array of integers. Note the use of `sizeof(int)` to obtain the size, in bytes, of an integer value. In lines 19–20, the allocation is indexed as an array using the notation `*(ptr+index)` where *index* corresponds to an element of the array:

```
19     for (i=0; i<ARRAYSIZE; i++)
20      *(dynamic_array+i) = i;
```

This operation works because of the similarities between C pointers and C arrays. You can demonstrate this by writing a short section of code to create a pointer to an array, and then index the array using the pointer, like this:

```
int * ptr;
int anarray[20];
...
ptr = &anarray[0];
for (i=0; i<20; i++) *(ptr + i) = i;
for (i=0; i<20; i++) printf("%d  ", anarray[i]);
```

In C, the expression `anarray[i]` is exactly equivalent to `*(anarray + i)`. See Listing 5.7 for an example of a dynamically allocated array.

LISTING 5.7. EXAMPLE USING THE `calloc()` FUNCTION.

```
 1  /*
 2    CALLOC.C
 3    Demonstrates use of calloc().
 4  */
 5
 6  #include <stdio.h>
 7  #include <alloc.h>
 8
 9
10  #define ARRAYSIZE 100
```

continues

LISTING 5.7. CONTINUED

```
11
12  void main(void)
13  {
14    int i;
15    int * dynamic_array;
16
17    dynamic_array = (int *) calloc(ARRAYSIZE, sizeof(int) );
18
19    for (i=0; i<ARRAYSIZE; i++)
20      *(dynamic_array+i) = i;
21
22    for (i=0; i<ARRAYSIZE; i++)
23      printf("%d  ", *(dynamic_array+i) );
24
25    free( dynamic_array );
26
27  }
```

USING *REALLOC()*

When you must change the size of an existing dynamic memory block, you can either free the block and allocate a new one, or you can call the `realloc()` function. In many instances, calling `realloc()` to change block size is more efficient than calling `free()` and then `malloc()` again, particularly if you are shrinking an existing block. You call `realloc()` by passing the original pointer and a new size. `realloc()`'s function prototype is

```
void * realloc( void *block, size_t size );
```

where *`block`* is the pointer to the existing memory allocation and *`size`* is the new desired size.

When you shrink an existing block, `realloc()` merely alters the size of the block. If you need to increase the size of the block, `realloc()` might be able to use the adjacent memory space if it is available. If not, `realloc()` finds a new memory block and copies the bytes from the existing allocation to the new block, returning the new pointer as its result. If insufficient memory is available to change the block's size, `realloc()` returns null.

USING *ALLOCA()*

`alloca()` is a special-purpose function whose purpose is to eliminate the problems related to using local pointers to keep track of allocated blocks. However, `alloca()` comes with its own host of problems and is probably best left unused in your programs. From the interface perspective, `alloca()` is identical to `malloc()`:

```
void * alloca( size_t size );
```

The difference is that `alloca()` places the allocation on the stack as part of the function's local variables. As a side effect, when the function exits, the allocation is automatically thrown away, as are the local variables. If you assign `alloca()`'s result to a local pointer, the allocated block and the local pointer are both discarded when the function exits, eliminating any concern about failing to free a locally allocated `malloc()` memory block.

Because `alloca()`'s drawbacks significantly outweigh its benefits, it is probably a solution that is best avoided. In addition to using precious stack space, `alloca()` doesn't work at all if your function has no local variables. In such a case, your program will crash. If you assign the result from `alloca()` to a global pointer, you are setting the stage for disaster. As soon as the function exits, the global pointer still contains a pointer to the now nonexistent stack allocation. Who knows what that pointer might be used to clobber?

DOS MEMORY ALLOCATIONS

The Borland C++ library (use the dos.h header) includes four nonstandard memory management routines: `allocmem()`, `_dos_allocmem()`, `setblock()`, and `freemem()`. These functions use the internal DOS INT 21H subfunction 0x48 to allocate memory blocks in a form that is incompatible with `malloc()` and `farmalloc()`. In other words, if you use `allocmem()` or `_dos_allocmem()`, you must not use `malloc()` or `farmalloc()` because these memory schemes are incompatible with one another.

`allocmem()` and `_dos_allocmem()` both use INT 21H subfunction 0x48 to allocate memory but behave slightly differently in low-memory situations. The prototypes for these functions are

```
int allocmem( unsigned size, unsigned *segp );
unsigned _dos_allocmem( unsigned size, unsigned *segp );
```

size is the requested block size in paragraphs and *segp* is the address of a word that will be assigned the memory segment containing the allocated block. A *paragraph* is 16 bytes. To convert bytes to paragraphs, divide the desired number of bytes by 16, then add one in order to round up.

If you must change the size of a DOS memory block, use either `setblock()` or `_dos_setblock()`. These functions are defined as

```
int setblock( unsigned segx, unsigned newsize );
unsigned _dos_setblock( unsigned newsize, unsigned segx, unsigned *maxp );
```

segx is the segment address returned by `allocmem()` or `_dos_allocmem()` when the memory was allocated. *newsize* is the new requested block size, in paragraphs. `setblock()` returns –1 if the reallocation is successful. If the reallocation is not successful, `setblock()` returns the size (in paragraphs) of the largest available memory block. If `_dos_setblock()` succeeds, it returns 0 (note the inconsistency with `setblock()`). If it fails, it returns a DOS error code, sets the global variable's `errno` to `ENOMEM` (a constant in dos.h), and sets the word pointed to by *maxp* to the size of the largest available memory block.

To deallocate a block of DOS memory, use `freemem()` to discard a block that was allocated with `allocmem()` and use `_dos_freemem()` to free a block that was allocated with `_dos_allocmem()`. These functions each have a single unsigned *segx* parameter, corresponding to the memory segment to be freed:

```
int freemem( unsigned segx );
unsigned _dos_freemem( unsigned segx );
```

FARMALLOC() AND RELATED ROUTINES

The far memory allocation routines are nearly identical to the basic set of memory allocation routines, except that `farmalloc()` and its related functions may allocate blocks that are greater than 64K. You should use `farmalloc()` only with programs compiled as compact, large, or huge. `farmalloc()` allocates from the far heap (as compared to the near heap of the small data model programs). You should use huge pointers (or use the huge memory model) when accessing blocks greater than 64K. You may use ordinary far pointers for all smaller block sizes.

To discard a block, call `farfree()` instead of `free()`. To change a block's size, call `farrealloc()` in place of `realloc()`. Use `farcalloc()` in place of `calloc()`. Finally, to

obtain the number of bytes of free memory, call `farcoreleft()`, which returns a long value indicating the number of bytes between the top of the stack segment and the current top of the heap. Due to calls to `farfree()`, there may be unused blocks within the heap, and these free spaces are not reflected in the value returned by `farcoreleft()`.

A sample program illustrating the use of the far memory allocation routines is shown in the dmalloc.c program of Listings 5.8 through 5.10. dmalloc.c implements a simple discardable memory allocation scheme. A discardable memory scheme is one that distinguishes between locked and discardable memory blocks. When you allocate a memory block by calling any of the standard C library routines, the block remains allocated until you discard it by calling `free()` or `farfree()`. Sometimes, though, you might not want to tie up all of memory, especially if you are storing items in the memory blocks that can be easily re-created.

Consider a file copy program that displays a source and a destination directory and enables you to select and copy files from the source to the destination. For efficiency, you would probably read the disk directory into a memory-based data structure. As soon as the files have been selected, you need to allocate some large file buffers to provide for fast file copying. If many TSRs or network interface software are installed on your system, it might not be possible to create large file buffers. The solution is to discard the directory data structures to make room for the file copy operation. As soon as the file copy is completed, the original disk directory information can be recreated by scanning the file directory again.

A convenient and automatic way to handle this type of memory allocation is to distinguish between memory blocks that must remain *locked* in memory and those that can be *discarded*. dmalloc.c implements a layer above the normal `farmalloc()` and `farfree()` routines that manages *locked* and *discardable* memory segments. Its primary interface uses a call to `ddm_allocate()` to allocate a block, and `ddm_deallocate()` to discard a block. The `ddm_` prefix is short for *dynamically discardable memory*.

To allocate a block of memory, use `ddm_allocate()`, passing to it the address of the pointer that will own the block, the block's requested size, and the block's type. `ddm_allocate()`'s function header is

```
void * ddm_allocate( void * p, unsigned int size, int block_type );
```

The *block_type* parameter should use the ddm_LOCKED or ddm_DISCARDABLE macro symbols. Use ddm_DISCARDABLE to mark memory blocks as disposable in the event that a subsequent call to ddm_allocate() would cause an out-of-memory condition. When an out-of-memory condition is encountered, ddm_allocate() throws away sufficient discardable memory blocks to make new memory allocations. When a discardable block is disposed, the owner's pointer is set to NULL so that subsequent code can determine when a block has been discarded. This means that any routine that uses a discardable memory block must not assume that its memory block is valid; it should first check that the pointer to the block is non-NULL.

Listing 5.8 contains dmalloc.h, the header file for the dynamically discardable allocation routines. For the purposes of this sample program, a simple array maintains a list of discardable blocks. The size of this array, set by the MAXDISCARD macro (see Listing 5.8, line 5), determines the maximum number of permitted discardable blocks. The number of locked blocks is limited only by the available memory. A more advanced implementation would track discardable memory blocks using a dynamically allocated list that could accommodate any number of discardable segments.

LISTING 5.8. THE HEADER FILE FOR THE DISCARDABLE MEMORY ALLOCATION SCHEME.

```
 1  /*
 2    DMALLOC.H
 3  */
 4
 5  #define MAXDISCARD 100     /* Max # of discardable memory segments */
 6  #define ddm_DISCARDABLE 1  /* Parameters to ddm_allocate() */
 7  #define ddm_LOCKED 0
 8
 9  void ddm_initallocate(void);
10  void * ddm_allocate( void * p, unsigned int size, int block_type );
11  void ddm_deallocate( void * ptr );
```

To use these routines, you must first call ddm_initallocate() prior to using ddm_allocate() for the first time. Although the implementation uses the farmalloc() routine, you should limit each block to less than 64K because the block size parameter and internal values used by the dmalloc module are unsigned integers. If you want to use these routines to allocate huge data blocks (greater than 64K), you need to change the unsigned int parameter and certain other

variables to the `long` data type. To discard a memory block, call `ddm_deallocate()`. Listing 5.9 presents the implementation of the memory manager. See testdm.c in Listing 5.10 for a sample program that uses the discardable memory block manager.

LISTING 5.9. THE DISCARDABLE MEMORY MANAGER LISTING.

```
/* DMALLOC.C
   Demonstrates use of the farmalloc and farfree functions by implementing
   a simplified discardable memory block structure.
   Compile using the large memory model.

   A simple improvement to this code would be to store the owner and size
   values within the allocated memory block. This would reduce the
   memory requirements of the discard_list. You could also get rid of
   the discard list altogether by storing a discardable flag byte within
   each allocated memory block, and stringing the blocks together in a list.
   This way, there would be no fixed limit on the number of discardable
   blocks that could be allocated.
*/

#include <alloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>


#include "dmalloc.h"


/* Contains info about each discardable block */
typedef struct tagBlockInfo {
  void * block;
  void * owner;
  unsigned int size;
};
struct tagBlockInfo discard_list[MAXDISCARD];

long total_discardable; /* Total # of bytes in discardable blocks */
int last_discarded;     /* Last table entry of discarded block */
```

continues

LISTING 5.9. CONTINUED

```
38  /*************************
39  Function:     ddm_initallocate()
40  Purpose:        Must be called prior to using these allocation routines.
41  */
42  void ddm_initallocate(void)
43  {
44    int i;
45
46    for(i=0; i<MAXDISCARD; i++ ) discard_list[i].block = NULL;
47    total_discardable = 0L;
48    last_discarded = 0;
49    return;
50  };
51
52
53
54  /*************************
55  Function:     ddm_add_to_list
56  Purpose:        Internal routine only.  Adds an allocated block
57                into the discard list.
58  */
59  int ddm_add_to_list ( void * ablock, void * theowner, unsigned int size )
60  {
61    int i;
62    unsigned int total_scanned = 0;
63
64    /* Look for a free entry in the discard list */
65    i=0;
66    while( discard_list[i].block != NULL)
67    {
68      if (i++ == MAXDISCARD) i=0;
69      if (total_scanned++ == MAXDISCARD)
70        return 1; /* Error, out of discardable table space */
71    };
72
73    /* Having found a free entry, set up the record information about the
74       allocation */
75    discard_list[i].block = ablock;
76    discard_list[i].owner = theowner;
77    discard_list[i].size = size;
78    total_discardable += size;
79
80    return 0;
81  };
82
83
```

```
 84
 85  /*************************
 86  Function:     discard_entry
 87  Purpose:        Internal routine only. Deletes the item denoted by entry
 88                from the internal discard_list.
 89  */
 90  void discard_entry(int entry)
 91  {
 92    void * p;
 93    long * backpointer;
 94
 95    /* Set the owner's pointer to NULL, letting it know that its block
 96       was discarded. The casting to a long is nonportable and is used
 97       to coerce the compiler into copying the 4-byte seg:off pair. */
 98
 99    (void *) backpointer = discard_list[entry].owner;
100    *backpointer = 0L;
101
102    /* And free the block */
103    farfree( discard_list[entry].block );
104
105    /* Remove entry from the discard list */
106    discard_list[entry].block = NULL;
107
108    /* Remove the block's size from your discardable memory total */
109    total_discardable -= discard_list[entry].size;
110
111    return;
112  };
113
114
115
116
117  /*************************
118  Function:     ddm_can_discard
119  Purpose:      If possible, calls discard_entry() to throw away
120                discardable memory blocks until sufficient space
121                is available for the new allocation. Note the use of
122                the last_discarded variable. By keeping track of the
123                last entry that was discarded, this routine will begin
124                searching at the next entry past that--this way it
125                doesn't throw away the last block allocated!
126  */
127  int ddm_can_discard(unsigned int desired_size)
128  {
129    int total_scanned;
130    long total_discarded;
131
```

continues

LISTING 5.9. CONTINUED

```
132   if (desired_size > total_discardable)
133     return 1; /* Out of memory, really */
134
135   total_discarded = 0;
136   total_scanned = 0;
137   /* Scan the discardable list and get rid of some entries */
138   do {
139     if (last_discarded++ == MAXDISCARD)
140       last_discarded = 0;
141     if (discard_list[last_discarded].block != NULL) {
142       total_discarded += discard_list[last_discarded].size;
143       discard_entry(last_discarded);
144       if (total_discarded >= desired_size)
145     return 0; /* All done; report success */
146     };
147   } while (total_scanned++ < MAXDISCARD);
148
149   return 1; /* Unable to find sufficient free memory; out of memory */
150 };
151
152
153
154 /**************************
155 Function:     ddm_allocate
156 Purpose:      Allocates a block of memory of size 'size' bytes,
157               returning a pointer to the allocated block. Parameter
158               p is the address of the pointer that will hold the pointer
159               to the allocated block; this address is stored in the
160               discard list so that the owning pointer can be set to null
161               if the block is discarded. block_type is either
162               ddm_DISCARDABLE if this block should be a discardable block,
163               or ddm_LOCKED if the block is not discardable.
164 Returns:      Pointer to allocated block, or NULL if unable to allocate.
165 */
166 void * ddm_allocate( void * p, unsigned int size, int block_type )
167 {
168   void * new_block;
169
170   do {
171     new_block = (void *) farmalloc( size );
172
173     /* When unable to allocate more memory, attempt to discard some
174        memory blocks */
175     if (new_block == NULL)
176       if (ddm_can_discard( size ) == 1)
177     return NULL; /* Was unable to discard any more blocks */
178       /* If get here, then a sufficiently large block(s) was discarded */
```

```
179    } while (new_block == NULL);
180
181    if (block_type == ddm_DISCARDABLE) {
182      /* Then add it to the discardable blocks list */
183      if (ddm_add_to_list( new_block, p, size ) == 1)
184        /* If get a return result of 1, then out of space in discard list */
185        { return NULL; };
186    };
187
188    return new_block;
189  };
190
191
192
193  /*************************
194  Function:     ddm_discardable
195  Purpose:      Internal routine that determines if a mem. block pointed
196                to by ptr is discardable.
197  */
198  int ddm_discardable( void * ptr )
199  {
200    int i;
201
202    /* Search the discard list. If ptr is in the discard list, then
203       delete it from the discard list. */
204    for( i=0; i<MAXDISCARD; i++ )
205      if (discard_list[i].block == ptr) {
206        discard_entry( i );
207        return 1;
208      };
209    return 0;
210  };
211
212
213
214  /*************************
215  Function:     ddm_deallocate
216  Purpose:      Throws away the allocated block.
217  */
218  void ddm_deallocate( void * ptr )
219  {
220    int i;
221
222    if ( ptr != NULL )
223    /* If the block is discardable, it will be deleted by the ddm_discardable
224       code. Otherwise, if it is LOCKED, delete using farfree() */
225      if(!ddm_discardable( ptr ))
226        farfree( ptr );
227  };
```

LISTING 5.10. A SAMPLE PROGRAM THAT DEMONSTRATES AND TESTS THE DISCARDABLE MEMORY MANAGER.

```
 1  /*
 2    TESTDM.C
 3    Tests and demonstrates usage of the dmalloc.c routines.
 4    Compile using the large memory model.
 5  */
 6  #include <stdio.h>
 7  #include <alloc.h>
 8  #include <stdlib.h>
 9  #include "dmalloc.h"
10
11  void main(void)
12  {
13    char * p[20];
14
15    int i;
16
17    ddm_initallocate();
18    printf("\nMemory(before)=%lu\n", farcoreleft() );
19
20    for (i=0;i<20;i++)
21      if( (i % 2) == 0)
22      {
23        p[i] = ddm_allocate( &p[i], 4000, ddm_DISCARDABLE );
24        if (p[i] == NULL)
25      { puts("Out of memory.\n");
26      exit(1);
27      };
28      }
29      else
30      {
31        p[i] = ddm_allocate( &p[i], 4000, ddm_LOCKED );
32        if (p[i] == NULL)
33        { puts("Out of memory.\n");
34        exit(1);
35        };
36      };
37
38    puts("Test in progress.\n");
39
40    for (i=0; i<20; i++)
41      if (p[i] != NULL)
42        ddm_deallocate( p[i] );
43
44    printf("Memory(after)=%lu\n", farcoreleft() );
45
46  };
```

USING C++ *NEW/DELETE* FOR SIMPLE DATA TYPES

C++ provides an improved memory allocation scheme that does not require calling `malloc()` or `farmalloc()` or recasting the returned result (like `malloc()`) nor does it use the `free()` or `farfree()` functions. Instead, C++ defines the `new` and `delete` operators for allocating memory blocks and instantiating objects. `new` has the form

```
pointer = new name-initializer;
```

Here's an example using `new` to allocate space for an integer quantity:

```
int *p;
...
p = new int;
```

For simple data types, a call to `new` is equivalent to calling `malloc()` like this:

```
(* data_type ) malloc( sizeof( data_type ));
```

Note that when using `new`, you do not need to cast the returned value to the type of the pointer. Memory that is allocated using `new` is discarded by calling `delete`, such as in this example:

```
delete p;
```

Even if you are not accustomed to writing C++ code, you can still use `new`. The Borland C++ compiler expects C programs to have a .c extension on the filename and C++ programs to have a .cpp extension. You can set your source filename extensions to .cpp and continue to write C code because C is a subset of C++. As soon as you have enabled the C++ language use in the compiler, you may use some or all of the C++ language features, together with your existing C code. Indeed, as the programming community shifts towards C++, the use of `new` and `delete` is rapidly replacing the use of the older `malloc()` and `free()` functions, even for non-object-oriented programming.

Another benefit of `new` is that it may preinitialize your allocation to a value that you specify. Again, using the pointer to an `int` data type, the allocation may be initialized to a particular value, such as 999, like this:

```
p = new int(999);
```

The `new` operator works for array data types too. Listing 5.11 shows an example that allocates an 80-byte-long character string array. Again, note the similarity to calling `malloc( sizeof( char[80] ))`. This program also illustrates the use of the `delete` operator to discard the allocation owned by the pointer `p`.

Unfortunately, preinitialization is not available for array types.

To allocate a multidimensional array, indicate each array dimension as

```
p = new char[10][80];
```

In general, each array dimension must be a compile-time constant, except for the far left array dimension, which may be a variable. This means that for single-dimension arrays, the size of the array be set at runtime. For instance, it is acceptable to write

```
int size = 80;
char * p;
...
p = new char(size);
```

Listing 5.11 is a short program illustrating the use of `new` and `delete` to allocate a string variable.

LISTING 5.11. A SAMPLE PROGRAM THAT ALLOCATES AN ARRAY USING THE C++ new OPERATOR.

```
 1  // char80.cpp
 2  #include <stdio.h>
 3  #include <string.h>
 4
 5  void main(void)
 6  {
 7    char * p = NULL;
 8
 9    p = new char[80];
10
11    strcpy( p, "I'm a character string allocated using the new operator." );
12
13    printf("%s\n", p );
14
15    delete p;
16
17  }
```

TRAPPING ALLOCATION ERRORS USING *SET_NEW_HANDLER*

If `new` is unable to allocate the desired memory block, `new` returns `NULL`. You should compare the `new` return result to `NULL` to ensure that the memory

allocation is completed. If you want to, you can create your own out-of-memory handler and use the `set_new_handler()` function to install your memory handler. As soon as you have done this, if `new` encounters an out-of-memory condition, it calls your handler. Listing 5.12 shows how simple it is to detect out-of-memory conditions and display your own error message. You may also use this technique to intercept the out-of-memory condition and perform some memory cleanup, throwing away unnecessary memory blocks. Then, let the statement that called `new` try its allocation request again.

LISTING 5.12. AN EXAMPLE OF THE `set_new_handler()` FUNCTION.

```
 1  // newerror.cpp
 2  // Demonstration of setting up an out-of-memory error handler
 3  // written using minimal C++.
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <string.h>
 7  #include <new.h>
 8
 9  void out_of_mem( )
10  {
11    printf("You have run out of memory.\n");
12    printf("Program is terminating.\n");
13    exit(1);
14  };
15
16  void main(void)
17  {
18    char * p = NULL;
19    unsigned index;
20
21    set_new_handler( out_of_mem );
22
23    for ( index=0; index < 65535; index++ ) {
24      p = new char[500];
25    };
26
27    printf("Successfully allocated a huge amount of memory!\n");
28    // Note: allows program termination to recover memory.
29  }
```

POINTER PROBLEMS AND MEMORY TRASHERS

Earlier I pointed out some situations that can cause unpredictable program activity when you are using pointers. Probably the most common error is to make use of a pointer after its memory block was discarded. The standard deallocation routines, such as `free()`, do not set the pointer to `NULL`. As a safety feature, you might want to include code, at least during the checkout of your program, to reset pointers to null after discarding memory. It's easy to check for `NULL` as part of your debugging and testing; you can use the `assert()` function (see the paragraph following Listing 5.13) to conditionally halt your program if a pointer is not `NULL`, helping you to track down problems.

You might set a pointer to `NULL` after each call to `free()` by using conditional compilation directives so that your checkout code can be removed in the final version of your software. Or, you can hide the call to `free()` inside a macro. Redefine the macro to either reset the pointer or leave the pointer as is. Listing 5.13 shows a sample macro for performing this function.

LISTING 5.13. EXAMPLES OF TWO WAYS TO SET A POINTER TO NULL.

```
// NULLPTR.C
// Ways of setting a pointer to NULL after discarding
// a memory block.
#include <stdio.h>
#include <alloc.h>
#define CHECKOUT 1
#define FREE( x ) free(x); x=NULL

void main (void)
{

  char * p;

  p = malloc( 10000 );

  free( p );
#if CHECKOUT
  p = NULL;
#endif
  if (p == NULL) puts("p was reset to NULL using first method.\n");
  else puts("p was not reset to NULL due to conditional code removed.");

  p = malloc( 10000 );
// Alternate form using a macro layer above free():
  FREE( p );
```

```
26
27    if (p == NULL) puts("p was reset to NULL using second method.\n");
28    else puts("p was not reset to NULL : ERROR!.\n");
29
30  }
```

The `assert()` function is a macro:

```
void assert( int test );
```

If the conditional expression, represented by *test*, evaluates to zero, `assert()` displays the following message and calls `abort`:

```
Assertion failed: test, file filename, line linenum
```

To use `assert()`, include assert.h in your source file. Place assertion tests throughout your source code wherever it can help you detect unexpected or inappropriate conditions. You may use `assert()` to detect a null pointer reference by placing the following code fragment before using a pointer:

```
assert( p != NULL );
```

This statement is equivalent to saying "I assert that `p` is not equal to `NULL`. If this is not true, then abort the program."

When you want to compile your program with all assertions removed, place

```
#define NODEBUG
```

on the statement preceding `#include <assert.h>`.

If you use a pointer after its memory block has been discarded, you may encounter the dreaded *memory trasher* phenomenon. This occurs when some portion of memory is mysteriously written over during program execution. Memory trashers can prove to be quite perplexing, because when they occur, you do not know where the errant code is located. Setting pointers to `NULL` and using assertion statements provide a first line of defense against some memory trashers.

Next, you can use the Turbo Debugger's Breakpoint at Changed Memory Location command (see Chapter 11, "Debugging Techniques") to keep a watch on the memory that is damaged. Turbo Debugger halts your program when certain memory conditions are altered. This usually tells you exactly where the problem is occurring. Sometimes, though, this only shows you where you have a bad pointer or variable containing an array index or size. You might still need to find the reason that your pointer has gone bad.

Other frequent causes of memory trashers include writing data beyond the end of an array. For instance, if you define an 80-byte character string, the following code will damage memory:

```
char s[80];
...
for (i=1; i<=80; i++) s[i] = ' ';
```

In C and C++, arrays always begin at the zeroth element. An array declared as

```
char s[80];
```

has elements `s[0]` to `s[79]`. The reference to `s[80]` is beyond the end of the allocated space for array `s`. By inadvertently exceeding the limits of an array, you can overwrite portions of your stack, causing your program to hang. When working with strings, be sure to allocate an extra byte for the trailing `'\0'` null byte that marks the end of most strings.

C and C++ have a large number of risky routines. By risky I mean that these routines enable you to move or copy bytes from any location in memory to any other. If you do not set the parameters to these functions precisely, you might find yourself copying too much or too little data. In the first case you might trash memory. In the second case you won't trash memory, but because some of the bytes in the destination area will be incorrect, the problem will look like a memory trasher on the loose.

For example, if you have two pointers to strings, you can use the `memcpy()` function to perform a block copy of bytes from one string to the other:

```
char * s1;
char * s2;
...
s2 = memcpy( s2, s1, size );
```

If the value you specify for *`size`* is incorrect, you can easily overwrite important sections of memory.

Off-by-1 errors are a frequent cause of problems related to block copy functions. For example, if you want to index the fifth through tenth elements of an array, you might be tempted to type

```
s2 = memcpy( s2, s1, 10 - 5 );
```

This specifies five bytes to copy. However, if you enumerate the elements from 5 to 10, you will see that there are six elements, not five:

```
[5] [6] [7] [8] [9] [10]
```

Instead, you must type

```
s2 = memcpy( s2, s1, 10 - 5 + 1);
```

In this example, the problem is not that you have run off the end of an allocated variable, but that you have failed to copy the entire data set. This produces symptoms similar to a memory trasher. To keep this type of *off-by-1* error straight in your mind, think of a fence with five sections: Does it have five or six fence posts? If the fence had only five fence posts, the last fence section would not be terminated. Hence, a five-section fence must have six fence posts. Therefore, you must add one to the difference between the start and the end of the fence—and to the start and the end of your length computation.

CHAPTER 6

USING LIBRARY ROUTINES

Libraries provide selections of commonly used, prewritten functions. By using a routine from a library, you speed up program development by saving yourself the time required both to write and to test a certain routine. Borland provides several libraries, including the standard C and C++ library, the container class libraries, and several specialized libraries used to support Turbo Vision and ObjectWindows applications.

The first part of this chapter provides examples for some of the file-oriented library functions that are frequently used in most applications but whose use often sparks many questions from C programmers. The selection of functions that are described is based on questions that are frequently posted on on-line programming forums. New and intermediate-level programmers will find this discussion

Working with filenames

Reading and searching directories

Using `TFileDialog` *in Turbo Vision and ObjectWindows*

The container class libraries

helpful. These functions include the filename manipulation functions, the exec*XXX* and spawn*XXX* application launchers, and specialized library features.

For advanced programmers, this chapter looks at some of the Turbo Vision and ObjectWindows components to provide open file dialog user interface control. The chapter concludes with information on and examples of the container class libraries. The container class libraries are a powerful set of classes that can manage many types of data structures efficiently and robustly. By using these library components, you can significantly improve your productivity.

As an aid to those who are porting source code between Borland C++ and Microsoft C++, or who must switch between those two platforms, I have tried to point out some of the library differences. Chapter 13, "Using Borland C++ with Other Products," provides additional details on porting applications to or from Microsoft C/C++.

WORKING WITH FILENAMES

Nearly all useful programs open and close disk files. Many must prompt the user for the name of the file to use. After obtaining the filename, the program must manipulate the filename to separate the filename portion from any subdirectory information. The Borland C++ library provides several routines, for example, `fnsplit()`, to process filenames. There are also routines to help you create uniquely named temporary files, adjust file attributes, create and delete subdirectories, and search through subdirectories. These and other functions are described in this section.

PARSING FILENAMES

To split an existing filename, such as c:\bc3\bin\bc.exe, into separate pieces such as drive name (c:), path (\bc3\bin), and filename and extension (bc and .exe), use the `fnsplit()` function (defined in dir.h). You typically use `fnsplit()` when your program displays the currently active filename. Word processors and spreadsheets, for instance, usually display the name of the document that is currently shown on-screen. With `fnsplit()` you can quickly isolate the filename

portion of a potentially lengthy drive and pathname combination. Another use is in applications that must construct new filenames based on the filename portion of the string. For example, when the compiler parses source.c, it may create other files named source.asm, source.obj, source.exe, or source.map.

`fnsplit()` takes as input a character array holding a filename and breaks it into four separate pieces, storing the pieces in four separate character strings:

```
int fnsplit( const char *path, char *drive, char *dir,
     char *name, char *ext );
```

`path` is the input string that `fnsplit()` splits apart. After splitting, *`drive`* contains the disk drive ID, *`dir`* contains the subdirectory name, *`name`* contains the filename, and *`ext`* contains the extension, including the leading period (.) character. The maximum length of each component is given by the constants shown in Table 6.1.

TABLE 6.1. MAXIMUM STRING LENGTH CONSTANTS DEFINED FOR `fnsplit()` PARAMETERS.

Constant	*Description*
`MAXPATH`	Maximum length of the path string
`MAXDIR`	Maximum length of the directory string
`MAXDRIVE`	Maximum length of the drive string
`MAXEXT`	Maximum length of the filename extension
`MAXFILE`	Maximum length of the filename field

You should use these constants when declaring character strings to hold the components. This way you ensure that the bounds of your character arrays will not be exceeded.

`fnsplit()` returns an integer value whose bit settings indicate which components were identified in the complete path filename. You can check the return result by performing a bitwise `&` on the result using the constants defined in Table 6.2. These constant values are defined in dir.h.

TABLE 6.2. `fnsplit()` RETURNS RESULT BIT MASKS.

Constant	*Description*
`DIRECTORY`	A subdirectory component was found.
`DRIVE`	The drive name was specified.
`EXTENSION`	The filename contained an extension.
`FILENAME`	The file specification included a filename.
`WILDCARDS`	Wildcards were found in the file specification.

A sample usage of `fnsplit()` is shown in Listing 6.1, together with its companion function, `fnmerge()`.

LISTING 6.1. EXAMPLE OF `fnsplit()` AND `fnmerge()`.

```
// FNSPLIT.CPP
// Demonstrates use of fnsplit() and fnmerge() functions.
#include <string.h>
#include <dir.h>
#include <iostream.h>

void main(void)
{
  char bigname[MAXPATH];
  char drivename[MAXDRIVE];
  char dirname[MAXDIR];
  char filename[MAXFILE];
  char extname[MAXEXT];

  cout << "Enter a fully qualified filename:  ";
  cin >> bigname;

  fnsplit( bigname, drivename, dirname, filename, extname );

  cout << "The filename components are: " << drivename << ", " <<
       dirname << ", " << filename << ", and " << extname << "\n";

  strcpy( bigname, "" );
  fnmerge( bigname, drivename, dirname, filename, extname );
  cout << "The reconstructed filename is:  " << bigname << "\n";
}
```

When you call `fnsplit()`, you may optionally pass `NULL` in place of a variable for any of the four components. When `fnsplit()` encounters the `NULL` address, it parses the component but does not return it in any variable. In other words, to obtain just the filename and extension, you might call `fnsplit()` like this:

```
fnsplit( path, NULL, NULL, filename, extension );
```

You can combine all the strings back together into a complete file by calling `fnmerge()`, which concatenates each of the component strings and returns a full path specification. `fnmerge()` is defined as

```
void fnmerge( char *path, const char *drive, const char *dir,
     const char *name, const char *ext );
```

For compatibility with Microsoft C/C++, you may use `_splitpath()` in place of `fnsplit()` and `_makepath()` in place of `fnmerge()`. Microsoft C/C++ does not define `fnsplit()` and `fnmerge()`. `_splitpath()` and `_makepath()` are defined as

```
void _splitpath( const char *path, char *drive, char *dir,
                 char *name, char *ext );
```

and

```
void _makepath( char *path, const char *drive, const char *dir,
                const char *name, const char *ext );
```

Like `fnsplit()`, `_splitpath()` splits apart a fully qualified filename into its constituent parts. The only difference is that `_splitpath()` has no return result and its parameter strings are defined using the constants shown in Table 6.3. `_makepath()` performs the inverse operation to `_splitpath()`.

TABLE 6.3. MAXIMUM STRING LENGTH CONSTANTS USED WITH `_splitpath()` PARAMETERS

Constant	*Description*
`_MAX_PATH`	Maximum length of the path string
`_MAX_DIR`	Maximum length of the directory string
`_MAX_DRIVE`	Maximum length of the drive string
`_MAX_EXT`	Maximum length of the filename extension
`_MAX_FILE`	Maximum length of the filename field

_FULLPATH()

Another function, provided in Borland C++ (and compatible with Microsoft C/C++), is the `_fullpath()` function. `_fullpath()` adds the current file directory to a *`filename.ext`* string, producing a fully qualified filename, including drive and subdirectory. For example, if the current DOS directory is C:\SOURCE and you use `_fullpath()` with a filename of datatrac.dat, `_fullpath()` returns `C:\SOURCE\datatrac.dat`.

`_fullpath` is defined as

```
char * _fullpath( char *buffer, const char *path, int buflen );
```

where *`path`* is the input filename and *`buffer`* is either `NULL` or the address of a character array having buflen size where the result may be stored. When *`buffer`* is `NULL`, `_fullpath()` uses `malloc()` to allocate a buffer of *`buflen`* bytes and returns a pointer to the allocated buffer. This dual mode of operation is illustrated in Listing 6.2. In the first example, `_fullpath()` returns its result through the parameter `FullName`. In the second example, `_fullpath()` returns its result through an allocated buffer. Note that it is up to your code to discard the buffer when it is no longer needed. Failure to discard the buffer, especially when the pointer is defined locally within a function, causes the allocated memory to be unclaimed and unreusable during the remainder of the program's execution.

LISTING 6.2. SAMPLE USAGE OF THE `_fullpath()` FUNCTION.

```
 1  // fullpath.cpp
 2  #include <stdlib.h>
 3  #include <alloc.h>
 4  #include <stdio.h>
 5  void main( void )
 6  {
 7    // Method #1 of using _fullpath
 8    char FullName[_MAX_PATH];
 9    _fullpath( FullName, "bc.exe", _MAX_PATH );
10    puts( FullName );
11
12    // Method #2, alternate use of _fullpath
13    char * pFullName;
14    pFullName = _fullpath( NULL, "bc.exe", 80 );
15    if (pFullName != NULL) puts( pFullName );
16    free( pFullName );
17  }
```

CREATING TEMPORARY FILES

Many programs must create temporary disk files. Many word processors, for example, store the edited text in a combination of RAM memory and temporary disk files. This permits the word processor to edit files much larger than would otherwise fit in memory. When applications store temporary data in a disk file, they must create a unique filename to ensure that existing files are untouched. When the program terminates, the temporary file is deleted. Users of such programs generally are unaware of the existence of the temporary file.

Some programs create unique filenames by creating semirandom text from the current time and date functions. However, Borland C++ provides several functions in its library that may be used to create temporary filenames and files automatically. These functions include the following:

`creattemp()`	Makes a unique filename, creates the file, and opens it for access through a file handle.
`mktemp()`	Makes a unique filename.
`tempnam()`	Makes a unique filename, verifies uniqueness, and creates the file.
`tmpnam()`	Makes a unique filename and verifies uniqueness.
`tmpfile()`	Makes a unique filename and opens the file for use in C streams. The file is automatically deleted when it is closed or when your program terminates.

As you can see, there is quite a variety of temporary file functions. You should select a temporary file function based on the needs of your application. If you want to create a temporary file that always disappears after your program has completed its execution, use `tmpfile()`. If you plan to create temporary files in a standardized directory, use `tmpnam()`. `tmpnam()` manufactures a unique filename and determines that the filename has not already been used by checking the TMP environment variable. The use of the TMP environment variable is an industry standard. Use the DOS command SET TMP=*directory*\ to initialize this variable if such a command has not already been inserted into your autoexec.bat file by some software's automatic installation process. Be sure to include a trailing backslash on the TMP directory specification. These routines do not work properly with the TMP (or alternate TEMP) variables unless the backslash is included.

To merely manufacture a unique name, use `mktemp()`. To create the name, verify that the filename is unique, and create the file, use `tempnam()`. `creattemp()` is suited for programs that use handle-based I/O, especially for large block I/O operations.

TMPFILE() AND *RMTMP()*

`tmpfile()` is especially useful for C programs that need to open a scratch file and access the file using C stream I/O. `tmpfile()` creates and opens a temporary file, returning a pointer to a `FILE` type. `tmpfile()` is defined as:

```
FILE *tmpfile(void);
```

To use `tmpfile()`, call the function and assign the result to your file variable like this:

```
FILE * tempfile;
...
tempfile = tmpfile();
```

The file is automatically opened in *w+b* (writeable binary) stream format. As a side benefit, you do not need to close or delete temporary files created with `tmpfile()`. When your program exits, the file is deleted automatically. If you close the file by calling `fclose()`, it will be deleted during the close process. If you have several temporary files, all created using `tmpfile()`, you can close and dispose of all of them by calling `rmtmp()`. `rmtmp()` returns the number of files that it successfully closed and deleted.

Where is the temporary file placed? If you've defined a `TMP` environment variable, the file will be created in that directory. If you don't have a `TMP` variable but you do have the older `TEMP` variable, the file will be created in the `TEMP`-specified directory. (`TEMP` has been largely supplanted by the use of `TMP` to specify a temporary files directory, but because it is still used by some software, it is available here.) If neither `TMP` nor `TEMP` has been defined, `tmpfile()` creates its temporary file in the current directory.

TMPNAM()

`tmpnam()` is the most standardized of the temporary filename routines because it is available in Borland C++, Microsoft C++, UNIX, and ANSI C compatible compilers. `tmpnam()` is defined in stdio.h as

```
char * tmpnam( char * s);
```

`tmpnam()` makes a unique filename and either sets its `s` parameter to point to that filename or, if `s` is `NULL`, returns a pointer to the filename string. When `tmpnam()` returns a pointer, the pointer points to a static object defined within the library. Therefore, if you need to open more than one temporary file, you should copy the result to a local string in your application. `s` should have a maximum length of `L_tmpnam`, where `L_tmpnam` is a constant defined in stdio.h.

The Borland documentation states that `tmpnam()` creates the file. This is not true; `tmpnam()` only returns a unique temporary filename. `tmpnam()` does check to see that the filename is unique by checking the subdirectory indicated by the `TMP` variable. If `TMP` does not exist, it checks for `TEMP` and checks the directory specified by `TEMP`. If neither `TMP` nor `TEMP` is defined, `tmpnam()` checks the current directory.

`tmpnam()` does not return the fully qualified path name, only the temporary filename. For this reason, `tmpnam()` is perhaps of limited usefulness. I recommend that you use `tempnam()` instead.

TEMPNAM()

`tempnam()` is similar to `tmpnam()`, except that it checks only the `TMP` variable (not `TEMP`), and it enables you to specify the first part of the temporary filename, plus your own optional temporary files directory. `tempnam()` does not *create* the temporary file, but it does check to see that there is no other file having this name, thereby ensuring uniqueness. `tempnam()` is defined as

```
char * tempnam( char *dir, char *prefix );
```

where *`dir`* is either the name of a subdirectory or `NULL`, and *`prefix`* is a string of up to five characters that will be used as the first five characters of the temporary filename. `tempnam()` returns a pointer to a `malloc()`-created string containing the complete name of the temporary file (including drive and subdirectory). Listing 6.3 is an example that uses `tempnam()`. Note that `free()` must be called to dispose of the string returned by `tempnam()`.

LISTING 6.3. EXAMPLE USING tempnam().

```
 1  #include <stdio.h>
 2  #include <alloc.h>
 3  void main( void )
 4  {
 5    char * FullName;
 6    FullName = tempnam( NULL, "TEMP" );
 7    if (FullName !=NULL)
 8      puts(FullName);
 9    else
10      puts("Not created.\n");
11    free( FullName );
12  }
```

tempnam() selects a subdirectory for the temporary file following this sequence of rules:

1. If TMP is defined, tempnam() attempts to use the directory specified by TMP.
2. If *dir* is not NULL, tempnam() attempts to use the directory specified by *dir*.
3. tempnam() attempts to use the default directory specified in the P_tmpdir definition in stdio.h. This string is hard-coded into the tempnam() library code and cannot be changed merely by editing stdio.h.
4. If all of the preceding fail, tempnam() places the temporary file in the current directory.

In the preceding list of rules, note the use of the word *attempt*. If an attempt to create a directory using the current rule fails, tempnam() advances to the next rule. If the directory specified by TMP or *dir* does not exist, tempnam() continues down the list of rules. The string result returned by tempnam() contains the fully qualified pathname for the temporary file.

MKTEMP()

mktemp() creates a temporary filename but enables you to select the first two characters. mktemp() uses its parameter string as a template to manufacture a unique filename and checks for the presence of the TMP and TEMP environment variables to determine which directory should hold the temporary file. mktemp()

also checks the appropriate directory to ensure that no other file has the name it has created. To use `mktemp()`, set the template parameter to a string where the first two characters are your choice, followed by six letter Xs:

```
char * FullName;
FullName = mktemp( "TPXXXXXX" );
puts(FullName);
```

CREATTEMP()

Use `creattemp()` to create a unique filename in a specific subdirectory for use with handle-based file I/O functions. `creattemp()` is defined as

```
int creattemp( char *path, int attrib );
```

where *`path`* is the name of the subdirectory (ending in a backslash character, for example, `C:\source\`) where the temporary file should be created, and *`attrib`* is set to one of the constants shown in Table 6.4.

TABLE 6.4. CONSTANT VALUES USED FOR THE attrib PARAMETER.

Constant	*Description*
`FA_RDONLY`	Makes the file read-only.
`FA_HIDDEN`	Makes the file hidden.
`FA_SYSTEM`	Sets the system file attribute.

`creattemp()` creates a temporary filename, creates and opens the file in the desired subdirectory, and returns the full, temporary filename in the *`path`* parameter. The return result is set to the DOS file handle for the opened file. To read or write data to handle-based files (as compared to C streams), you must use the `read()` and `write()` functions. To specify the file mode, set the global `_fmode` variable to either `O_TEXT` for text files or `O_BINARY` for binary files.

If the `creattemp()` operation fails for some reason, it indicates the error condition by setting the global variable `errno` to one of the constants shown in Table 6.5. `errno` is defined in both errno.h and stdlib.h, but the constants are defined only in errno.h.

Constant	*Description*
ENOENT	Could not locate the subdirectory.
EMFILE	Too many files already open.
EACCESS	Access permission is denied.

USING FILE ATTRIBUTES

Each file located on a disk has an associated set of *file attributes*. The file attributes indicate whether the file is read-only, writeable, a hidden file, or a system file. To set or get these attributes for a particular file, use the `_chmod()` function. You'll need to `#include` both dos.h and io.h. `_chmod()` is declared as

```
int _chmod( const char *path, int func, int attrib);
```

Set *path* to the name of the file, including subdirectory, whose attributes you want to change. Set *func* to 1, and set *attrib* to a bit mask determined by the constants in Table 6.6. For example, to set the attributes of the file named mtype1.c to become a read-only file, use

```
_chmod( "mtype1.c", 1, FA_RDONLY );
```

You also can use `_chmod()` to obtain the current file attributes of an existing file. In this form, set the *func* parameter to 0. You do not need to include the *attrib* parameter. For example, to obtain the current attributes of a file, write

```
attributes = _chmod( "mtype1.c", 0 );
```

Use the bit mask constants in Table 6.6 to examine the result. For example, to test for a read-only file, use a statement like this:

```
if (attributes & FA_RDONLY) puts("File is read-only.");
```

TABLE 6.6. BIT MASK CONSTANTS USED TO SET OR READ FILE ATTRIBUTES.

Constant	*Description*
FA_ARCH	Archive bit
FS_DIREC	Filename is a directory.

Constant	*Description*
FA_HIDDEN	Hidden file
FA_LABEL	Disk volume label
FA_RDONLY	Read-only
FA_SYSTEM	System file

You also can use chmod() to set file attributes. This alternative function, defined in \borlandc\include\sys\stat.h, is chiefly for compatibility with Microsoft C/C++ and UNIX. Note that Borland's chmod() is equivalent to Microsoft's _chmod() function; Borland's _chmod() has no equivalent in Microsoft C/C++.

chmod() has just two parameters:

```
int chmod( const char *path, int amode );
```

path is the name of the file to set, and *amode* is set to one or more of the bit mask values in Table 6.7.

TABLE 6.7. BIT MASK VALUES FOR THE chmod() FUNCTION.

Bit Mask Value	*Description*
S_IWRITE	Makes the file have write access.
S_IREAD	Makes the file have read access.
S_IREAD ¦ S_IWRITE	Makes the file have read/write access.

A matching industry-standard function, access(), reads the attribute information. access() is similar to chmod(), but it returns the current access rights rather than setting them. access() is defined as

```
int access( const char *filename, int amode );
```

To use access(), set *amode* to one of the values in Table 6.8. access() returns 0 if the particular attribute is set, or –1 if the attribute is not set. If the file cannot be found, the global errno variable is set to ENOENT. errno is defined in both errno.h and stdlib.h, but the constants are defined only in errno.h.

TABLE 6.8. SET amode TO ONE OF THESE VALUES TO CHECK A FILE'S ACCESS RIGHTS.

Value	*Description*
0	Checks to see whether the file exists. If this condition is true, the file is also readable.
1	Checks to see whether the file can be executed.
2	Checks to see whether the file can be written to.
4	Checks to see whether the file can be read from (all DOS files can be read from).
6	Checks to see whether the file can be both read from and written to.

CREATING AND DELETING SUBDIRECTORIES

You've seen how to create temporary files and how to set and check file attributes. Depending on your needs, you also might need to create and delete subdirectories, change the current working directory, and change the current working disk drive. To create a new subdirectory, use `mkdir()`. To delete an existing subdirectory, use `rmddir()`. Use `chdir()` to change the currently active directory and `_chdrive()` to change the active disk drive selection.

MKDIR(), *RMDIR()*, AND *CHDIR()*

`mkdir()` creates or *makes* a new subdirectory in the current subdirectory. `mkdir()` is defined as

```
int mkdir( const char *path );
```

where *path* is the name of the subdirectory to create. `mkdir()` returns 0 if the operation was successful and –1 if the operation failed. Here is an example:

```
if (mkdir( "data-dir" )) puts("Error creating directory.");
```

If `mkdir()` fails, check the global variable `errno` (from errno.h). If the directory already exists, `mkdir()` returns the `EACCES` constant.

If no path or subdirectory is specified, the new directory is created inside the currently active directory. If you want to create the subdirectory elsewhere, you can specify an optional drive letter plus existing subdirectory name, like this:

```
mkdir( "C:\\books\\tpr" );
```

When typing directory information in C string constants, be sure to use the double backslash as the directory separator. The C language uses the backslash as a special escape character to enter nonprintable code such as `\n` or `\0`; two backslashes translate into a single backslash. This requirement applies only to string constants in the C source file. You are not required to type double backslashes when entering a subdirectory in response to an executing program's prompt.

If `mkdir()` cannot find the specified directory (for example, C:\books), `mkdir()` returns the `ENOENT` error code.

To select a new directory or an existing directory as the current or default directory, call `chdir()`. `chdir()` is defined as

```
int chdir( const char *path );
```

`chdir()` makes the directory in *`path`* the current default DOS directory. If the directory does not exist, `chdir()` returns `ENOENT`.

To remove a directory, use `rmdir()`, defined as

```
int rmddir( const char *path );
```

This deletes the directory named in *`path`*, provided that the named directory is empty, is not the default directory, and is not the root directory. Possible error codes are `EACCES`, indicating that the conditions just mentioned were not satisfied, or `ENOENT` if the specified directory does not exist.

DRIVE SELECTION FUNCTIONS

When you need to switch the default drive, use `_chdrive()`, defined in direct.h as

```
int _chdrive( int drive );
```

where *`drive`* is set to 1 for drive A, to 2 for drive B, and so on. If the drive selection succeeds, `_chdrive()` returns 0; otherwise, `_chdrive()` returns –1. To determine the current default drive, call `_getdrive()`. `_getdrive()` returns 1 for

drive A, 2 for drive B, 3 for drive C, and so on. `_chdrive()` and `_getdrive()` are implemented identically in both Borland C++ and Microsoft C/C++.

READING AND SEARCHING DIRECTORIES

Many programs display directory listings or perform directory searches. Directory listings help users choose a file to operate on, and directory searches can help locate files that are squirreled away in a twisty maze of subdirectories. In Borland C++, there are three separate ways to obtain directory listings. Each of these methods is described in this section.

Another problem faced by many applications is the need to search for their own data and configuration files. Thanks to the DOS `PATH` environment variable, you can launch an application from any subdirectory on the PC system. When an application starts, it must do some hunting to determine where its configuration files are located, unless the start-up directory is hard-coded into the program. Hard coding a directory name is one solution, but it is generally frowned on by users because it forces them to place your program into a specific directory. Several routines make it possible for your program to be located in any directory yet gain access to its location and files in a fairly straightforward manner.

THREE WAYS TO OBTAIN A DIRECTORY LISTING

In Borland C++, there are actually three ways you can obtain a directory listing. None of them are particularly difficult to use, and one is downright easy.

Surprisingly, perhaps, you can access the DOS `dir` command directly from within your Borland C++ programs. Use the `system()` function (from stdlib.h) to issue the `dir` command directly to DOS, just as if you had typed the command at the DOS command line:

```
system( "dir" );
```

This displays the directory listing on your screen wherever the cursor happens to be located. You don't have much control over the location and format of the display when using this technique. You might be able to improve the situation by using this command instead:

```
system( "dir >output" );
```

This uses the DOS redirection operator (>) to redirect the output of the `dir` command into a disk file named output. As soon as the directory listing has been written to the disk file, your application can open and read this text file. The beauty of the `system()` command is its simplicity. If you want to add directory listings in a pinch, the `system()` function might provide you with a quick-and-dirty solution.

ACCESSING A DIRECTORY AS A FILE

You can open a directory as you would open a disk file. The `opendir()` function opens a disk directory in a manner similar to opening a disk file; you then use the `readdir()` function to read through the directory entries. `closedir()` closes an open directory stream; `rewinddir()` resets the directory back to its beginning position so that future `readdir()` calls will start at the beginning of the directory.

To scan through a directory using these functions, you must first open the directory using `opendir()`, defined in `<dirent.h>` as

```
DIR *opendir( char *dirname);
```

The structure `DIR` is not intended for direct use, but it is used as a parameter to the `readdir()` function. Its layout is shown in Listing 6.4. `opendir()` opens the directory as a stream, positioning to the first entry in the directory. If the directory cannot be opened, `opendir()` returns a `NULL` pointer and sets the global variable `errno` (from errno.h) to `ENOENT` if the directory does not exist, or `ENOMEM` if it is unable to allocate a memory block for the `DIR` structure. When you are finished reading the directory, be sure to discard the `DIR *` pointer by calling `closedir()`.

LISTING 6.4. THE DIR TYPE DEFINED IN DIRENT.H FOR USE WITH THE opendir() AND readdir() FUNCTIONS.

```
typedef struct
{
    char          _d_reserved[30];      /* Reserved */
    struct dirent _d_dirent;            /* Filename part */
    char    _FAR *_d_dirname;           /* Directory name */
    char          _d_first;             /* First file flag */
    unsigned char _d_magic;             /* For handle verification */
} DIR;
```

As soon as the directory is opened, you must use `readdir()` to read the filenames from the directory. `readdir()` is defined as

```
struct dirent * readdir(DIR *dirp);
```

`dirp` must point to the `DIR` directory block returned by the `opendir()` function. `readdir()` returns a `dirent` structure that contains a single field, `d_name`, holding the next filename. `dirent` is shown in Listing 6.5. When you have finished working with the directory stream, call the `closedir()` function:

```
int closedir( DIR *dirp);
```

`closedir()` returns 0 if the close was successful and –1 if an error occurred (typically meaning that *`dirp`* is invalid).

LISTING 6.5. THE `dirent` STRUCTURE IS USED TO RETURN FILENAME ENTRIES.

```
struct dirent
{
    char        d_name[13];
};
```

The Borland C++ documentation incorrectly defines `readdir()` as

```
struct dirent readdir(DIR *dirp);
```

This description omits the `*` pointer component from the function header. This is incorrect. The definition of `readdir()` shown earlier is correct. The documentation also incorrectly defines the `closedir()` function as

```
void closedir(DIR *dirp);
```

Listing 6.6 shows a sample program that scans through a directory using these functions. `readdir()` returns all subdirectory entries, including files and subdirectories, and the special subdirectory entries . and .... All files,

even hidden and system files plus volume labels, are read by this function. This function, however, does not provide any attribute information that you can use to determine what each of the entries might be. If you need to distinguish between filenames and subdirectory names, you should use the `findfirst()` and `findnext()` functions.

LISTING 6.6. SAMPLE USAGE OF THE `opendir()` AND RELATED DIRECTORY-READING FUNCTIONS.

```
/* OPENDIR.CPP
   Demonstrates use of the opendir(), readdir(), and
   closedir() functions.
*/
#include <stdio.h>
#include <dirent.h>
void main(void)
{
  DIR * pDirectory;
  struct dirent * pEntry;

  pDirectory = opendir("C:\\SBM");
  if (pDirectory == NULL)
    puts("Unable to open the directory.\n");
  else do {
     pEntry = readdir( pDirectory );
     if (pEntry != NULL) puts( pEntry->d_name );
  } while (pEntry != NULL);

  closedir( pDirectory );
}
```

The last and probably the most commonly used method for reading directories is use of the `findfirst()` and `findnext()` methods. These functions provide complete control of the directory listing and search process, enabling you to search using wildcards and to separate filenames from directory entries. These functions are described next.

DIRECTORY SEARCHING

The Borland C++ library provides several files to help you locate specific files. The `findfirst()` and `findnext()` functions enable you to manually control the

search process, giving you the greatest flexibility but at the expense of requiring you to write additional code. A couple of extra functions, `searchpath()` and `_searchenv()`, automatically scan through the list of directories listed in the DOS `PATH` variable, or in a specific environment variable. Both of these functions look for a specific file.

The `findfirst()` and `findnext()` functions are used in conjunction with one another to scan through the directory structure. You must call `findfirst()` to initialize the search, then call `findnext()` repeatedly to read the content of each directory. These functions scan through only the specified directory; however, you can write code that uses the return result to search through each subdirectory.

`findfirst()`'s prototype is

```
int findfirst( const char *pathname, struct ffblk *ffblk, int attrib );
```

You use `findfirst()` to initialize the search process, setting *`pathname`* and *`attrib`* according to how the search should be conducted. Set *`pathname`* to the name of the subdirectory and file search pattern—such as "c:\borlandc\bin*.exe"—to locate all files having an .exe extension within the c:\borlandc\bin directory. *`attrib`* must be set to one or more attributes using the bit mask constants in Table 6.9. Using these constants you can selectively confine the search so that, for example, only public files or only directories are displayed. If you set *`attrib`* to `FA_HIDDEN ¦ FA_NORMAL ¦ FA_DIREC`, `findfirst()` will return any file that is hidden, normal, or a directory entry.

TABLE 6.9. FILE ATTRIBUTE BIT MASK CONSTANTS (FROM DOS.H).

Constant	*Description*
`FA_ARCH`	Finds files having their archive bit set.
`FA_DIREC`	Finds entries that are directories.
`FA_HIDDEN`	Finds all hidden files.
`FA_LABEL`	Finds volume label entries.
`FA_NORMAL`	Finds all nondirectory files with no attributes set.
`FA_RDONLY`	Finds all files that are read-only.
`FA_SYSTEM`	Finds system files.

`findfirst()` returns the structure `ffblk`, which contains information that is used in tracking the progress of the directory search and which contains the file's name, creation date, time, and so forth. This structure is shown in Listing 6.7. You must use the `ffblk` structure when calling `findnext()`. See the section "Using the `ff_fdate` and `ff_ftime` Fields" for more information about the date and time fields.

LISTING 6.7. THE `ffblk` STRUCTURE USED IN `findfirst()` AND `findnext()` CALLS.

```
struct  ffblk   {
    char        ff_reserved[21];
    char        ff_attrib;        // File attribute byte
    unsigned    ff_ftime; // File creation time
    unsigned    ff_fdate; // File creation date
    long        ff_fsize; // File size in bytes
    char        ff_name[13];      // Filename
};
```

`findfirst()` returns the first matching directory entry. Thereafter, you must pass the `ffblk` structure to the `findnext()` function to obtain the next directory entry. `findnext()` is defined as

```
int findnext( struct ffblk *ffblk );
```

`findnext()` sets *`ffblk`* to the next file. If there are no matching files, `findfirst()` and `findnext()` both return a nonzero value.

Listing 6.8 shows how `findfirst()` and `findnext()` may be used to display a list of all files on the system. By setting the initial starting subdirectory to a directory other than the root (c:\), you can restrict the display to all directories beneath a specific directory. For example, to display all files in the Borland C++ installation, start the search like this:

```
searchdirectory( "c:\\borlandc\\");
```

The function `searchdirectory()` uses `findfirst()` to initialize the disk search. Line 18 sets the search pattern to `*.*`. If you want to look at all files, including subdirectories, you must set the search pattern to `*.*`. If you want to look only for files ending in .obj, you could set this value to `*.obj`. However, this would then miss any subdirectories unless they happen to end in `.obj`.

The `findnext()` function also returns the special . and . . DOS directory entries. Because there is no need to display these filenames, line 22 includes a check to throw out any filenames beginning with a single period (.). Because DOS permits filenames to begin with multiple decimal points, this code could erroneously throw out valid entries, although it seems very unlikely that many useful files will be named beginning with a period.

The interesting part of the routine is in lines 24–30. If a directory entry is encountered, `searchdirectory()` adds the subdirectory name to the current path and then calls itself recursively to search deeper into the directory structure.

You can modify this code to search for a specific file. Change line 23 to display the directory and filename only if the directory entry matches the filename for which you are searching. You also could modify this routine to display all the files in the current directory first, followed by the files in the respective subdirectories. One approach you could take would be to scan through the current directory and display all files. Then, restart the scan from the beginning of the directory and call `searchdirectory()` for each subdirectory that is encountered.

Listing 6.8 shows an example of `findfirst()` and `findnext()` used to display a directory listing.

LISTING 6.8. USING `findfirst()` AND `findnext()` TO DISPLAY A DIRECTORY LISTING.

```
 1  // FINDDEMO.CPP
 2  // This program demonstrates the use of the
 3  // findfirst() and findnext() functions to
 4  // display a directory listing.
 5  #include <dir.h>
 6  #include <iostream.h>
 7  #include <string.h>
 8  #include <dos.h>
 9
10  void searchdirectory (char * directory )
11  {
12    char tempdirectory[MAXPATH];
13    int last_one;
14    struct ffblk fileinfo;
15
16    strcpy (tempdirectory, directory);
```

```
17    // Note use of wildcard in next line; see text for details.
18    strcat( tempdirectory, "*.*" );
19    last_one = findfirst( tempdirectory, &fileinfo,
20                FA_NORMAL ¦ FA_DIREC ¦ FA_RDONLY );
21    while (!last_one) {
22      if (fileinfo.ff_name[0] != '.') {
23        cout << directory << fileinfo.ff_name;
24        if (fileinfo.ff_attrib & FA_DIREC) {
25    cout << "  <DIRECTORY>\n";
26    strcpy( tempdirectory, directory );
27    strcat( tempdirectory, fileinfo.ff_name );
28    strcat( tempdirectory, "\\" );
29    searchdirectory( tempdirectory );
30        };
31        cout << "\n";
32     };
33     last_one = findnext( &fileinfo );
34    };
35  };
36
37  void main( void )
38  {
39
40    // Set starting directory here
41    searchdirectory( "C:\\" );
42
43  };
```

USING THE *FF_FDATE* AND *FF_FTIME* FIELDS

File date and time information is specially encoded in the bits of the `ff_ftime` and `ff_fdate` fields. The following code fragment shows how to extract the data:

```
int hours, minutes, seconds;
int year, month, day;
...
hours = fileinfo.ff_ftime >> 11;
minutes = (fileinfo.ff_ftime >> 5) & 63;
seconds = fileinfo.ff_ftime & 63;
year = (fileinfo.ff_fdate >> 9) + 1980;
month = (fileinfo.ff_fdate >> 5) & 15;
day = fileinfo.ff_fdate & 31;
cout << "  " << year << "-" << month << "-" << day;
```

USING *_SEARCHENV()* AND *SEARCHPATH()*

When you need to check for the presence of an environment variable that specifies a file subdirectory, use the `_searchenv()` function. `_searchenv()` examines the DOS environment area, looking for the desired variable, such as `TMP`, `INCLUDE`, `HELP`, or whatever else you've defined for your program. If the variable is found, `_searchenv()` then scans through each of the specified subdirectories to see whether the desired file is found. If the file is found, `_searchenv()` sets a parameter to the full pathname of the file.

`_searchenv()` is especially useful if your program places data files or creates output files in special directories that have been specified using the DOS `SET` command and environment variables. When you set up an environment variable for use with `_searchenv()`, you may specify multiple file directories by separating each directory with a semicolon. For example:

```
set helpfile=c:\win31;c:\sbm;c:\wp
```

It is very important that there be no blanks on either side of the equal sign (`=`).

`_searchenv()` is defined as

```
void _searchenv( const char *file, const char *varname, char *buf);
```

`file` is the name of the file for which you are looking, *`varname`* is the name of the environment variable to check, and *`buf`* is a character array to store the complete pathname of the file, if found. Listing 6.9 provides an example. In this code fragment, `_searchenv()` searches for a file named HELPFILE.DAT using the `HELPFILE` environment variable. You must use uppercase text when typing the filename and environment variable name; if you use lowercase text, `_searchenv()` fails to make the match.

LISTING 6.9. SAMPLE USE OF THE `_searchenv()` FUNCTION.

```
char pathname[80];

 _searchenv("HELPFILE.DAT", "HELPFILE", pathname );
if (strlen(pathname) != 0)
  cout << "File was found at " << pathname << "\n";
else
  cout << "File was not found.\n";
```

> Borland's documentation incorrectly defines `_searchenv()` as
>
> ```
> char * _searchenv( const char *file, const char *varname, char *buf);
> ```

When you need to search for a specific file that is located or expected to be located in one of the directories specified by the DOS PATH statement, use the `searchpath()` function instead. `searchpath()` scans each of the directories listed in the PATH variable, looking for the filename you have specified. In effect, `searchpath()` is roughly equivalent to

```
_searchenv( yourfile, "PATH", pathname );
```

where *yourfile* contains the filename you are looking for.

`searchpath()` is defined as

```
char * searchpath( const char *file );
```

where *file* is the filename to locate. `searchpath()` returns a pointer to a static memory location that contains the complete pathname for the file.

Many applications such as word processors, data base software, presentation graphics programs, and so forth use the PATH variable to locate their installation directory. As soon as they have found their installation directory, they gain access to their configuration and support files, including on-line help data bases, font files, and so on.

To see how a program determines its own subdirectory, consider how a DOS program is launched from the command line. In DOS, the only way to execute a program is to type the program's name at the command line. When you type the program name, DOS uses these rules to locate the program:

1. If the program name is fully specified, including a subdirectory, DOS runs the program directly.
2. Next, DOS looks in the current default directory.
3. If the program was not found, DOS then scans through the directories listed in the PATH variable.

For an application to locate its own directory, it must perform roughly the same steps used by DOS. Listing 6.10 shows sample code that you can include in your programs. Line 42 uses the global variable `_argv[0]` to obtain the

program name that was typed on the DOS command line (this works only for DOS 3.0 and later). Function `find_file()` uses `fnsplit()` to parse out a possible subdirectory. If the user entered the directory name when the program was launched, `find_file()` returns the original command line filename. Otherwise, `find_file()` uses `searchpath()` to automatically check *both* the default directory and the directories listed in the PATH variable.

LISTING 6.10. SAMPLE CODE TO LOCATE AN APPLICATION'S OWN SUBDIRECTORY.

```
// SEARCHP.CPP
// Demonstrates use of searchpath() to locate
// an application's own files.
#include <dir.h>
#include <iostream.h>
#include <string.h>
#include <dos.h>

int find_file( const char * filename, char * pathname )
/* Determines the location of a filename by checking for
   an explicit subdirectory, checking the default directory,
   and then the directories specified by PATH. If found,
   returns 0 and sets pathname to the subdirectory. If not
   found, returns -1. */
{
  char * location;

  // First see if the directory was given explicitly in the filename
  if (fnsplit( filename, NULL, NULL, NULL, NULL ) & DIRECTORY) {
    strcpy( pathname, filename);
    return 0;
  }
  location = searchpath( filename );
  if (location == NULL)
    return -1;
  else {
    strcpy( pathname, location );
    return 0;
  }
};

void main(void)
{
  char pathname[MAXPATH];

  // Get the program name from the command line and pass
  // it to find_file() for searching.
```

```
38    if (find_file( _argv[0], pathname ) == 0)
39      cout << "File is located at " << pathname << "\n";
40    else
41      cout << "File was not found.\n";
42
43  }
```

ACCESSING COMMAND-LINE PARAMETERS

Listing 6.10, line 38, uses the `_argv[0]` variable to obtain the original program name that was typed on the command line. The other elements of the `_argv[]` array contain the remaining command-line arguments. For example, if I launch my text editor program by typing

```
C:> EDIT RADIO.RC
```

DOS (and Borland C++) sets up the `_argv[]` array so that `_argv[0]` points to a string containing the program name, and `_argv[1]` points to a string containing the filename argument radio.rc. The variable `_argc` is a count of the number of valid elements in `_argv[]`. Both symbols, `_argc` and `_argv[]`, are defined in dos.h. Listing 6.11 shows how these variables are used to access each of the command-line arguments.

Although the use of `_argc` and `_argv[]` is well known by C programmers, Borland provides an interesting twist that makes processing filenames containing wildcard characters a lot easier. If you link in a Borland-provided object module named wildargs.obj (located in the \borlandc\lib directory), all wildcard filenames on the command line are automatically expanded into a list of matching files. If, for instance, I type

```
C:> EDIT *.RC
```

the code brought in from wildargs.obj will expand this into a list of all matching files, such as this:

```
EDIT RES1.RC RES2.RC RADIO.RC
```

You do not need to make any changes to your program. Merely add the wildargs.obj file to your project so that it will be linked into your .exe file. Thereafter, anytime a wildcard filename is encountered in the command-line options, it will be expanded into a complete file list.

For compatibility with other compilers, you may also use the `argc` and `argv[]` variables, by defining these as parameters to the `main()` function, like this:

```
void main( int argc, char *argv[] );
```

Listing 6.11 shows how `_argc` and `_argv[]` are used to access each of the command-line arguments.

LISTING 6.11. SAMPLE USE OF THE `_argc` AND `_argv[]` SYSTEM VARIABLES.

```
 1  #include <dos.h>
 2  #include <stdio.h>
 3
 4  void main( void )
 5  {
 6    int i;
 7
 8    for( i=0; i< _argc; i++ )
 9      printf("%s\n", _argv[i] );
10
11  }
```

USING ENVIRONMENT VARIABLES

DOS environment variables store a variety of information available to all applications running on the PC. You have already seen how the PATH variable is used to locate the subdirectory where a program has been launched. Many applications designate their own variables, such as TMP, HELP, LIB, INCLUDE, and so forth. The content of a DOS environment variable may include items other than subdirectories. Indeed, many applications use environment variables to set information about their configuration. Some applications, such as command-line-operated utilities, may let you store commonly used command-line switches in an environment variable, as in this example:

```
c:>set options=-r -v -mh
```

Each time the program is run, it looks for the environment variable. If it is found, the program uses these settings as its default command-line options.

You can control the use of environment variables using the `getenv()` function to obtain the value of an existing variable, and `putenv()` to set a temporary variable for use during your program's execution. `getenv()` is defined in stdlib.h as

```
char *getenv( const char *name );
```

`getenv()`looks for an environment variable specified by *name* and returns a pointer to the content of the variable, or `NULL` if the variable does not exist. The variable name should be in all capital letters. Here's an example that fetches the value of the `OPTIONS` environment variable:

```
char *varstr;
varstr = getenv( "OPTIONS" );
```

You may also obtain a list of all environment variables by referencing the `environ[]` array defined in dos.h. `environ[]` is an array of pointers to strings that you can use to scan through the complete list of variables, as shown in this code segment:

```
int index = 0;
while (environ[index] != NULL)
  printf("%s\n", environ[index++]);
```

Do not use the `environ[]` pointers to alter the environment variables. Instead, use `putenv()`. You can temporarily change an environment variable by calling `putenv()`, defined in stdlib.h as

```
int putenv( const char *name );
```

where *name* contains the environment variable assignment statement. For example,

```
putenv("EDITOPTIONS=/BW /50");
```

You may also use `putenv()` to modify an existing variable (use `getenv()` first, make your changes, and then use `putenv()` to place it back in the environment area) or to delete a variable. To delete a variable, set the variable to an empty string, like this:

```
putenv("EDITOPTIONS=");
```

`putenv()`'s modifications are temporary. When the program terminates, the modified environment goes away and the system's original environment is restored. The string that you pass to `putenv()` should be either a static character string or a global variable. If a local variable is used, the results can become unpredictable. A convenient solution is to use `getenv()` to return a pointer to `getenv()`'s own static character array. Then, use this pointer to set up your variable assignment, and pass the address to `putenv()`.

INTERCEPTING CTRL-BREAK

In MS-DOS, pressing Ctrl-Break is used to halt executing programs. Sometimes it is not appropriate to terminate a program when someone presses Ctrl-Break. For instance, inadvertently pressing Ctrl-Break while editing a 50-page document in your word processor or while typing several new pages of source code in the IDE could ruin your whole day. To prevent this, your program can intercept Ctrl-Break and take its own action or ignore the Ctrl-Break keystrokes altogether.

The Library has a routine named, appropriately, `ctrlbrk()` that may be used to set a function to execute whenever the Ctrl-Break keystroke is hit. The function that is then activated can do whatever is appropriate for the application, including aborting the program (return 0) or continuing execution (return nonzero value). Alternatively, the function may use the `longjmp()` function to transfer control to some other location in the program. Listing 6.12 shows an example of how the `ctrlbrk()` function is used.

LISTING 6.12. HOW TO INTERCEPT THE CTRL-BREAK KEYSTROKE.

```
1  // ctrlbrk.cpp
2  // Shows how to intercept a Ctrl-Break to avoid
3  // terminating a program. Will not work if launched
4  // from the IDE!  Execute from command line to
5  // see it in operation.
6  #include <stdio.h>
7  #include <conio.h>
8  #include <dos.h>
```

```
 9
10  int IgnoreCtrlBrk(void)
11  // Return 0 to abort program; nonzero to continue
12  { return 1; }
13
14  void main(void)
15  {
16    ctrlbrk( IgnoreCtrlBrk );
17    while (1) {
18      printf("Press <space> to halt; Ctrl-Break is intercepted...");
19      if (kbhit())
20        { if (getch()==' ') return; }
21    };
22  }
```

USING *TFILEDIALOG* IN TURBO VISION

Turbo Vision is a complete class library for creating modern user interfaces that use pull-down menus, dialog boxes, and the mouse. A number of components also provide support for specialized data structures and keyed data base files (or resource files). *Secrets of the Borland C++ Masters* has a number of program examples that use the Turbo Vision library. It is not our intent, though, to provide a tutorial for the use of Turbo Vision. Instead, if you are unfamiliar with the Turbo Vision Application Framework, these examples give a brief glimpse of how Turbo Vision can be put to work in your programs. If you already program in Turbo Vision, then we hope that our examples will highlight features that are useful to you.

The Turbo Vision class library contains a large number of predefined classes that implement a remarkable amount of program functionality. By deriving your own classes from the existing class libraries, or merely by instantiating the functions that are already provided, you can create sophisticated user interfaces quickly. In this section, the `TFileDialog` class is introduced. You can use `TFileDialog` to implement Open File and Save As dialog boxes. `TFileDialog` contains prebuilt support for the standardized types of Open File dialog boxes that most programs use. You can see an example of the `TFileDialog` each time you select File | Open... in the Borland C++ IDE. Another user interface example is the File | Change dir... dialog box. This dialog box is also available as a prepackaged class, `TChDirDialog`, in the Turbo Vision library.

You can put the `TFileDialog` dialog object into your Turbo Vision applications very easily. Assuming that you already have a Turbo Vision application, you need only instantiate a `TFileDialog` object, passing a filename pattern and other options to the `TFileDialog` constructor method. To see how this works, look at tvdialog.cpp, shown in Listing 6.13.

LISTING 6.13. A SAMPLE TVISION APPLICATION THAT USES `TFileDialog`.

```
// TVDIALOG.CPP
// Demonstrates use of TFileDialog to implement a File Open
// or File Save dialog box in a TVISION application.
#define Uses_TApplication
#define Uses_TKeys
#define Uses_TRect
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_TStatusLine
#define Uses_TStatusItem
#define Uses_TStatusDef
#define Uses_TDeskTop
#define Uses_TFileDialog
#define Uses_TView
#include <tv.h>
#include <dos.h>
#include <string.h>

// This constant value is the ID returned for File ¦ Open cmd.
const int cmFile_Dialog = 200;

class DemoApp : public TApplication
{
public:
  DemoApp();
  static TStatusLine *initStatusLine( TRect r );
  static TMenuBar *initMenuBar( TRect r );
  virtual void handleEvent( TEvent& event);
};

DemoApp::DemoApp() :
  TProgInit( &DemoApp::initStatusLine,
      &DemoApp::initMenuBar,
      &DemoApp::initDeskTop
    )
// Constructor defaults to ancestor's constructor.
{
```

```
39  }
40
41  TStatusLine *DemoApp::initStatusLine(TRect r)
42  {
43  // Initializes the on-screen status line.
44    r.a.y = r.b.y - 1;
45    return new TStatusLine( r,
46      *new TStatusDef( 0, 0xFFFF ) +
47         *new TStatusItem( 0, kbF10, cmMenu ) +
48         *new TStatusItem( "~Alt-X~ Exit", kbAltX, cmQuit )
49        );
50  }
51
52  TMenuBar *DemoApp::initMenuBar( TRect r )
53  {
54  // Initializes the menu bar and pull-down menu.
55    r.b.y = r.a.y + 1;
56    return new TMenuBar( r,
57      *new TSubMenu( "~F~ile", kbAltF )+
58        *new TMenuItem( "~O~pen", cmFile_Dialog, kbF3, hcNoContext, "F3" )+
59        newLine()+
60        *new TMenuItem( "E~x~it", cmQuit, cmQuit, hcNoContext, "Alt-X" )
61      );
62  }
63
64  // The following function is from Borland's TVEDIT sample program
65  ushort execDialog( TDialog *d, void *data )
66  {
67      TView *p = TProgram::application->validView( d );
68      if( p == 0 )
69        return cmCancel;
70           else
71        {
72        if( data != 0 )
73            p->setData( data );
74        ushort result = TProgram::deskTop->execView( p );
75        if( result != cmCancel && data != 0 )
76            p->getData( data );
77        TObject::destroy( p );
78        return result;
79        }
80        }
81
82
83  void DemoApp::handleEvent(TEvent& event)
84  // Processes all event messages. Most messages are
85  // passed to TApplication for processing, but this
86  // function does handle the File ¦ Open command and
87  // shows how to use the TFileDialog class.
```

continues

LISTING 6.13. CONTINUED

```
 88  {
 89    char pathname[MAXPATH];
 90
 91    TApplication::handleEvent(event);
 92    if( event.what == evCommand )
 93    {
 94      switch( event.message.command )
 95      {
 96      case cmFile_Dialog: {
 97        // Set pathname to the default filename or wildcard pattern
 98        // to first appear in the dialog box.
 99        strcpy( pathname, "*.*" );
100        if ( execDialog( new TFileDialog( pathname, "Open File", "Filename",
101      fdOpenButton ¦ fdHelpButton, 100 ), pathname) != cmCancel);
102        // Then open the file
103        break;
104         };
105      default:
106        return;
107      }
108      clearEvent( event );
109      }
110  }
111
112
113  int main( void )
114  {
115    DemoApp FileDlgDemo;
116    FileDlgDemo.run();
117    return 0;
118  }
```

The only function that tvdialog.cpp has is to display a menu bar having a single item, the File menu. This menu contains just two functions: Open and Exit. When Open is selected, the program displays the dialog box that is created by the `TFileDialog` object. Like all Turbo Vision applications, the heart of this operation is implemented in the `HandleEvent()` method, in lines 83–110. Lines 96–104 handle the processing of the `cmFile_Open` command, the command designated to be returned when the File | Open... menu selection is made.

Lines 100–101 call a special function, `execDialog()`, that processes the instantiated `TFileDialog` object. The first parameter to `TFileDialog`'s constructor, shown in this code as `pathname`, is the filename or wildcard pattern to display as

the default filename. The next two strings correspond to the dialog box title and the input field name, respectively. The last two parameters select specific features of the `TFileDialog`. As shown, the constants `fdOpenButton` and `fdHelpButton` instruct `TFileDialog` to place Open and Help buttons into the dialog box. To change this to a Save As dialog box, you need only to change the parameter strings in the constructor, and possibly the selection of dialog buttons.

`execDialog()` is a function provided by Borland in its TVEDIT1.CPP through TVEDIT3.CPP sample programs (in the \borlandc\tvision\demos directory). This function copies the value of its pathname parameter into the data transfer record of the `TFieldDialog` object, then displays `TFileDialog` as a modal dialog box. Upon exit from the dialog box, `execDialog()` retrieves the newly updated value of `pathname` from the dialog box's data transfer area.

If you are not a Turbo Vision programmer, I hope that this short example whets your appetite to learn more. Turbo Vision has a fairly steep learning curve, so you should be familiar with the use of C pointers and C++ classes and objects before starting to study Turbo Vision. But as soon as you master Turbo Vision, you can create top-quality Turbo Vision applications quickly and provide your applications with the look and feel of a modern user interface.

USING *TFILEDIALOG* IN OBJECTWINDOWS

ObjectWindows programmers can easily display a file dialog box of the type used in most Windows applications. Figure 6.1 shows a sample dialog box.

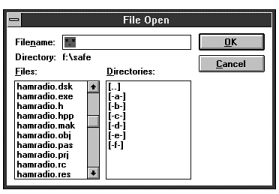


Figure 6.1. The standard Open File dialog box used in Windows applications.

You can insert this dialog box into your application using the remarkably small code fragment shown in Listing 6.14. This sample code uses the `TFileDialog` class provided by ObjectWindows. `TFileDialog` implements a

full-featured Open File (or Save As) dialog box. The dialog box features a scrollable list box showing all files in the active directory, plus a list of directories you can use to navigate through the file structure. The code shown in Listing 6.14 implements the Open File dialog box. To select the Save As dialog box, substitute SD_FILESAVE in place of the SD_FILEOPEN constant. The variable FileName must be declared to be MAXPATH (from dos.h) bytes long. You should initialize the FileName string to the default filename or wildcard pattern to appear in the Filename: field of the dialog box.

LISTING 6.14. HOW TO USE TFileDialog IN AN OBJECTWINDOWS APPLICATION.

```
char FileName[MAXPATH];
strcpy( FileName, "*.*" );
if (GetApplication()->ExecDialog(new TFileDialog( this,
                                SD_FILEOPEN, FileName )) == IDOK)
{
  // Open the file using standard file access functions
};
```

To incorporate the dialog box into your application, you must add two files to the resources for your application. These files are \borlandc\owl\include\owlrc.h and \borlandc\owl\include\filedial.dlg. If your resource file is defined using a resource script file, you can include these resources by writing

```
#include <owlrc.h>
rcinclude filedial.dlg
```

If you use the Resource Workshop, use the File | Add to project function to add both of these resources into the resource file for your application.

THE CONTAINER CLASS LIBRARIES

To set the record straight, *switching from C to C++ will not make you a more productive programmer—right away*. On the contrary, every new C++ programmer and every new project I have seen undertaken in C++ probably have taken longer to reach product completion than if a traditional language such as C or Pascal had been used. What happened?

The feature set of C++ is extensive and comes with a rather steep learning curve that must be climbed by new C++ programmers. Learning C++ is not merely learning new syntax but also new ways of thinking about software design and development. The power of C++ as a productivity enhancer does not come until after programmers have crossed the pinnacle of the learning curve *and* when their applications are able to inherit from preexisting classes. For your first few applications, you might not have any classes from which you can begin to inherit your application's features. Instead, you must write your classes from scratch. Writing those classes for the first time might take longer than if you had coded them using non-OOP methods. The classes might take longer to build initially because of the tendency to make them complete, accurate, and useful for future projects.

So what can you do to make your initial use of C++ more productive? The answer is to borrow from existing class libraries. Borland provides three primary class libraries—ObjectWindows, Turbo Vision, and the container libraries—to implement Microsoft Windows applications, DOS character-based windowing software, and data structures, respectively. For ObjectWindows and Turbo Vision, your application can inherit all the facilities needed to produce a complete, menu-driven application that supports dialog boxes and mouse pointing. The sample Turbo Vision application shown in Listing 6.13 is an example. With just a few statements (one for each menu item), you can expand this sample application to quickly display an application's complete pull-down menu structure. With just over 100 lines of code, you've created the entire infrastructure for your mouseable application.

UNDERSTANDING THE CONTAINER LIBRARIES

The container libraries can be used in any C++ program. These libraries provide generic data structure support for dynamically sized arrays, queues, lists, stacks, and specialized data structures such as Btrees, sorted arrays, bags, and dictionaries. Member functions provide searching and sorting capabilities. By using the predefined class libraries, either directly or by deriving new classes from them, your application can begin to put the power of true C++ programming to work *now*.

Borland provides two separate container class libraries—one based on *objects* and the other based on *templates*. Of these, I have found the objects-based library, which is described in this section, to be easier to use than the

templates-based model. The templates-based model, however, provides advanced functionality, especially in its capability to store any type of object, not just those derived from the `Object` class. Using the container libraries is not difficult (see the section "Using the Container Libraries"), but a few traps can slow your progress when you are working with container libraries for the first time. The sample programs in this section give you some specific sample code that you can put to work right away.

The name for the container libraries comes from the metaphor for their operation: they act like a container (a box, for example) that holds assorted objects of any type. Note the last statement carefully. A traditional array, by contrast, contains an array of elements in which each element is the same type. A container contains objects (or elements, if you prefer) that can be of any type. As an example, a container can be used to implement an array in which each element in the array varies from, for example, an integer here to a string there, or even a structure.

Each element that is placed in a container is derived from a common ancestor—the `Object` class (or in some cases, from the `Sortable` class, which is itself derived from `Object`). Because descendants of an `Object` class are type-compatible with their ancestor, there is no problem with storing differing `Object`-class descendants within a single container. Each object derived from `Object` has two virtual functions, called `IsA()` and `nameOf()`, that report the object's class identification and name. You define these functions for your object type so that at any time you can query an object in a container to learn what type it is.

USING THE CONTAINER LIBRARIES

Each element that goes into a container must be derived from the `Object` class. Several of the `Object` class functions are pure virtual functions, making the `Object` class itself an *abstract* class. You must provide definitions for these pure virtual functions in order to instantiate a useful object. Sample definitions are shown in Listing 6.15, a sample program using the `Bag` container library. A `Bag` is a data structure for tracking unordered collections of objects. A `Bag` is similar to a shopping bag—it contains a random, unordered collection of objects and may contain more than one of the same object (such as three cartons of whole milk, two cartons of skim milk, and 15 cartons of chocolate milk).

LISTING 6.15. A SAMPLE PROGRAM DEMONSTRATING USE OF THE BAG CONTAINER.

```
// BAGS.CPP
// Demonstrates use of the bag container library.
// Compile using the large memory model.
#include <iostream.h>
#include <object.h>
#include <bag.h>
#include <string.h>
#include <stdlib.h>

class TLogEntry : public Object {
public:
  TLogEntry(char * NewCallSign,
 char * NewContactNum,
 char * NewExchange,
 int NewMode,
 int NewBand,
 long NewTime,
 long NewDate ) : Object () {
    strcpy( CallSign, NewCallSign );
    strcpy( ContactNum, NewContactNum );
    strcpy( Exchange, NewExchange );
    Mode = NewMode;
    Band = NewBand;
    ContactTime = NewTime;
    ContactDate = NewDate;
       }
  virtual hashValueType hashValue() const;
  virtual classType isA() const {return __firstUserClass;}
  virtual int isEqual( const Object& testObject) const
    { if (stricmp( ((TLogEntry&)testObject).CallSign, CallSign ))
       return 0;
       else return 1;
     }
  virtual char *nameOf() const {return "TLogEntry";}
  virtual void printOn( ostream& outputStream) const
      { outputStream << CallSign;}
  friend void Display(Object& o, void *);
private:
  char CallSign[11];
  char ContactNum[11];
  char Exchange[35];
  int Mode, Band;
  long ContactTime;
  long ContactDate;
};

hashValueType TLogEntry::hashValue() const
```

continues

LISTING 6.15. CONTINUED

```
48  {
49    // This hash computation algorithm is adapted
50    // from Borland's STRNG.CPP implementation
51  /*
52    hashValueType   hash = hashValueType(0);
53    int len = strlen( CallSign );
54    for( int i = 0; i < len; i++ ) {
55      hash ^= CallSign[i];
56      hash = _rotl( hash, 1 );
57    };
58    return hash;
59  */
60    return atoi( ContactNum );
61  }
62
63
64  void Display(Object& o, void * )
65  {
66    cout << ((TLogEntry&)o).ContactNum << " " <<
67      ((TLogEntry&) o).CallSign << "\n";
68  };
69
70  void main(void)
71  {
72    Bag LogBook( 30 );
73    TLogEntry Contact1( "KF7VY", "0001", "59 EWA", 0, 0, 0, 0 );
74    TLogEntry Contact2( "N7VPL", "0002", "59 EWA", 0, 0, 0, 0 );
75    TLogEntry Contact3( "W6ZRJ", "0003", "59 SCV", 0, 0, 0, 0 );
76    TLogEntry Contact4( "N6IIU", "0004", "59 SF", 0, 0, 0, 0 );
77    TLogEntry Contact5( "N7LCG", "0005", "59 WWA", 0, 0, 0, 0 );
78
79    LogBook.add( Contact1 );
80    LogBook.add( Contact2 );
81    LogBook.add( Contact3 );
82    LogBook.add( Contact4 );
83    LogBook.add( Contact5 );
84
85    cout << "Total number of items in bag = " <<
86        LogBook.getItemsInContainer() << "\n";
87    LogBook.forEach( Display, NULL );
88
89    LogBook.flush();
90  }
```

The key to using the bag or any other container is to first define the object that will be placed into the bag. Lines 10–45 are the definition for `TLogEntry`, derived from `Object`. `TLogEntry` implements a data record from an actual ObjectWindows-based database program used to track ham radio contacts during a contest. In that application, a container library logs each contact. As such, the `TLogEntry` contains information about the contact, including

- The station callsign
- A sequential contact number
- The brief information exchange used during a contest, such as signal strength and location
- The mode of communication used (voice, code, packet, radio-teletype, or television) stored as an integer
- The radio band used, also stored as an integer
- The date and time that the contact took place

Private fields for this information are laid out in lines 38–44 and are set to appropriate values by calling the constructor function.

To successfully instantiate an object of this class, you must provide definitions for each of the pure virtual functions: `hashValue()`, `isA()`, `isEqual()`, `nameOf()`, and `printOn()`. It is extremely important that when you define these functions, you copy their definitions exactly. If you miss a `const` keyword, an `&` symbol, or a function parameter, misspell a function name, or inadvertently use a lowercase letter where you should have used an uppercase letter, C++ creates an overloaded function rather than redefining the inherited pure virtual function. When this occurs, the compiler outputs the error message `Cannot create an instance of abstract class '`*`classname`*`'`. Unfortunately, the compiler doesn't give you a clue as to which function is missing (Borland needs to improve this aspect of the compiler). Your only recourse is to stare at your code until you find the problem. Because it is easy to make a troublesome typographical error when deriving from an abstract class, you might want to copy the class definition directly from the object.h file (or the sample programs in this section).

The `hashValue()` member function computes a hash code using one or more data fields and a hash algorithm of your choice. A hash algorithm converts a key, usually text, into a numeric representation. An ideal hash algorithm produces a different number for each possible input string. In real life, though,

hash algorithms normally do not produce a unique hash code for different input strings. Consult a book on data structures or data structure techniques if you want to learn more about hash computation and the use of hash codes in hash table data structures. To use the container methods, you need only to know how to compute a hash code. Use the preceding sample code to do that for your applications.

`TLogEntry`'S `hashValue()` member function (lines 47–66) shows two methods of computing a hash code. The method shown within the comment brackets is a generic algorithm that is suitable for most string data. For the purposes of this application, however, the `ContactNum` field contains a unique numeric identifier. Converting the contact number to an integer provides a quick and straightforward hash computation for this application.

The `isA()` and `nameOf()` member functions serve to identify the object's class and name. The identifier `__firstUserClass` (see line 28) comes from clstypes.h, a definitions file automatically included by object.h. You can use this enumerated value directly or you can add an offset to it. `nameOf()` returns a pointer to an identifying string.

`isEqual()` compares two objects to one another to determine equality. In the sample code in Listing 6.15 (lines 30–32), the key field, the `CallSign` field, is tested for equality using the `stricmp()` string function from the standard library.

`printOn()` is called by the overloaded (`<<`) operator and is used to output the object to a stream. For the purposes of this example, the only value that is output is the `CallSign` field, but you can output other values too.

Defining the object to be placed into a container is the hard part. The easy part is creating the container, shown in line 72:

```
Bag LogBook( 30 );
```

This instantiates a `Bag` object named `LogBook`, having space for 30 items. Lines 73–77 create several `TLogEntry` objects, which are added into the `Bag` by calling the `Bag`'S `add` member function (lines 79–83). Use `getItemsInContainer()` to find out how many objects are currently stored in the container.

THE *FOREACH* ITERATOR

`ForEach` is a member function of the `Bag` class. *For each* object in the `Bag`, the `ForEach` member function calls a function that you have defined. Because of

this behavior, the `ForEach` member function usually is called an *iterator* function. This function is free to perform any appropriate operation, such as performing calculations on the data or, as in this example, displaying the content of the `LogBook` bag on the screen. `ForEach` is defined as

```
void ForEach( void (*actionFuncPtr()(Object& o, void *), void *args);
```

`actionFuncPtr` is the address of the function to execute for each item in the bag. In the sample code, it is a friend function named `Display` (see lines 37 and 64–68). The `void *` parameter is provided so that your programs may pass optional additional information to the called function. Line 87 invokes the `ForEach` iterator. Your program can have more than one function called by the iterator. To do this, define additional friend functions, as illustrated by `Display()`, and then call `ForEach` with the address of the other functions.

COMPILING THIS EXAMPLE

To compile this example (using the IDE), you should select the large memory model. (The example should work fine in other memory model configurations, but I encountered memory model conflicts when compiling with the small memory model). You must add the `classlib` directories to the compiler's search path. Do this by opening the Directories dialog box (Options | Directories...) and adding `\borlandc\classlib\include` to the Include Directories edit field and `\borlandc\classlib\lib` to the Libraries Directories edit field.

Select Options | Linker | Libraries... and select the Static or the Dynamic Container Class Libraries option, as appropriate. This option causes the linker to automatically include the appropriate container class library. If you do not select this option, you must add the container class library (TCLASSx.LIB, where x is S, C, or L, as per the memory model in use).

USING THE *SORTEDARRAY* CONTAINER

`SortedArray` implements a container that maintains an array of elements but keeps the elements in sorted order. Listing 6.16 shows an example based on the previous code that implemented a `Bag`. The most significant difference is that `TLogEntry` is derived from `Sortable` instead of `Object`. `Sortable` defines an additional pure virtual function, `isLessThan()`, used to determine the sort order of the elements in the array (see lines 34–38). `isLessThan()` returns 1 when its test object is less than the current object; it returns 0 otherwise.

LISTING 6.16. AN EXAMPLE OF THE SortedArray CONTAINER CLASS.

```
// SORTED.CPP
// Demonstrates use of the SortedArray class.
// Compile using the large memory model.
#include <iostream.h>
#include <sortable.h>
#include <sortarry.h>
#include <string.h>
#include <stdlib.h>

class TLogEntry : public Sortable {
public:
  TLogEntry(char * NewCallSign,
  char * NewContactNum,
  char * NewExchange,
  int NewMode,
  int NewBand,
  long NewTime,
  long NewDate ) : Sortable () {
    strcpy( CallSign, NewCallSign );
    strcpy( ContactNum, NewContactNum );
    strcpy( Exchange, NewExchange );
    Mode = NewMode;
    Band = NewBand;
    ContactTime = NewTime;
    ContactDate = NewDate;
       }
  virtual hashValueType hashValue() const;
  virtual classType isA() const {return __firstUserClass;}
  virtual int isEqual( const Object& testObject) const
     { if (stricmp( ((TLogEntry&)testObject).CallSign, CallSign ))
       return 0;
       else return 1;
     }
  virtual int isLessThan( const Object& Obj1 ) const
  {  if (stricmp( ((TLogEntry&)Obj1).CallSign, CallSign ) > 0)
       return 1;
       else return 0;
  }
  virtual char *nameOf() const {return "TLogEntry";}
  virtual void printOn( ostream& outputStream) const
      { outputStream << CallSign;}
  friend void Display(Object& o, void * );
private:
  char CallSign[11];
  char ContactNum[11];
  char Exchange[35];
  int Mode, Band;
```

```
48    long ContactTime;
49    long ContactDate;
50  };
51
52  hashValueType TLogEntry::hashValue() const
53  {
54    return atoi( ContactNum );
55  }
56
57
58  void Display(Object& o, void * )
59  {
60    cout << ((TLogEntry&)o).ContactNum << " " <<
61      ((TLogEntry&) o).CallSign << "\n";
62  };
63
64  void main(void)
65  {
66    SortedArray LogBook( 30 );
67    TLogEntry Contact1( "KF7VY", "0001", "59 EWA", 0, 0, 0, 0 );
68    TLogEntry Contact2( "N7VPL", "0002", "59 EWA", 0, 0, 0, 0 );
69    TLogEntry Contact3( "W6ZRJ", "0003", "59 SCV", 0, 0, 0, 0 );
70    TLogEntry Contact4( "N6IIU", "0004", "59 SF", 0, 0, 0, 0 );
71    TLogEntry Contact5( "N7LCG", "0005", "59 WWA", 0, 0, 0, 0 );
72
73    LogBook.add( Contact1 );
74    LogBook.add( Contact2 );
75    LogBook.add( Contact3 );
76    LogBook.add( Contact4 );
77    LogBook.add( Contact5 );
78
79    cout << "Total number of items in bag = " <<
80      LogBook.getItemsInContainer() << "\n";
81    LogBook.forEach( Display, NULL );
82
83    LogBook.flush();
84  }
```

Other than the inclusion of `isLessThan()`, the reference to `SortedArray` in line 66, and replacing `#include<object.h>` with `#include<sortable.h>` and `#include <bag.h>` with `#include<sortable.h>`, the sample program is nearly identical to the `Bag` example. If you would like to change the data structure to a `Btree` type, you can do that easily. Change `#include<sortarray.h>` to `#include<btree.h>`, and change line 66 to read

```
Btree LogBook;
```

WRITING ROBUST AND REUSABLE CLASSES

BY PETE BECKER

For a couple years now you've been hearing about how object-oriented programming will make your life easier by providing you with reusable code. You've also probably noticed that so far you haven't seen much code that's truly reusable, either from outside vendors or from the other programmers you work with. It's often too hard to understand, too limited in its capabilities, or simply too buggy. I think that's mostly because programmers haven't yet given enough thought to the problem of how to design code so it can be reused. Reusability doesn't happen by accident. It has to be designed into your code.

Getting the interface right

Documenting your code carefully

Designing your code for strength

Testing thoroughly

My work at Borland during the past two years has involved designing and implementing two reusable class libraries: Turbo Vision and the Borland International Data Structures (BIDS) library. Neither is perfect, but both have provided me with some insights into how to write code that's reusable. No one has the ultimate answer to reusability, but I hope that I can give you some useful tips.

Several years ago, a man I worked for told me a story about one of the planes used during World War II. I don't remember which one, but the rest of the story stuck with me. Aircraft typically require lots of maintenance, and during a war, that's a major drain on resources. Therefore, this plane was designed to be easy to maintain. The designers went to a lot of trouble to be sure that critical systems were easy to get at, that they worked as independently as possible, and so on. The plane was a major success—not because maintenance was so much easier, but because it turned out that in the course of designing the plane to make maintenance easy, they designed a plane that didn't need much maintenance.

The same applies to programming: if you make your code easy to maintain, it probably will also be easy to use. So I'm not going to make a clear distinction between users and maintainers. Both are equally important. In fact, I generally refer to them as *customers*. Don't forget that the customer is always right.

You need to do only four things to keep your customers happy: get the interface right, document carefully, design for strength, and test thoroughly.

GET THE INTERFACE RIGHT

One of the most obvious differences between C and C++ is the addition of classes. Viewed narrowly, classes enable the programmer to define a set of member functions that operate on a common set of data. The other side of the coin is that classes also provide exclusion: for private data, no other access paths are allowed. Both of those sides are critical in understanding what a class does.

For example, you probably recall struggling with the functions `fopen()` and `fread()` in the C library. It took a while to understand that they were meant to work together. Eventually you figured out the normal sequence: `fopen()`, `fread()`, `fclose()`. Then you had to figure out that `open()` didn't fit in. The Library Reference probably helped, but its discussion of how these functions interact

is minimal. You pretty much had to figure out the interconnections yourself.

When functions are grouped in classes, their interrelationship is much clearer. Because the class must be completely defined in one place, you don't have to hunt through long lists of functions looking for something that might be what you want, or to rely on an editor's ability to figure out the right cross-references. You only have to look at the class definition to find what you want. If it isn't there, it doesn't exist.

This also means that when you are designing a class interface, you must be sure that it's right. It must provide everything that belongs to that class, and nothing that doesn't—which, in turn, means that you must know what belongs to that class and what doesn't. You must be extra careful if you try to hack together a class definition on-the-fly. Getting the interface right requires foresight and planning.

That doesn't mean that you must write the definition first and then never change it. It does mean that you must think of the consequences whenever you consider adding a member function to a class. As you use the class, you'll undoubtedly think of things that it might be nice to have in the class. That's not a good enough reason to add them to the class definition, though. In addition, you must be convinced that the new function is consistent with what the class does.

BEHAVIOR DEFINES CLASSES

So far I've talked about a narrow view of classes as a mechanism for grouping data and functions. That's helpful for a C programmer moving up to C++, but it understates the power of classes. A broader view is that each class is an actor, capable of playing certain roles. Each class is defined by the roles it is capable of playing.

When I'm confused about a class I'm working on, I find it helpful to try to describe, in ten words or less, what the class does. Occasionally, when I've suggested this to other people, I get an indignant response—"Of course I know what my class does"—mostly from programmers who are having a terrible time deciding what belongs in that class and what doesn't.

If I can't describe a class's role completely and succinctly (all right, maybe it takes more than ten words), I don't understand the class. But once I have that succinct description in mind, it's easy to decide whether some proposed feature

should be part of that class. If the feature fits the class's role, it can be added; if it's out of character, it can't.

For example, suppose that you're sketching tentative class definitions for a windowing system for a text editor. What will some of the key classes be?

Obviously there must be a `Text` class to hold the text being edited and display it on-screen. To me, the word *and* in the previous sentence is a red flag: it sounds like this class is doing two different things, which might mean that it's really two classes. So try separating the text itself from the mechanism used to display it. This gives us two classes: a `View` class that represents a rectangular drawing region on the screen, and a `Text` class that contains editable text. From these brief descriptions, it's easy to put together a preliminary list of needed functions:

```
class Text        // Contains editable text
{
public:
      const char *textAt( int line, int col );
      void insert( const char *text, int line, int col );
      void delete( int line, int col, int length );
};

class View        // Represents a rectangular
{                 // drawing region on the screen
public:
      void move( int xPos, int yPos );
      void resize( int xSize, int ySize );
};
```

Nothing here allows scrolling of the text within a view. Where does that go? The description of the class `Text` doesn't say anything about figuring out how much of its contents to display, and the class `View` doesn't seem to know anything about what it's displaying. So how can you possibly scroll text in a `View`? For that matter, how can you display anything at all? At first glance, it looks like these abstractions of the behaviors of the two classes have lost what you were after in the first place: the capability to display text on the screen.

That's true. And it gives you a clue in the search for the right classes. You already have a class that can edit text, and you have a class that can manage a rectangle on the screen. Now you need a class that combines these two sets of behaviors. Let's call it `TextView`. `TextView` coordinates a `Text` object and a `View` object in order to display and edit text within a window on the screen.

```
class TextView          // Coordinates a Text object and a View object
{                       // in order to display and edit text within
```

```
public:                 // a window on the screen
      void draw();
      void scroll( int xDelta, int yDelta );
};
```

Okay, I admit it: I let an *and* slip back into the summary of what the class `TextView` does. In fact, there are two of them. It doesn't bother me nearly as much here as it did in the first case, because here it's clear that `TextView` is combining the behaviors of two other classes. It's hard to describe that without using "and"! What we've accomplished, though, is to factor out the operations that belong to the class `Text` and to factor out the operations that belong to the class `View`. This means that you can put whatever effort you need into designing and implementing `View` and `Text` more or less independently. The details of how they interact are important only when you're building a `TextView`. By focusing on the roles these classes play, you've come up with three logically consistent classes and eliminated the temptation to hack scrolling into the innards of the `Text` class or the `View` class. That means that both are simpler than they would have been otherwise, and it means that changes to either are much less likely to affect the behavior of a `TextView`.

You may have noticed that the preliminary definition of `TextView` doesn't make any commitment to how `Text` and `View` are combined. I suspect that `TextView` probably will inherit from both of them, but this isn't the time to make that decision. For now, all that's needed is to understand that `TextView` coordinates `Text` and `View`. In fact, even when examining existing class definitions to be sure that they are sensible, you should ignore the details of how the classes are implemented. Otherwise, you're likely to focus on implementation details and miss seeing the big picture.

As soon as you've identified the behavior of a class, by putting together a succinct description of the class and a list of its functions, it's time to move on to a more complete interface.

MAKE IT COMPLETE

As soon as you've roughed out a few classes for use in your current project, give some thought to how you might use those classes in the future. The time to fill them out is now, when you have the best understanding of what those classes are all about. If you wait until you need to reuse them, you'll have to figure them out all over again.

One of the best clues for finding additional features that should be added is to look for holes in the current set of functions. For example, what's missing here?

```
class complex
{
public:
      complex( double re, double im );
      double real() const;
      double imag() const;
      friend complex operator + ( complex c1, complex c2 );
      friend complex operator - ( complex c1, complex c2 );
      friend complex operator * ( complex c1, complex c2 );
};
```

Even though your current project doesn't need to divide complex numbers, this class demands that a division operator be added. Take the time to do it now.

Another area that must be complete is construction, destruction, and assignment. It's easy, in the rush to get code that works, to overlook some of the helpful things the compiler does for you. These helpful things might cause trouble for you later. Take a look at a simple example:

```
class String
{
public:
      String( const char *s ) { str = strdup(s); }
      ~String() { free(str); };
private:
      char *str;
};
```

Now suppose that your code does something like this:

```
void demo()
      {
      String s1( "Hello, world!\n" );
      String s2(s1);
      }
```

That code compiles correctly, but when you run it, it probably will crash. That's because the compiler generated a copy constructor to use for the construction of `s2`, and that constructor simply copied the pointer stored in `s1` into `s2`. When the destructors for the two classes were called, that block of storage was deleted twice. That's not a good thing to do.

The solution, of course, is to add your own copy constructor that duplicates the string:

```
class String
{
public:
      String( const char *s ) { str = strdup(s); }
      String( const String& st ) { str = strdup( st.str ); }
      ~String() { free(str); };
private:
      char *str;
};
```

Now the preceding code example will work correctly. But you're not done yet!

```
void demo()
      {
      String s1( "Hello, world!\n" );
      String s2( "Goodbye, cruel world!\n" );
      s2 = s1;
      }
```

Same problem: The compiler provided an assignment operator that copied the pointer, and the result, again, is that the same block of memory got deleted twice. The solution is also the same: Write your own assignment operator.

```
class String
{
public:
      String( const char *s ) { str = strdup(s); }
      String( const String& st ) { str = strdup( st.str ); }
      ~String() { free(str); };
      String& operator = ( const String& st )
            {
            if( str != st.str )          // Protect against s = s;
                  {
                  free(str);
                  str = strdup( st.str );
                  }
            }
private:
      char *str;
};
```

In general, any class that allocates any resources must have a copy constructor, a destructor, and an assignment operator. Otherwise, you'll end up with resources that can't be freed or resources that are freed multiple times. If you've written any one of these three members, you probably need the other two.

A third area to look at is output. There are two reasons that you might want to include a member in your class that writes the internals of the class out in text form. First, someone might want to use it in his or her program. What good is

a complex number if you can't display its value? Second, when debugging a program, it's often very helpful to be able to dump out the contents of an object so that you can see what's being done to it. Debugging is an integral part of developing classes. Make it easier on yourself by building in the things that you're likely to need later.

KEEP IT CONCISE

When I was first learning C++ I wrote the ultimate rectangle class. I looked at three different ways to describe a horizontal rectangle: specifying the coordinates of two points on a diagonal, specifying the positions of each of the four sides, and specifying the location of one corner and the height and width. The class I wrote let you talk about the same rectangle in any of these three ways, and you could use all three views interchangeably at any time. I chose wonderfully descriptive names for all the accessor functions, and I made them all inline and very tight and efficient. The class definition was about three pages long and impossible to read. Fortunately, the early version of Borland's C++ compiler that I was using at the time couldn't compile any serious program that used that class definition. The compiler didn't have enough memory capacity to handle that much inlining. I couldn't use that marvelous class definition, and it eventually disappeared from my hard disk.

A rectangle is a simple object. A rectangle class should also be simple.

That's not to say that those three different views of a rectangle aren't all useful. It's just that they aren't useful all at once. A much better design would have been to have three classes for rectangles, each providing one of the three views, and to have conversion functions from each class to each of the others. That keeps the mental clutter to a minimum for a customer who uses only one of those views.

Classes are cheap. Don't be afraid to use them.

NAMES ARE IMPORTANT

If you've taken a course in writing, you might have heard the rule "Never write anything unless you have a dictionary and a thesaurus within arm's reach." The same thing applies to programming.

I spend a lot of time thinking about the right names for classes and their members. It's just like writing well-polished English: If you choose the wrong word, it's hard for someone to understand what you're saying. Don't forget that you're writing for the benefit of your customers as well as yourself. Don't make it hard for them. Choose your words carefully.

Because a class is an actor, its name usually should be a noun. It should summarize the various actions that the class can perform.

The name of a function should describe what it does. That usually means that the name contains a verb: `seek()`, `showData()`. Sometimes the name can be a noun, if that's the best way to describe what the function does. This usually works well for accessor functions: `location()`, `size()`.

One of the worst names around, by the way, is `main()`. It doesn't tell you anything about what that function does. Rather, it tells you what role that function plays in the language. That puts the emphasis in the wrong place. In my code, `main()` never does much. In fact, if you've used Turbo Vision, you've seen how simple `main()` can often be:

```
int main()
{
      TVApplication tvApp;
      tvApp.run();
      return 0;
}
```

Maybe it's because I've done a lot of writing, but I like to use names that I can pronounce. When I'm looking at code, I read it to myself. After all, the more senses we use, the more likely we are to remember things. If the code is filled with names that are unpronounceable, that's much harder to do.

One of the worst offenders in this regard is *Hungarian notation*, promoted by Microsoft. It's unpronounceable, and like `main()`, it tells you about an object's role in the programming language, not about its behavior. Five years ago, when C was much more sloppy about type checking, it was a great idea. Today languages exist that are much more careful about type checking, and the benefits of Hungarian notation have largely gone away. The cost in readability and maintainability is now too high.

Choose your names carefully. They should clearly and concisely describe the things they represent.

USE STANDARD IDIOMS

One of the mistakes made in the first version of Borland's container class library was not putting enough emphasis on the `operator <<()` that it provides. That library has a base class named `Object`, and the library provides an inserter that takes a reference to `Object`:

```
ostream& operator << ( ostream& os, const Object& obj )
{
      obj.printOn( os );
      return os;
}
```

Each class derived from `Object` provides a definition of `printOn()`. Because `Object` declares `printOn()` to be a virtual function, the right version will always be called. So, regardless of context, you can always use the standard iostream idiom:

```
cout << data << '\n';
```

Somehow, though, this piece of information gets lost, and people end up calling `printOn()` directly:

```
data.printOn( cout );
```

To experienced C++ programmers, this just looks wrong. Iostreams use inserters, and that's the way they should be written.

The best way to learn to recognize standard idioms is to read code. I suggest that you be skeptical about general programming magazines that run an occasional feature on C++, though. Much of the code that I've seen there is awful. If you stick with the magazines that are devoted to C++ specifically, you'll get much better quality code.

DON'T GET CARRIED AWAY WITH OPERATOR OVERLOADING

Operator overloading is a powerful tool in the right context. It can be a disaster in the wrong one. I once saw a proposal for an interface to a database manager that used overloaded operators to invoke search operations:

```
dataBase <<= sample;
```

Obvious, isn't it? This line of code searches `dataBase` for all records that match `sample` and flags them as matches. Presumably, this will be followed by some other equally cryptic command that does something with the records that were found.

The most obvious place to use operator overloading is in the solution of problems that are basically mathematical. The code you end up writing in this case resembles the way that the problem would be expressed in the real world by a mathematician. As the problem that you're solving gets farther and farther away from mathematics, the usefulness of operator overloading diminishes.

When you create a class library to solve some set of problems, you're creating a language that can be used to express those solutions. Keep in mind that the people who need solutions to these problems usually have a technical language of their own. The language you create should be as much like their language as possible. If they don't use mathematical operators in describing their problems, you probably shouldn't use them in solving their problems.

The goal of programming is not to show how clever you can be. It is to provide a solution to a problem in a form that can be understood, used, and maintained by your customers.

DOCUMENT CAREFULLY

Code that isn't documented isn't reusable. This has always been true, of course, but it has become a much more serious problem as programming languages become more powerful and libraries do much more than they have in the past. I don't think that we know yet how to adequately document large class libraries. Just look at the documentation you get with any of the libraries you have. Can you really understand what the classes do without looking at the source code?

Documentation is more than printed manuals. It includes comments in the source code. Even if you don't make source code available to your customers, the header files should contain comments that give a summary of what each class is for and further notes on anything that isn't fairly obvious from reading the class definition.

Documentation also includes sample programs. Their complexity should be determined by the complexity of the library. Ideally, these samples should show everything that someone reasonably can do with the library. Just because it's obvious to you how the library should be used, don't assume that it's obvious to your customers. They might need to have their hands held for a while. Good sample programs can help.

THE PUBLIC AND PROTECTED INTERFACES

Whenever you document a class, you should keep in mind that there are three categories of programmer who will have to use that class. First is the user, who takes whatever you've provided and uses it in a program. The user needs to know about all the public members of the class. Second is the extender, who takes your code and adds to it by deriving classes from yours. The extender needs to know about protected members and any private virtual functions. Third is the maintainer, the programmer who must maintain the source code and who needs to know about everything.

In my class definitions I like to use the newspaper style of writing: put the most important stuff first. Make it easy to find the parts that will be needed most often. The same thing applies when writing documentation.

Begin by describing the public interface. This description should begin with what the class does and should be followed by a description of the public interface functions. Give examples.

Next, describe the protected interface. Assume that readers have already read about the public interface, and work from there. Describe the sorts of things an extender of the class is likely to do, and how those things can be done. Give more examples.

Finally, if appropriate, describe the private parts of the class. For internal documentation, this is essential. For a library that you will be distributing without source code, it's not needed.

DESIGN FOR STRENGTH

At one time or another you've undoubtedly had to deal with code that was so fragile that if you changed anything it broke. To help ease this problem for your customers, you should do all you can to insulate them from changes that you make when you revise your code and from errors that they make when they use your code. By designing and coding defensively, you make your code much more robust.

HIDING THE IMPLEMENTATION DETAILS

Anything that isn't part of the public or protected interface should be private. Usually the private part of a class consists of the data that keeps track of what's going on inside the class and a set of helper functions that deal with that data.

All data should be private.

Often when I say that, people respond with the example of a "simple" `Point` class:

```
class Point
{
public:
      Point( int x, int y );
      int xPos, yPos;
};
```

They object to making `xPos` and `yPos` private because they don't want to give up the flexibility of being able to assign directly to them and read directly from them. Aside from the fact that this is C thinking, there's a serious problem lurking here. Suppose that you've written a program using this definition of a `Point` as the basis for manipulation of windows on the display screen. Then you get the assignment of moving your code to a different operating system that has built-in windowing functions that reverse the sense of the vertical coordinates. That is, in your system, the top of the screen is at vertical position 0, and the bottom of the screen is, for example, 319. Under the new system, the bottom of the screen is 0 and the top is 319. What can you do? One possibility is that whenever you call the display routines in the system, you can convert your coordinates to the coordinates that the display system expects. That's fairly easily done, but what if the application you're writing is a CAD system, where you often pass coordinates to the display system, but you change them much less often? You might want to use the system's representation inside the `Point` class to eliminate all those conversions when you have to draw something. If your program reads and writes `yPos` directly, you're in for quite a bit of rewriting.

Now consider what would happen if `Point` had been written using accessor functions instead:

```
class Point
{
public:
      Point( int initX, int initY ) : x(initX), y(initY) {}
      int xPos() { return x; }
```

```
      int yPos() { return y; }
      void xPos( int newX ) { x = newX; }
      void yPos( int newY ) { y = newY; }
private:
      int x, y;
};
```

This version of `Point` is no less efficient than the previous one, but it does require thinking about a `Point` differently. Now, porting your program is trivial: just rewrite the accessor functions and recompile:

```
class Point
{
public:
      Point( int initX, int initY ) : x(initX), y(initY) {}
      int xPos() { return x; }
      int yPos() { return 320-y; }
      void xPos( int newX ) { x = newX; }
      void yPos( int newY ) { y = 320-newY; }
private:
      int x, y;
};
```

The key here is to realize that the representation of data within a class is rarely dictated by the class itself. Rather, it is often something that can be affected by outside factors such as the operating system that you're working with. By hiding it inside the private part of the class, you make the class much more flexible, because your customers cannot make assumptions about the particular data representation that you have chosen. This makes it possible for you to change the internal representation of the class without requiring customers to change their programs.

A NOTE ON ACCESS RESTRICTIONS IN GENERAL

When you are designing a class, members should be private unless you have a reason to make them protected or public. This can get you in trouble when users of your class find that there's a member function that does just what they need, but they can't get at it because it's private. If what they want to do makes sense, then making that function private was a mistake. Some programmers try to avoid this sort of mistake by reversing the rule just stated: They make every member public unless they can satisfy themselves that it doesn't need to be public. This is very dangerous. As soon as you've announced to the world that something is part of the public interface to the class, you cannot take it out

without breaking someone's code. It's perfectly okay to release a new version of a library with access restrictions eased for some members, because code written with the previous, tighter restrictions will still work. Going the other way and tightening restrictions is guaranteed to make your customers unhappy.

INTERNAL CHECKING

A truly useful class library protects itself from misuse. That means that its functions check that they have been called with valid parameters, that the data fields in the class make sense, and that their results make sense. All this checking can be expensive, though, so you should also provide your customers with a way to build their applications without this debugging code. Whether they want to risk doing that is up to them.

The `assert()` macro has been part of the C language for a long time, but it's grossly underused. Get to know it. It's one of the most powerful tools in the language. It's also very easy to use. Suppose that you're writing the code to move the cursor around inside a `TextWindow`. One of the design decisions that has already been made is that the result of trying to move the cursor to a position outside the window is undefined. That is, it's something the programmer shouldn't do. This may or may not be the right design choice, but as soon as it's made, the library should help programmers keep from running afoul of it. That's where `assert()` comes in handy:

```
#include <assert.h>

void TextWindow::moveCursor( int xPos, int yPos )
{
assert( xPos >= minX && xPos < maxX && yPos >= minY && yPos < maxY );
// Code to move cursor goes here
}
```

The `assert()` macro evaluates the expression that it gets as a parameter, and if the result is 0 it displays an error message and aborts the program. If the result is not 0, execution continues. In this example, the expression that it evaluates determines whether the parameters passed in would result in the cursor's being placed outside the window. If so, its result is 0, and you get an error message:

```
Assertion failed: xPos >= minX && xPos < maxX && yPos >= minY && yPos < maxY, file
test.cpp, line 10
Abnormal program termination
```

As soon as you have a working program, if you decide that performance is more important than such strict safety checking, the `assert()` can be removed by #defining NDEBUG. The easiest way to do that is in the .CFG file or in the project file. In a .CFG file, just add the line

```
-DNDEBUG
```

In a project file, select Options | Compiler | Code Generation | Defines, add NDEBUG to any other defines that may be there, and press Enter. Now recompile the entire project, and all the asserts will be gone.

The class libraries use a more flexible variation of `assert()`. The header file checks.h provides two macros, `PRECONDITION()` and `CHECK()`, that work in much the same way as `assert()`. In fact, they are implemented in almost exactly the same way, except that they give a different message when a failure occurs, and they have a different control scheme.

A `PRECONDITION()` is intended to test conditions at entry into a function. That is, if the precondition is not satisfied, the function cannot be expected to work correctly. Typically this is used to check input parameters. A `CHECK()` is used to test internal conditions within the function and to check for general problems. It's used for a general sanity check during development. For example:

```
1  void *allocate( int sz )
2  {
3         PRECONDITION( sz > 0 );
4         CHECK( heapcheck() >= 0 );
5         return new char[sz];
6  }
```

At line 3, the function checks whether it has been called with a valid parameter. Because `operator new` expects an unsigned value, giving it a negative number would result in a silent conversion to a rather large positive number. Because that's not what was intended here, the `PRECONDITION()` results in an error message if that happens. At line 4, the function checks whether the heap is corrupt. If so, it generates an error message.

`PRECONDITION()` is intended to flag usage errors. `CHECK()` looks for other sorts of problems. While you are developing your library, you should enable both forms of test. That's the default in checks.h. As soon as you're confident that the code itself is solid, you can disable the `CHECK()` macro and continue to use the `PRECONDITION()` macro. That enables you to continue to validate input while removing the internal checking. In fact, that's how the debugging versions of

the classlib that ship with your compiler are built: PRECONDITION() tests are enabled and CHECK() tests are disabled. To do that you #define __DEBUG to have the value 1. (Note that __DEBUG has two underscores in front of it.) In a .CFG file, this means adding this line

```
-D__DEBUG = 1
```

If you're using project files, it's Options | Compiler | Code Generation | Defines, and you add __DEBUG=1 to whatever is already there. Building a nondebugging version works the same way, except that you use the value 0 instead of 1.

TEST THOROUGHLY

Testing is a part of the process of developing robust software. Many programmers don't like to test, perhaps because they don't feel that it's productive. They would rather write code. That's not the right approach—code that isn't tested won't be robust. You must allow time for testing as part of your project plan, and you must do the testing.

JUST BECAUSE IT WORKS, DON'T ASSUME THAT IT'S RIGHT

One of the most common complaints that we hear when we release a new version of Borland's compiler is "My program worked perfectly when I compiled it with the old compiler; and now, when I compile it with the new one, it crashes. What's wrong with the compiler?" Almost always, the answer is that there's nothing wrong with the compiler. The program has a mistake in it that used to be harmless because of the way the old compiler laid out data, and now it happens to be fatal because of the way the new compiler lays out data. A dramatic example of this sort of problem actually came up in the Microsoft Windows SDK. One of their sample programs worked "perfectly" when compiled with their compiler, but it crashed when it was compiled with ours. I won't repeat the exact code here, but in essence, what the code did was this:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #pragma option -r-
5
```

```
 6  void getString( char *str, int len )
 7  {
 8      strncpy( str, "zzzzzzzzzzzzzzzzzzz", len-1 );
 9      str[len-1] = '\0';
10  }
11
12  int main()
13  {
14      int i;
15      int j;
16      char buf[18];
17
18      j = 0;
19      getString( buf, 20 );
20      i = j;
21      printf( "%d %d\n", i, j );
22      return 0;
23  }
```

When this program runs, it prints

```
122 122
```

Both values should be 0. Oddly enough, if you interchange lines 14 and 15, the program works "perfectly." That is, it displays the expected results.

Of course, you've spotted the bug in this program: the call to `getString()` at line 19 passes the wrong length, so too much data is copied. The value stored in `j` gets overwritten, and when that value is copied into `i`, i also gets the wrong value. When lines 14 and 15 are interchanged, `i` is overwritten. Because its value isn't used, this doesn't matter, and the program appears to work correctly. That's exactly the reason that the original code worked when compiled with Microsoft's compiler: it laid out the data in a different order, and it just happened that the bad function call didn't hurt anything.

If you replace line 19 with this, it will work correctly in all cases:

```
getString( buf, sizeof(buf) );
```

This sort of problem can be hard to recognize. Try compiling your code in a different memory model. That often flushes out hidden bugs. Or try changing the data alignment (`-a` on the command line). If you have a different compiler, try using it.

TEST EARLY, TEST OFTEN

The sooner you find bugs, the easier it is to fix them. That's why `PRECONDITION()` and `CHECK()` are so useful: they provide a painless way of catching serious problems. I assume that you eventually run all your code through a comprehensive test suite. But during the course of developing and maintaining a library, you also should be testing it. I tend to recompile and test my code after any significant changes. That sometimes means compiling and testing several times an hour. Of course, it's not practical to run a full test suite that often, because I would end up spending much more time testing than writing code. So what I usually do is write my own miniature test suite. I write a couple of short programs to test the area of code that I'm currently working on. I run these constantly. Somewhat less frequently, I also do what's known in the world of software testing as *touch testing*. That is, I run test code that executes every function at least once. It doesn't try to hit the obscure, nasty conditions that a more sophisticated test suite would look for. It's just a test of basic functionality, which I can use to be sure that I haven't made any major mistakes. For the class libraries, my private test suite produces about 30 executable files, which between them perform a total of about 2000 tests.

WRITE MEANINGFUL TESTS

When you're writing test code, make sure that the reports it gives are meaningful. When I first came to work at Borland, some of the runtime library test programs did things like this:

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

int main()
{
      printf( "cosh(0.5) == %lf\n", cosh(0.5) );
}
```

What does it mean if you run this program and it produces this output?

```
cosh(0.5) = 1.127626
```

Is the result correct?

Test programs must be self-documenting. At the very least, a program like the preceding one should also tell you what the result should be. Even better, it would tell you whether the result was correct:

```
#include <math.h>
#include <iostream.h>

int main()
{
      double res = cosh(0.5);
      const double correct = 1.127626;
      if( fabs(res-correct) < 1e-6 )
            {
            cout << "Result is " << res << ", passed\n";
            return 0;
            }
      else
            {
            cout << "Result is " << res << ", should be " << correct << ",
                     FAILED\n";
            return 1;
            }
}
```

Writing individual tests in separate programs makes for slow testing, though. This is just an example. You should combine tests into a few programs, but be sure that each test program provides meaningful reports.

KEEP YOUR CUSTOMER SATISFIED!

Whether your customer is someone who will be buying your libraries, or someone else who works for your company, or even yourself, six months from now, your success as a programmer depends on keeping that customer happy. That means thinking about your customer's needs, and designing your code to satisfy those needs. Get the interface right, document carefully, design for strength, and test thoroughly.

VIEWPOINT GRAPHICS IN C++

BY JOHN DLUGOSZ

This chapter covers the essentials of graphics programming and, in particular, a C++ approach to graphics programming using the ViewPoint library. The ViewPoint library was designed from the ground up to use C++ effectively. This chapter discusses the ViewPoint library, provides details on how it was designed, and describes the C++ features it takes advantage of.

The first section of this chapter is a quick introduction to ViewPoint. Then *world-coordinate* concepts and details are presented, followed by some tips on C++ class library design. This is followed by a tour of raster graphics primitives using ViewPoint for specific functions and a discussion of color on the PC, with details of palettes, hi-color, and gamma correction.

Transforming coordinates with `scaler`

Using modularly designed components

Touring ViewPoint's features

Using the mouse

Displaying graphics files

The chapter concludes with a sample application of a simple business graphics class, showing modular design and abstract base classes, as well as the use of some graphics primitives. The mouse is also discussed. Two sample programs to load and display Targa graphics files and PCX graphic files are also presented. The ViewPoint graphics library is a commercially sold product. The program examples in this chapter require the ViewPoint library in order to be compiled.

INTRODUCTION TO VIEWPOINT

Listing 8.1 shows an example of ViewPoint, introducing the overall flavor of the library. This sample program is the graphical equivalent of a "Hello world" program—one that shows how to compile and link and do something simple. This example program draws a square with an X centered in the middle of the screen. This example is mostly housekeeping and overhead, since the program itself does not do much.

LISTING 8.1. A SIMPLE GRAPHICS PROGRAM THAT USES THE VIEWPOINT C++ GRAPHICS CLASS LIBRARY.

```
 1  // VPINTRO.CPP
 2  #include "usual.h"
 3  #include "vp.h"
 4  #include "device.h"
 5  #include "extras\drvinit.h"
 6  #include "getkey.h"
 7
 8  screen_device d;
 9
10  int main()
11  {
12  activate (d);
13  viewport v (d);  //the whole screen
14  v.set_scale (0,0,999,999);
15  v.rectangle (333,333, 666,666);
16  v.line (400,400, 600,600);
17  v.line (400,600, 600,400);
18  key::get();  //pause
19  }
```

There are a number of include files for various features. ViewPoint is highly modular, and you include only the classes you need. More on this philosophy later.

Line 8 defines `d`, a `screen_device`. `screen_device` is a shell around loadable device drivers. The `device` class handles primitive drawing and such things as switching to graphics mode and back. When the program exits, the destructor will switch back to text mode. That is why `d` is a global variable rather than local to `main`—if the program terminates with an `exit()`, the destructor is called for all static variables and the screen is restored to text mode.

The first line in `main()` (line 12) is a call to `activate()`, which checks the display hardware, loads the proper driver, activates graphics mode, and does complete error checking and reporting along the way. It also provides for driver and mode overrides via an environment variable (see "Using Environment Variables" in Chapter 6, "Using Library Routines," for information about accessing and using DOS environment variables). Note that `activate()` is not a member function—it isn't even part of the library proper. Unlike the Borland Graphics Interface (see Chapter 9, "Graphics Programming in Borland C++"), which has detection and preassigned codes for several drivers and requies manual handling of "other" drivers, ViewPoint's screen device class knows nothing about detecting or driver filenames or paths. There is a member that loads a file, and another that activates a successfully loaded driver. The detection routines and any video standards are completely outside the class, and use only the public members. It can be rewritten or modified as desired.

After being activated, `d` is an active device, and any rendering primitives (member functions of class `device`) can be used on it. However, these drawing functions really are quite primitive, and they use absolute device pixels.

Depending on which driver you used, the number of pixels on the screen can vary, so drawing a rectangle in the center requires some calculation. You can determine the resolution of the display by checking `d.xmax()` and `d.ymax()`, and use some computations to adjust every point you plot to the actual display size. Performing these computations is a lot of trouble, but the library offers *world coordinates*, which are a way for the library to do all that calculating for you. `viewport` is a high-level class that implements the world-coordinate system and the mapping of world coordinates to screen coordinates.

A viewport is attached to some device. The definition of `v` (see line 13) specifies constructor arguments selecting the viewport to be attached to `d` and

taking up the whole screen. Now you can use the high-level commands in `v` instead of the low-level commands in `d`, but the scaling problem still remains—the pixels in `v` are exactly the same as those in `d`.

Line 14 solves this problem. Instead of having to adapt my code to deal with whatever display resolution is active at run time, I turn it around and tell it what size coordinate system I want to use. The call to `v.set_scale()` specifies the values that I want to call the upper-left and lower-right pixels in the viewport. Now, all use of `v` will use the resolution I specified, automatically mapping world coordinates to the physical device. This idea is explained in detail in the next section.

Line 15 draws a rectangle, centered in the 1000×1000 viewport. Lines 16 and 17 draw the diagonal lines to form the X. The last line waits for a keypress.

Note that there is no code for cleaning up. When `v` goes out of scope, the viewport is destroyed. When `d` goes out of scope (at program termination) its destructor will put the screen back in text mode, as if `d.finish()` were called.

WORLD COORDINATES

World coordinates refers to the use of a definable coordinate system rather than the natural system used by the screen. In the first example, the program centered and scaled the drawing assuming a 1000×1000 coordinate system. The alternative would be to examine the device resolution and perform computations to figure out what endpoints to use. Instead, the scaling ability is built into the library.

Besides making the whole screen appear in some arbitrary resolution, coordinate mappings can also be applied to small areas of the screen, or be used to make the coordinate system have the origin in the lower-left rather than the upper-left corner, or to implement many other drawing effects.

See the demo program SEGMENT for a demonstration of world coordinates. This program manipulates a test object like Silly Putty.

HOW IT WORKS

In the simplest system, you can scale the coordinates with simple multiplication. You can also add a constant to move the origin around. A formula such as

$$x' = a_1 x + c_1$$
$$y' = a_2 y + c_2$$

will be very general purpose. The scaling values a_1 and a_2 will magnify or reduce the axis, and you can map a 1000×1000 display into any size screen. To put the origin in the middle, set values to the *c* variables as well. To put the axis in the lower rather than upper corner, make a_2 a negative number.

The above function is general, but ViewPoint is even fancier. It includes an additional term, which lets one part of the coordinate interact with the other part. This allows for rotation and shearing. The *affine transform* is

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \vec{M} + \vec{c}$$

$$x' = a_1 x + b_1 y + c_1$$
$$y' = a_2 y + b_2 y + c_2$$

although it is usually expressed as a matrix. The *a* and *b* terms go into a square matrix, and the *c* terms (which just reposition the origin) are added in last.

IMPLEMENTATION

Clearly, a class can be written to abstract a transformation of this kind. The class will hold the six values, and have a member that takes points and returns transformed points. However, it is not as easy as it looks; doing it well is difficult. First, the values can't be integers. Scaling 1000 down to 640 real pixels, for example, requires a_1 to be 0.64. For the PC, floating point is too slow. Instead, fixed point numbers are used. Second, very close attention must be paid to the coding to prevent round-off errors. The `set_scale()` member, for example, is designed to hit the boundaries exactly, with no round-offs around the border of the rectangles. There are problems with the inverse operation. Instead of using the same six values with the inverse formulas, six different values are maintained, because doing it the simple way would be inaccurate.

CLASS *SCALER*

The result is a class called `scaler`, implemented by my colleague, Robert N. Goldrich, with an eye to detail and perfection.

For transforming coordinates, `scaler` provides a `scale()` function and an inverse function, `unscale()`. To set up a coordinate system, use the `set_scale()` function. In general, you give it two rectangles. The *physical rectangle* specifies screen coordinates. The *logical rectangle* specifies how you want to address the screen. After setting the scale, the idea is that scaling the logical rectangle gives you the physical rectangle as the result. So if you wanted to plot in that rectangle as a 100×100 grid with the origin in the standard place, the logical rectangle would be ((0,0),(99,99)) and the physical rectangle is the desired region of the screen.

The most common form is to assign a viewport to a specific area of the screen and establish a coordinate system in that viewport. Class `viewport` is derived from `scaler`, so all the scaling functions are available directly. New forms of `set_scale()` are provided to make it simpler to use. The sample program shows this: It specifies the desired size, assumes zero goes in the corner, and takes the physical rectangle from the viewport's bounds.

Other members of `scaler` modify the coordinate system. There are simple functions to change the scale, move the origin, mirror, rotate, and shear. The SEGMENT program on the companion disk shows all the interesting features available (source on disk). It always draws the same shape (a working digital clock), but lets you change the coordinate system.

PIE2.CPP (in Listing 8.6) always draws the same thing, yet it can fit in any rectangle anywhere on the screen, thanks to scalers.

DRAWING WITH SCALING

Although you have a simple way to transform points, you are not completely home free. Drawing a point is easy enough—transform, then plot. Ditto for lines—you transform both endpoints first and then plot.

But how about a rectangle? The rectangle may be rotated, so you can't use the fast frame rectangle drawing primitive (which uses horizontal and vertical lines). For that matter, it may be squished—it may look more like a diamond than a rectangle and contain no 90-degree angles.

In general, the amount of work done depends on the nature of the coordinate system. At its worst, a rectangle is rendered as a general polygon. Also, if the axes are aligned to the physical axis, it can do two transforms to find all the physical endpoints, while otherwise it must transform all four individually.

The device primitives know nothing of scaling and work entirely in device coordinates. The viewport-level drawing functions, on the other hand, always apply transforms. However, the viewport functions apply transforms in an intelligent manner, minimizing the computation that needs to be done. A number of property flags in class `scaler` will tell it if the axes are aligned, for example.

MODULAR DESIGN

As a library, the components of ViewPoint are designed to be independent of each other. There are layers, in that viewports use devices, but even in such cases the relationship is one-way. You can write a program that does not use viewports and uses device-level functions only. In this form, you do not have to link all the viewport support code into the program.

Even though viewports use devices, you as the programmer do not have to. The `viewport` class is in vp.h and the device classes are in device.h. You can include the former without having to include the latter. Your source file will know nothing about devices other than the fact that the name exists. Likewise, you don't have to include the pixarray.h file for class `pixarray`, even though both devices and viewports have commands to get and put `pixarray`s.

Each part of the library has its own header, and can often be used in any order and independently of each other. This does mean that a program, especially a simple program, has a lot of include files. Contrast this with other libraries that have a single huge include file.

There are several benefits to having lots of small headers instead of one big one. First, the "small" headers are not all that small. pixarray.h is 94 lines. vp.h is 280 lines, and device.h is nearly as large. There is a lot of stuff in the library, including several major class definitions. Cutting down on the size of the includes speeds up the compiler and uses less memory.

A more important benefit to the user of the library is the correspondence between the file's includes and its coupling with other modules. The quest to minimize the number of include files results in a module with fewer ties to other parts of the system. It also makes more explicit exactly what is used. For example, if you have a good-sized source file and make a change to one of the functions, then suddenly find you have to include another header, you know that a whole new part of the library is being used by that source file that was not being used before.

In general, knowing what components your modules are dependent upon helps promote good programming.

A TOUR OF FEATURES

Viewpoint is a rendering library. It has many functions for drawing various simple things. This section provides an overview of the drawing features in the ViewPoint library.

Most rendering functions use a number of common drawing parameters, which are collected into a style structure. When you draw an object, it appears in some color or colors, using some logical operation and some pattern. All this is in an instance of class `gfxstyle`, and passed to the device as a single parameter.

Each viewport has its own style. You never pass the style, because it is contained in the viewport and implicitly used in all drawing operations in that viewport. You can have more than one at the same time, each with its own settings.

In the first example program, the color of the drawing could have been specified by writing

```
v.style.FPen= 7;  // foreground pen is white
```

or something similar. There is an `FPen` (foreground pen) and a `BPen` (background pen).

For drawing lines, you can set a 16-bit line pattern. The 1s in the pattern are drawn using the `FPen`, and the 0s are drawn using the `BPen` or may be left transparent. A logic operation is applied to all pixels drawn: `REPLACE`, `AND`, `OR`, or `XOR`.

When drawing a polyline (several connected line segments), the line pattern is continuous across the entire polyline. It does not start over with each segment, like most other libraries do. This is a general ability: The style can be set to preserve line patterns in progress across multiple calls, so lines connect smoothly.

Another fine touch for using patterns is the *open interval*. Say you draw from A to B and then from B to C. Point B is drawn twice. If you are using `XOR` mode, this is bad; if you are trying to make your line pattern continuous across the segments, this is bad. With an open interval, the first line does not draw from A to B but skips the last pixel. B is not drawn twice. With diagonal lines and scaled coordinate systems, this would be nearly impossible to do manually. But ViewPoint supports it as a low-level primitive, where it belongs.

There are other `gfxstyle` fields for dealing with filled shapes. A filled shape can be drawn in a solid color using the `FPen`. Or it can use an arbitrarily sized fill pattern. A one-plane fill pattern is just like the line pattern in two dimensions. That is, it uses `FPen` for the 1s and your choice of `BPen` or transparent for the 0s. Or a fill pattern can be full color. In all cases, the logic modes apply.

Fill patterns work the same way on all filled shapes—rectangles, polygons, ellipses, even flood filling.

LINES

As you have already seen, there are functions to draw lines. Thanks to overloading, almost all of them are called `line()`. There is a `line()` member of the `device` class, which is a primitive function. There are five functions called `line()` in the `viewport` class. Different overloaded forms let you draw between two points to a point from the current point. You can specify the points as `x` and `y` `int`s, or as a pair structure. Thanks to overloading, the library can be made easier

to use. Instead of many different functions, a single function name will do. It is more practical to supply many variations using different parameter types; you can use whatever you have on hand rather than fitting the data to call. This simplifies your program.

Besides `line()` there are other functions to draw polylines, move the current point, and move in single directions. Calls can be chained, so you could say something like

```
v.right(10).down(10);
```

FILLED POLYGONS

The `viewport` class contains six forms of the `filled_polygon()` function. Parameters can be arrays of integers or pairs, and a starting point can be specified as two integers, a pair, or left out. Polygons are very difficult to do correctly in raster graphics. ViewPoint implements correctly meshing polygons. That is, two polygons that have the same edge in common will mesh together seamlessly. No pixel is drawn twice. Only points that are strictly on the inside of the polygon are drawn (called *rounding in*), and a decision process is used to handle points on the edges and corners. This means that a polygon mesh can be drawn to form complex shapes.

But sometimes you don't want meshing polygons. For a single polygon sitting out there by itself, it can appear "shaved." You want it to round off rather than rounding in. You don't want the left corner pixel to be zapped. So you can also specify polygons with Bresenham edges; that is, the polygon will exactly fit over a set of ordinary lines drawn between the same points. You also have your choice of odd/even or winding fill rules.

ELLIPSES

There are 21 ways to draw lines, and six ways to draw filled polygons. And that's just counting the function calls, not the variations in the mode parameter for polygons or all the different settings in the style. How many ways can you draw ellipses and ellipse-related things? More than a few dozen—more like hundreds!

For ellipse drawing, you can specify center-radii, specify bounding boxes in several different ways, and so on. Then you can draw an arc, where the arc can be specified in several different ways, or you can draw the whole thing. The number of functions you'd need would be into the double digits at the minimum. Overloading may be handy, but this is too much to be practical, and there are ambiguity problems, too.

So a better way was invented. The different properties of the rendering command are specified by individual calls, and several forms of each are available. This means that the total number of effective calls is equal to the product of the numbers of its parts.

Here are some examples:

```
v.ellipse(center,rad_x,rad_y).draw(); //draw whole ellipse
v.ellipse(bounding_box).arc(angle1,angle2).draw(); //draw arc
```

Notice that the second line is three functions. The first call, `ellipse()`, could have used up to six different forms to specify the same shape. The second call, `arc()`, could use two different forms or be left out entirely to draw the whole thing. And finally, the third call, `draw()`, does the rendering of the built-up definition. Instead of `draw()` you could have used `filled()` or `pieslice()` or others. This paragraph touches on 54 (though only 48 are useful) effective rendering functions.

FLOOD FILLS

Viewports support flood fills. As you've probably guessed by now, there are several ways to do it. First of all, flood fills use the fill pattern in style data the same way as any other filled shape. Even if you are filling with a pattern and a logical operation, it will not get confused. Any shape can be filled properly no matter what modes, colors, or styles are in use. You can have flood fill search out a border color and stop at the border. Or you can have it fill over a region of a color and stop when it hits something that is not that color. For simpler fills, you can specify the fast fill mode, which does not work with arbitrary shapes but is just fine for convex shapes and is much faster. There is also a `change_color()` function that will change all pixels of one color inside a specified rectangle into another color. This function is extremely fast, and is great for things like highlighting menu choices.

BITMAPS

There are move and scroll functions that copy rectangular blocks of pixels from one place on the screen to another. Naturally, you have many choices of parameters and modes.

Moving is really a special case of bitmap manipulation—`get` and `put` functions. Bitmaps are a major part of the ViewPoint library, and they even have their own class: class `pixarray` has over three dozen functions in it.

A `pixarray` holds a bitmap image. It manages its own memory, and will use the far heap even in a small model program. You don't have to allocate memory for it; that is automatic. `get`ing an image is as simple as

```
pix= v.get(r);
```

where *`pix`* is a `pixarray`, *`v`* is a viewport, and *`r`* is a `rect`. You can use different forms for the parameters besides a `rect`. Bitmaps can be used in expressions as high-level objects.

The flip side of a `get` is a `put`. Just putting a `pixarray` at some position would be too boring. Using the logical operations is better. But there is even more variety: You can `put` just part of a `pixarray`, magnify while `put`ing, or both.

Besides a place to hold an image between a `get` and a `put`, class `pixarray` can be used to manipulate the image. A simple example is to recolor it. You can map one set of colors onto another, or extract individual colors. You can query the `pixarray` for its size and other information, read and write individual pixels within it, and even copy parts from one bitmap to another. A more exotic function is `ortho_transform()`, which is really eight functions rolled into one. It can mirror, transpose, and rotate (in 90-degree increments) an image. There are also functions to load and save `pixarray`s to disk files.

FONTS

ViewPoint has a set of font classes. There is an abstract base class `font` that provides an interface independent of the actual font implementation. There are derived classes for fixed and proportional fonts. Over 40 fonts are supplied with the library.

MICE

ViewPoint features *intelligent mouse classes*. These are a progressive, modern way to handle the mouse in a C++ program. Use of the mouse is explained and demonstrated in more detail in "Using the Mouse," later in this chapter.

COLOR

There are a number of types of color in current PC video systems. The simplest is the monochrome display. There is one bit per pixel, which is either bright or dark. The Hercules display adaptor card is monochrome and is still in common use because it can cohabit a machine with a VGA card, giving your PC two screens. Also, the monochrome monitors and cards are very cheap.

In the early days, there was also CGA. This has a monochrome mode and a four-color mode that is very crude by modern standards. You have a choice of several palettes, each with three fixed colors. The fourth color could be set to one of 16 possible colors.

The next round, EGA, introduced proper 16-color graphics. There are 16 colors, each of which may be selected from a palette of 64. VGA added more resolutions, and upped the specification to provide 16 out of 2^{18} or 262,144 colors. Meanwhile, the MCGA offered a 256-color mode, using the same 262,144 color palette. Super VGAs work the same way for 256 colors, but with greater resolution.

Some new VGAs give 16 or 256 colors out of a palette of 2^{24} or 16,777,216 colors.

COLOR PALETTES

So you can see the pattern: Typical color video systems operate in a paletted mode offering 16 or 256 colors out of a selection of many more. This is like the painter's palette—out of all the millions of kinds of paint manufactured or mixable, he will set up a few blobs of paint on his small wooden board. He paints the entire picture with only a few colors, not the millions possible.

Attention to the palette is important in computer graphics, too. Mixing the colors is done with light, and the intensity of the red, green, and blue primaries

is individually selected. On the EGA, there are two bits for each value. So red can be off, low, medium, or high. Likewise for green and blue. The result can be encoded as 6 bits (2 bits for each color), providing 26 or 64 choices.

The VGA uses 6 bits for each primary. This gives 2^{18} choices, or over a quarter million separate color combinations. Some boards (and this is getting more and more popular) use 8-bit primaries instead, providing 2^{24} or over 16 million possible colors.

There are two problems with arranging your palette. The first is that each generation of card uses a completely different way to set the palette. ViewPoint solves this problem because the `set_pen_color()` member works the same way for any card and tries to hide the differences. The second problem is more serious. The program might be designed for rich color abilities but have to work with a display that is limited to showing fewer colors.

ViewPoint has a pair of members in the device class. `set_pen_color()` sets one palette (color table) entry, and `set_pen_colors()` sets a group of entries given an array of values. The actual colors are specified as 32-bit numbers, with one byte each for the red, green, and blue entries. It is simple to write these as hex constants in the form 0x*RRGGBB*, with two hex digits for each component, where *RR* corresponds to the red value, *GG* to the green value, and *BB* to the blue value. On a VGA with an 8-bit DAC (the DAC is the *digital-to-analog converter*, the part that sends the signals to the monitor. The palette color table is stored in the DAC chip); the numbers are taken as is. On a common VGA with a 6-bit DAC, the numbers are rounded off. On an EGA, the numbers are severely rounded. But it is the same function for all devices with palettes.

TRUE COLOR

Some VGA modes do not use palettes. A 16-bit hi-color mode will have 16-bit pixels, with the meaning of each pixel fixed. No lookup table is used. Rather, there are 5-bit components within the pixel. Other modes will have 5 bits for two components and 6 for the third, and 24-bit color mode has 8 bits for each component.

So instead of a program creating a palette, it has to find the pen value to draw in the proper color. This means knowing how the pixel is encoded. For example, in the 5-5-5 hi-color modes you can do the trick with

```
(red<<10)¦(blue<<5)¦(green)
```

The problem is that this is different for various cards and modes. The `device` class has some fields that will describe the coding scheme used (as well as tell you that it is a true color device in the first place), but it also has members to do it all for you. `device::truecolor_encode()` will take the 32-bit color specification as used with `set_pen_color()` and return the closest pen that draws in that color. There is also a `truecolor_decode()` function.

GAMMA CORRECTION

The *gamma* of a monitor describes the way color intensity works. Ideally, if you double the color value of a primary, it would appear twice as bright on the screen. Well, it does not work that way. Most cards and monitors operate linearly so that doubling the pixel value doubles the brightness. However, the human eye senses light changes logarithmically, not linearly. For example, a photographer will measure light in *stops*, where each stop looks to the eye like a linear progression but actually contains twice as much light.

Monitors vary as to how bright they display color, and video cards vary as to how hard they drive the signals. If the eye were linear, this would simply change the overall brightness. However, changing the brightness changes the position of the log curve, which affects the overall range of brightness values. So the image can become too compressed with darks and lights too close together, or washed out with darks and lights too far apart. Since the effect applies to each RGB primary, it can shift the colors too, making the image too "warm."

The bottom line is that, to display any given intensity α from 0 to 100%, you apply a "gamma correction" by raising the original value to a power. Computing α^{γ} will give you a correct image, once the proper γ for your monitor is found. Typical values on a PC are 1.3 to 1.8.

A graphics file on a PC is typically encoded to take the gamma correction into account. That is, just display the values in the file and you see what the artist intended. However, to do any kind of image manipulaion you need to "uncorrect" the gamma first.

A SIMPLE BUSINESS GRAPHICS CLASS

Enough theory; now on with some actual programs. Say you want to write a program that, among other things, displays some results as simple graphs. If a couple of different graphs are required, and if this may be used in more than one program later on, you should design a good general-purpose class. The class should be flexible and reusable. That's the whole point of making general-purpose code.

Start thinking *objects*. Obviously, a graph is an object. It can display itself somewhere on the screen. Also, you need to set it up and initialize it with the data and parameters.

On the abstract level, it matters very little what *kind* of a graph it is. That matters only to the drawing function. Therefore a good design might use an abstract base that represents some kind of graph, and a derived class for say, a pie chart. Other kinds of graphs (such as a bar chart) can be added later, and the program that uses it can switch between pies and bars with no changes. The example program on disk uses a stacked bar chart as well as a pie chart, and can switch between them.

Besides the all-important `draw()` function, the chart class needs to be told what to draw. It has members to tell it what the data values are and what the styles are. Each element (pie wedge, bar, whatever) is drawn using any specified colors and pattern. The second function specifies the colors and patterns, and draws in each element. A third function gives a list of labels for the data, which is not yet used for anything.

LISTING 8.2. THE `chart` DRAWING CLASS.

```
 1  // chart.h
 2  struct chartstyle {
 3     pentype FPen;
 4     pentype BPen;
 5     unsigned Flags;
 6     pixarray* pattern;
 7     int patternpage;
 8     };
 9  extern chartstyle default_style_list[];
10  class chart {
11  protected:
```

```
12     int* Values;
13     int count;
14     int valuescale;
15     chartstyle* Styles;
16     char** Labels;
17  public:
18     chart():count(0),Labels(0),Styles(default_style_list) {}
19     void data (int values[], int count, int valuescale);
20     void styles (chartstyle[]);
21     void labels (char* []);
22     virtual void draw (viewport& v) =0;
23     };
24
25  class piechart : public chart {
26  public:
27     void draw (viewport& v);
28     };
```

From the class definition, you can see how it is put together. The various things are stored inside the class, and members are provided to set them. There are also suitable defaults, so you don't have to give explicit labels, and if you don't give it any styles it will use the built-in list. The idea is to make it minimally usable with very little work, yet customizable as needed.

The `data()` member takes a list of `int`s and a scale value. The scale value is there to prevent the need for fractions, yet retain flexibility in specifying data. If the scale is 100, the data values are percentages. But rather than being hard-wired to take values scaled by 100, the class lets that be specified too. On a pie chart, scaling by 360 enables you to specify values in degrees.

Listing 8.3 shows the implementation of the `chart` class. The interesting part here is the `piechart::draw()` function. It has access to all the data and settings, so it can just do the drawing and not worry about where it all came from.

LISTING 8.3. IMPLEMENTATION OF THE `chart` CLASS.

```
1  #include "usual.h"
2  #include "vp.h"
3  #include "patterns.h"
4  #include "chart.h"
5
6  chartstyle default_style_list[]= {
7     { 1,0,0,0,0 },  //solid blue
```

continues

LISTING 8.3. CONTINUED

```
 8     { 2,0,0,0,0 },  //solid green
 9     { 4,5,USEFILLPATTERN,&macpat,3},
10     { 6,7,USEFILLPATTERN,&macpat,4},
11     { 4,1,USEFILLPATTERN,&macpat,5}
12  };
13
14  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
15
16  void chart::data (int values[], int count_, int valuescale_)
17  {
18  Values= values;
19  count= count_;
20  valuescale= valuescale_;
21  }
22
23  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
24
25  void chart::styles (chartstyle styles[])
26  {
27  Styles= styles;
28  }
29
30  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
31  void chart::labels (char* labels[])
32  {
33  Labels= labels;
34  }
35
36  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
37  void setstyle (gfxstyle& style, const chartstyle& chs)
38  {
39  style.FPen= chs.FPen;
40  style.BPen= chs.BPen;
41  style.Flags= chs.Flags;
42  style.FillPattern= chs.pattern;
43  style.PatternPage= chs.patternpage;
44  }
45
46  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
47
48  void piechart::draw (viewport& v)
49  {
50  const int radius= 400;
51  v.set_scale (1000,1000);
52  v.move_axis (500,500);
53  int startangle= 0;
54
```

```
55  long totalvalue= 0;
56  for (int loop= 0;  loop < count;  loop++) {
57     totalvalue += Values[loop];
58
59     //convert to degrees
60     int endangle= (totalvalue * 360) /valuescale;
61     setstyle (v.style, Styles[loop]);
62
63     v.circle(0,0,
64        radius).arc(startangle,endangle).filled_pieslice();
65     startangle= endangle;
66     }
67  }
```

The graph class will draw in any specified viewport, so it can appear anywhere on-screen and be any size. This is specified outside of the class. The viewport is created by the caller and passed in. That also keeps the class simpler and more flexible.

Before drawing a pie chart, the drawing area is prepared. Since it can be drawn anywhere at any size, it creates its own scale in the viewport. Now subsequent code does not care what the actual size is. It always draws a circle that is 400 units in radius.

For each wedge to draw, the start and end angles are computed. This is done by finding the ratio of the cumulative value of that slice to a 360-degree circle. The code assumes that the values add up to the value scale value. That is, for a pie, the values should all be fractions of the whole. Note the expression for `endangle` is arranged to prevent round-off errors or truncation problems.

To draw the slice, the values in the style for that slice need to be stuffed into the viewport's style. Since this will be done for all kinds of graphs, not just pies, this was made into a function. It is general-purpose and could be moved up to the base class and used elsewhere.

Now comes the interesting part. The line that draws the wedge is made up of three function calls. The first specifies the overall shape of the ellipse (center and radius in this case). The second specifies the angles, and the third tells it what to do with all that.

Listing 8.4 is a test program that creates a viewport using the entire screen and draws a pie. (Listing 8.5 in the next section is a bit fancier.)

LISTING 8.4. A PROGRAM TO TEST THE PIE CHART DRAWING CAPABILITY.

```
 1  // TESTPIE.CPP
 2  #include "usual.h"
 3  #include "device.h"
 4  #include "extras\drvinit.h"
 5  #include "vp.h"
 6  #include "getkey.h"
 7  #include "chart.h"
 8
 9  screen_device d;
10
11  int sampledata[]= {10,15,30,35,10};
12
13  void main()
14  {
15  activate (d);
16  piechart pie;
17  pie.data (sampledata, 5, 100);
18
19  viewport v1 (d);
20  v1.style.BPen= 8;
21  v1.clear();
22  pie.draw(v1);
23
24  key::get();
25
26  }
```

USING THE MOUSE

The mouse classes are powerful and simple to use. Rather than encapsulating the primitive mouse cursor control that is built into the mouse driver, the mouse classes are desiged from scratch.

The main class involved is `mouse_cursor`. A mouse cursor describes the appearance of the mouse and where it appears. The cursor can be any size and can contain any number of colors and logic modes. It also works on all devices, including super VGA modes, hi-color modes, and on fancy GUI accelerator cards.

The mouse cursor is associated with a rectangular region of the screen and can also have other constraint flags. Every time the mouse moves, all cursors are searched and the proper one found based on the region and flags. You can easily have the mouse change appearance as it moves to different areas in the screen. There is no work once you have defined the cursors! And when cursors go out of scope or are destroyed, their destructors take care of updating the system.

The class has members to ask about the mouse, such as its current position and the state of the buttons. It is fairly simple to use, though completely polled. The mouse tells you what is happening at the time the call is made.

THE MOUSE EVENT QUEUE

There is also a mouse event queue built in. With the queue activated, mouse events are buffered, and you can ask about the next unread event. There are various queuing modes. You can set it so the queue is not overwhelmed with mouse events, or completely ignores mouse events. There are also modes for specifying what to do if the queue overflows.

There is a `mouse_event` class which has a member to get the next record from the queue. Other members wait for specific events alone or in combination. After that returns, data members of the mouse event object will hold the information about the mouse at the time the queue entry was made, including the position, button states, and the time.

Listing 8.5 shows a function that lets the user select a rectangle on the screen using a rubber-band box. It uses the mouse queue.

LISTING 8.5. IMPLEMENTATION OF A BOX THAT CAN BE STRETCHED USING THE MOUSE.

```
 1  // RUBBER.CPP
 2  bool get_rubber_box (rect& r)
 3  {
 4  mouse_event e;
 5  gfxstyle style (15,0,XOR,0,0);
 6  e.wait_any (2¦8);  //wait for either left or right button down
 7  if (e.buttons & mouse_cursor::Rbutton) return FALSE;  //quit
 8  mouse_cursor::global_hide();  //turn off cursor
 9  r.a= r.b= e.pos;
10  d.rectangle (style, r.a.x, r.a.y, r.b.x, r.b.y);
```

continues

LISTING 8.5. CONTINUED

```
11
12  do {
13     e.wait();
14     //erase the old
15     d.rectangle (style, r.a.x, r.a.y, r.b.x, r.b.y);
16     r.b= e.pos;
17
18    //draw the new
19     d.rectangle (style, r.a.x, r.a.y, r.b.x, r.b.y);
20     } while (e.buttons & mouse_cursor::Lbutton);
21
22  // erase box
23  d.rectangle (style, r.a.x, r.a.y, r.b.x, r.b.y);
24  mouse_cursor::global_hide_off();  //restore cursor
25  return TRUE;
26  }
27
28  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
```

THE PIE CHART REVISITED

Listing 8.6 is a program that ties together several topics. It draws pie charts as did the previous program. But it demonstrates scaling transforms by letting the user select an area of the screen and drawing the chart to fit. It uses the rubber-band box for input.

LISTING 8.6. THIS PROGRAM USES THE MOUSE TO SELECT A REGION INTO WHICH A PIE CHART IS DRAWN.

```
 1  // PIE2.CPP
 2  #include "usual.h"
 3  #include "device.h"
 4  #include "extras\drvinit.h"
 5  #include "vp.h"
 6  #include "mouse.h"
 7  #include "mouseq.h"
 8
 9  #include "chart.h"
10
11  screen_device d;
12  int sampledata[]= {10,15,30,35,10};
```

```
13
14  void showchart (chart& ch, const rect& r)
15  {
16  viewport v (d, r);
17  setz (v);  //hide mouse cursor
18  v.style.BPen= 8;
19  v.clear();
20  ch.draw(v);
21  clearz();  //mouse cursor normal
22  }
23
24  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
25
26  int main()
27  {
28  static mouse_event queue[16];
29  activate (d);
30  piechart pie;
31  pie.data (sampledata, 5, 100);
32  if (!mouse_cursor::init(d)) return 3;
33  mouse_event::create_queue
34       (queue,16, mouse_event::CollapseMoves);
35
36  for (;;) {
37     rect r;
38     if (!get_rubber_box (r)) break;
39     showchart (pie, r);
40     }
41  mouse_cursor::finish();
42  return 0;
43  }
44
45  \endlisting
```

DISPLAYING GRAPHICS FILES

Many graphics images are generated and stored by different software, and you may want to access these images in your program. Reading a graphics file is a simple matter once you know the format.

As a first example, Listing 8.7 shows a program that displays a 16-bit Targa file on a hi-color equipped card. This lets me view the output from my favorite ray-tracing software.

The program is very short, mainly because it deals with only a single file format. TGA files come in many flavors, and can have different options. The program is meant for 16-bit pixels with no compression, stored from top to bottom and left to right.

The bulk of the program is spent checking the graphic image file to make sure it is suitable.

LISTING 8.7. THE TARGA FILE READER.

```
// SHOWTGA.CPP
#include "usual.h"
#include "device.h"
#include "extras\drvinit.h"
#include "ezfile.h"
#include "pixarray.h"
#include "getkey.h"
#include <stdlib.h>  //need exit()
screen_device d;

void error (char* message, int error_code= 3)
{
error_out (message);
exit (error_code);
}

int main (int argc, char* argv[])
{
if (argc != 2) error ("no filename given.", 1);
ezfile f;
struct {
   byte misc[8];
      // various fields, some mis-aligned so
      // I can't use a proper struct.

   // the interesting part (following) is
   // all aligned OK, though
   int xorg, yorg;
   int width, height;
   byte bits_per_pixel;
   byte flags;
   } TGAheader;     //Targa file image header

if (! f.open (argv[1], &TGAheader, sizeof TGAheader, 0,0))
   error ("can't open file");
// check over the file
if (TGAheader.misc[2] != 2)
```

```
38     error_out ("expecting True color, uncompressed image.");
39  if (TGAheader.misc[1] != 0)
40     error ("can't handle a color map");
41
42  if (TGAheader.bits_per_pixel != 16)
43     error ("program expecting 16-bit TGA file");
44
45  //got that far, now activate the graphics card
46  drvinit_default= "svga16.drv";
47  activate (d);  //assume resolution was chosen
48                 //externally to the program!
49
50  pixarray row (d.organization(), TGAheader.width, 1);
51  gfxstyle style (0,0,REPLACE,0,0);
52  style.clip (rect (0,0, d.xmax(),d.ymax()));
53
54  for (int loop= 0;  loop < TGAheader.height;  loop++) {
55     f.read (row.rawdata(), 2*TGAheader.width);
56     d.put (style, row, 0,loop);
57     }
58  key::get();
59  }
```

A structure is defined for the TGA file header. The `ezfile` class loads a header and optionally checks for a file signature while it opens a file, so upon opening I can immediately refer to the values in the header structure. Three more tests are made to assure that the file is the kind I'm looking for. Other fields specify the size of the image, which is used in the actual display loop.

Once everything is known to be okay, the program enters graphics mode. It uses the general `activate()` command, so you can set the resolution and pick a driver in an environment variable. This simplifies the program, since nothing about selecting modes needs to be programmed. By default, it will load the hi-color driver and use the default mode.

The following line defines a `pixarray` that is the width of the image and one row tall. Next is a definition of a style. The only part of the style that is needed is the `REPLACE` mode for putting and then clipping. Clipping is set to the screen boundaries in case the file is actually larger than the current video mode.

The actual display logic is a `for` loop with two lines in the body. The `read()` call loads a row of data into the existing `pixarray`'s bitmap area. The file stores pixels exactly as the hi-color device likes it, so no processing is needed. The second line puts the row onto the screen.

The result is quite fast. It is faster than general-purpose file viewers by a noticeable margin, and it is quite useful. The program is easily extended to add more features once this display file core is in place.

PCX FILES

The Targa file viewer is a good first program because the format is so simple to load. A PCX file is a bit more work, but the program follows essentially the same outline: Set up the file and the device, and then, in a loop, read and put a row at a time. The big difference is that the reading of a row requires a fancy function rather than a direct file read. Listing 8.8 contains the source code for the PCX file viewer.

LISTING 8.8. A PCX FILE VIEWER.

```
// SHOWPCX.CPP
#include "usual.h"
#include "device.h"
#include "extras\drvinit.h"
#include "ezfile.h"
#include "internal\doscall.h"
#include "pixarray.h"
#include "getkey.h"
#include <stdlib.h>  //need exit()
#include <string.h>  //need memmove();

screen_device d;

const bufsize= 2048;

void error (char* message, int error_code= 3)
{
d.finish();
error_out (message);
exit (error_code);
}

/* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */

class PCX_reader {
   pixarray input_line;
   ezfile f;
   void read_pal();
```

```
29     int Height;
30     pixarray pix;
31     byte* buffer;
32     byte* bufcur;  //current position in buffer
33     int buflen;   //how much left in buffer
34     int planes;
35     void refill_buffer();
36     void check_buffer (int val)
37      { if (buflen < val) refill_buffer(); }
38
39  public:
40     PCX_reader (unsigned organization, char* filename);
41     ~PCX_reader() { delete[bufsize] buffer; }
42     void read_line();
43
44     // the decoded and translated line
45     int height() { return Height; }
46     pixarray& read_row();
47
48     // palette info
49     unsigned long pens[256];
50     int pencount;
51     };
52
53  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
54
55  void PCX_reader::read_pal()
56  {
57  // more than one plane means no palette
58  if (planes > 1) {
59     pencount= 0;
60     return;
61     }
62
63  // depending on version and bits fields, either
64  // EGA or VGA palettes are used.
65  // this code only reads VGA palette at end of file
66
67  byte buf[1+256*3];
68
69  mylib_seek (f.handle, -(1+3*256), 2);
70  f.read (buf, sizeof buf);
71  if (buf[0] != 12) error ("palette signature error");
72
73  byte* p= buf+1;
74  for (int loop= 0;  loop < 256;     loop++) {
75     pens[loop]=
76     ((unsigned long)(p[0])<<16)¦(unsigned(p[1])<<8)¦p[2];
77     p += 3;
```

continues

LISTING 8.8. CONTINUED

```
78     }
79  pencount= 256;
80  }
81
82  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
83
84  void PCX_reader::refill_buffer()
85
86  {
87  memmove (buffer, bufcur, buflen);
88  bufcur= buffer;
89  f.read (buffer+buflen, bufsize-buflen);
90  buflen= bufsize;
91  }
92
93  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
94
95  PCX_reader::PCX_reader (unsigned organization, char* filename)
96  {
97  struct {
98     byte sig;  //always 0x0a, checked by file open
99     byte version;
100    byte encoding;  //always 1
101    byte bits;       //bits per pixel per plane
102    rect imagebounds;
103    pair resolution;
104    byte header_palette[48];
105    byte reserved;  //always zero
106    byte planes;
107    int rowwid;   //bytes per line
108    int header_palette_interp;
109    pair vidsize;
110    char padding[54];
111    int width() { return imagebounds.width(); }
112    int height() { return imagebounds.height(); }
113    } PCXheader;
114
115 if (! f.open (filename, &PCXheader, 128, "\x0a",1))
116    error ("can't open file");
117
118 pix.reformat (organization, PCXheader.width(), 1);
119 unsigned org;
120 planes= PCXheader.planes;
121 if (planes == 1) {  //chunky mode
122    org= PCXheader.bits ¦ 0x100;
123    }
124 else {  // planar mode
```

```
125     org= PCXheader.planes;
126     }
127  input_line.reformat (org, PCXheader.width(), 1);
128  Height= PCXheader.height();
129  read_pal();
130  mylib_seek (f.handle, 128, 0);
131  bufcur= buffer= new byte [bufsize];
132  f.read (buffer, bufsize);
133  buflen= bufsize;  //full
134  }
135
136  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
137
138  pixarray& PCX_reader::read_row()
139  {
140  byte far* dest= input_line.rawdata();
141
142  // here is the actual PCX decoder
143  int bytes_to_go= planes * input_line.prim()->rowwid;
144  check_buffer(2*bytes_to_go);
145  while (bytes_to_go > 0) {
146     byte b= *bufcur++;
147     buflen—;
148     if ((b & 0xc0) == 0xc0) {
149        int count= b & 0x3f;
150        byte value= *bufcur++;
151        buflen—;
152        bytes_to_go -= count;
153        while (count—) *dest++ = value;
154        }
155     else {
156        *dest++ = b;
157        bytes_to_go —;
158        }
159     }
160
161  if (input_line.organization() ==
162      pix.organization()) return
163  input_line;
164  input_line.map (pix, 0);
165  return pix;
166  }
167
168  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
169  /* /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\ */
170
171  int main (int argc, char* argv[])
172  {
173  activate (d);  //assume resolution was chosen
```

continues

LISTING 8.8. CONTINUED

```
174                //externally to the program!
175
176  PCX_reader file (d.organization(), argv[1]);
177
178  if (file.pencount)
179     d.set_pen_colors (0, file.pencount, file.pens);
180  gfxstyle style (0,0,REPLACE,0,0);
181  style.clip (rect (0,0, d.xmax(),d.ymax()));
182  for (int loop= 0;  loop < file.height();  loop++) {
183     pixarray& row= file.read_row();
184     d.put (style, row, 0,loop);
185     }
186  key::get();
187  }
```

PCX files also come in different flavors. This program is designed to read 16-color and 256-color files. There are also 24-bit and 16-bit PCX file formats that are not addressed here. There is a monochrome format, and this program is robust enough to deal with it even though that was not part of the original design. The program will display an image on any palette-type device with enough colors. In particular, it will display a 16-color file on a 256-color VGA or Super VGA screen, as well as on its native 16-color planar display.

In either file format, the run-length encoding is the same. But the 16-color format reads 4 lines, one for each plane. Both formats are modeled after the display's memory layout, so `pixarray`s also easily accommodate both formats. After a row is read into a one-line `pixarray`, it is converted to the format used for the actual display. The library does all the work with a single call, and it can handle any display format this way. That takes most of the work out of format conversion.

Listing 8.8 describes a class for reading PCX files. The constructor opens the file and loads the header and reads the palette. This program is different from the first in that it goes into graphics mode first, so the PCX reader class can know the organization of the target device (how many bits per pixel, among other things) so an extra line was added to `error()`, to switch back to text mode before displaying the error message.

The direct file read in the first program is replaced by a call to the `read_row()` member of the PCX file class. This function is what decodes one row of data and (if necessary) converts it to the proper format.

There are a couple of interesting points in the program. In the constructor, notice that `PCXheader` is a unnamed structure. Yet it has member functions. This is defined locally inside a function, so any members have to be implicit-inline. This shows a case where a simple member function did make the code more readable.

The `read_row()` member decodes the row to a `pixarray` organized in the way that matches the image file. Yet it has to return a `pixarray` that matches the target display. In particular, it may convert a 16-color planar format to a one-byte-per-pixel format. This is done with the `pixarray::map()` function. The first argument is the destination `pixarray`, and the second is a table of color mappings. Since the table is 0, it just converts the format and does not remap the colors. This function could be used to map the image into an existing palette in a more complex program, instead of having to use the palette that is specified in the file.

CHAPTER 9

GRAPHICS PROGRAMMING IN BORLAND C++

Borland C++ provides extensive two-dimensional computer graphics support in an easy to use graphics library. The library contains routines for basic drawing (lines, rectangles, circles, ellipses, polygons, and so on), multiple text fonts and text output routines, plus special purpose routines for creating bar, pie, and three-dimensional bar charts. Borland C++ supports CGA, EGA, VGA, IBM 8514, and other display devices.

Selecting colors

Fixing aspect ratio problems

Charting

Drawing a bar chart

Graphics drivers and font files

The graphics routines described here apply only to DOS-based Borland C++ applications. Turbo C++ for Windows does not have a graphics library. Instead, all Windows applications rely on use of the Microsoft Windows Application Programming Interface to call routines inside Windows. All graphics drawing is handled by the Graphics Device Interface, a component of Microsoft Windows.

INTRODUCTION TO BORLAND C++ GRAPHICS

Listing 9.1 illustrates basic Borland C++ graphics. The graph1.cpp program initializes the graphics system and draws some objects, a pie chart, and some text to the graphics screen. The resulting output is shown in Figure 9.1. Depending on your graphics adaptor, this picture may range from drab white on black up to a beautiful multicolored pie chart. On CGA screens, the output is white on black (in other words, boring). On VGA screens, the pie chart displays in a variety of colors.

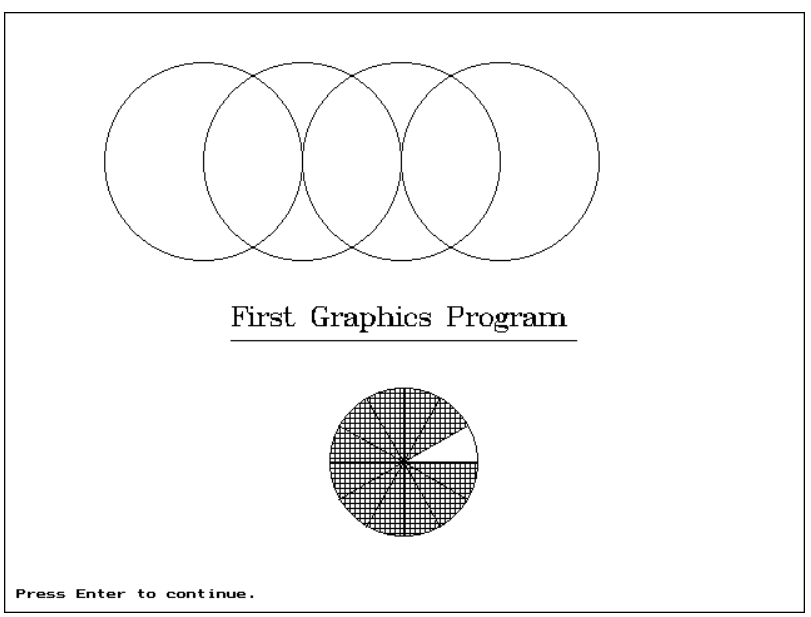


Figure 9.1. Example output produced by the graph1.cpp program.

LISTING 9.1. A SAMPLE BORLAND GRAPHICS PROGRAM.

```
 1  //
 2  //  GRAPH1.CPP -- Introduces Turbo C++ Graphics.
 3  //
 4  //  Displays three overlapping circles, some text, and a
 5  //  simple pie chart. Followed by a prompt "Press Enter to
 6  //  continue."
 7  //
 8  //  In the examples that follow, all drawings are relative
 9  //  to the current maximum X and maximum Y coordinates.
10  //  These values vary depending upon the graphics modes
11  //  supported by the computer (e.g. CGA versus VGA). By
12  //  making all drawing commands relative to the screen size,
13  //  this program will operate correctly on any monitor.
14  //
15
16  #include <graphics.h>
17  #include <iostream.h>
18  #include <stdlib.h>
19  #include <conio.h>
20
21  //
22  //      Function prototypes
23  //
24  void InitGraphics(const char *pBGIPath);
25  void DrawSomeCircles(int n);
26  void DrawSomeText(const char *pText);
27  void DrawPieChart(int nSlices);
28  void DrawFrame(int nThickness);
29  void PromptAndWait(const char *pText);
30
31  //
32  //      Initialize the graphics system via auto detection.
33  //  Supply the path to the BGI files on your system.
34  //
35  void InitGraphics(const char *pBGIPath)
36  {
37    //      Request auto detection of graphics driver.
38    int graphDriver = DETECT;
39    int graphMode;
40
41
42    //  Initialize graphics system.
43    //  Look for files in specified directory.
44    initgraph(&graphDriver, &graphMode, pBGIPath);
45
46    int graphError = graphresult();
47    if(graphError != grOk) {
```

continues

LISTING 9.1. CONTINUED

```
48            cout << "Graphics error: "
49                     << grapherrormsg(graphError) << '\n';
50            exit(1);
51    }
52  }
53
54  //
55  // Draw some circles
56  //
57  void DrawSomeCircles(int n)
58  {
59    const int xmax = getmaxx();
60    const int ymax = getmaxy();
61    const int radius = ymax / 6;
62
63    for(int i=0; i<n; i++)
64            circle(xmax/n + i*radius, ymax/n, radius);
65  }
66
67  //
68  //      Draw some text and underline it.
69  //
70  void DrawSomeText(const char *pText)
71  {
72    const int xmax = getmaxx();
73    const int ymax = getmaxy();
74
75    // Output some text, centered vertically and horzontally
76    settextjustify(CENTER_TEXT, CENTER_TEXT);
77    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 3);
78    outtextxy(xmax/2, ymax/2, pText);
79
80    // Save the current drawing color and set a new one
81    int saveColor = getcolor();
82    setcolor(CYAN);
83
84    // Underline the text we drew on the screen
85    const int textWidth = textwidth(pText);
86    const int textHeight = textheight(pText);
87    line(xmax/2 - textWidth/2, ymax/2 + textHeight,
88             xmax/2 + textWidth/2, ymax/2 + textHeight);
89
90    // Restore the drawing color
91    setcolor(saveColor);
92  }
93
94  //
```

```
95  //      Draw a pie chart with the given number of slices.
96  //
97  void DrawPieChart(int nSlices)
98  {
99    const int xmax = getmaxx();
100   const int ymax = getmaxy();
101
102   const int dAngle = 360/nSlices; // slice size (degrees)
103   const int radius = ymax/8;
104
105   const int maxColor = getmaxcolor(); // max colors
106   int color    = 0;                      // current color
107
108   for(int angle=0; angle<360; angle += dAngle) {
109
110      // Select a cross-hatched appearance and a color
111      setfillstyle(HATCH_FILL, color);
112
113      // Draw the slice
114      pieslice(xmax/2, ymax*3/4,             // center
115               angle, angle+dAngle,  // start, end angles
116               radius);
117
118      // Set the color for the next slice
119      color = (color + 1) % maxColor;
120   }
121 }
122
123 //
124 //      Draw a frame around the entire screen.
125 //  nThickness is the thickness in pixels.
126 //
127 void DrawFrame(int nThickness)
128 {
129   const int xmax = getmaxx();
130   const int ymax = getmaxy();
131
132   for(int i=0; i<nThickness; i++)
133           rectangle(i, i, xmax-i, ymax-i);
134 }
135
136 //
137 //      Display a prompt and wait for a keypress.
138 //
139 void PromptAndWait(const char *pText)
140 {
141         settextjustify(LEFT_TEXT, BOTTOM_TEXT);
142         settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
143         outtextxy(10, getmaxy()-10, pText);
```

continues

LISTING 9.1. CONTINUED

```
144          getch();
145  }
146
147  void main()
148  {
149          InitGraphics("C:\\BC3\\BGI");
150
151          DrawSomeCircles(4);
152          DrawSomeText("Sample Text!");
153          DrawPieChart(12);
154          DrawFrame(5);
155           PromptAndWait("Press any key to continue.");
156
157          closegraph();
158  }
```

The graphics library is located in graphics.lib. Programs that use the graphics library must `#include` graph.h. You must also add graphics.lib to your project or make file so the graphics library will be linked into your executable program. Each of the programs in this chapter includes a call to a function named `initgraphics()`. Be sure to edit the statement containing `initgraphics()` to ensure that the parameter specifies the directory where the Borland Graphics Interface files are stored on your computer.

Each graphics program must select a *graphics driver* to act as the interface between the graphics program and the computer hardware (see lines 35–52). The actual graphics driver files (and graphic character font files) are located in the \borlandc\bgi subdirectory. If you use the sample code presented in the graphics demonstration programs in this chapter, you must edit the source and change the subdirectory name to the name of the directory where the files are stored on your system. It is also possible to bind the necessary driver files directly into a completed .exe file so that your users don't clutter their disks with files having peculiar names. A technique for doing this is described in the "Linking Driver and Font Files" section.

Line 44 calls `initgraph()`, which, when its first parameter is set to the value of the `DETECT` constant, causes `initgraph()` to automatically detect the presence of graphic hardware and to determine the appropriate graphics driver for use with the program.

When using the autodetect feature, the `graphmode` parameter returns an integer value indicating which graphic mode of the driver has been chosen. For example, typical VGA or EGA graphics hardware can support multiple color and resolution settings. With an EGA monitor, you can select 640×350 by 16 colors, or 640×350 by 4 colors, or even a low resolution 640×200 by 16 colors. `graphmode` indicates which of these resolutions is the one currently in use by the graphics driver.

By setting the `graphdriver` parameter to a value other than `DETECT`, such as `CGA` or `EGA`, you can explicitly select a particular graphics driver and choose the operating mode manually. The constant values used to manually or automatically select graphics drivers and graphics modes are described in Borland's reference documention. Refer to the description of `initgraph()` in the Borland Library Reference.

Graphics routines return an error through the `graphresult()` function that resets the current error condition code to `grOk` (or zero) when called. After performing a graphics function, copy the `graphresult()` value to a variable and compare it to the `grOk` constant. There are over a dozen possible return result codes. Refer to the description of `graphresult()` in the Borland Libary Reference for a complete list of possible error codes. When an error has occurred, you can either check the value returned by `graphresult()`, or pass it directly to the `grapherrormsg()` function, which translates the error code into a descriptive text message about the problem.

THE GRAPHICS COORDINATE SYSTEM

Before you can understand how objects are drawn on the screen, you need to briefly explore the coordinate system that describes how objects are placed. For all drawings, the upper left corner of the screen is coordinate `(0, 0)`, and the lower right is located at coordinate `(getmaxx(), getmaxy())`. The coordinates and their relationship to the screen are shown in Figure 9.2

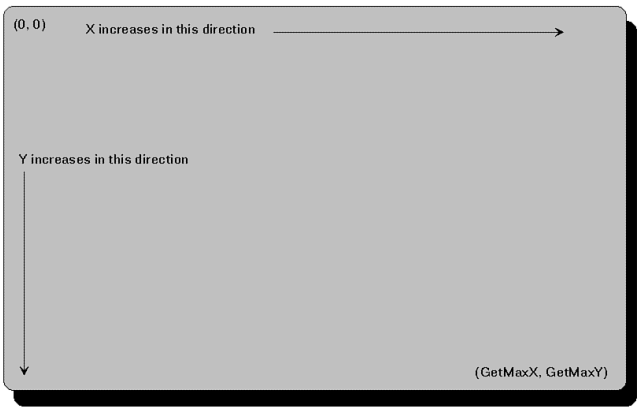


Figure 9.2. The Borland graphics coordinate system places `(0,0)` *at the upper left corner of the screen.*

`getmaxx()` and `getmaxy()` are functions returning the maximum X and maximum Y values, respectively, of the graphics coordinates on the screen. The actual values vary depending on the screen resolution (CGA, EGA, or VGA). It's important that your programs refer to `getmaxx()` and `getmaxy()`, rather than hard coding actual coordinate values. When I wrote the sample program in Listing 9.1, I couldn't possibly know the resolution of your computer's graphics screen. So, instead of writing a program that works only on my Super VGA monitor, I made all X and Y coordinate values relative to `getmaxx()` and `getmaxy()`. This way, the sample drawing is scaled to fit your computer screen (although, due to different aspect ratios, it will be less than optimal on some displays, especially CGA).

DRAWING CIRCLES

Lines 57–65 of Listing 9.1 call the graphics.lib function `circle()` to display a set of four overlapping circles, centered across the top quarter of the screen. The circle's size, or radius, is determined by dividing the maximum Y value by 6. If the maximum `Y` resolution is 480 pixels, then then resulting radius is 480 divided by 6, or 80 pixels. The first circle is drawn in the leftmost quarter of the screen (`getmaxx() / 4`), and successive circles are drawn one radius to the right.

DISPLAYING TEXT

When you need to display text on the graphics screen, don't use `printf()` or `puts()`. Instead, use combinations of the graphical text output routines such as `outtext()`, `outtextxy()`, and `settextjustify()`. `settextjustify()` tells the graphics system how subseqent calls to `outtext()` or `outtextxy()` should position the text around the X and Y coordinates. Options are available to format the text so that `(X,Y)` can mark any corner of the string's graphic representation—upper left, lower left, upper right, or lower left—as well as center the string about the specified coordinates. See the section called "SelectingFonts and Text Justification" later in this chapter.

`settextstyle()` chooses a character font, `DEFAULT_FONT`, requests that it be displayed horizontally (if you wish, you can write text vertically up the screen), and selects a character size of 3. The character size works like a multiplier: A size value of *2* requests a character size *twice* as big as the normal or standard character for that font. `DEFAULT_FONT` selects a standard bit-mapped font, with the base character described in an 8×8 bitmap. For fun, try changing `DEFAULT_FONT` in the sample program to one of the following constants: `TRIPLEX_FONT`, `SMALL_FONT`, `SANS_SERIF_FONT`, `GOTHIC_FONT`, `SCRIPT_FONT`, `SIMPLEX_FONT`, `TRIPLEX_SCR_FONT`, `COMPLEX_FONT`, `EUROPEAN_FONT`, or `BOLD_FONT`. These fonts are called stroked fonts and are smoothly scalable in size, unlike the `DEFAULT_FONT`, which looks rather clunky when enlarged more than two or three times.

Finally, the string is written to the display with a call to `outtextxy()`, taking an X,Y coordinate and the contents of the string to write. This last parameter may be either a string constant or a variable, as shown in the example.

In line 82 of listing 9.1, `setcolor()` changes the default color selection for subsequent screen output. Here, the default color is temporarily changed to cyan from the default color of white. The `line()` function (lines 87–88) draws a line between two coordinates, hence the need for two sets of X and Y values. The line's position is calculated to neatly underline the `"First Graphics Program"` text that was output with `outtextxy()`. The functions `textwidth()` and `textheight()` calculate the size of the string in pixels (which can vary depending on font, resolution, and the font magnification parameter of `SetTextStyle`). Finally, at line 91, the color selection is restored to its original value.

Drawing a basic pie chart is easy in Borland C++. The function `pieslice()` draws individual pie slices on-screen. You use `pieslice()` by specifying a starting angle (in degrees, with 0 being on a horizontal line headed to the right), and an ending angle, which describes an arc in the counterclockwise direction.

You can make many graphic objects contain an interior pattern by calling `setfillstyle()`. In this example, `HATCH_FILL` fills each each pie slice with a cross-hatched pattern. `setfillstyle()` also selects the color for the interior region. In this section of code the `color` variable is incremented from 0 to 15, and then reset back to 0, so that each pie slice is drawn in a separate color. Depending on your monitor and the graphics mode in effect, when you run graph1.cpp you may not see 16 colors, but instead may see as few as two colors or sections that are blinking on the display.

Lines 114–116 call `pieslice()`, passing to it the X and Y coordinates specifying the middle of the pie chart, the start and ending angles, and the radius of the circle (or size of the pie).

Lastly, a rectangle is drawn around the entire screen in function `DrawFrame()` (lines 127–134). Lines 141 to 144 display a prompt message `Press Enter to continue`, followed by a `call to getch()` to wait for input. `closegraph()` in line 157 shuts down the graphics system and returns the screen to text mode operation.

The Graph1 example program illustrates a number of Borland C++ graphics features within a simple program. The following sections detail additional concepts and elaborate on various procedures and functions found in the Borland C++ graphics interface.

SELECTING FONTS AND CHARACTER SIZES

Before performing text output with `outtext()` or `outtextxy()`, you should call `settextstyle()` to select a character font, display direction (either horizontal or vertical), and relative size of the graphical text. If you do not call `settextstyle()`, all output appears in the default 8×8 bitmap font and in the smallest size available (8 pixels by 8 pixels).

`settextstyle()` is defined as:

```
void far settextstyle(int font, int direction, int charsize);
```

The first parameter selects the font, the second the direction, and the third sets the character size. Here's an example that selects the TRIPLEX_FONT and prints it in the horizontal direction with a scaling factor of 3:

```
settextstyle() ( TRIPLEX_FONT, HORIZ_DIR, 3 );
settextjustify ( LEFT_TEXT, BOTTOM_TEXT );
outtextxy()( getmaxx() / 2, getmaxy() / 2, "Hello, World!");
```

The fonts are stored in a series of .chr files, in the same directory as the .bgi driver files. If `settextstyle()` cannot locate the appropriate .chr file for the font selected, `graphresult()` returns an error code. To ensure program reliability, you should test the condition of `graphresult()` whenever you select a new font.

You select one of the standard fonts by passing one of these constants to `settextstyle()`:

```
DEFAULT_FONT
TRIPLEX_FONT
SMALL_FONT
SANS_SERIF_FONT
GOTHIC_FONT
SCRIPT_FONT
SIMPLEX_FONT
TRIPLEX_SCR_FONT
COMPLEX_FONT
EUROPEAN_FONT
BOLD_FONT
```

The DEFAULT_FONT is a simple 8×8 bitmap font. As a consequence, the font takes on a chunky appearance as it is scaled upwards in size. Therefore, the default font is convenient for small text items such as prompts and messages. The remaining fonts are *stroked fonts*, meaning that internally they are stored as graphical vectors rather than as bitmaps. Stroked fonts can be significantly enlarged in size with no degradation in appearance.

The *direction* parameter determines whether output is drawn on-screen horizonally, like normal text, or vertically. By setting this parameter to the constant HORIZ_DIR, you select the horizontal text. Use VERT_DIR to select the vertical text.

charsize sets a scaling factor to be applied to the font. For instance, a *charsize* of 1 for the default font produces the smallest 8×8 pixel characters. Setting the *charsize* to 3 displays the font in a 24×24 pixel array (but still with only 8×8 level resolution).

When using graphical fonts, you cannot rely on the usual `strlen()` function to determine the size of a string when it appears on-screen. Instead, call the `textheight()` and `textwidth()` functions to calculate the actual pixel height and width, respectively.

In addition to the `charsize` scaling factor set with `settextstyle()`, you can vary the character width and height of the stroked fonts in fine increments by calling `setusercharsize`. `setusercharsize` is defined as:

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

The *`multx`* and *`divx`* parameters set a scaling ratio for character width, and *`multy`* and *`divy`* set a scaling ratio for character height. For example, in the `TRIPLEX_FONT` used in the sample program (Listing 9.1) at the beginning of this chapter, you can make the characters 1.5 times wider than the normal font by writing the following:

```
setusercharsize ( 3, 2, 1, 1 );
```

The 3:2 ratio is applied to the character width, so that each character becomes 3/2 or 1.5 times wider. By varying both values, you can produce remarkably fine degrees of adjustment in the shape of the basic character set.

To make the characters small and skinny, write this:

```
setusercharsize ( 1, 4, 1, 1);
```

This produces a scaling multiplier of 1/4. While not shown in these examples, scaling values also apply to the Y axis. When both values, *`multx`* and *`divx`* or *`multy`* and *`divy`*, are set to 1, then no scaling adjustment is made to the respective axis.

Here's an example that uses `setusercharsize` to adjust the character scaling factors:

```
strcpy (thetext, "First Graphics Program" );
settextjustify (CENTER_TEXT, CENTER_TEXT);
settextstyle ( TRIPLEX_FONT, HORIZ_DIR, 3);
setusercharsize ( 3, 2, 1, 1 );
outtextxy(getmaxx()/2, getmaxy()/,tThTtxt);
```

SETTING TEXT JUSTIFICATION

When text is drawn on-screen using `outtext()` or `outtextxy()`, the graphical representation of the text is drawn relative to the X,Y coordinate. Think of the text as being like a rectangular region that just happens to be filled with text.

The region contains four corners. When you output text to the screen at position (X,Y), any one of the region's four corners can be placed at (X,Y) by using the `settextjustify()` procedure. `settextjustify()` is used to orient the text drawing about the X, Y coordinate.

For example, if you output a text string to `(0,0)`, you want `(0, 0)` to mark the upper left corner of the text region. If `(0,0)` marks the lower left corner of the text region, then all of the output would be drawn off the top of the screen. On the other hand, if you display the string at the bottom of the screen, `(0,getmaxy())`, you want `(0,getmaxy())` to mark the lower left corner of the string; if the coordinate marked the upper left corner, the entire string would fall off the bottom the screen.

`settextjustify()` is declared as follows:

```
void far settextjustify(int horiz, int vert);
```

You select the position of the output text around the X,Y coordinate by passing predefined constant values to the *`horiz`* and *`vert`* parameters. These constants are: `LEFT_TEXT`, `CENTER_TEXT`, and `RIGHT_TEXT` for the *`horiz`* parameter; `BOTTOM_TEXT`, `CENTER_TEXT`, and `TOP_TEXT` for the *`vert`* parameter.

You should use the `LEFT_TEXT`, `CENTER_TEXT`, and `RIGHT_TEXT` constants for the *`horiz`* parameter, and the `BOTTOM_TEXT`, `CENTER_TEXT`, and `TOP_TEXT` parameters for the *`vert`* parameter. Each constant positions one axis of the text box, as shown in Figures 9.3 and 9.4.

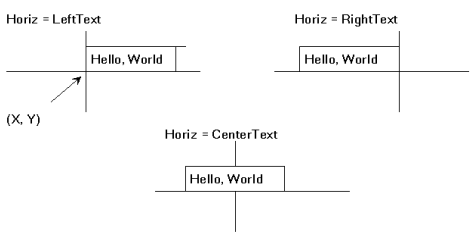


Figure 9.3. The effects of using `LEFT_TEXT`, `RIGHT_TEXT`, *and* `CENTER_TEXT` *on the horizontal positioning of graphical text output. In each drawing, the* `vert` *parameter of* `settextjustify()` *is set to* `BOTTOM_TEXT`.

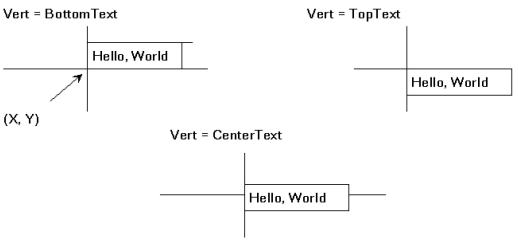


Figure 9.4. The effects of setting the `vert` *parameter of* `settextjustify()` *to* `BOTTOM_TEXT`, `TOP_TEXT`, *and* `CENTER_TEXT`. *In each example,* `horiz` *is set to* `LEFT_TEXT`.

VIEWPORTS

A *viewport* describes a window or region on-screen where all graphics drawing will take place. Initially, the viewport is set to encompass the entire screen. After a call to `setviewport()`, all subsequent drawing commands are mapped to screen positions relative to the location of the viewport region. Figure 9.5 shows a screen image containing a viewport region. By setting a `setviewport()` option, you can restrict your drawings to appear only within the viewport. Any portion of an object that falls outside the viewport is *clipped* at the edge of the viewport and does not draw outside the region.

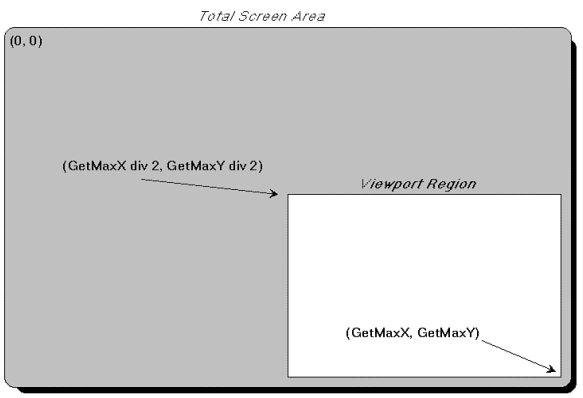


Figure 9.5. The outer box represents the entire physical screen. The smaller box at the lower right is a viewport region defined in the lower right quarter of the screen.

`setviewport()`'s parameters define an upper left and a lower right corner of a boundary box, and optionally activate viewport clipping. `setviewport()` is defined as:

```
void far setviewport(int left, int top, int right, int bottom, int clip);
```

For example, to set up a *viewing portal* with an upper left corner located at `(100,70)` and extending down to the lower right corner of the screen, write the following:

```
setviewport (100, 70, getmaxx(), getmaxy(), CLIP_ON);
```

If you now execute the drawing command

```
line (0, 0, 50, 50);
```

the line's location is mapped to the viewport region, so that `(0,0)` is at physical screen location `(100,70)` and `(50,50)` is at physical coordinate `(150,120)`. Effectively, this is equivalent to writing the following:

```
Line (100, 70, 150, 120 );
```

The result of this is shown in Figure 9.6.

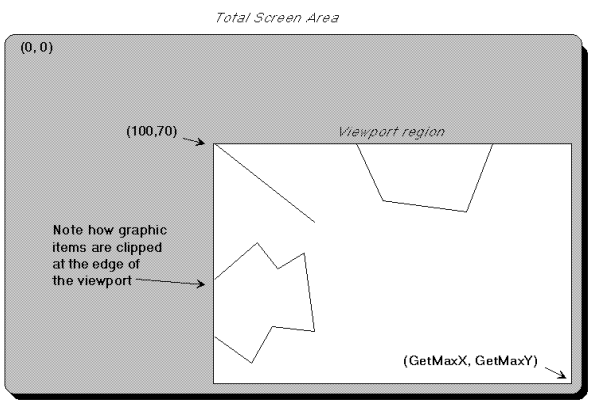


Figure 9.6. How a line and other drawing commands are mapped to a viewport region.

The last `setviewport()` parameter is set to `CLIP_ON`, a constant that means graphic elements falling outside the viewport should be cut or clipped out of the drawing. If set to `CLIP_OFF`, then graphic elements are allowed to extend beyond the borders of the viewport.

You can experiment with `setviewport()` by placing calls to `setviewport()` inside the Graph1 program. As a first step, you might try placing this statement just before the `Circle` drawing section:

```
setviewport (20, 20, getmaxx(), getmaxy()/3 * 2, CLIP_ON);
```

Note that this slides the drawing down and over by 20 pixels, restricting the Y coordinate to the top two-thirds of the screen. Give it a try and see what happens.

Viewports are often used to restrict new graphic drawings from overrunning other graphics on-screen. If your programs draw a rectangular border around the screen by calling `rectangle (0, 0, getmaxx(), getmaxy())`, you can protect the rectangle from being overwritten by calling

```
setviewport ( 1, 1, getmaxx() - 1, getmaxy() - 1, CLIP_ON );
```

This moves the viewport to one pixel inside the bounding rectangle and ensures that any items you subsequently draw will be clipped at the edge of the viewport before they can overwrite the border.

THE CURRENT POINTER

When you type text in the IDE editor, your current location in the edit window is indicated with a flashing cursor. The graphics system maintains a similar entity called the *current pointer*, although it is never visible on-screen. The current pointer is used for *relative* drawing commands and tracks the location where the next drawing command will take place.

To position the current pointer (CP) to a location on-screen, use the `moveto()` function. To position the CP to the center of the screen, write

```
moveto (getmaxx()/2, getmaxy()/2);
```

Next, you can draw an object relative to this starting point, such as

```
lineto(0, 0);
```

which results in a line drawn from the screen mid-point to the upper left corner.

The following drawing commands use and set the position of the CP (all other commands have no effect on the CP):

```
linerel()
lineto()
moverel()
moveto()
outtext()
```

These next procedures reset the CP's position to `(0,0)`:

```
cleardevice()
clearviewport()
graphdefaults()
initgraph()
setgraphmode()
setviewport()
```

You can find the current location of the current pointer using the `getx()` and `gety()` functions, which return the CP's current X and Y coordinates, respectively.

SELECTING COLORS

Borland C++ graphics support drawings that may have 2, 4, 16, or 256 color choices, depending on the graphics driver and graphics mode you select. Colors on the PC are handled in a manner similar to the way an artist mixes colors. The artist squeezes various colors onto a palette board, and then mixes a selection of color choices to be used in a drawing. Thereafter, the artist selects colors by choosing one of the paints from the palette.

On the PC, you draw a graphic object on-screen and select its color from a *color palette*. Rather than directly selecting, for example, red or purple, you select the palette entry that contains red or purple. Table 9.1 shows the standard color constants. Each of these constants selects a color from the corresponding standard 16 color palette.

TABLE 9.1. COLOR CONSTANTS USED IN CALLS TO `setpalette()` AND `setallpalette()`.

CGA Modes Color Constant	*EGA/VGA Modes Constant*
BLACK	EGA_BLACK
BLUE	EGA_BLUE
GREEN	EGA_GREEN
CYAN	EGA_CYAN
RED	EGA_RED
MAGENTA	EGA_MAGENTA
BROWN	EGA_LIGHTGRAY
LIGHTGRAY	EGA_BROWN
DARKGRAY	EGA_DARKGRAY
LIGHTBLUE	EGA_LIGHTBLUE
LIGHTGREEN	EGA_LIGHTGREEN
LIGHTCYAN	EGA_LIGHTCYAN
LIGHTRED	EGA_LIGHTRED
LIGHTMAGENTA	EGA_LIGHTMAGENTA
YELLOW	EGA_YELLOW
WHITE	EGA_WHITE

Because the Borland C++ graphics system is optimized for 16 color EGA displays, it does not provide simultaneous 256 color support on Super VGA monitors. Through a programming trick, however, you can redefine each of the 16 entries in the basic color palette to be any color you want. That's because the VGA has the capability to display 16 or 256 different colors, where each is selected from a palette of 262,144 separate colors. See the section later in this chapter called "Using `setrgbpalette()`" for details.

CHOOSING COLORS FROM THE COLOR PALETTE

To make the next object that you draw appear in the color red, call `setcolor` *before* drawing the object:

```
setcolor ( 4 );
line ( 10, 10, 70, 125 );
```

This selects the fourth entry in the color palette (for 16 color palettes, the palette is indexed with values from 0 to 15). The result is a red line from `(10,10)` to `(70,125)`. `setcolor()` changes the current or active drawing color and affects all subsequent drawing commands until you call `setcolor()` again.

If you are using an EGA or VGA monitor (but not the CGA), an interesting side effect of color palettes is that all of the screen's current colors can be rearranged by changing just the palette. There's no need to redraw any of the objects. Just change the underlying palette. Two procedures let you alter the palette entries: `setpalette()` to change an individual entry and `setallpalette()` to change several or all the entries in a single procedure call. For example, to change the fifth entry to blue, type

```
setpalette ( 5, BLUE );
```

Any objects previously drawn in color number 5 are instantly changed to blue.

AVAILABLE COLORS

The actual set of colors available in the color palette depends on the type of monitor in use, the graphics adaptor, and the resolution of the screen. Borland C++ graphics look best on EGA, VGA, or better displays. In the CGA 320×200 resolution mode you have a choice of four palettes, each with three foreground colors and one background color. These palettes are determined by the *graphmode* parameter to `initgraph()`, and since they are hard-wired into the CGA, they cannot be altered with calls to `setpalette()`.

Only the IBM 8514 supports 256-color mode. To run 256 colors simultaneously on VGA compatible monitors, you must obtain a VGA 256-color driver, available from Borland and other sources. To change the 256 color mode palette you should call `setrgbpalette()` instead of `setpalette()`.

USING *SETRGBPALETTE()*

For the IBM 8514 and VGA device drivers, the 16 basic palette entries are programmable. Using `setrgbpalette()` you can precisely specify the amount of red, green, or blue for each index in the palette.

Each of the color values ranges from 0 to 63, with 0 being the lowest intensity and 63 being the brightest intensity. By mixing various intensities of red, green, and blue, you can create custom colors up to a maximum of 262,144 different combinations. `setrbgpalette` is defined as:

```
void far setrgbpalette(int colornum, int red, int green, int blue);
```

For example,

```
setrgbpalette ( 0, 35, 20, 60 );
```

changes the background color (palette entry 0) to whatever color is produced by mixing these combinations of red (35), green (20), and blue (60). For the IBM 8514 only, *colornum* may range from 0 to 255.

Here is a code fragment that you may use to cycle through all 262,144 color combinations:

```
setcolor(3);
outtextxy(100, 100, "Here is some text in color #3");

/*Change the value of palette entry #3 to every possible color
This will take a while!*/
for( RedValue = 0; RedValue <= 63; RedValue++)
  for( GreenValue = 0; GreenValue <= 63; GreenValue++ )
    for( BlueValue = 0; BlueValue <= 63; BlueValue++ )
    {
      setrgbpalette( 3, RedValue, GreenValue, BlueValue );
      delay(50);
    };
```

SELECTING INTERIOR COLORS AND PATTERNS FOR OBJECTS

`setcolor()` chooses the active color for line drawing and point plotting operations, including the borders of circles, pie charts, and so on. To color the interior of a bounded object (circles, pie slices, rectangles, polygons, and so on), use the `setfillstyle()` function (or optionally `setfillpattern()`). With `setfillstyle()` you can choose a variety of standard patterns to fill the interior of an object, including empty, solid, dots, cross-hatched, and others.

Use `setfillpattern()` in conjunction with `setfillstyle()` to establish a custom-design pattern for filling the interior of all filled graphic objects (`fillpoly()`, `floodfill()`, `bar()`, `bar3d()`, `pieslice()`), and to select the color for that pattern (the interior color is set here or in `setfillstyle()`; the boundary color is set by `setcolor()`). `setfillpattern()` is declared as follows:

```
void far setfillpattern(char far *upattern, int color);
```

The example program in Listing 9.2 provides a custom-design editor that makes it easy to design your own patterns (it's a lot of fun to invent new designs).

LISTING 9.2. A SAMPLE PROGRAM THAT DEMONSTRATES THE USE OF `setfillpattern()` BY IMPLEMENTING A CUSTOM PATTERN EDITOR.

```
 1  //  CUSTOMPA.CPP
 2  //  Use this program to help you design a custom fill
 3  //  pattern. After the 8 x 8 grid displays, you can use
 4  //  the arrow keys to navigate to a specific bit, and set
 5  //  it to a 1 by pressing the 1 key, or clearing the bit
 6  //  by pressing the 0 key. Press the Esc key to terminate
 7  //  data entry and a sample filled circle is displayed on
 8  //  the screen. Press Enter to return to pattern editing,
 9  //  or press Esc key again to terminate the program.
10  //
11
12  #include <graphics.h>
13  #include <fstream.h>
14  #include <stdlib.h>
15  #include <conio.h>
16  #include <ctype.h>
17
18  typedef int BOOL;                  // boolean
19
20  //
21  //      Function prototypes
22  //
23  void PlotBit(int x, int y, BOOL fOn);
24  void DisplayPattern(const char *pPat);
25  unsigned GetChar();
26  int ReadPattern(const char *pFilename, unsigned char *pPat);
27  int WritePattern(const char *pFilename, unsigned char *pPat);
28  void InitGraphics(const char *pBGIPath);
29  void SetBit(unsigned char *pPat, int x, int y, BOOL f);
30  BOOL GetBit(const unsigned char *pPat, int x, int y);
31
32  // Keyboard values for extended keystrokes and '0' and '1'
```

continues

LISTING 9.2. CONTINUED

```
33  const unsigned keyUpArrow    = 72 << 8;
34  const unsigned keyLeftArrow  = 75 << 8;
35  const unsigned keyRightArrow = 77 << 8;
36  const unsigned keyDownArrow  = 80 << 8;
37  const unsigned keyEscape     = 27;
38  const unsigned key0          = 48;
39  const unsigned key1          = 49;
40
41  //
42  //      Upper-left corner of the grid goes at (ulx, uly).
43  //  1's and 0's are horizontally spaced by xmul locations.
44  //
45  const int ulx  = 5;
46  const int uly  = 5;
47  const int xmul = 4;
48
49  //
50  //      Set a bit in the pattern.
51  //
52  void SetBit(unsigned char *pPat, int x, int y, BOOL f)
53  {
54    if(f)
55      pPat[y] ^= (0x80 >> x);         // set
56    else
57      pPat[y] &= ~(0x80 >> x);       // clear
58  }
59
60  //
61  //      Return a bit's value (0 or 1).
62  //
63  BOOL GetBit(const unsigned char *pPat, int x, int y)
64  {
65     return (pPat[y] & (0x80 >> x)) != 0;
66  }
67
68  //
69  //      Display a '1' if a bit is on, '0' if it's off at the
70  //  appropriate location.
71  //
72  void PlotBit(int x, int y, BOOL fOn)
73  {
74     gotoxy(x*xmul + ulx, y + uly);
75     cout << (fOn ? '1' : '0');
76  }
77
78  //
79  //      Display the entire pattern on the screen.
```

```
80  //
81  void DisplayPattern(const unsigned char *pPat)
82  {
83     for(int x=0; x<8; x++)
84       for(int y=0; y<8; y++)
85         PlotBit(x, y, GetBit(pPat, x, y));
86  }
87
88  //
89  //  Read a character from the keyboard, placing the
90  //  extended value in the high byte of the result.
91  //
92  unsigned GetChar()
93  {
94    unsigned u = getch();
95    if(u == 0)
96       u = getch() << 8;
97    return u;
98  }
99
100 //
101 //      Read a pattern from a file.
102 //  Return 1 if success, 0 if failure.
103 //
104 int ReadPattern(const char *pFilename, unsigned char *pPat)
105 {
106   ifstream is(pFilename);   // open the file
107   if(!is)
108           return 0;
109
110   for(int i=0; i<8; i++)    // read 8 bytes
111           is >> *(pPat++);
112
113   return is.good();                  // get result
114 }
115
116 //
117 //      Write a pattern to a file.
118 //  Return 1 if success, 0 if failure.
119 //
120 int WritePattern(const char *pFilename,
121                   unsigned char *pPat)
122 {
123   ofstream os(pFilename);   // open the file
124   if(!os)
125     return 0;
126
127   for(int i=0; i<8; i++)    // write 8 bytes
128      os << *(pPat++);
129
```

continues

LISTING 9.2. CONTINUED

```
130    return os.good();                    // get result
131  }
132
133  //
134  //      Initialize the graphics system via auto detection.
135  //  Supply the path to the BGI files on your system.
136  //
137  void InitGraphics(const char *pBGIPath)
138  {
139     //      Request auto detection of graphics driver.
140     int graphDriver = DETECT;
141     int graphMode;
142
143
144     //  Initialize graphics system.
145     //  Look for files in specified directory.
146     initgraph(&graphDriver, &graphMode, pBGIPath);
147
148     int graphError = graphresult();
149     if(graphError != grOk) {
150             cout << "Graphics error: "
151                      << grapherrormsg(graphError) << '\n';
152             exit(1);
153     }
154  }
155
156  void main()
157  {
158    unsigned char userPattern[8];
159    memset(userPattern, 0, 8);      // initally all 0's
160
161    //
162    //  See if we should use a saved pattern.
163    //
164    clrscr();
165    cout << "Read existing pattern from file? <y/n>  ";
166    unsigned response = GetChar();
167    if(toupper(response) == 'Y')
168      if(ReadPattern("PATTERN.PAT", userPattern) == 0) {
169         cout << "Problem reading file!\n";
170         exit(1);
171       }
172
173    //
174    //  Setup graphics mode.
175    //
176    InitGraphics("C:\\BC3\\BGI");
```

```
177
178   unsigned keyPress = 0;
179   do {
180     restorecrtmode();
181     DisplayPattern(userPattern);
182
183     gotoxy(1,17);
184     cout << "Use arrow keys to navigate.\n"
185          << "Press 1 to set a bit, 0 to clear a bit.\n"
186          << "Press ESC twice to exit the program.\n";
187
188     int x = 0;
189     int y = 0;
190
191   //
192   //  Edit the pattern until user hits ESC
193   //
194   do {
195      gotoxy(x*xmul + ulx, y + uly);
196      switch(keyPress = GetChar()) {
197      case key0:                          // clear the bit
198              cout << '0';
199              SetBit(userPattern, x, y, 0);
200              break;
201      case key1:                // set the bit
202              cout << '1';
203              SetBit(userPattern, x, y, 1);
204              break;
205      case keyUpArrow:
206              if(y > 0) y--;
207              break;
208      case keyDownArrow:
209              if(y < 7) y++;
210              break;
211      case keyLeftArrow:
212              if(x > 0) x--;
213              break;
214      case keyRightArrow:
215              if(x < 7) x++;
216              break;
217     }
218   } while(keyPress != keyEscape);
219
220   //
221   // After editing the pattern, return to graphics mode
222   // and display an object containing the new pattern.
223   //
224   setgraphmode(getgraphmode());
225
226    setfillpattern(userPattern, 3);
```

continues

LISTING 9.2. CONTINUED

```
227     fillellipse(getmaxx()/2, getmaxy()/2,
228                 getmaxx()/3, getmaxy()/3);
229
230    } while(GetChar() != keyEscape);
231
232    closegraph();
233
234  //
235  //  See if the user wants to save his pattern.
236  //
237  cout << "Save pattern to file? <y/n> ";
238    if(toupper(response = GetChar()) == 'Y')
239      if(WritePattern("PATTERN.PAT", userPattern) == 0) {
240         cout << "Problem writing file!\n";
241         exit(1);
242      }
243  }
```

Use `setfillstyle()` to select one of the standard interior patterns for filled objects. The patterns are selected using one of the constants from Table 9.2. You may also create custom fill patterns using the `setfillpattern()` function described above. When the selected pattern is set to `USER_FILL`, `setfillstyle()` selects the previously registered custom fill pattern. This way you can flip back and forth between a standard fill pattern and a custom fill pattern without rebuilding the custom pattern each time.

The parameters to `setfillstyle()` select the desired pattern (using a constant from Table 9.2) and the color for the pattern. `setfillstyle()` is declared as follows:

```
void far setfillpattern(char far *upattern, int color);
```

Here's an example code fragment that selects slanted lines to the right (called *light slash* pattern) and proceeds to fill a pie slice with the pattern:

```
setfillstyle ( LTSLASH_FILL, 3 );
pieslice ( getmaxx()/2, getmaxy()/2, 0, 90, 100 );
```

TABLE 9.2. CONSTANTS USED IN CALLS TO SETFILLSTYLE().

Constant	*Description*
EMPTY_FILL	Uses the background color for a solid fill.
SOLID_FILL	Uses the specified color as a solid fill.
LINE_FILL	Outputs horizontal lines across the object.
LTSLASH_FILL	Outputs slanted lines (to the right) across the object.
SLASH_FILL	Outputs thick slanted lines (to the right) across the object.
BKSLASH_FILL	Outputs thick slanted lines (to the left) across the object.
LTBKSLASH_FILL	Outputs slanted lines (to the left) across the object.
HATCH_FILL	Displays a cross-hatch pattern.
XHATCH_FILL	Displays a thicker cross-hatch pattern.
INTERLEAVE_FILL	Outputs a tightly spaced line fill.
WIDE_DOT_FILL	Outputs a dot fill pattern with the dots widely spaced.
CLOSE_DOT_FILL	Displays a closely spaced dot fill pattern.
USER_FILL	Selects the user-defined fill pattern.

If you have an arbitrary bounded region, you can fill it with a selected pattern. For instance, if you draw an arbitrary shape on-screen—the outline of an automobile, for example—using line drawing functions, you can fill in the drawing by calling the `floodfill()` procedure.

Use `floodfill()` to fill the interior of any object bounded by a border of a single color. `floodfill()` is defined as follows:

```
void far floodfill(int x, int y, int border);
```

`floodfill()` uses its color parameter, specified by *`border`*, to locate the boundary edges of the object. To fill the interior, you must insure that `(x,y)` describes a point inside the object to be filled, and that the object is bounded. If `(x,y)` lies outside the object, then everything outside the given color boundary is filled. If there is a "hole" in the boundary area, then the color "leaks out" and potentially covers the entire screen. If `(x,y)` is outside the boundaries of the object, `floodfill()` fills the screen area on the outside of the object.

If a problem occurs during `floodfill()` execution, `graphresult()` is set to `grNoFloodMem` (–7). Here's an example:

```
/* Set border color */
setcolor (4);

/* Draw rectangle in that color */
rectangle ( 100, 100, 200, 200 );

/* Prepare to hatch fill the rectangle */
setfillstyle( HATCH_FILL, 5 );

/* Fill the interior to matching color 4 */
floodfill ( 110, 110, 4);
```

FIXING ASPECT RATIO PROBLEMS

If circles appear more like ellipses on your monitor (a common problem), it means that your monitor is slightly out of alignment. Rather than fixing the hardware, its easier to modify the graphics drawing algorithms by calling `setaspectratio()`. `setaspectratio()` has two parameters: *`xasp`* and *`yasp`*. By varying the *`xasp`* and *`yasp`* parameters, the circle drawing algorithm is modified until a true circle appears. Listing 9.3 shows a routine you can use to help calibrate the software for any monitor.

LISTING 9.3. A PROGRAM YOU CAN USE TO HELP CALIBRATE THE CIRCLE DRAWING ALGORITHM FOR ANY MONITOR.

```
1  //
2  //  ASPECTR.CPP
3  //
4  //  Demonstrates the effect of varying the aspect ratio
5  //  on circle drawing. Increasing Xasp results in an
```

```
 6  //  increasingly elliptical circle, in the vertical
 7  //  direction. Decreasing Xasp or increasing Yasp produces
 8  //  an increasing ellipse in the horizontal direction.
 9  //
10
11  #include <graphics.h>
12  #include <strstrea.h>
13  #include <iomanip.h>
14  #include <stdlib.h>     // for exit()
15  #include <values.h>                // for MAXINT
16  #include <conio.h>      // for getch(), kbhit()
17
18  //
19  //  Inline functions are safer than macros.
20  //
21  inline int min(int a, int b) { return (a<b) ? a : b; }
22  inline int max(int a, int b) { return (a>b) ? a : b; }
23
24  //
25  //      Function prototypes
26  //
27  void InitGraphics(const char *pBGIPath);
28  void ClearSection(int x1, int y1, int x2, int y2);
29
30  //
31  //      Initialize the graphics system via auto detection.
32  //  Supply the path to the BGI files on your system.
33  //
34  void InitGraphics(const char *pBGIPath)
35  {
36    //      Request auto detection of graphics driver.
37    int graphDriver = DETECT;
38    int graphMode;
39
40
41    //  Initialize graphics system.
42    //  Look for files in specified directory.
43    initgraph(&graphDriver, &graphMode, pBGIPath);
44
45    int graphError = graphresult();
46    if(graphError != grOk) {
47            cout << "Graphics error: "
48                    << grapherrormsg(graphError) << '\n';
49            exit(1);
50    }
51  }
52
53  //
54  //      Clear a section of the screen.
55  //
```

continues

LISTING 9.3. CONTINUED

```
56  void ClearSection(int x1, int y1, int x2, int y2)
57  {
58    setviewport(x1, y1, x2, y2, 0);              // no clipping
59    clearviewport();
60    setviewport(0, 0, getmaxx(), getmaxy(), 1); // clipping
61  }
62
63  void main()
64  {
65    InitGraphics("C:\\BC3\\BGI");
66
67    //  Screen dimensions and number of colors
68    const int xmax = getmaxx();
69    const int ymax = getmaxy();
70    const int maxColor = getmaxcolor();
71
72    // Want diameter to be 70% of smaller screen dimension,
73    // so the radius is half of that.
74    const int radius = min(xmax,ymax) * 35 / 100;
75
76    // Coordinates for text output
77    const xText = 10;
78    const yText = 10;
79
80    // X aspect is fixed; Y will cycle through its range.
81    const int xAsp = MAXINT / 10;
82    int yAsp = 0;
83
84    // Text color is constant, while the circles' varies.
85    const int textColor = RED;
86    int color = 0;
87
88    do {
89       // Adjust the aspect ratio;  wrap around as req'd.
90       yAsp -= max(3, yAsp/175);
91       if(yAsp <= 0) {
92               yAsp = MAXINT;
93               clearviewport();
94               settextjustify(LEFT_TEXT, BOTTOM_TEXT);
95               settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
96               setcolor(textColor);
97               outtextxy(10, getmaxy()-10, "Press any key to exit.");
98               settextjustify(LEFT_TEXT, TOP_TEXT);
99       }
100      setaspectratio(xAsp, yAsp);
```

```
101
102       // Draw the circle, cycling the color each time.
103       setcolor(color = (color + 1) % maxColor);
104       circle(xmax/2, ymax/2, radius);
105
106       //
107       // Display the aspect ratio values on the screen.
108       //
109       // First, format the output into a character buffer.
110       char buf[128];
111       ostrstream os(buf, 128);
112       os << "X aspect: "   << setw(7) << xAsp;
113       os << "  Y aspect: " << setw(7) << yAsp;
114       os << "  " << ends;
115
116       // Clear an area the width of the text...
117       ClearSection(xText, yText-2,
118                                  xText + textwidth(buf),
119                                  yText + textheight(buf));
120
121       // ... and out it goes.
122       setcolor(textColor);
123       outtextxy(10, 10, buf);
124
125    } while(!kbhit());
126
127    closegraph();
128 }
```

USING *DRAWPOLY()* AND *FILLPOLY()*

Use `drawpoly()` for drawing simple or complex polygon shapes, where each vertex is specified with an X,Y coordinate (parameter `*polypoints`) and the total number of vertices is indicated with `numpoints`. `drawpoly()` is defined as follows:

```
void far drawpoly(int numpoints, int far *polypoints);
```

The polygon is drawn in connect-the-dots fashion, moving from vertex 1 to vertex 2 to vertex 3, and so on. Figure 9.7 shows a house-shaped object drawn using `drawpoly()`.

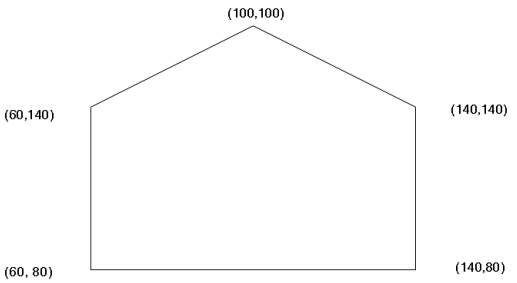


Figure 9.7. An example of an object that can be represented as a sequence of a points.

If the polygon encloses a region, then you must specify one of the points at least twice. In Figure 9.7, drawing the triangle requires `polypoints` to list vertices 1, 2, 3, and then draw the final segment from vertice 3 back to 1. As a result, `polypoints` must contain four sets of coordinates (1, 2, 3, and 1).

The polygon is drawn using the current `setcolor` color, and may be `XOR`'d on to the screen (for easy erasure) by selecting the exclusive-or write mode, by calling `setwritemode (XOR_PUT)` before calling `drawpoly()`.

To draw a filled in polygon, use `fillpoly()` instead of `drawpoly()`. `fillpoly()` is identical to `drawpoly()` except that it fills the interior of the object with the active fill pattern.

CHARTING

Charting is the graphics subspeciality concerned with displaying data as line, bar, pie, and other forms of statistical charts. Borland C++ provides special routines to support creation of these chart types, although realistically they require a substantial bit of supplementary code to create useful general-purpose routines. This section shows you how to create full-featured pie, bar, and line charts. The output from the sample programs is shown in Figures 9.8 through 9.10.

If you have a CGA display or select one of the low resolution modes on EGA or VGA displays, you may have to alter the font scaling size to keep text and objects from overlapping one another.

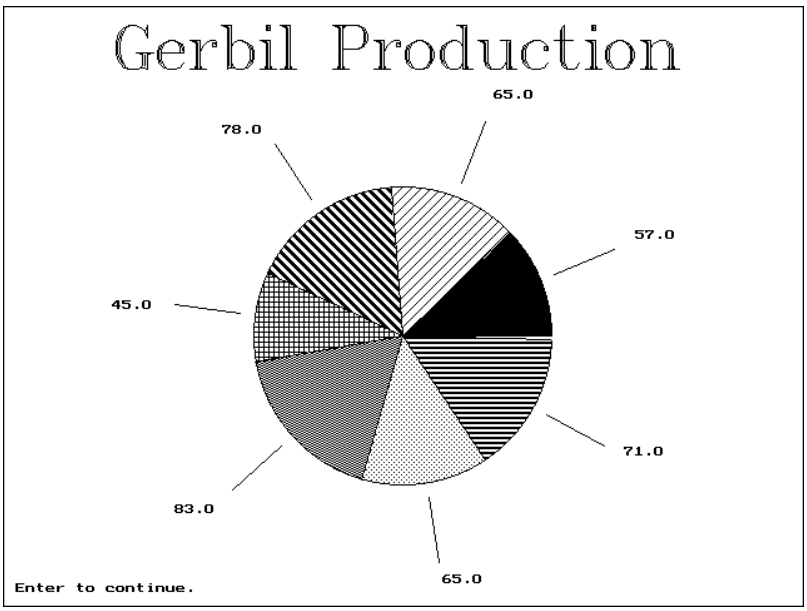


Figure 9.8. The display produced by the piechart program.

When you run the sample programs presented in this chapter, enter sample data when prompted. For these demonstration programs, you can type a value shown in the prompt to signify that you have entered the last data value. For the bar and line charts, the data value you type corresponds to the Y axis. After entering each value, you are prompted for a label. This label becomes the X axis label. For instance, if you draw a bar graph of sales per month, each bar or X axis position is labeled with the month, and the height of the bar corresponds to the data or Y value.

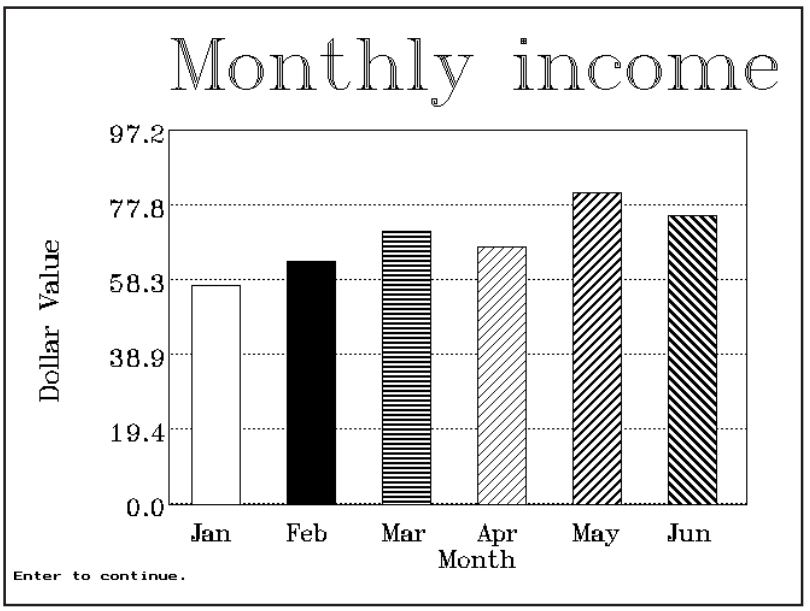


Figure 9.9. The display produced by the barchart program.

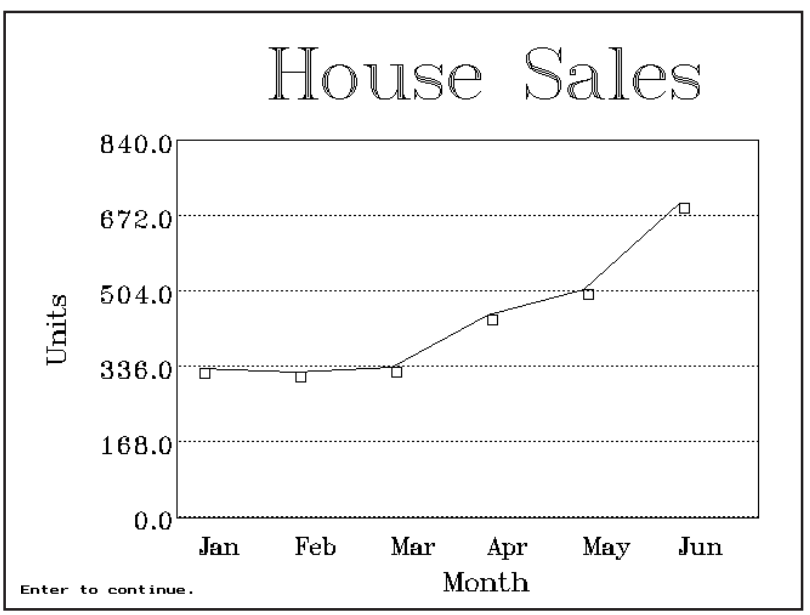


Figure 9.10. The display produced by the linechart program.

The main program body for each of these demonstration programs is pretty much the same, prompting for the graph's title and the actual data values to be graphed. You can enter optional X axis and Y axis titles for the bar and line chart. Once the data is entered, each program calls its respective drawing function: `DrawPieChart`, `DrawBarChart`, or `DrawLineChart`. These programs have been designed to make it easy for you to incorporate the drawing procedures directly into your programs.

THE PIE CHART

The pie chart routine converts data values into angular measurements. For example, if you enter three values—`10`, `20`, and `30`, for example—the size of each pie slice should correspond to the data value. The slice representing 10 is the smallest and the slice representing 30 is the largest. To convert these values to an angular measurement, first calculate the total or sum of the data values (lines 177–182). In this example, 10 + 20 + 30 produces a total of 60. The data value `30` represents 50% of the total, or half of the pie. So the angular measure is 50% of 360, or 180 degrees. Similarly, `10` corresponds to 16.6% of the total, and `20` corresponds to 33.3% of the total. Translated to angles, these produce 16.6% of 360 (60 degrees) and 33.3% of 360 (120 degrees). The calculations to determine the degrees of arc of each pie slice are performed in lines 191–192 of Listing 9.4. The addition of 0.5 in line 192 causes values to round upwards prior to conversion to the integer format.

LISTING 9.4. THE PIECHART PROGRAM.

```
 1  //
 2  //      PIECHART.CPP
 3  //
 4  //      Demonstrate how to create a pie chart.
 5  //
 6  #include <graphics.h>
 7  #include <strstrea.h>
 8  #include <iomanip.h>
 9  #include <stdlib.h>
10  #include <string.h>
11  #include <conio.h>
12  #include <math.h>
13
14  //
```

continues

LISTING 9.4. CONTINUED

```
15  //      Function prototypes
16  //
17  int  GetGraphInfo(char *pTitle, unsigned *pData);
18  void InitGraphics(const char *pBGIPath);
19  void DrawPieChart(const char *pTitle, float *pData,
20                int nVals);
21  void DrawLabel(int xCenter, int yCenter, int radius,
22              int startAngle, int endAngle, float data);
23  void PromptAndWait(const char *pText);
24  void SetTextJustification(float angle);
25
26  //
27  //      Global constants
28  //
29  const int maxDataValues = 20;
30  const int maxTitleLen = 128;
31
32  const float PI = asin(1.0) * 2.0;
33
34  //
35  //      Initialize the graphics system via auto detection.
36  //  Supply the path to the BGI files on your system.
37  //
38  void InitGraphics(const char *pBGIPath)
39  {
40    //      Request auto detection of graphics driver.
41    int graphDriver = DETECT;
42    int graphMode;
43
44
45    //  Initialize graphics system.
46    //  Look for files in specified directory.
47    initgraph(&graphDriver, &graphMode, pBGIPath);
48
49    int graphError = graphresult();
50    if(graphError != grOk) {
51       cout << "Graphics error: "
52            << grapherrormsg(graphError) << '\n';
53       exit(1);
54     }
55  }
56
57  //
58  //      Display a prompt and wait for a keypress.
59  //
60  void PromptAndWait(const char *pText)
61  {
```

```
  settextjustify(LEFT_TEXT, BOTTOM_TEXT);
  settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
  outtextxy(10, getmaxy()-10, pText);
  getch();
}

//
//      Read the graph title and data values.
//  Return the number of values read.
//
int GetGraphInfo(char *pTitle, float *pData)
{
  clrscr();

  // Use cin.get() rather than >> to allow whitespace.
  cout << "Enter graph title: " << endl;
  cin.get(pTitle, maxTitleLen);

  // Note that multiple values on one line are okay.
  cout << "Enter data values (0 when done):" << endl;
  int nVals = 0;
  float val;
  while(1) {
    cin >> val;
    if(val <= 0.0)
            break;
    pData[nVals++] = val;
  }

  return nVals;
}

//
//      Set text justification appropriately given an angle
//  in radians.  There are four possibilities.
//
void SetTextJustification(float angle)
{
  int hJust = LEFT_TEXT;
  if((angle > PI/2) && (angle < PI*3/2))
     hJust = RIGHT_TEXT;

  int vJust = BOTTOM_TEXT;
  if(angle > PI)
     vJust = TOP_TEXT;

  settextjustify(hJust, vJust);
}
```

continues

LISTING 9.4. CONTINUED

```
111  //
112  // Label a pie slice with its data value, given the center
113  // of the pie, its radius, this slice's start and end
114  // angles (in degrees), and the slice's data value.
115  //
116  void DrawLabel(int xCenter, int yCenter, int radius,
117            int startAngle, int endAngle, float data)
118  {
119    //
120    // Calculate the offset of the inner endpoint for the
121    // line that points to this slice, centered on the
122    // slice's arc at 1.1 times the arc's radius.
123    //
124
125    // The angle in radians:
126    float midRadians = (startAngle+endAngle)/2 * PI/180.0;
127
128    // Now the endpoint, adjusted for aspect ratio.
129    // Note that the y axis is inverted!
130    int xAsp;
131    int yAsp;
132    getaspectratio(&xAsp, &yAsp);
133    int dxIn = (int)(1.1 * radius * cos(midRadians));
134    int dyIn = - (int)(1.1 * radius * xAsp *
135                       sin(midRadians)/yAsp);
136
137    //
138    // Calculate the offset of the outer endpoint at the
139    // same angle but 1.4 times the radius.
140    //
141    int dxOut = (int)(dxIn * 1.4);
142    int dyOut = (int)(dyIn * 1.4);
143
144    // Draw the line
145    line(xCenter + dxIn,    yCenter + dyIn,
146         xCenter + dxOut, yCenter + dyOut);
147
148    //
149    // Draw the text slightly beyond the outer end of the
150    // line with the appropriate justification.
151    //
152    SetTextJustification(midRadians);
153
154    dxOut = (int)(1.1 * dxOut);
155    dyOut = (int)(1.1 * dyOut);
156
157    char buf[24];
```

```
158    ostrstream os(buf, 128);
159    // Always use fixed point
160    os.setf(ios::fixed, ios::floatfield);
161    // Always display the decimal point
162    os.setf(ios::showpoint);
163    // Always display 2 digits after the decimal
164    os << setprecision(2) << data << ends;
165    outtextxy(xCenter + dxOut, yCenter + dyOut, buf);
166  }
167
168  //
169  // Given a title, some data, and the number of values,
170  // draw a pie chart.  Label each slice with its data
171  // value.
172  //
173  void DrawPieChart(const char *pTitle, float *pData,
174                    int nVals)
175  {
176    //
177    // Calculate the degrees that each data value spans.
178    // For example, 10% of total spans 360/10 = 36 degrees.
179    //
180    float dataSum = 0.0;
181    for(int i=0; i<nVals; i++)
182            dataSum += pData[i];
183
184    // Allocate the array of degree values
185    int *pDegrees = new int[nVals];
186    if(pDegrees == NULL) {
187            cout << "Out of memory!" << endl;
188            exit(1);
189    }
190
191    for(i=0; i<nVals; i++)
192            pDegrees[i] = (int)(pData[i]/dataSum * 360 + 0.5);
193
194    // The pie's midpoint is centered horizontally and 10
195    // percent below the center of the y axis.
196    const int xCenter = getmaxx()/2;
197    const int yCenter = (int)(1.1 * getmaxy()/2);
198    const int radius  = getmaxy() / 4;
199
200    const int maxColor = getmaxcolor(); // max colors
201    const int maxFillStyle = 11;
202    const int minFillStyle = 1;
203
204    int startAngle = 0;
205    int color = 1;
206    int fillStyle = 1;
```

continues

LISTING 9.4. CONTINUED

```
207
208    //
209    //  Draw the pie and the labels.
210    //
211    for(i=0; i<nVals; i++) {
212
213       // Find the ending angle for this slice.  Adjust for
214       // roundoff error for last slice only.
215       int endAngle = startAngle + pDegrees[i];
216       if(i == nVals-1)
217               endAngle = 360;
218
219       // Draw this slice
220       setfillstyle(fillStyle, color);
221       pieslice(xCenter, yCenter, startAngle, endAngle,
222                         radius);
223
224       // Label it
225       DrawLabel(xCenter, yCenter, radius,
226                 startAngle, endAngle, pData[i]);
227
228       // Set the color and fill style for the next slice
229       color = (color + 1) % maxColor;
230       if(++fillStyle > maxFillStyle)
231               fillStyle = minFillStyle;
232
233       // It starts where this one ended
234       startAngle = endAngle;
235    }
236
237    //
238    // Now draw the title.  Make a copy and truncate it if
239    // it's too long.  Don't modify the original string.
240    //
241    settextjustify(CENTER_TEXT, CENTER_TEXT);
242    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 6);
243
244    char titleCopy[maxTitleLen];
245    strcpy(titleCopy, pTitle);
246    while(textwidth(titleCopy) > getmaxx() - 2)
247            titleCopy[strlen(titleCopy)-1] = '\0';
248
249    outtextxy(xCenter, textheight(pTitle)/2, titleCopy);
250
251    // Free the space for the degrees array
252    delete[] pDegrees;
253 }
254
```

```
255  void main()
256  {
257    char title[maxTitleLen];
258    float data[maxDataValues];
259
260     //  Get the title and data values
261     int nVals = GetGraphInfo(title, data);
262     if(nVals == 0) {
263             cout << "No data entered!" << endl;
264             exit(1);
265     }
266
267     InitGraphics("C:\\BC3\\BGI");
268
269     DrawPieChart(title, data, nVals);
270
271     PromptAndWait("Hit any key to continue...");
272     closegraph();
273  }
```

The first pie slice is drawn beginning at angle 0. If you imagine that the screen holds an old-fashioned analog clock, angle 0 corresponds to the 3 o'clock position, or a horizontal line running to the right. As the degree measurement increases, the position moves around counterclockwise—90 degrees is 12 o'clock, 180 degrees is at 9 o'clock, and so on.

To draw the pie, the code uses `StartAngle` and `EndAngle` to mark the starting point and end point of each slice (see lines 204, 215, and 234). The radius or size of the pie is determined by dividing the maximum number of Y axis pixels by 4 (see line 198). This produces a pie that fills half the screen.

Each slice of the pie is drawn in its own color and with its own interior fill pattern (see line 220). This insures that the pie chart is visually interesting on color monitors, but that it may also be viewed on monochrome screens, because the differing patterns make each slice appear unique. At this point, you could call it quits. However, most pie charts include labels around the graph indicating what each slice represents. For the purposes of this demonstration, use the actual data values themselves, but you could easily substitute descriptive labels, such as *January*, *February*, and so on.

The labels are placed outside the pie, with a line drawn from the label back to the pie slice. The position of the line is calculated by converting the angular measure to X,Y coordinates (see function DrawLabels in lines 112–165). Lines 133–134 calculate the starting position for the line by converting from polar

coordinates (using the radius and the angle) to cartesian coordinates. The result, stored in `dxIn` and `dyIn`, is the starting point for the line that points back to the pie slice. Lines 141–142 compute the ending point of the line, storing the result in `dxOut` and `dyOut`. The line is drawn in lines 145–146. The actual label is written in lines 157–165.

Lastly, the main title is drawn on-screen centered above the pie. To prevent a long title string from overflowing the screen, lines 244–247 remove trailing characters from the title string to ensure that it fits in the allowed space.

There are a number of modifications that you can make to this routine. For instance, it is common practice to sort the data values into ascending or descending order; this can be done by sorting the `Data` array. You should throw out any negative values, because a pie chart cannot represent a mixture of positive and negative values. You can make an *exploded pie*—a pie where one or more slices are moved slightly outwards from the main pie for emphasis—by adjusting the center X and Y values used in calling `pieslice()`. Use a polar to Cartesian coordinate calculation similar to that used for positioning the labels, such as

```
xCenter + (int) cos( AngleInRadians ) * 10
```

and

```
yCenter - (int) sin( AngleInRadians ) * 10
```

This has the effect of shifting the center point outwards, and therefore, the entire pie slice. You can vary the distance by changing the value of the constant `10` in the equations.

When drawing pies, or any graph, you want to avoid mixing extremely large data values with very small data values. If you do mix small and large data values, the effect is a few very thin slivers and one gigantic piece of pie! If the pie doesn't appear round on your display, take a look at the "Solving Aspect Ratio Problems" section earlier in this chapter.

THE BAR CHART

Drawing the actual bar in a bar chart is easy: Borland C++ provides a nifty function `bar()` specifically for drawing the bar portion of a bar chart. But there's more to drawing a bar chart than the bar itself. For one, if the chart is going to

show more than relationships between data, you need to add a Y-axis and a grid to indicate the approximate value of each bar. Along the bottom or X-axis of the chart, you need to add labels identifying what each bar represents. And if your chart has negative values, you need to ensure that positive values are drawn above the Y-axis 0 line and that negative values are drawn below the 0 line. Listing 9.5 contains a complete, albeit lengthy, bar chart drawing program.

LISTING 9.5. THE BARCHART PROGRAM.

```
 1  //
 2  //  BARCHART.CPP
 3  //
 4  //  Demonstrates how to create a bar chart.
 5  //
 6  //
 7  #include <graphics.h>
 8  #include <strstrea.h>
 9  #include <iomanip.h>
10  #include <stdlib.h>
11  #include <string.h>
12  #include <conio.h>
13  #include <math.h>
14
15  typedef int BOOL;                  // boolean
16
17  //
18  //      Global constants
19  //
20  const int maxDataValues = 20;
21  const int maxTitleLen = 128;
22  const int maxLabelLen = 64;
23
24  //
25  //      Inline functions
26  //
27  inline void swap(int &a, int &b)
28  {
29    int t;
30    t = a; a = b; b = t;
31  }
32
33  //
34  //      Function prototypes
35  //
36  void CalcMinAndMax(float &min, float &max, BOOL &fThousands,
```

continues

LISTING 9.5. CONTINUED

```
                    float *pData, int nVals);
void DrawBarChart(char *pMainTitle, char *pxAxisTitle,
                    char *pyAxisTitle,
                    char   pLabels[][maxLabelLen],
                    float *pData, int nVals);
void DrawBars(int xLeft, int yTop, int xRight, int yBottom,
              float minVal, float maxVal,
              float *pData, int nVals);
void DrawTitles(char *pMainTitle, char *pxAxisTitle,
                  char *pyAxisTitle,
                  int xLeft, int yTop,
                  int xRight, int yBottom,
                  int textColor = WHITE);
void DrawXAxisLabels(int xLeft, int xRight, int yBottom,
                    int nVals, char pLabels[][maxLabelLen],
                    int textColor = WHITE);
void DrawYAxisInfo(int xLeft, int yTop,
                    int xRight,int yBottom,
                     float minVal, float maxVal, int nDivs,
                     int lineColor = WHITE,
                     int textColor = WHITE);
int GetGraphInfo(char *pMainTitle, char *pxAxisTitle,
               char *pyAxisTitle,
               char  pLabels[][maxLabelLen],float *pData);
void InitGraphics(const char *pBGIPath);
void PromptAndWait(const char *pText,
                    int textColor=WHITE);

//
//  Initialize the graphics system via auto detection.
//  Supply the path to the BGI files on your system.
//
void InitGraphics(const char *pBGIPath)
{
  //      Request auto detection of graphics driver.
  int graphDriver = DETECT;
  int graphMode;


  //  Initialize graphics system.
  //  Look for files in specified directory.
  initgraph(&graphDriver, &graphMode, pBGIPath);

  int graphError = graphresult();
  if(graphError != grOk) {
          cout << "Graphics error: "
                   << grapherrormsg(graphError) << '\n';
```

```
84            exit(1);
85    }
86  }
87
88  //
89  //      Display a prompt and wait for a keypress.
90  //
91  void PromptAndWait(const char *pText, int textColor)
92  {
93    settextjustify(LEFT_TEXT, BOTTOM_TEXT);
94    settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
95    setcolor(textColor);
96    outtextxy(10, getmaxy()-10, pText);
97    getch();
98  }
99
100 //
101 //      Get the graph titles and data values.
102 //  Return the number of values read.
103 //
104 int GetGraphInfo(char *pMainTitle, char *pxAxisTitle,
105                               char *pyAxisTitle,
106                               char  pLabels[][maxLabelLen],
107                               float *pData)
108 {
109   clrscr();
110
111   // Use cin.getline() rather than >> to allow whitespace.
112   // Use getline() instead of get() to discard '\n'.
113   cout << "Enter main graph title: " << endl;
114   cin.getline(pMainTitle, maxTitleLen);
115   cout << "Enter x-axis title: " << endl;
116   cin.getline(pxAxisTitle, maxTitleLen);
117   cout << "Enter y-axis title: " << endl;
118   cin.getline(pyAxisTitle, maxTitleLen);
119
120   // Get all the data
121   // Note that multiple values on one line are okay
122   cout << "Enter data values (99999 when done): " << endl;
123   int nVals = 0;
124   float val;
125   while(1) {
126           cin >> val;
127           if(val == 99999)
128                   break;
129           pData[nVals++] = val;
130   }
131
132   // Discard any trailing characters
```

continues

LISTING 9.5. CONTINUED

```
133   cin.ignore(255, '\n');
134
135   // Get a label for each piece of data
136   for(int i=0; i<nVals; i++) {
137           cout << "Enter label #" << i << ": " << endl;
138           cin.getline(pLabels[i], maxLabelLen);
139   }
140
141   return nVals;
142 }
143
144 //
145 //  Find the minimum and maximum values in a dataset.
146 //  Set fThousands TRUE if the absolute value of any entry
147 //  is greater than 1000.
148 //
149 void CalcMinAndMax(float &min, float &max, BOOL &fThousands,
150                                float *pData, int nVals)
151 {
152   min = max = 0.0;
153   for(int i=0; i<nVals; i++) {
154           if(pData[i] > max)
155                   max = pData[i];
156           if(pData[i] < min)
157                   min = pData[i];
158   }
159   fThousands = ((fabs(min) > 1000) || (fabs(max) > 1000));
160 }
161
162 //
163 //  Draw the y-axis divisions and value labels given the
164 //  pixel coordinates for the upper-left and lower-right
165 //  corners of the display area, the range of values that
166 //  the axis spans, the number of divisions to use, and
167 //  the color for the lines and text.
168 //
169 void DrawYAxisInfo(int xLeft, int yTop,
170                     int xRight,int yBottom,
171                     float minVal, float maxVal, int nDivs,
172                     int lineColor, int textColor)
173 {
174   settextjustify(LEFT_TEXT, CENTER_TEXT);
175   settextstyle(TRIPLEX_FONT, HORIZ_DIR, 2);
176   setlinestyle(DOTTED_LINE, 0, NORM_WIDTH);
177
178   float range = maxVal - minVal;
179   float dy = range / nDivs;
```

```
180    int dyPix = (yBottom - yTop) / nDivs;
181
182    // Calculate # of decimal places to show
183    int ndigits;
184    if(fabs(dy) < 10/nDivs)
185            ndigits = 2;
186    else if(fabs(dy) < 100/nDivs)
187            ndigits = 1;
188    else
189            ndigits = 0;
190
191    for(int i=0; i<=nDivs; i++) {
192
193       // Pixel value for line and text output
194       int yPix = yTop + i*dyPix;
195
196       // Don't want a dotted line on the edges!
197       if((i != 0) && (i != nDivs)) {
198               setcolor(lineColor);
199               line(xLeft, yPix, xRight, yPix);
200       }
201
202       // Draw the label to the left of the graph
203       float label = maxVal - i*dy;
204       char buf[24];
205       ostrstream os(buf, 24);
206       // Always use fixed point
207       os.setf(ios::fixed, ios::floatfield);
208       // Display the decimal point, sometimes
209       if(ndigits) {
210               os.setf(ios::showpoint);
211               os << setprecision(ndigits);
212       }
213       // Display ndigits after the decimal
214       os << setprecision(ndigits) << label << ends;
215       setcolor(textColor);
216       outtextxy(xLeft - textwidth(buf), yPix, buf);
217    }
218
219    // Draw the 0 line if it's in the display area
220    if((minVal < 0) && (maxVal > 0)) {
221            int pixRange = yBottom - yTop;
222            int yPix = yTop + (int)(maxVal/range * pixRange);
223            setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
224            setcolor(lineColor);
225            line(xLeft, yPix, xRight, yPix);
226    }
227 }
228
229 //
```

continues

LISTING 9.5. CONTINUED

```
230  //  Draw the x-axis labels given the left and right pixel
231  //  coordinates, the number of values, and the labels.
232  //
233  void DrawXAxisLabels(int xLeft, int xRight, int yBottom,
234                       int nVals, char pLabels[][maxLabelLen],
235                       int textColor)
236  {
237    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 1);
238    settextjustify(CENTER_TEXT, TOP_TEXT);
239    setcolor(textColor);
240
241    // The width of each x division, in pixels
242    int dxPix = (xRight - xLeft) / nVals;
243
244    for(int i=0; i<nVals; i++) {
245
246       // Truncate each label if it's too long
247       // Use a copy — don't modify the original
248       char copy[maxLabelLen];
249       strcpy(copy, pLabels[i]);
250       while(textwidth(copy) > dxPix * 0.9)
251               copy[strlen(copy)-1] = '\0';
252
253       // Write the label in the center of this division
254       int xMid = xLeft + i*dxPix + dxPix/2;
255       outtextxy(xMid, yBottom, copy);
256    }
257  }
258
259  //
260  // Draw the bars themselves.
261  //
262  void DrawBars(int xLeft, int yTop, int xRight, int yBottom,
263                        float minVal, float maxVal,
264                        float *pData, int nVals)
265  {
266    // Line style for each bar's bounding rectangle
267    setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
268
269    // The width of each x division, in pixels
270    int dxPix = (xRight - xLeft) / nVals;
271
272    // The percent width of each bar in its pixel division
273    // 0.8 means 80 percent
274    const float widthPct = 0.8;
275    const float barWidth = widthPct * dxPix;
276
```

```
277    const int maxColor = getmaxcolor(); // max colors
278    const int maxFillStyle = 11;
279    const int minFillStyle = 1;
280
281    int color = 1;
282    int fillStyle = 1;
283
284    //
285    //  Calculate the base line. It will be 0 if 0 is on
286    //  the graph, or the top or bottom edge otherwise.
287    //
288    float yrange = maxVal - minVal;
289    int ypixRange = yBottom - yTop;
290    int yBase = yTop + (int)(maxVal/yrange * ypixRange);
291    if(yBase < yTop)
292            yBase = yTop;
293    if(yBase > yBottom)
294            yBase = yBottom;
295
296    for(int i=0; i<nVals; i++) {
297
298      // Calculate the left edge of the bar such that it
299      // will be centered in its division
300      int x = xLeft + i*dxPix + (dxPix - barWidth)/2;
301
302      // Calculate the top and bottom edges of the bar.
303      float yPct = (maxVal - pData[i]) / yrange;
304      int yBarTop = yTop + (int)(yPct * ypixRange);
305      int yBarBottom = yBase;
306
307      // Insure that yBarTop is always the top edge.
308      if(yBarTop > yBarBottom)
309              swap(yBarTop, yBarBottom);
310
311      // Draw the bar and its bounding rectangle.
312      setfillstyle(fillStyle, color);
313      bar      (x, yBarTop, x+barWidth, yBarBottom);
314      rectangle(x, yBarTop, x+barWidth, yBarBottom);
315
316      // Set the color and fill style for the next bar
317      // Varying both color and fill style accomodates
318      // monochrome displays, too
319      color = (color + 1) % maxColor;
320      if(++fillStyle > maxFillStyle)
321              fillStyle = minFillStyle;
322    }
323  }
324
325  //
326  // Draw the titles.
```

continues

LISTING 9.5. CONTINUED

```
//
void DrawTitles(char *pMainTitle, char *pxAxisTitle,
                            char *pyAxisTitle,
                            int xLeft, int yTop,
                            int xRight, int yBottom,
                            int textColor)
{
  // Use this to make string copies that we can modify
  char copy[maxTitleLen];

  int xWidth = xRight - xLeft;
  int xCenter = xLeft + xWidth / 2;
  int yHeight = yBottom - yTop;
  int yCenter = yTop + yHeight / 2;

  setcolor(textColor);

  // Main title
  strcpy(copy, pMainTitle);
  settextjustify(CENTER_TEXT, TOP_TEXT);
  settextstyle(TRIPLEX_FONT, HORIZ_DIR, 7);
  while(textwidth(copy) > getmaxx() - 2)
          copy[strlen(copy)-1] = '\0';
  outtextxy(getmaxx()/2, 5, copy);

  // x-axis title
  strcpy(copy, pxAxisTitle);
  settextjustify(CENTER_TEXT, BOTTOM_TEXT);
  settextstyle(TRIPLEX_FONT, HORIZ_DIR, 2);
  while(textwidth(copy) > xWidth)
          copy[strlen(copy)-1] = '\0';
  outtextxy(xCenter, getmaxy() - 5, copy);

  // y-axis title
  strcpy(copy, pyAxisTitle);
  settextjustify(LEFT_TEXT, CENTER_TEXT);
  settextstyle(TRIPLEX_FONT, VERT_DIR, 3);
  while(textwidth(copy) > yHeight)
          copy[strlen(copy)-1] = '\0';
  outtextxy(5, yCenter, copy);
}

//
// Draw a bar chart, complete with bars, axes, and titles.
//
void DrawBarChart(char *pMainTitle, char *pxAxisTitle,
                              char *pyAxisTitle,
```

```
374                                 char  pLabels[][maxLabelLen],
375                                 float *pData, int nVals)
376  {
377    float minVal;
378    float maxVal;
379    BOOL fThousands;          // true if abs(any data) > 1000
380
381    //
382    // Find the minimum and maximum values.  Then, if the
383    //  any of the data is in thousands, scale it down.
384    //
385    CalcMinAndMax(minVal, maxVal, fThousands, pData, nVals);
386    if(fThousands) {
387            for(int i=0; i<nVals; i++)
388                     pData[i] /= 1000.0;
389            minVal /= 1000.0;
390            maxVal /= 1000.0;
391    }
392
393    //
394    // Scale the minimum and maximum values so there's some
395    // space at the top and bottom of the graph.
396    //
397    const float minScaleVal = 0.1;
398    const float maxScaleVal = 0.1;
399    if(minVal > 0)
400            minVal *= (1.0 - minScaleVal);
401    else
402            minVal *= (1.0 + minScaleVal);
403
404    if(maxVal > 0)
405            maxVal *= (1.0 + maxScaleVal);
406    else
407            maxVal *= (1.0 - maxScaleVal);
408
409    //
410    //      Draw the rectangle around the charting area.
411    //
412    const int xmax = getmaxx();
413    const int ymax = getmaxy();
414    const float xLeft   = 0.20 * xmax;
415    const float xRight  = 0.95 * xmax;
416    const float yTop    = 0.20 * ymax;
417    const float yBottom = 0.85 * ymax;
418
419    rectangle(xLeft, yTop, xRight, yBottom);
420
421    // If using scaled data, say so!
422    if(fThousands) {
```

continues

LISTING 9.5. CONTINUED

```
423     const char *pThous = "In 1,000's";
424     settextjustify(LEFT_TEXT, BOTTOM_TEXT);
425     outtextxy(xLeft, yTop-3, pThous);
426   }
427
428   // Draw the y axis divisions and numerical labels
429   const int nYDivs = 5;
430   DrawYAxisInfo(xLeft, yTop, xRight, yBottom,
431                               minVal, maxVal, nYDivs);
432
433   // Draw the x axis labels
434   DrawXAxisLabels(xLeft, xRight, yBottom, nVals, pLabels);
435
436   // Draw the data bars
437   DrawBars(xLeft, yTop, xRight, yBottom, minVal, maxVal,
438                     pData, nVals);
439
440   // Draw the titles
441   DrawTitles(pMainTitle, pxAxisTitle, pyAxisTitle,
442              xLeft, yTop, xRight, yBottom, RED);
443 }
444
445 void main()
446 {
447   char mainTitle[maxTitleLen];
448   char xAxisTitle[maxTitleLen];
449   char yAxisTitle[maxTitleLen];
450   char labels[maxDataValues][maxLabelLen];
451   float data[maxDataValues];
452
453   //  Get the title and data values
454   int nVals = GetGraphInfo(mainTitle, xAxisTitle,
455                            yAxisTitle, labels, data);
456   if(nVals == 0) {
457           cout << "No data entered!" << endl;
458           exit(1);
459   }
460
461   InitGraphics("C:\\BC3\\BGI");
462
463   DrawBarChart(mainTitle, xAxisTitle, yAxisTitle,
464                               labels, data, nVals);
465
466   PromptAndWait("Hit any key to continue...");
467   closegraph();
468 }
```

Line 385 calls `CalcMinAndMax()` to determine the maximum and minimum data values, and set a flag, `fThousands`, if any value exceeds 1,000. If there are values over 1,000, then all the data values are divided by 1,000 and a notation `In 1,000s` is placed on the chart. This way, the number of digits shown on the Y-axis won't become so large that they do not fit.

The maximum value is adjusted slightly upwards (see lines 404–407) so that the top of the grid will be slightly higher than the highest data value entered. If we didn't do this, the bar representing the largest data value would push right up against the top of the chart, and would not be aesthetically pleasing.

Lines 414–417 compute the location of the upper left and lower right corners of the bounding rectangle that will contain the bar chart. The size of the bounding rectangle is computed as a percent of the total screen area, as determined by `xmax` and `ymax`. Function `DrawYAxisInfo()` (lines 163–227) draws grid lines across the bounding rectangle, corresponding to various `Y` values that are shown along the Y-axis, and also displays the Y-axis labels. The grid helps the user interpret the bar chart. You can vary the number of grid lines by changing the constant `nYDivs` (see line 429), here set to 5. After the grid lines are drawn, a grid line marking the 0th value along the Y-axis is added (see lines 219–226). This is particularly useful when the bar chart contains both positive and negative vlues.

Function `DrawBars()` (lines 260–323) displays the actual bars and their labels. As with the pie chart, each bar is given both a different pattern and a different color, so you may use this code on either color or monochromatic displays. Finally, the bar itself is drawn. If the bar represents a positive value, the code in lines 285–294 handles proper placement of the bar above the 0 line; if the bar represents a negative value, the code places the bar below the 0 line. In either case, a solid line rectangle is drawn around each bar for a more pleasing look, because the `bar()` library function does not draw a border around the bar that it displays. Instead of using `rectangle()` to draw a border, you may use the graphics library function `bar3d()` and set `bar3d`'s `Depth` parameter to zero.

The main title is positioned above the bar chart and the X-axis title is positioned along the bottom. The Y-axis is unique in that its title is drawn vertically up the left side of the graph, by choosing the `VERT_DIR` option for `settextstyle()` (see line 363). Each of the title items is truncated, if necessary, to fit within the region allotted to it.

As with the pie chart, there are a number of options you might consider in modifying this code for your use. The constant values used within `DrawBarChart` (lines 414–417) can be set to reposition the top, bottom, left, and right sides of the charting area. Because the code is written to work with varous screen resolutions, these constants define a percent of the screen rather than actual pixel locations. Therefore, `0.20` means that the leftmost edge of the chart appears 20% of the screen width from the left edge of the screen. The right edge is set to 95% (or `0.95`) of the screen width to the right. The top is positioned 20 percent of the pixels from the top of the screen, reserving the top 20 percent of the screen for the main title. You can adjust these values up or down, as desired, to create more or less free space in the chart. `widthPct` (line 274) specifies how wide the bar appears. The actual width depends on the number of data elements in the data set. If there are four data values, each bar potentially could occupy up to 25 percent of the horizontal graph space. `widthPct` specifies how much of this potential space should actually be used. The default value of `0.8` means that only 80 percent of this space is used for each bar, leaving a small amount of unused area between each bar.

THE LINE CHART

The `DrawLineChart` procedure in Listing 9.6 is very similar to `DrawBarChart`. The significant change is to draw each point and a line connecting one point to the next. The primary differences appear in lines 297–354, with line 348 drawing the actual connecting line from the *current point* (depending on how you look at it, this might also be called the *previous point*), to the next point. `DrawPoint()` draws a tiny rectangle to mark each data point. The third parameter to `DrawPoint()` selects a green rectangle for positive values, or a red rectangle for negative values.

Often, multiple data series are drawn on a single chart, giving 2 or 3 data lines, for example. If you choose to modify the routines to handle this type of graph, you may wish to change `DrawPoint` so that it uses a different shape for the data values in each series.

LISTING 9.6. THE LINCHART SAMPLE PROGRAM.

```
1  //
2  //  LINCHART.CPP
3  //
4  //  Demonstrate how to create a line chart.
```

```
 5  //
 6  //
 7  #include <graphics.h>
 8  #include <strstrea.h>
 9  #include <iomanip.h>
10  #include <stdlib.h>
11  #include <string.h>
12  #include <conio.h>
13  #include <math.h>
14
15  typedef int BOOL;                  // boolean
16
17  //
18  //      Global constants
19  //
20  const int maxDataValues = 20;
21  const int maxTitleLen = 128;
22  const int maxLabelLen = 64;
23
24  //
25  //      Function prototypes
26  //
27  void CalcMinAndMax(float &min, float &max, BOOL &fThousands,
28                     float *pData, int nVals);
29  void DrawLineChart(char *pMainTitle, char *pxAxisTitle,
30                     char *pyAxisTitle,
31                     char  pLabels[][maxLabelLen],
32                     float *pData, int nVals);
33  void DrawLines(int xLeft, int yTop, int xRight, int yBottom,
34                 float minVal, float maxVal,
35                 float *pData, int nVals,
36                 int lineColor=WHITE,
37                 int posPtColor=GREEN, int negPtColor=RED);
38  void DrawPoint(int x, int y, int color = WHITE, int size=8);
39  void DrawTitles(char *pMainTitle, char *pxAxisTitle,
40                  char *pyAxisTitle,
41                  int xLeft, int yTop,
42                  int xRight, int yBottom,
43                  int textColor = WHITE);
44  void DrawXAxisLabels(int xLeft, int xRight, int yBottom,
45                  int nVals, char pLabels[][maxLabelLen],
46                  int textColor = WHITE);
47  void DrawYAxisInfo(int xLeft, int yTop,
48                   int xRight,int yBottom,
49                   float minVal, float maxVal, int nDivs,
50                   int lineColor = WHITE,
51                   int textColor = WHITE);
52  int GetGraphInfo(char *pMainTitle, char *pxAxisTitle,
53                 char *pyAxisTitle,
```

continues

LISTING 9.6. CONTINUED

```
54                char  pLabels[][maxLabelLen],float *pData);
55  void InitGraphics(const char *pBGIPath);
56  void PromptAndWait(const char *pText, int textColor=WHITE);
57
58  //
59  //  Initialize the graphics system via auto detection.
60  //  Supply the path to the BGI files on your system.
61  //
62  void InitGraphics(const char *pBGIPath)
63  {
64    //      Request auto detection of graphics driver.
65    int graphDriver = DETECT;
66    int graphMode;
67
68
69    //  Initialize graphics system.
70    //  Look for files in specified directory.
71    initgraph(&graphDriver, &graphMode, pBGIPath);
72
73    int graphError = graphresult();
74    if(graphError != grOk) {
75       cout << "Graphics error: "
76            << grapherrormsg(graphError) << '\n';
77       exit(1);
78    }
79  }
80
81  //
82  // Display a prompt and wait for a keypress.
83  //
84  void PromptAndWait(const char *pText, int textColor)
85  {
86    settextjustify(LEFT_TEXT, BOTTOM_TEXT);
87    settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
88    setcolor(textColor);
89    outtextxy(10, getmaxy()-10, pText);
90    getch();
91  }
92
93  //
94  //      Get the graph titles and data values.
95  //  Return the number of values read.
96  //
97  int GetGraphInfo(char *pMainTitle, char *pxAxisTitle,
98                   char *pyAxisTitle,
99                   char  pLabels[][maxLabelLen],
100                  float *pData)
```

```
{
  clrscr();

  // Use cin.getline() rather than >> to allow whitespace.
  // Use getline() instead of get() to discard '\n'.
  cout << "Enter main graph title: " << endl;
  cin.getline(pMainTitle, maxTitleLen);
  cout << "Enter x-axis title: " << endl;
  cin.getline(pxAxisTitle, maxTitleLen);
  cout << "Enter y-axis title: " << endl;
  cin.getline(pyAxisTitle, maxTitleLen);

  // Get all the data
  // Note that multiple values on one line are okay
  cout << "Enter data values (99999 when done): " << endl;
  int nVals = 0;
  float val;
  while(1) {
    cin >> val;
    if(val == 99999)
            break;
    pData[nVals++] = val;
  }

  // Discard any trailing characters
  cin.ignore(255, '\n');

  // Get a label for each piece of data
  for(int i=0; i<nVals; i++) {
    cout << "Enter label #" << i << ": " << endl;
    cin.getline(pLabels[i], maxLabelLen);
  }

  return nVals;
}

//
//  Find the minimum and maximum values in a dataset.
//  Set fThousands TRUE if the absolute value of any entry
//  is greater than 1000.
//
void CalcMinAndMax(float &min, float &max, BOOL &fThousands,
                   float *pData, int nVals)
{
  min = max = 0.0;
  for(int i=0; i<nVals; i++) {
    if(pData[i] > max)
            max = pData[i];
    if(pData[i] < min)
```

continues

LISTING 9.6. CONTINUED

```
150             min = pData[i];
151   }
152   fThousands = ((fabs(min) > 1000) || (fabs(max) > 1000));
153 }
154
155 //
156 //  Draw the y-axis divisions and value labels given the
157 //  pixel coordinates for the upper-left and lower-right
158 //  corners of the display area, the range of values that
159 //  the axis spans, the number of divisions to use, and
160 //  the color for the lines and text.
161 //
162 void DrawYAxisInfo(int xLeft, int yTop,
163                    int xRight,int yBottom,
164                    float minVal, float maxVal, int nDivs,
165                    int lineColor, int textColor)
166 {
167   settextjustify(LEFT_TEXT, CENTER_TEXT);
168   settextstyle(TRIPLEX_FONT, HORIZ_DIR, 2);
169   setlinestyle(DOTTED_LINE, 0, NORM_WIDTH);
170
171   float range = maxVal - minVal;
172   float dy = range / nDivs;
173   int dyPix = (yBottom - yTop) / nDivs;
174
175   // Calculate # of decimal places to show
176   int ndigits;
177   if(fabs(dy) < 10/nDivs)
178           ndigits = 2;
179   else if(fabs(dy) < 100/nDivs)
180           ndigits = 1;
181   else
182           ndigits = 0;
183
184   for(int i=0; i<=nDivs; i++) {
185
186     // Pixel value for line and text output
187     int yPix = yTop + i*dyPix;
188
189     // Don't want a dotted line on the edges!
190     if((i != 0) && (i != nDivs)) {
191             setcolor(lineColor);
192             line(xLeft, yPix, xRight, yPix);
193     }
194
195     // Draw the label to the left of the graph
196     float label = maxVal - i*dy;
```

```
197     char buf[24];
198     ostrstream os(buf, 24);
199     // Always use fixed point
200     os.setf(ios::fixed, ios::floatfield);
201     // Display the decimal point, sometimes
202     if(ndigits) {
203             os.setf(ios::showpoint);
204             os << setprecision(ndigits);
205     }
206     // Display ndigits after the decimal
207     os << setprecision(ndigits) << label << ends;
208     setcolor(textColor);
209     outtextxy(xLeft - textwidth(buf), yPix, buf);
210   }
211
212   // Draw the 0 line if it's in the display area
213   if((minVal < 0) && (maxVal > 0)) {
214     int pixRange = yBottom - yTop;
215     int yPix = yTop + (int)(maxVal/range * pixRange);
216     setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
217     setcolor(lineColor);
218     line(xLeft, yPix, xRight, yPix);
219   }
220 }
221
222 //
223 //  Draw the x-axis labels given the left and right pixel
224 //  coordinates, the number of values, and the labels.
225 //
226 void DrawXAxisLabels(int xLeft, int xRight, int yBottom,
227                      int nVals, char pLabels[][maxLabelLen],
228                      int textColor)
229 {
230   settextstyle(TRIPLEX_FONT, HORIZ_DIR, 1);
231   settextjustify(CENTER_TEXT, TOP_TEXT);
232   setcolor(textColor);
233
234   // The width of each x division, in pixels
235   int dxPix = (xRight - xLeft) / nVals;
236
237   for(int i=0; i<nVals; i++) {
238
239     // Truncate each label if it's too long
240     // Use a copy -- don't modify the original
241     char copy[maxLabelLen];
242     strcpy(copy, pLabels[i]);
243     while(textwidth(copy) > dxPix * 0.9)
244             copy[strlen(copy)-1] = '\0';
245
```

continues

LISTING 9.6. CONTINUED

```
246     // Write the label in the center of this division
247     int xMid = xLeft + i*dxPix + dxPix/2;
248     outtextxy(xMid, yBottom, copy);
249   }
250 }
251
252 //
253 //      Draw the titles.
254 //
255 void DrawTitles(char *pMainTitle, char *pxAxisTitle,
256                 char *pyAxisTitle,
257                 int xLeft, int yTop,
258                 int xRight, int yBottom,
259                 int textColor)
260 {
261   // Use this to make string copies that we can modify
262   char copy[maxTitleLen];
263
264   int xWidth = xRight - xLeft;
265   int xCenter = xLeft + xWidth / 2;
266   int yHeight = yBottom - yTop;
267   int yCenter = yTop + yHeight / 2;
268
269   setcolor(textColor);
270
271   // Main title
272   strcpy(copy, pMainTitle);
273   settextjustify(CENTER_TEXT, TOP_TEXT);
274   settextstyle(TRIPLEX_FONT, HORIZ_DIR, 7);
275   while(textwidth(copy) > getmaxx() - 2)
276           copy[strlen(copy)-1] = '\0';
277   outtextxy(getmaxx()/2, 5, copy);
278
279   // x-axis title
280   strcpy(copy, pxAxisTitle);
281   settextjustify(CENTER_TEXT, BOTTOM_TEXT);
282   settextstyle(TRIPLEX_FONT, HORIZ_DIR, 2);
283   while(textwidth(copy) > xWidth)
284           copy[strlen(copy)-1] = '\0';
285   outtextxy(xCenter, getmaxy() - 5, copy);
286
287   // y-axis title
288   strcpy(copy, pyAxisTitle);
289   settextjustify(LEFT_TEXT, CENTER_TEXT);
290   settextstyle(TRIPLEX_FONT, VERT_DIR, 3);
291   while(textwidth(copy) > yHeight)
292           copy[strlen(copy)-1] = '\0';
```

```
293   outtextxy(5, yCenter, copy);
294 }
295
296 //
297 //  Draw a square data point centered at the given pixel
298 //  coordinates in the given color and with sides of the
299 //  given size.
300 //
301 void DrawPoint(int x, int y, int color, int size)
302 {
303   int saveColor = getcolor();
304   setcolor(color);
305
306   // cut the size in half for convenience
307   size /= 2;
308   rectangle(x-size, y-size, x+size, y+size);
309
310   setcolor(saveColor);
311 }
312
313 //
314 //  Draw each data point and connect them with lines.  The
315 //  line will be drawn in lineColor, positive points in
316 //  posPtColor, and negative points in negPtColor.
317 //
318 void DrawLines(int xLeft, int yTop, int xRight, int yBottom,
319                float minVal, float maxVal,
320                float *pData, int nVals,
321                int lineColor,
322                int posPtColor, int negPtColor)
323 {
324   setcolor(lineColor);
325   setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
326
327   // The width of each x division, in pixels
328   int dxPix = (xRight - xLeft) / nVals;
329
330   // Pixel values to keep track of the last point drawn
331   int xPrev;
332   int yPrev;
333
334   for(int i=0; i<nVals; i++) {
335     // The mid point in this division
336     int xMid = xLeft + i*dxPix + dxPix/2;
337
338     // The y coordinate for this point
339     float yPct = (maxVal - pData[i]) / (maxVal-minVal);
340     int yPix = yTop + (int)(yPct * (yBottom - yTop));
341
```

continues

LISTING 9.6. CONTINUED

```
342      // Plot the point in the appropriate color
343      DrawPoint(xMid, yPix,
344                (pData[i] >= 0) ? posPtColor : negPtColor);
345
346      // Draw a line for all points except the first
347      if(i > 0)
348              line(xPrev, yPrev, xMid, yPix);
349
350      // So we'll know where to draw from next time
351      xPrev = xMid;
352      yPrev = yPix;
353    }
354  }
355
356  //
357  // Draw a line chart, with points, axes, and titles.
358  //
359  void DrawLineChart(char *pMainTitle, char *pxAxisTitle,
360                char *pyAxisTitle,
361                char  pLabels[][maxLabelLen],
362                float *pData, int nVals)
363  {
364    float minVal;
365    float maxVal;
366    BOOL fThousands;          // true if abs(any data) > 1000
367
368    //
369    //  Find the minimum and maximum values.  Then, if the
370    //  any of the data is in thousands, scale it down.
371    //
372    CalcMinAndMax(minVal, maxVal, fThousands, pData, nVals);
373    if(fThousands) {
374            for(int i=0; i<nVals; i++)
375                    pData[i] /= 1000.0;
376            minVal /= 1000.0;
377            maxVal /= 1000.0;
378    }
379
380    //
381    //   Scale the minimum and maximum values so there's some
382    //  space at the top and bottom of the graph.
383    //
384    const float minScaleVal = 0.1;
385    const float maxScaleVal = 0.1;
386    if(minVal > 0)
387            minVal *= (1.0 - minScaleVal);
388    else
```

```
389             minVal *= (1.0 + minScaleVal);
390
391     if(maxVal > 0)
392             maxVal *= (1.0 + maxScaleVal);
393     else
394             maxVal *= (1.0 - maxScaleVal);
395
396     //
397     //  Draw the rectangle around the charting area, using
398     //  cyan as the color.
399     //
400     const int xmax = getmaxx();
401     const int ymax = getmaxy();
402     const float xLeft   = 0.20 * xmax;
403     const float xRight  = 0.95 * xmax;
404     const float yTop    = 0.20 * ymax;
405     const float yBottom = 0.85 * ymax;
406
407     setcolor(CYAN);
408     rectangle(xLeft, yTop, xRight, yBottom);
409     setcolor(WHITE);
410
411     // If using scaled data, say so!
412     if(fThousands) {
413             const char *pThous = "In 1,000's";
414             settextjustify(LEFT_TEXT, BOTTOM_TEXT);
415             outtextxy(xLeft, yTop-3, pThous);
416     }
417
418     // Draw the y axis divisions and numerical labels
419     const int nYDivs = 5;
420     DrawYAxisInfo(xLeft, yTop, xRight, yBottom,
421                                 minVal, maxVal, nYDivs,
422           CYAN);
423
424     // Draw the x axis labels
425     DrawXAxisLabels(xLeft, xRight, yBottom, nVals, pLabels);
426
427     // Draw the lines
428     DrawLines(xLeft, yTop, xRight, yBottom, minVal, maxVal,
429                         pData, nVals);
430
431     // Draw the titles
432     DrawTitles(pMainTitle, pxAxisTitle, pyAxisTitle,
433                          xLeft, yTop, xRight, yBottom);
434   }
435
436   void main()
437   {
```

continues

LISTING 9.6. CONTINUED

```
438    char mainTitle[maxTitleLen];
439    char xAxisTitle[maxTitleLen];
440    char yAxisTitle[maxTitleLen];
441    char labels[maxDataValues][maxLabelLen];
442    float data[maxDataValues];
443
444    //  Get the title and data values
445    int nVals = GetGraphInfo(mainTitle, xAxisTitle,
446                                yAxisTitle, labels, data);
447    if(nVals == 0) {
448            cout << "No data entered!" << endl;
449            exit(1);
450    }
451
452    InitGraphics("C:\\BC3\\BGI");
453
454    DrawLineChart(mainTitle, xAxisTitle, yAxisTitle,
455              labels, data, nVals);
456
457    PromptAndWait("Hit any key to continue...");
458    closegraph();
459  }
```

GRAPHICS DRIVERS AND FONT FILES

Borland C++ supports several types of graphic display monitors and several modes for each monitor type. This support is provided through a set of graphics driver files contained in the \borlandc\bgi (the default location) subdirectory, each ending in .bgi. These files are shown in Table 9.3.

The function `initgraph()` is responsible for loading (and optionally, automatically choosing) the correct driver file. These driver files provide the interface between the Borland C++ graphics routines and the underlying hardware. Once you have finished using the graphics driver, you terminate the Borland C++ graphics system by calling `closegraph()`, which unloads the driver file from memory.

TABLE 9.3. GRAPHICS DRIVERS PROVIDED BY BORLAND.

Filename	*Device Supported*
att.bgi	AT&T 6300 in 400 line resolution
cga.bgi	CGA and MCGA adaptors
egavga.bgi	EGA and VGA monitors
herc.bgi	Hercules monochrome graphics adaptor
ibm8514.bgi	IBM 8514 high resolution display
pc3270.bgi	Supports the IBM 3270 PC

After calling `initgraph()`, the program switches to graphics mode. Normal `printf()` statements will not send your output to the graphics screen. (You can, however, switch back and forth between graphics mode and text mode by calling `restorecrtmode()` to switch back to text mode, and calling `setgraphmode()` when you wish to return to graphics mode.)

The sample program description presented at the beginning of this chapter included additional information on the use of `initgraph()` to automatically detect or manually select an appropriate graphics driver file. If you wish to support the IBM 8514 display, your program must explicitly select the driver by setting the *`graphdriver`* parameter to the `IBM8514` constant when calling `initgraph()`. The automatic detect feature of `initgraph()` thinks that the 8514 is a VGA display; hence, to use the 8514 you must set the *`graphdriver`* variable equal to the `IBM8514` constant before calling `initgraph()`.

LINKING DEVICE DRIVERS AND FONT FILES

Borland C++'s support for a variety of graphics display adaptors and a selection of stroked character fonts is nice, but dragging around the .bgi driver and .chr font files can be a real bother, especially if your software is going to run on someone else's computer. Fortunately, there's a way you can combine the

various .bgi and .chr files into your .exe file so that your users get one big .exe file instead of a collection of small driver and font files. An advantage of linked .chr files is that drawing performance may be improved if you frequently switch back and forth between different stroked fonts (see `registerfarbgifont` in the System Library Reference for more detals). The steps to linking the driver and font files together with your .exe file are presented in this section.

You can optionally link one or many of the .bgi files, as well as one or many of the .chr files. Generally, if you are linking some of the .bgi files, you will also link at least one of the .chr files. It would be unusual to use graphics support without using at least one character font.

CONVERTING .BGI AND .CHR FILES INTO .OBJ FILES

The first step in the link process is to convert the needed .bgi and .chr files into .obj object file format using the Borland provided utility program BGIOBJ. BGIOBJ is located in the \borlandc\bgi directory. For ease of use, I suggest that you copy bgiobj.exe to the \borlandc\bgi directory. Alternately, you may place \borlandc\utils into your DOS PATH statement.

BGIOBJ is executed from the DOS command line, with three command-line parameters.

```
BGIOBJ inputfile outputfile public name
```

The first parameter, *`inputfile`*, is the name of the file that needs converting; *`outputfile`* is the name of the .obj file to create. *`public name`* sets this name as the entry point for the module. This name is important and is used by your program to load the driver or font information from the .exe file (see the section called "Modifying Your Program to Reference the Linked .bgi and .chr Files"). In the case of the Borland supplied font and driver files, you can run BGIOBJ with only the *`inputfile`* specified. The rest of the values will be set to appropriate defaults. For example, to convert the egavga.bgi driver file to .obj file format, type the following:

```
BGIOBJ /F egavga
```

This will create an object file named fegavga.obj. Use BGIOBJ to convert each of the driver files that you will use in your project. You should also use BGIOBJ, as shown in this example, to convert all desired font files:

```
BGIOBJ /F goth
```

This produces a file named gothf.obj.

Add each of the generated .obj files to your program's project or make file. This will ensure that the converted driver and font files are added into the .exe application file when the .exe is linked.

MODIFYING YOUR PROGRAM TO REFERENCE THE LINKED .BGI AND .CHR FILES

In addition to linking the necessary drivers and fonts, your program must register the respective drivers and fonts with the graphics systems. This is done by calling the `registerfarbgidriver` or `registerfarbgifont` functions, respectively.

Listing 9.7 demonstrates how to call these functions. The procedure names used in these external statements match the corresponding *`public name`* used in the bgiobj command line statement or a default name determined by the BGIOBJ program. Lines 72–81 and 89–96 show the format of the default names for the driver and font files, respectively. It's very important that you ensure that the same *`public name`* is used in both places: on the BGIOBJ command line and in the external declaration within your program. You don't need to list all of the drivers or all of the fonts; declare only those that you need to use in your application.

LISTING 9.7. A DEMONSTRATION OF THE SOURCE CODE CHANGES NEEDED TO LINK .BGI AND .CHR FILES.

```
 1  //
 2  //  DEMOOBJ.CPP
 3  //
 4  //  Demonstrates how to use BGI fonts and drivers that have
 5  //  been converted to OBJ files with BGIOBJ.EXE and linked
 6  //  into the EXE as part of your project.
 7  //
 8  //
 9  //  To build this program, do the following:
10  //
11  //  1. Run BIGOBJ with the /F switch on the following font
12  //   files, located in the \BORLANDC\BGI directory:
13  //
14  //        GOTH.CHR, LITT.CHR, SANS.CHR, TRIP.CHR
```

continues

LISTING 9.7. CONTINUED

```
15  //
16  //     Example:  BIGOBJ /F GOTH
17  //
18  //     This will produce 4 OBJ files with the names
19  //
20  //       GOTHF.OBJ, LITTF.OBJ, SANSF.OBJ, TRIPF.OBJ
21  //
22  //
23  //  2. Run BGIOBJ /F on the following driver files, located
24  //     in the \BORLANDC\BGI directory:
25  //
26  //        ATT.BGI, CGA.BGI, EGAVGA.BGI, HERC.BGI,
27  //        IBM8514.BGI, PC3270.BGI
28  //
29  //     Example:  BGIOBJ /F ATT
30  //
31  //     This will produce 6 OBJ files with the names
32  //
33  //        ATTF.OBJ, CGAF.OBJ, EGAVGAF.OBJ, HERCF.OBJ,
34  //        IBM8514F.OBJ, PC3270F.OBJ
35  //
36  //  3.  Add all 10 OBJ files to your project.  Make sure
37  //      that you specify the correct path.
38  //
39  //  4.  Build and run!
40  //
41
42  #include <graphics.h>
43  #include <iostream.h>
44  #include <stdlib.h>
45  #include <conio.h>
46
47  //
48  //  Function prototypes.
49  //
50  void BGIProblem(char *pProblem);
51  void DoDemo();
52  void InitGraphics(const char *pBGIPath);
53  void PromptAndWait(const char *pText, int textColor=WHITE);
54  void RegisterDrivers();
55  void RegisterFonts();
56
57  //
58  //  If a problem occurs during BGI registration, this
59  //  function is called.  It prints an error and exits.
60  //
61  void BGIProblem(char *pProblem)
```

```
62  {
63      cout << "Error:  " << pProblem << endl;
64      exit(1);
65  }
66
67  //
68  //  Register all the drivers.
69  //
70  void RegisterDrivers()
71  {
72    if(registerfarbgidriver(ATT_driver_far) < 0)
73            BGIProblem("AT&T Driver");
74    if(registerfarbgidriver (CGA_driver_far) < 0)
75            BGIProblem("CGA Driver");
76    if(registerfarbgidriver (EGAVGA_driver_far) < 0)
77            BGIProblem("EGA/VGA Driver");
78    if(registerfarbgidriver (Herc_driver_far) < 0)
79            BGIProblem("Hercules Driver");
80    if(registerfarbgidriver (PC3270_driver_far) < 0)
81            BGIProblem("PC3270 Driver");
82  }
83
84  //
85  //  Register all the fonts.
86  //
87  void RegisterFonts()
88  {
89    if(registerfarbgifont(gothic_font_far) < 0)
90            BGIProblem("gothic");
91    if(registerfarbgifont(sansserif_font_far) < 0)
92            BGIProblem("sans serif");
93    if(registerfarbgifont(small_font_far) < 0)
94            BGIProblem("small font");
95    if(registerfarbgifont(triplex_font_far) < 0)
96            BGIProblem("triplex font");
97  }
98
99  //
100 //      Initialize the graphics system via auto detection.
101 //  Supply the path to the BGI files on your system.
102 //
103 void InitGraphics(const char *pBGIPath)
104 {
105   //      Request auto detection of graphics driver.
106   int graphDriver = DETECT;
107   int graphMode;
108   detectgraph(&graphDriver, &graphMode);
109
110   //  Initialize graphics system.
```

continues

LISTING 9.7. CONTINUED

```
111    //  Look for files in specified directory.
112    initgraph(&graphDriver, &graphMode, pBGIPath);
113
114    int graphError = graphresult();
115    if(graphError != grOk) {
116            cout << "Graphics error: "
117                 << grapherrormsg(graphError) << '\n';
118            exit(1);
119    }
120  }
121
122  //
123  //      Display a prompt and wait for a keypress.
124  //
125  void PromptAndWait(const char *pText, int textColor)
126  {
127    settextjustify(LEFT_TEXT, BOTTOM_TEXT);
128    settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
129    setcolor(textColor);
130    outtextxy(10, getmaxy()-10, pText);
131    getch();
132  }
133
134  void DoDemo()
135  {
136    settextstyle(GOTHIC_FONT, HORIZ_DIR, 6);
137    outtextxy(10, 10, "Gothic font");
138
139    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 6);
140    outtextxy(10, 110, "Triplex font");
141
142    settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 6);
143    outtextxy(10, 210, "Sans Serif font");
144
145    settextstyle(SMALL_FONT, HORIZ_DIR, 6);
146    outtextxy(10, 310, "Small font");
147  }
148
149  void main()
150  {
151    //
152    //  Register the drivers and fonts that we linked into
153    //  the EXE.
154    //
155    RegisterDrivers();
156    RegisterFonts();
157
```

```
158    //
159    //  Initialize the graphics system.
160    //
161    //  Note that by supplying an empty path string we're
162    //  insuring that it won't use external drivers and
163    //  fonts.
164    //
165    InitGraphics("");
166
167    DoDemo();
168
169    PromptAndWait("Hit any key to continue...");
170    closegraph();
171  }
```

In the main body of your program, before calling `initgraph()`, the program must register each of the drivers and fonts. `registerfarbgidriver` uses the address of the external routine as its parameter, and sets up the graphics system to use the linked driver file, rather than loading the driver from a disk file. `registerfarbgidriver` returns a negative value if an error occurs during the registration process. A positive return value is the internal driver number assigned by the system and can be ignored by your program.

Similar statements are added to call `registerfarbgifont`. That's all there is to linking the .bgi and .chr files to your .exe application.

CHAPTER 10

AUDIO OUTPUT AND SOUND SUPPORT UNDER DOS

BY BRIAN HERRING

BORED OF THE BEEP

Sound output without special hardware under DOS

Sound output from PC audio cards under DOS

"If a computer beeps in the forest and nobody is around to hear it, does it make a sound?" The answer to this not-so-old riddle is a resounding "Who cares?" Face it, beeps are boring. Yet, this trivial sound is all you hear from most DOS programs. Why? Unfortunately, the software and hardware that generate sound in an off-the-shelf

PC are extremely limited. You might invest in one of several add-on sound cards now available. You can program these cards to produce terrific sound output from your applications, but your applications will then require that that card be installed on each user's machine. To make matters worse, manufacturers charge extra for the information and tools you need to use their sound cards within your code.

How does that leave the vast majority of DOS programmers who are tired of the "beep" but are without a sound card? Not as helpless as you might think. With some clever programming techniques and the application of digital audio theory, your PC can produce sound effects, melodies—even polyphonic music. Of course, the sound produced by the tiny speaker in your PC won't be the greatest, but with the techniques presented in the chapter, you can go "beyond the beep!"

This chapter discusses techniques for generating sound effects and music under DOS without using special hardware. Sound cards also are discussed at the end of the chapter.

THE PHYSICS OF SOUND

Before reading about digital audio reproduction on the PC, you may need a bit of analog sound theory. This section gives you the minimum knowledge required to understand the topics that follow.

Sound is the energy emitted from a vibrating source through a medium, such as air. This energy, which is in the form of moving pressure variances, pushes and pulls on everything in its path.

These pressure variances cause the eardrum to vibrate like the original source of the sound. For the sound to be heard, the pressure variance energy must be translated into electrochemical impulses inside the brain.

A device that translates one form of energy into another is a *transducer*. One common man-made transducer is the microphone. The human ear actually consists of two transducers: the eardrum and the inner ear. The eardrum translates pressure variances into motion, or *kinetic energy*. The inner ear translates this kinetic energy into electrochemical brain impulses. Finally, these impulses are processed by the brain and perceived as sound.

Each unit of vibration (a complete push-pull on the eardrum) is a *cycle*. The number of cycles per unit time is the *frequency*. Frequency is measured in units of *hertz*, or cycles per second. One hertz (Hz) is equal to one cycle per second. The frequency range of human hearing is about 20 Hz to 20,000 Hz. The maximum displacement of the eardrum per cycle is the *amplitude* of the sound. Amplitude is perceived as volume: the greater the amplitude, the louder the sound. Displacement of the eardrum at any discrete instant in time is the *instantaneous amplitude* of the sound.

The simplest of all sounds is called a *pure tone*. The frequency and amplitude of a pure tone do not vary. (Of course, the instantaneous amplitude varies; otherwise there would be no vibration at all.) A pure tone is heard as a single, clear note that does not vary in pitch or volume. Such sounds are rare in nature. Even something as simple as a vibrating string produces a much richer and more complex sound than a simple tone. These complex vibrations are called *compound waveforms* because they actually consist of two or more simple waveforms added together (see Figure 10.1).

COMPUTERS AND SOUND: PRINCIPLES OF DIGITAL AUDIO

This section provides a brief introduction to digital audio concepts. Read this section to understand the code (found later in this chapter) that generates and plays digital audio data.

DIGITAL RECORDING

Converting sound energy into a stream of digital data is conceptually simple. The computer is connected to an external device called an *analog-to-digital converter* (a-to-d converter), which in turn is connected to an input transducer (a microphone), as shown in Figure 10.2.

The microphone converts sound energy into electrical energy, or *voltage*. The voltage at any moment in time is proportional to the instantaneous amplitude of the sound wave at that moment. The a-to-d converter changes these amplitude values into numbers. The computer records these numbers at discrete intervals of time.

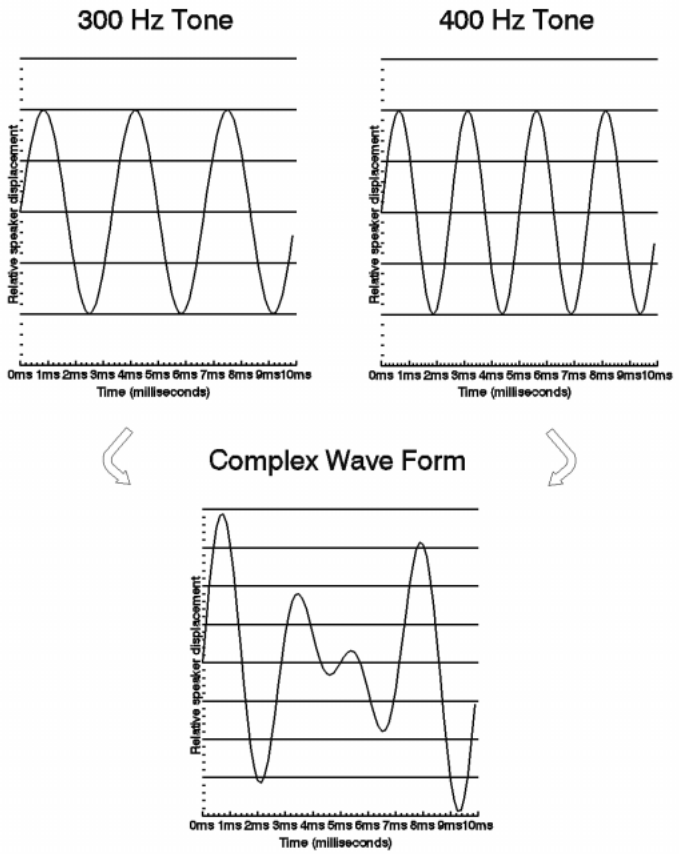


Figure 10.1. A complex waveform is two or more simple waveforms added together.

Each number stored by the computer is a *sample*. The number of samples recorded per unit time is the *sampling frequency*. The greater the sampling frequency, the more accurate the sound reproduction, and the more memory is used per second of recording. The unit Hertz quantifies the sampling frequency. One Hertz is equal to one sample per second.

Sound frequency (cycles per second) and sampling frequency (samples per second) are easy to confuse. Sound frequency is perceived as pitch, but sampling frequency is perceived only indirectly as sound quality.

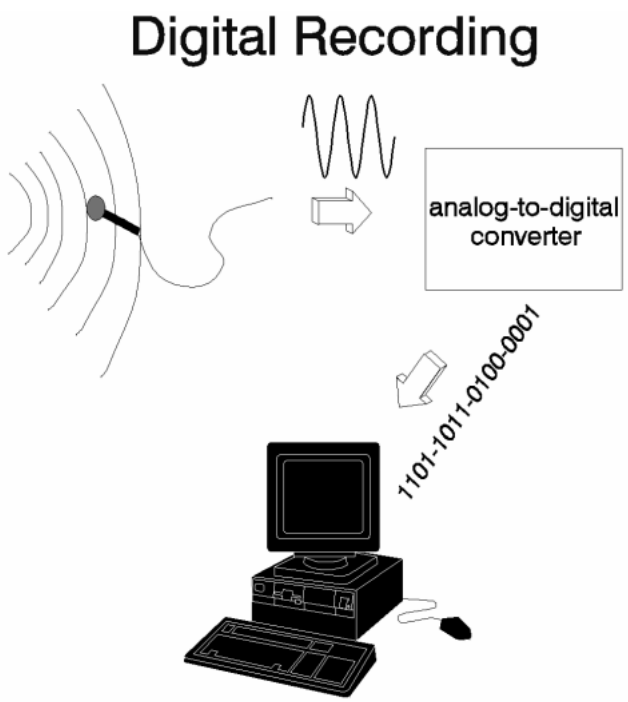


Figure 10.2. Sound energy is converted to digital data via an analog-to-digital converter.

Since the PC does not come with a microphone input jack or an a-to-d converter, you might wonder how to record digital audio without special hardware. Well, you can't. But don't worry—there are other ways to generate digital recordings besides sampling them from the real world. The technique used in this chapter to generate digital sound is *compound waveform generation*. This technique produces a stream of digital sound data for output to the PC speaker.

DIGITAL PLAYBACK

To play back recorded digital sound, you simply reverse the recording process (see Figure 10.3). The PC is connected to an external device, a digital-to-analog converter (d-to-a converter), which converts numbers into electrical energy (voltage). The voltage from this converter is connected to an output transducer (a speaker) which converts the voltage into sound vibrations. The complete process of digital recording and playback is illustrated in Figure 10.4.

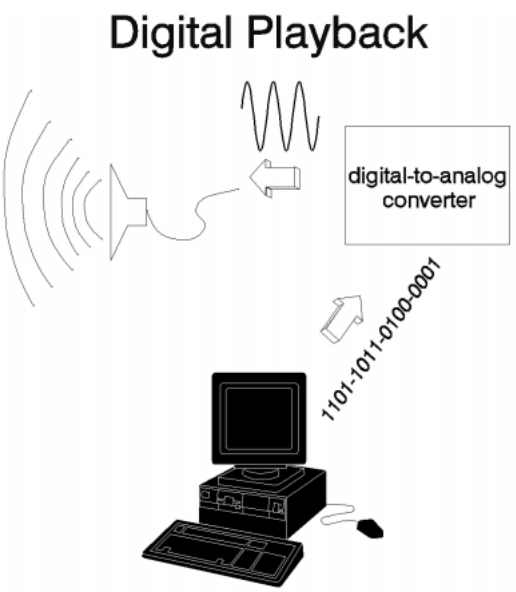


Figure 10.3. Digital data is converted into sound via a digital-to-analog converter.

All PCs come with a speaker, but not a d-to-a converter. Instead, DOS provides direct access to the PC speaker. Later in this chapter, a software d-to-a converter is developed using direct speaker manipulation techniques. The section "Playing PCM Data" covers this topic.

PULSE CODE MODULATION

Pulse Code Modulation, or *PCM*, describes the digital technique for recording and storing sound or other information usually transmitted by waves. *Pulse* means each sample in the digital data stream represents a periodic instantaneous sound amplitude. *Code* means that the sequence of pulses conveys information—in this case, audio information. The word *modulation* simply describes any process that encodes information for storage or transmission. Techniques for generating and playing PCM data on the PC are discussed in the section "Producing Music."

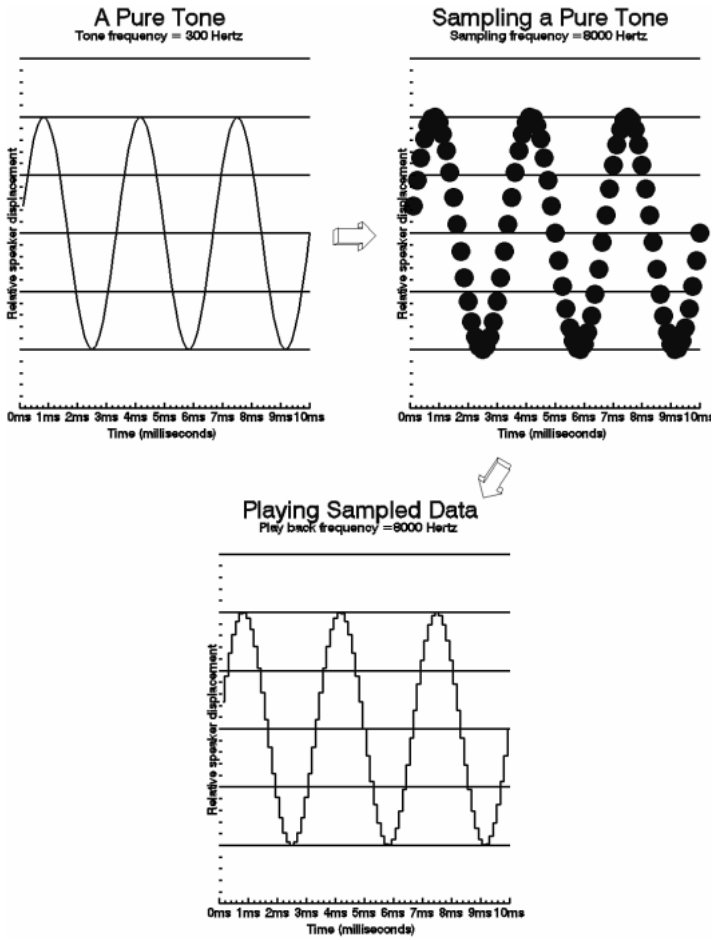


Figure 10.4. "Doing the wave": From the original analog signal, to digital samples, to digital output.

SOUND OUTPUT WITHOUT SPECIAL HARDWARE UNDER DOS

This section explores sound output using an off-the-shelf PC with no special hardware. Functions are developed to produce tones, sound effects, notes, melodies, chords, and songs. Both direct and indirect speaker manipulation techniques are presented. This section also develops the tools for generating and playing back PCM files.

THE PC SPEAKER: "WHAT MAKES IT BEEP?"

All speakers vibrate in response to the voltage variances from the sound sources to which they are connected. The voltage from an analog device like an AM radio varies continuously within a specific range in response to radio signals.

The PC speaker is connected to hardware capable of sending only two voltage values: high and low. You don't need to know these voltage values; rather, know that the low-voltage state corresponds to pulling the speaker cone the maximum distance towards the speaker magnet and that the high state corresponds to pushing the speaker cone the maximum distance away from the speaker magnet.

Imagine you are driving a car with only two throttle positions: idle and floored. Driving that car is challenging, to say the least. To maintain a steady speed between the two extreme possibilities requires you to switch between the two throttle positions at such a pace that the average throttle position is the one desired.

This scenario, though tedious and awkward for a human, is well suited to a computer. In fact, driving the speaker at a specific frequency is easy for a computer. To drive the speaker at a specific frequency, you set up a loop that transitions the speaker from the low-to-high-to-low states at the same rate as the frequency you want to output. For instance, to produce a sound at 440 hz, you transition the speaker from low to high to low again, 440 times per second. Varying the volume (amplitude) of the speaker output is more difficult.

To vary the amplitude of the sound on the PC speaker, apply the same principle used to drive your car with the two-state throttle. By transitioning the speaker between states very rapidly, you produce effective speaker cone

displacements between the two extremes. The frequencies used to achieve these average speaker displacements must be higher than the range of human hearing, or they are heard as noise in the speaker output. Fortunately, the PC speaker can be oscillated well beyond the range of human hearing to effectively produce a range of instantaneous amplitudes.

DIRECT SPEAKER MANIPULATION

The PC provides direct access to the hardware that controls the internal speaker through *I/O ports*. I/O is short for *input/output*. The word *port* indicates a gateway between hardware and software. The PC uses I/O ports to communicate with and control devices such as the keyboard, a printer, and the internal speaker.

You address I/O ports in the range 0 to 0xFFFF. Each number in this range specifies a unique I/O port. The I/O port that controls the speaker is port 0x61.

Each I/O port is eight bits wide. Often, you can use the bits within a single I/O port to control multiple hardware devices. For port 0x61, you are interested only in the two lowest-order bits for speaker manipulation. Bit zero controls whether the speaker vibrates in response to a special timer, which is discussed later in this section. Bit one controls the current voltage state of the speaker, either high or low.

Alternating the speaker voltage bit of I/O port 0x61 is the most direct way to vibrate the speaker. The following function oscillates the speaker at a steady frequency (note the technique for not disturbing other bits of port 0x61 that you don't care about):

```
 1  void PlaySound1( void )
 2  /*
 3    This function pulls the PC speaker in & out as fast
 4    as possible.
 5    The frequency will be higher on faster machines.
 6
 7    This function is on the companion disk in source file:
 8    EXAMPLES.CPP.
 9  */
10  {
11    #define DELAY_COUNT 1000
12
13    // Holder for current port value.
```

```
14    // Used so we don't disturb other
15    // bits.
16    unsigned char portByte;
17
18    // Reset speaker...
19    nosound();
20
21    // Read current value of port 0x61
22    portByte = inp(0x61);
23
24    // Turn off bits 0 and 1 of portByte
25    portByte &= 0xFC;
26
27    // Pull speaker in & out
28    // as fast as possible.
29    for (long i = 0; i < 5000; i++)
30    {
31    // Pull speaker in by
32    // sending portByte to port 0x61,
33    // with bit 1 turned off.
34    outp(0x61, portByte);
35
36    // Delay a bit, otherwise
37    // no sound is produced
38    // on fast machines...
39    for (short j = 1; j < DELAY_COUNT; j++);
40
41    // Push speaker out by
42    // sending portByte to port 0x61, with
43    // bit 1 turned on.
44    outp(0x61, (portByte ¦ 2));
45    }
46  }
```

This code produces a frequency dependent on the CPU speed of the particular computer being used. An 80486/50 MHz machine running this code produces a much higher frequency tone than an old 8086 machine does. This code is an interesting and quick qualitative benchmark program to run on different machines.

Obviously, this code is not sufficient for producing tones at a specific absolute frequency. To do that, you must measure the relative speed of the computer prior to executing the loop and then insert a delay between iterations to produce any frequency lower than the maximum possible on that machine. Fortunately, there is a better way.

The PC comes with programmable timer hardware that can be controlled from software. This hardware consists of three timers operating on separate channels. Timer 0 is the *system clock timer*, which the PC normally uses to keep track of the time by generating an interrupt 18.2 times per second. An *interrupt* is a call to a special function, an *interrupt handler*, that performs a specific task. Interrupts are aptly named because they halt the normal flow of execution in your program while the interrupt is processed. After the interrupt is processed, control is returned to your program, right where it left off. When Timer 0 goes off, it activates a specific interrupt handler that increments the *system clock counter*, which is used by the PC to keep track of the time. You can reprogram and use Timer 0 for other purposes, but doing that disrupts the computer's sense of time. You need to reprogram Timer 0 to produce sound with variable amplitudes.

The next timer, timer 1, is off-limits to programmers because it is used by the PC to refresh random access memory.

By design, the last timer, Timer 2, manipulates the PC speaker. At each timer firing, the speaker state changes from low to high to low again. Timer 2 is capable of firing 1,193,180 times per second. If programmed to fire that often, the speaker produces a vibration at a frequency of 1,193,180 Hz, which you do not perceive as sound. In fact, the speaker cannot move that fast; it would be effectively locked in the "high" state.

All three timers are capable of firing (generating an interrupt) at a maximum rate of 1,193,180 times per second. To vary the firing rate, each timer has an associated countdown value. Each time the clock ticks, the countdown value is decremented. When the countdown value of Timer 2 reaches zero, a special interrupt handler is called, and then the countdown value is reset to the value you last set it to. In this way, the timer continues to fire at the same rate until a new countdown value is set.

The frequency at which the timer fires is reciprocal to the countdown value; that is, Timer 2 fires at the rate of 1,193,180 divided by the countdown value. The countdown value is a 16-bit value; thus, its range is 0 to 65,535. Therefore, Timer 2 is capable of driving the PC speaker in the range 1,193,180 Hz (where the countdown value is zero) to 18.2 Hz (where the countdown value is 65,535).

Using Timer 2 is desirable for several reasons. First, it doesn't matter how fast your CPU is. The timer chip acts completely independently of the CPU, and the timer chip on the slowest PC is as fast as the timer chip on the most powerful model, which means you can program the speaker to vibrate at any frequency between 18.2 Hz (the minimum rate at which the timer chip fires) and 1,193,180 Hz. Timer 2 provides plenty of frequencies above the range of human hearing that you can use to generate any effective instantaneous amplitude (see the section "The PC Speaker: 'What Makes it Beep?'").

As a final fringe benefit, using Timer 2 to oscillate the speaker leaves the CPU free to continue executing your code, so you can produce simple tones at any frequency, without halting the execution of the rest of your program.

You access Timer 2 through two I/O ports: 0x42 and 0x43. Also, Port 0x61 enables and disables the connection between Timer 2 and the speaker; otherwise, the speaker vibrates constantly, since the timer always fires at some rate between its minimum and maximum. The following function vibrates the PC speaker at 44O Hz using Timer 2:

```
 1  void PlaySound2( void )
 2  /*
 3    This function uses system timer 2 to oscillate the PC
 4    speaker at a specific frequency. Speaker will continue
 5    to oscillate at the same frequency until speaker is
 6    disconnected from timer 2, or until timer 2 is reset to
 7    a different countdown value.
 8
 9    This function is on the companion disk in source file:
10    EXAMPLES.CPP.
11  */
12  {
13    // Define frequency and base clock frequency.
14    #define FREQUENCY_2 440
15    #define TIMER_TICKS_PER_SECOND 1193180
16
17    // Holder for current port value.
18    // Used so we don't disturb other bits.
19    unsigned char portByte;
20
21    // Store timer 2 countdown value.
22    unsigned short count_down =
23      TIMER_TICKS_PER_SECOND / FREQUENCY_2;
24
25    //Reset speaker...
26    nosound();
```

```
27
28    // Tell timer 2 that a two
29    // byte count is coming.
30    outp(0x43, 0xB6);
31
32    // Send low count-down byte to timer 2.
33    outp(0x42, LO_BYTE(count_down));
34
35    // Send high count-down byte to timer 2.
36    outp(0x42, HI_BYTE(count_down));
37
38    // Read current value of port 0x61
39    portByte = inp(0x61);
40
41    // Turn on bits 0 & 1.
42    portByte |= 3;
43
44    // Connect timer 2 to speaker,
45    // Start the sound!
46    outp(0x61, portByte);
47  }
```

HIGH-LEVEL SPEAKER MANIPULATION

Using I/O ports is a powerful technique for speaker manipulation. The port technique gives you direct control over all aspects of sound generation. If all you need to do is produce a series of simple tones, the direct technique is a bit tedious. Fortunately, DOS provides three high-level functions that enable your code to produce simple tones, regardless of whether you know anything about system timers or I/O ports. These functions are `sound`, `delay`, and `nosound`.

The `sound` function takes an integer argument that specifies the frequency to be output. The frequency this function produces is limited to the range of positive integer values 0 to 37,367, a practical range for producing any simple tone within the range of human hearing. The sound function does the work of calculating and setting the Timer 2 countdown value and enabling the connection between Timer 2 and speaker.

The `delay` function is a general-purpose routine that causes the execution of your program to pause for a specific period of time. The `delay` function takes an unsigned integer argument that specifies the number of milliseconds to pause. This function enables your code to hold a simple tone for an exact duration, without you knowing the execution speed of the CPU on which your code is

running. Even though the execution of your code is halted for the duration of the delay, the CPU is not idle. The CPU still can process interrupts or continue to execute code in other resident programs such as print spoolers.

The `nosound` function takes no arguments; it simply disables the connection between Timer 2 and the speaker, stopping the sound. The following code duplicates the functionality of Listing 10.2 using the `sound` function calls:

```
 1  void PlaySound3( void )
 2  /*
 3    This function uses system timer 2 to oscillate the PC
 4    speaker at a specific frequency. Speaker will continue
 5    to oscillate at the same frequency until speaker is
 6    disconnected from timer 2, or until timer 2 is reset to
 7    a different countdown value.
 8
 9    This function is on the companion disk in source file:
10    EXAMPLES.CPP.
11  */
12  {
13    // Define note frequency.
14    #define FREQUENCY_3 440
15
16    //Reset speaker...
17    nosound();
18
19    // Program timer 2, enable
20    // connection to speaker.
21    sound(FREQUENCY_3);
22  }
```

BEYOND THE BEEP

Having mastered the beep, it's time to explore the possibilities of producing richer sounds using the high-level speaker functions.

THE *PLAYTONE* FUNCTION

You can use the functions `sound`, `delay`, and `nosound` to write a function that takes a frequency and duration as arguments and plays one simple tone. The `playtone` function is the first building block on the way to producing sound effects and music.

```
 1  void PlayTone( short frequency, short number_of_milliseconds )
 2  /*
 3    This function uses system timer 2 to oscillate the PC
 4    speaker at the given frequency for the given duration.
 5
 6    This function is on the companion disk in source file:
 7    EXAMPLES.CPP.
 8  */
 9  {
10    // Reset speaker...
11    nosound();
12
13    // Start oscillating at frequency.
14    sound(frequency);
15
16    // Wait for the duration of the tone
17    delay(number_of_milliseconds);
18
19    // Silence the speaker
20    nosound();
21  }
```

SOUND EFFECTS

By placing a call to the function inside a loop and varying the frequency at each call, you now can create several sound effect functions. The following code demonstrates how to produce a simple sound effect:

```
 1  void PlaySiren( short repeat )
 2  /*
 3    Plays an up/down siren for the given number of repetitions.
 4    Will rise & fall more quickly on faster machines.
 5
 6    This function is on the companion disk in source file:
 7    EXAMPLES.CPP.
 8  */
 9  {
10    // Define low and high siren limits.
11    #define FRQ_MIN 440
12    #define FRQ_MAX 3000
13
14    // Define time to delay between iterations.
15    #define DELAY_MS 1
16
17    // Reset speaker...
18    nosound();
19
20    // Start repeat loop.
```

```
21    for (short loops = repeat; loops > 0; loops--)
22    {
23      // Rising frequency loop.
24      for (short up_frq = FRQ_MIN; up_frq < FRQ_MAX; up_frq++)
25      {
26        sound(up_frq);
27        delay(DELAY_MS);
28      }
29
30     // Falling frequency  loop.
31     for (short down_frq = FRQ_MAX; down_frq > FRQ_MIN; down_frq--)
32      {
33        sound(down_frq);
34        delay(DELAY_MS);
35      }
36    }
37    // Shhhhh!
38    nosound();
39  }
```

There are other sound effects functions on the companion disk in the effects.cpp file.

PRODUCING MUSIC

One of the most rewarding uses of the techniques in the chapter is the output of music from the PC speaker. This section describes how to produce melody and harmony using the `PlayTone` function as the basis of all sound output. A set of classes for adding music to your applications is presented in this section. Music theory is beyond the scope of this chapter. Some musical terms are used without being explained. Don't worry; you don't need to know anything about music to enjoy it, and the companion disk contains all the code and an example song.

NOTES

A *musical note* is a single tone of a specific frequency held for a specific duration. The `PlayTone` function provides this functionality, but requires that you know the frequency of each note to be played. Figure 10.5 lists note frequencies for 12 consecutive notes (one *octave*), starting with middle C.

Note Frequencies

Starting with middle C

Note	Frequency
C	262
C#	277
D	294
D#	311
E	330
F	349
F#	370
G	392
G#	415
A	440
A#	466
B	494

One octave

Figure 10.5. The frequencies of the notes in one octave, starting with middle C.

To distinguish the notes in different octaves, a number designates each octave register. The note middle C is in octave register four, so its designation is C4.

PLAYING A MAJOR SCALE

The C major scale consists of the notes C, D, E, F, G, A, and B. To play this scale, you just make seven calls to the `PlayTone` function, using the frequencies in Figure 10.5. The following code plays the notes of the major scale, starting with C4 and ending with C5 (actually eight notes):

```
 1  void PlayMajorScale( void )
 2  /*
 3    This function plays one octave of the major scale using
 4    the PlayTone() function.
 5
 6    This function is on the companion disk in source file:
 7    EXAMPLES.CPP.
 8  */
 9  {
10    // Define the frequency values for the middle C octave.
11    #define NOTE_C4 262
12    #define NOTE_D4 293
13    #define NOTE_E4 330
14    #define NOTE_F4 349
15    #define NOTE_G4 392
16    #define NOTE_A4 440
```

```
17    #define NOTE_B4 494
18    #define NOTE_C5 523
19
20    // Play each note with a 1/2 second duration.
21    PlayTone(NOTE_C4, 500);
22    PlayTone(NOTE_D4, 500);
23    PlayTone(NOTE_E4, 500);
24    PlayTone(NOTE_F4, 500);
25    PlayTone(NOTE_G4, 500);
26    PlayTone(NOTE_A4, 500);
27    PlayTone(NOTE_B4, 500);
28    PlayTone(NOTE_C5, 500);
29
30    // Shhhhh.
31    nosound();
32  }
```

NOTE STRINGS

Looking up the frequency in Figure 10.5 for each note you want to play is quite tedious. Also, Figure 10.5 only lists a small number of possible notes. What is needed is a function that converts a *note string* (`A`, `B`, `C`, and so on) into a frequency value. Your note strings will consist of three parts: an uppercase note letter (`A`, `B`, `C`), followed by an optional accidental (`#` or `b`), and then an octave register (0–9). `Notestringtofreqency` is provided on the companion disk in the examples.cpp file.

Here is a new function, `PlayNote`, which uses NoteStringToFrequency to play a note from its note string:

```
 1  void PlayNote( char *note_str, short duration_ms )
 2  /*
 3    Converts <note_str> to a frequency, then plays a tone
 4    of that frequency for <duration_ms> milliseconds.
 5
 6    This function is on the companion disk in source file:
 7    EXAMPLES.CPP.
 8  */
 9  {
10    // Convert note string to frequency.
11    short note_frq = NoteStringToFrequency(note_str);
12
13    // Play the note!
14    PlayTone(note_frq, duration_ms);
15  }
```

CHORDS

In music, *chords* consist of two or more notes played simultaneously. This idea seems simple, but playing chords presents some new challenges. First, you cannot play two notes simultaneously using the tools developed so far. Later, this chapter develops code that plays notes simultaneously, but one technique for playing chords works with the tools you have now. The idea is to cycle through the notes in the chord quickly so that the ear perceives the notes together. This works fairly well and is easy to implement.

The following code plays a C Major chord consisting of the notes C4, E4, and G4 for a duration of two seconds:

```
 1  void PlayCMajorChord( void )
 2  /*
 3    Plays a C Major chord for a duration of about two
 4    seconds.
 5
 6    This function is on the companion disk in source file:
 7    EXAMPLES.CPP.
 8  */
 9  {
10    // Define number of milliseconds between each chord note,
11    // for each chord cycle, based on a 3 note chord.
12    #define MS_BETWEEN_NOTES 40
13
14    // Define total number of milliseconds to play this chord.
15    #define DURATION_MS 2000
16
17    // Calculate the total number of chord cycles.
18    short num_cycles = DURATION_MS / (MS_BETWEEN_NOTES * 3);
19
20    for (short repeat = 0; repeat < num_cycles; repeat++)
21    {
22      PlayNote("C4", MS_BETWEEN_NOTES);
23      PlayNote("E4", MS_BETWEEN_NOTES);
24      PlayNote("G4", MS_BETWEEN_NOTES);
25    }
26
27    // Silence the speaker.
28    nosound();
29  }
```

At this point, the code for playing music is sufficiently messy to move to a higher level. Chords are much easier to deal with as a class. Here is a definition for class `Chord`:

```
 1  class Chord
 2  {
 3
 4  private:
 5    short note_array[MAX_NOTES+1];
 6    short duration;
 7    short num_notes;
 8    short beats_per_minute;
 9
10    void  PlaySquare( short beats_per_minute );
11
12  public:
13    Chord ( short number_of_16ths, short beats_per_minute );
14    ~Chord ( void );
15
16    void AddNote( char* note_str );
17    void AddNote( short frq );
18
19    void Play( short beats_per_minute = 60 );
20
21    //Inline access methods
22    short GetNote( short index ) {
23      return this->note_array[index];
24    }
25
26    short GetDuration( void ) {
27      return this->duration;
28    }
29
30    short GetNumNotes( void ) {
31    return this->num_notes;
32    }
33  };
```

The constructor takes two parameters that determine the duration of the chord. The unit of duration for chords is the sixteenth note. The beat unit is the quarter note. To play a chord for one second, you set the beats per minute to 60 and the number of sixteenth notes to four. Add notes to chords one at a time using either `Add Note` method. Add notes by frequency or by note string. The `Play` method outputs the chord to the PC speaker.

The implementation of class `Chord` is on the companion disk, in the file MUSIC.CPP.

SONGS

Songs (for our purposes) consist of multiple chords, played one after the other. Here is the code that plays the first three chords of "America":

```
 1  void ASliceOfAmerica( void )
 2  /*
 3    Plays first three chords of "America", using class Chord.
 4
 5    This function is on the companion disk in source file:
 6    EXAMPLES.CPP.
 7  */
 8  {
 9    // Define number of 16th notes per chord.
10    #define NOTES_PER_CHORD 4
11
12    // Define beats per minute.
13    #define TEMPO 60
14
15    // Build the chords.
16
17    Chord* firstChord  = new Chord(NOTES_PER_CHORD, TEMPO);
18    firstChord->AddNote("G4");
19    firstChord->AddNote("B4");
20    firstChord->AddNote("G5");
21
22    Chord* secondChord = new Chord(NOTES_PER_CHORD, TEMPO);
23    secondChord->AddNote("E4");
24    secondChord->AddNote("B4");
25    secondChord->AddNote("G5");
26
27    Chord* thirdChord  = new Chord(NOTES_PER_CHORD, TEMPO);
28    thirdChord->AddNote("C4");
29    thirdChord->AddNote("E5");
30    thirdChord->AddNote("A5");
31
32    // My Country...
33    firstChord->Play();
34    secondChord->Play();
35    thirdChord->Play();
36
37    delete firstChord;
38    delete secondChord;
39    delete thirdChord;
40  }
```

You can now take the final step by creating class `Song`, which stores and manipulates `Chords`. Here is the definition of class `Song`:

```
1  //max # chords per song
2  #define MAX_CHORDS 100
3
4  class Song
5  {
6
7  private:
8
9    short   beats_per_measure;
10   short   beats_per_minute;
11   short   num_chords;
12   Chord*  chord_array[MAX_CHORDS];
13
14 public:
15
16   Song( short beats_per_minute = 60, short beats_per_measure = 1 );
17
18   ~Song( void );
19
20   void AddChord( short number_of_16ths, char* note_str1, char* note_str2 = 0,
21     char* note_str3 = 0, char* note_str4 = 0, char* note_str5 = 0 );
22
23   void AddChord( short number_of_16ths, short frq1, short frq2 = 0,
24     short frq3 = 0, short frq4 = 0, short frq5 = 0 );
25
26   void Play( void );
27
28 };
```

Songs are now much easier to generate. Add chords with a single call, instead of one call per chord note. Use the title parameter in the constructor only if the song is output as a PCM file. The section "Generating PCM Data" covers this topic.

The implementation of class `Song` is on the companion disk.

Here's the code to play the entire song "America," using class `Song`:

```
1  void PlayAmerica( void )
2  /*
3    Plays the entire song "America",
4    using class Song.
5  */
6  {
7    // America, two, three & four voice harmony
8    Song* mySong = new Song(90, 3, "AMERICA");
9
10   /* measure 1 */
11   mySong->AddChord(4, "G4", "B4", "G5");
```

```
12    mySong->AddChord(4, "E4", "B4", "G5");
13    mySong->AddChord(4, "C4", "E5", "A5");
14
15    /* measure 2 */
16    mySong->AddChord(6, "D4", "D5", "F#5");
17    mySong->AddChord(2, "D4", "E5", "G5");
18    mySong->AddChord(4, "D4", "F#5","A5");
19
20    /* measure 3 */
21    mySong->AddChord(4, "G4", "D5", "B5");
22    mySong->AddChord(4, "E4", "G5", "B5");
23    mySong->AddChord(4, "C4", "E5", "C6");
24
25    /* measure 4 */
26    mySong->AddChord(6, "D4", "D5", "B5");
27    mySong->AddChord(2, "D#4","C5", "A5");
28    mySong->AddChord(4, "E4", "B4", "G5");
29
30    /* measure 5 */
31    mySong->AddChord(4, "C4", "C5", "A5");
32    mySong->AddChord(4, "D4", "B4", "G5");
33    mySong->AddChord(4, "D4", "C5", "F#5");
34
35    /* measure 6 */
36    mySong->AddChord(12, "G3", "B4", "G5");
37
38    /* measure 7 */
39    mySong->AddChord(4, "G4", "B5", "D6");
40    mySong->AddChord(4, "B4", "B5", "D6");
41    mySong->AddChord(4, "D5", "B5", "D6");
42
43    /* measure 8 */
44    mySong->AddChord(6, "G4", "B5", "D6");
45    mySong->AddChord(2, "G4", "A5", "C6");
46    mySong->AddChord(4, "G4", "G5", "B5");
47
48    /* measure 9 */
49    mySong->AddChord(4, "D4", "A5", "C6");
50    mySong->AddChord(4, "F#4", "A5", "C6");
51    mySong->AddChord(4, "A4", "A5", "C6");
52
53    /* measure 10 */
54    mySong->AddChord(6, "D4", "A5", "C6");
55    mySong->AddChord(2, "D4", "G5", "B5");
56    mySong->AddChord(4, "D4", "F#5", "A5");
57
58    /* measure 11 */
59    mySong->AddChord(4, "G4", "G5", "B5");
60    mySong->AddChord(2, "G4", "C6");
```

```
61    mySong->AddChord(2, "G4", "B5");
62    mySong->AddChord(2, "G4", "A5");
63    mySong->AddChord(2, "G4", "G5");
64
65    /* measure 12 */
66    mySong->AddChord(6, "G4", "G5", "B5");
67    mySong->AddChord(2, "A4", "G5", "C6");
68    mySong->AddChord(4, "B4", "G5", "D6");
69
70    /* measure 13 */
71    mySong->AddChord(2, "C5", "G5", "E6");
72    mySong->AddChord(2, "C5", "G5", "C6");
73    mySong->AddChord(4, "D5", "G5", "B5");
74    mySong->AddChord(4, "D4", "C5", "F#5", "A5");
75
76    /* measure 14 */
77    mySong->AddChord(12, "G4", "B4", "G5");
78
79    // Play it!
80    mySong->Play();
81    delete mySong;
82
```

PRODUCING POLYPHONIC MUSIC

This section develops code that generates and plays PCM data from the chords stored in a Song object. Using this technique, the PC plays chords in true *polyphony;* that is, the chord notes sound simultaneously. You achieve this by applying the techniques in the section "Computers and Sound: Principles of Digital Audio."

GENERATING PCM DATA

The first step when making polyphonic music is to generate the PCM data representing the complex waveform of multiple notes of arbitrary frequencies sounding simultaneously. The task becomes simpler when you realize that complex waveforms consist of one or more simple waveforms added together. The process involves producing an instantaneous amplitude for each sample in the PCM stream. First, examine generating the data for a single waveform.

Using the `cosine` and `exp` functions in conjunction with an iterative algorithm that can generate the instantaneous amplitude of the wave for any moment in time produces the waveform of a damped harmonic oscillator (like a vibrating

tuning fork). The oscillator functions in this section simulate the vibration of a tuning fork. It is beyond the scope of this chapter to explain the mathematics behind the tuning fork simulator. The complete code is on the companion disk in the file PCM.CPP. This section lists and discusses the key sections of the code.

Here are the definitions of class `PcmNote` and `PcmFile`:

```
 1  class PcmNote
 2  {
 3    friend PcmFile;
 4  private:
 5    double
 6        const1,const2,t1,t2,damping;
 7    long
 8        samples_remaining;
 9    short
10        frq;
11  protected:
12    double PcmNote::PcmMaxAmplitude( short frq );
13  public:
14    PcmNote( short frq, long number_of_samples, double damping );
15  };
16
17  class PcmFile
18  {
19  private:
20    long
21        samples_per_chunk,
22        current_chunk_offset;
23    double
24        average, std_dev, min_sample, max_sample,
25        damping;
26
27    SAMPLE_TYPE* data;
28    FILE* file_ptr;
29  protected:
30    char file_name[128];
31
32    long AddNoteToChunk( PcmNote* pcmNote );
33    void ClearAdjust( void );
34    void AdjustData( void );
35    void InitChunk( void );
36    void WriteChunk( void );
37    void PlayChunk( void );
38    void Close(void);
39  public:
40    PcmFile( char* file_name, long samples_per_chunk = SAMP_RATE,
```

```
41          double damping = -1.5 );
42      ~PcmFile( void );
43
44      void AppendChord( Chord* chord );
45      void Play( void );
46  };
```

The job of class `Pcmfile` is to generate a file consisting of PCM data which represent digitized chords. The chords are built one note at a time using the method `AddNoteToChunk`. A *chunk* is a short section of the song being generated. Saving data in chunks is a technique for limiting the amount of memory required for generating and playing back the data. The best chunk size to use is one that matches the number of samples per beat of the song being generated. In this way, the slight delay that occurs when the next chunk is read from disk is less disruptive to the flow of the music. The first two bytes of the generated file contain the chunk size, in samples per chunk. The rest of the file consists of a continuous stream of PCM data.

The complete implementation of class `PcmFile` is on the companion disk, in the file PCM.CPP.

PLAYING PCM DATA

This section develops the code necessary to drive the PC speaker to play the PCM data generated in the last section. This code can be adapted to play any other PCM data for which the sampling rate is known to be 16,000 Hz or less.

Two problems need to be solved to play PCM data over the PC speaker. The first problem is reproducing the exact sampling frequency at which the data was recorded. This frequency is necessary so the speaker can be set to produce a different instantaneous amplitude for each sample in the data stream at precisely the right moment. The second problem is reproducing the instantaneous amplitude (speaker displacement) for each sample in the data stream.

To solve the first problem, Timer 0 (the system clock timer) is reprogrammed to fire at the sampling frequency. The system clock timer interrupt handler is temporarily replaced by a custom interrupt handler, which is called each time the timer fires. This interrupt handler changes the instantaneous amplitude of the speaker to the next value in the PCM data stream and then advances the data stream pointer.

Here is the code that reprograms Timer 0 and sets up your interrupt handler:

```
1  void pcm_speaker_init( SAMPLE_TYPE *data, long number_of_samples )
2  /*
3    Initialize system timers & interrupt handler.
4  */
5  {
6    //Storage for timer value.
7    short lo_timer_byte;
8    short hi_timer_byte;
9
10   // Disable interrupts.
11   pcm_speaker_cli();
12
13   // Save interrupt mask.
14   ps.irq_mask = inp(0x21);
15
16   // Mask all but system timer interrupt.
17   outp(0x21, 0xFE);
18
19   // Save current interrupt handler.
20   ps.save_interrupt = _dos_getvect(8);
21
22   // Turn-off the speaker.
23   outp(0x61, (inp(0x61) & 0xFC));
24
25   // Tell Timer 2 that a count is coming.
26   // Put in the mode to receive LSB only
27   outp(0x43, 0x90);
28
29   // Clear Timer 2.
30   outp(0x42, 0x0);
31
32   // Enable the connection between Timer 2
33   // and the speaker.
34   outp(0x61, (inp(0x61) ¦ 3));
35
36   // Tell Timer 0 that count is coming.
37   outp(0x43, 0x36);
38
39   // Set Timer 0 to fire at SAMP_RATE
40   ps.timer_zero_count = 1193180 / SAMP_RATE;
41   hi_timer_byte = ((ps.timer_zero_count & 0xFF00) >> 8);
42   lo_timer_byte = (ps.timer_zero_count & 0x00FF);
43
44   // Send the timer countdown value to Timer 0.
45   outp(0x40, lo_timer_byte);   // lo-byte
46   outp(0x40, hi_timer_byte);   // hi-byte
47
48   // Hook-up our interrupt handler
```

```
49    _dos_setvect(8, &pcm_speaker_timerInt);
50
51    // Enable interrupts (start sound!)
52    pcm_speaker_sti();
53  }
```

The code at lines 10–20 manipulates software interrupts. At line 11, all software interrupts are disabled. This is necessary because you don't want any interrupts to occur until you finish reprogramming the interrupts and the system timer.

Next, the code reads and saves the current value of I/O port 0x21 in order to later restore the interrupt bits.

Line 17 clears all bits in port 0x21 except for bit 0. This step masks all interrupts except for the system timer interrupt. If you don't do this, other interrupts tie up the CPU, and the digital playback sounds noisy and uneven.

Next, the code saves the current interrupt handler for the system clock timer in order to later restore it.

Lines 25–30 reprogram system Timer 2 to receive countdown values only in the least significant byte, instead of receiving two bytes. Since the samples are only one byte big, you don't need to send the upper byte, which saves time inside your interrupt handler because only one call, instead of two, is made to `outp`. The faster the code within your interrupt handler executes, the clearer the sound reproduces.

Lines 36–46 reprogram Timer 0 to fire at the sampling rate of the sound to be played. Next, line 49 hooks the custom interrupt handler (`pcm_speaker_timer_int`) to Timer 0, and then line 52 reenables interrupts. This starts the sound.

Setting the instantaneous amplitude of the PC speaker is relatively simple. All you do is set Timer 2's countdown value to match the current sample value being played. The samples generated by `PcmFile` range from 1 to 72. Setting Timer 2's countdown to these values displaces the speaker cone at various positions away from the magnet of the speaker. At low PCM values, the cone locks at nearly the closest position to the speaker magnet, corresponding to the low voltage state. At higher PCM values, the speaker cone moves to a position between the low voltage state and the high voltage state. With this technique, you can now effectively reproduce different instantaneous amplitudes.

Here is the interrupt handler code that programs Timer 2 according to the current PCM data value:

```
 1  static void interrupt pcm_speaker_timerInt (...)
 2  /*
 3    Handle Timer 0 interrupts.
 4  */
 5  {
 6    // Set Timer 2 (change the instantaneous amplitude).
 7    // "ps" is the global struct holding the pcm play data.
 8    outp(0x42, ps.cur_sample);
 9
10    // Increment sample counter.
11    ps.sample_number++;
12
13    // Load next frequency value.
14    ps.cur_sample = ps.data_ptr[ps.sample_number];
15
16    // End this interrupt.
17    outp(0x20, 0x20);
18  }
```

The code within the custom interrupt handler is very simple. Line 8 sets the Timer 2 countdown value to the new sample. Lines 11–14 prepare the next sample value. Line 17 tells the PC's software interrupt controller that you are done handling this interrupt.

THE SAMPLE APPLICATION: DASS.EXE

A sample application called DASS.EXE is on the companion disk to this book. DASS is an acronym for Digital Audio and Sound Support. DASS.EXE demonstrates the concepts presented in this chapter.

MENU/DIALOG TOUR

The menus and dialogs in DASS.EXE are organized like this chapter. Starting with simple tones, the menus progress to sound effects, notes, chords, and finally a song. Some dialogs give you the choice of output techniques. *Square* means play the note using the `sound`, `delay`, and `nosound` functions which produce a simple square-wave tone. *PCM* means generate and play PCM data using the

techniques in the section "Producing Polyphonic Music." The PCM technique requires generating and saving PCM data to disk and can take a while. For the Song dialog, the file AMERICA.PCM is generated only if it doesn't already exist.

MODULE TOUR

DASS.EXE has four modules: DASSMAIN, EFFECTS, MUSIC, and PCM. A brief description of each module is given here:

DASSMAIN	User interface code. Uses Borland's TurboVision libraries.
EFFECTS	Sound effects functions. Uses "Square" sound generation technique.
MUSIC	C++ classes for writing and playing music. Uses both the "PCM" and "Square" sound generation techniques.
PCM	Code for generating and playing PCM data over the PC speaker, without special hardware.

SOUND OUTPUT FROM PC AUDIO CARDS UNDER DOS

Several popular sound cards are available for the PC. These cards are powerful tools for producing high-quality sound output.

SOUND CARD FEATURES

Sound cards provide several different flavors of sound generation, including digital recording and playback with built-in data compression and a multi-voice *music synthesizer*, which produces polyphonic music from a high-level description language. Some cards also provide *MIDI* ports, which control the music synthesizer from an external source. The MIDI ports can also control an external synthesizer. Other features include an output amplifier, jacks for external headphones or speakers, an input jack for a microphone, a volume control, and a joystick port.

A sound card offers you several big advantages. The most obvious advantage is that the sound output quality is much greater than output that can be achieved using only the PC hardware. Another advantage is that the CPU is not tied up playing the sounds, because the sound card does most of the work. Applications can continue displaying graphics and processing user input, even during digital audio playback. Programmers also deal with sound output functions at a much higher level. All the sound output code discussed in this chapter can be reduced to a few calls to a sound card driver.

If sound cards are so great, why doesn't everybody have one? It's just like anything else in the PC world: there are several different cards, each with different features and programming interfaces. Microsoft Windows took a big step toward solving the interface problem by providing a high-level interface that acts as a layer between Windows programs and the sound card drivers. Windows users can chose from among several sound cards and know that all their Windows applications using the high-level interface layer work with their card.

DOS developers don't have it so easy. They're stuck using (at extra cost) the drivers and development kits provided by the sound card manufacturers. Sorely needed are third-party software libraries that provide high-level programming interfaces for multiple sound card drivers under DOS. Genus Microprogramming, a company based in Houston, Texas, has developed a library called Genus GX Effects that, among other things, supports reading and playing Sound Blaster's VOC file format through the Sound Blaster card.

RECORDING AND PLAYING SOUND CARD PCM FILES

Most sound cards come with DOS utility programs that let you record digital audio through a microphone and save the PCM data into that card's native digital audio format. Another utility enables you to read back the native format and play the file through the card. It also is possible to extract the PCM data from the native sound card file and, using the techniques discussed in this chapter, play the PCM data on a system that has no sound card. Data compression should not be used when recording the PCM file unless you know how to decompress it at playback time.

Most sound card manufacturers make their internal digital data format public. You should be able to get the file format for your sound card at no cost. It's then a simple matter of programming to read the native sound card file, extracting the PCM data, and writing the file back in any way you wish. Using the code discussed in this chapter, you can play the data on any PC without special hardware!

DEBUGGING TECHNIQUES

In a perfect world, programs would not have defects. Someday programming methodologies will be such that you will rarely find defects in software (see Chapter 7, "Writing Reusable, Robust Classes," for some suggestions that may help you write more reliable code). For now, however, you owe it to those who use your programs to identify and remove as many programming defects as possible. A number of debugging techniques, which are described in this chapter, can help you identify and correct program errors.

Before you can begin removing defects from your software, you must perform *software testing*. Software testing is the discipline of isolating all paths through the software to ensure that each input produces an expected output with no unexpected side effects. Because of the importance of software testing, this topic is described in the first section of this chapter, "Program Testing Strategies."

Program testing strategies

Isolating programming defects

Debugging techniques

Using Turbo Debugger

Turbo Debugger for Windows

When you get to the debugging stage of a project, several debugging techniques are available. The three basic techniques are

- *Adding extra program statements* to your source code, to output results or to check for specific conditions.
- *Using the IDE's built-in debugger* to single-step through your program's execution, at the source line level, examining and changing the values of variables if needed.
- *Using Turbo Debugger*. Turbo Debugger provides everything that is contained in the IDE's built-in debugger, plus access to disassembled machine code, all the processor registers and memory, and a built-in assembler to modify code while the program is executing. Turbo Debugger is essential if your program is too big to be run in the IDE, because Turbo Debugger is capable of executing and debugging programs up to the maximum size allowed by DOS.

This chapter describes each of the debugging tools and includes sections about software testing and debugging strategies. These tips can help you locate some of the most common problems encountered by C and C++ programmers. During the process of testing and debugging your software, keep in mind that there is rarely a defect-free program. One of the amusing aspects of software development occurs when a programmer holds up finishing the software to fix "the last bug." Invariably, another "last bug" is found shortly thereafter. According to some researchers, 5 percent of all program defects in commercially produced software are not discovered until after the program arrives in the customer's hands! This doesn't mean you should produce defect-ridden software. It does mean that you should try harder to produce the highest-quality software, at every step of a project—from concept to design, implementation, and the final test. When defects are found, it's time to begin debugging.

PROGRAM TESTING STRATEGIES

To find and fix a program error, you first must recognize that your program is not doing what it is intended to do. Some program errors are obvious

(1 + 2 = 1,784,342), but in other cases the program errors are more subtle. The only reliable way to discover such problems is to perform rigorous software testing.

The best approach to finding and correcting software defects is to write the software correctly the first time. Realistically, the best you can do is test small sections of code prior to incorporating new code into the total program. This approach is called *unit testing*. By ensuring that individual code sections work correctly, you increase the probability that the entire program will work.

For the purposes of unit testing, a testing unit does not necessarily correspond to a source module or class. A unit, in this context, refers to any segment of code for which it makes sense to begin testing. A testing unit may be an individual function, or it may be an entire module—whatever makes sense in the context of your application and where the program is in its phase of development.

Consider the testing of a single function. To test an individual function, you may write special-purpose test code to call the function and pass to it appropriate parameter values. Check to see that this results in the desired operation. Be certain to test for the following types of conditions:

- Test using typical, normal values that would be expected during program operation.
- Next, test the boundary areas, where parameter values approach the limits of acceptable input to the function. For example, if a function is expected to justify a line of text (by adding blanks to make the line exactly 80 characters long, for instance), check what happens when the procedure is given a blank line, or a line already containing exactly 80 characters.
- Test invalid inputs. What happens if the hypothetical justify procedure receives a line with 128 characters? Does your procedure have a mechanism for detecting possible errors and indicating the result of those errors to the caller?

The goal of unit testing is to stabilize the code before proceeding. Instead of linking half a dozen new functions and testing them all at once, there is a greater likelihood that the group will work correctly if you test each function individually. After the functions are individually tested, you should run appropriate

tests on the group. Your goal is to provide a solid foundation of reliable code before advancing to higher levels of functionality. As you link new code into the set of core routines, any new problems should be related to the new code that you have added (although it's possible that your new code has such side effects as modifying a global value, which in turn effects existing code).

Another testing category uses Turbo Profiler (described in Chapter 12, "Program Optimization and Turbo Profiler") to record an execution history of the program as it runs. A proper set of tests ensures that every statement in a program is executed at least once. If Turbo Profiler reports that some lines are not being executed, this is a sign that the test is inadequate, the program logic is flawed, or the extra statements are superfluous and can be deleted, thereby saving a bit of memory.

CATCHING SOFTWARE DEFECTS BEFORE THEY HAPPEN

To avoid designing defects into the code, software engineers have devised some techniques that help to identify problems before the code even reaches the testing stage. These techniques include design and code walkthroughs, as well as the maintenance of modification histories to help track down problems later.

MODIFICATION HISTORIES

With any sizeable software development, significant changes will be made to the software over the course of the project. If you know when a defect was introduced, you can consult the modification history to determine what changes may have caused the defect to occur. This history can often help in tracking down hard-to-find problems.

Modification histories are not just for projects written by large teams of programmers. Any source code, over time, can be hard to follow, even when the code is your own original code. In addition to keeping adequate records of your work, commercial source version control tracking systems (see Chapter 4, "Version Control Systems") can produce automatic modification histories. Version control systems enable you to track every change made in your source code, and can automatically reproduce the source code as it existed several revisions in the past.

DESIGN REVIEWS

Before starting the coding phase (or, in some instances, in parallel with the start of coding), the overall design should be reviewed. Certainly the design should be given an overall review by the designer, but the best method to snuff out problems early is to have others review the design. When a team is developing a project, this can often be accomplished by having team members review each other's design.

By having each designer give a presentation to the other team members, conflicting assumptions can be identified early. For instance, Designer A may make an assumption about how much memory is available for A's data structures. However, Designer B may have assumed that B's data structures could occupy most of memory, leaving inadequate free memory for the portions of the product to be produced by Designer A. By bringing these issues out in the open prior to coding, you can avoid a lot of grief later.

CODE WALKTHROUGHS

After the code has been written, it should be examined by other team members, much as the design is reviewed. Although code reviews do detect problem areas, they can often result in improved sharing of code and data resources. One programmer may spot, buried in someone's module, a routine that she has written for one of her own modules. By finding these common sections of code, team members can avoid duplication, which reduces the program's memory requirements.

Other items to look for during a code walkthrough include ensuring that the code handles errors properly (due to time pressures, laziness, or perhaps too little espresso, proper error checking is often neglected). Also look for correct handling of boundary conditions, and ensure that source comments are kept up-to-date and in agreement with the code's implementation.

SOURCE COMMENTS

Popular and long-standing programming wisdom encourages the use of extensive internal source code comments. In general, this is a good idea.

continues

continued

During the course of a project's development, however, time pressures often result in code changes being implemented without a corresponding edit and update of the source comments. Eventually, when you are trying to track down a problem or perhaps add a new feature, you may find yourself relying on inaccurate comments. This results in the maxim *A wrong comment is worse than no comment at all*. Therefore, strive to ensure that comments are kept up-to-date and are consistent with the implementation. You should check the accuracy of source comments during code walkthroughs.

ISOLATING PROGRAMMING DEFECTS

Most programming errors are due to some very common problems, such as uninitialized variables or index values that are off by 1. The problem, of course, is trying to find which variable or index value is out of whack. The following is a partial list of some of the more common types of problems encountered during software development.

LOGIC ERRORS

A logic error is when the program algorithm is wrong. The algorithm may be correctly implemented; however, because it is the wrong algorithm or based on flawed assumptions, the underlying methodology is itself wrong. Neither the best debugging tools nor whittling the code is likely to make it better.

You can check for flawed logic by testing the individual module for simple test cases. If the module works for simple cases, the basic logic is probably correct. When problems still show up, the problem may be in the *implementation* of the desired algorithm (see the sections that follow for suggestions on tackling implementation errors).

Procedural logic can sometimes be tested the old-fashioned way—by running through the code manually. That means running through the code *in your head*, line by line, noting the values of variables on a pad of paper, and checking for fundamental issues.

UNINITIALIZED VARIABLES

Forgetting to initialize a variable is a common problem, particularly when you are using global values. Also, remember that automatic or local variables do not retain their values between calls to the function in which they are declared (although locally defined static variables do retain their values between function calls). You should never depend on the value of a locally declared variable; always initialize such variables to an appropriate value.

Failure to initialize a variable may result in variables being set to zero, or more often, set to random values each time the code is executed. Occasionally, such random values are due to other problems (see "Clobbering Memory" later in this chapter, and "Pointer Problems and Memory Trashers" in Chapter 5).

When your variables have seemingly random values, check your code carefully to ensure that the variable was actually assigned the value you think the variable should have. Use the debugging tools described later to check the value of the variable at key locations in the program and determine if some other section of code is using the variable for some other purpose. For variables that should not vary, precede their declaration with the `const` keyword, as in the following:

```
const int maximum_horses = 100;
```

C allows you to declare variables local to a block of code (such as the code block appearing with paired brackets `{ }` after the `for` or `while` loop statements). Because many programmers tend to use simple variables like `i`, `j`, and `k` inside `for` loops, it's too easy to inadvertently define one of these variables a second time, which causes much mischief in your code. If you must declare local variables, make sure that the symbols you use have not already appeared in the local declarations for the current function. As a rule of thumb, it is also a good idea to use accurate variable names for loop counters, rather than the simple `i`, `j`, or `k`.

UNINITIALIZED OR ERRONEOUS POINTER VALUES

The use of pointers can cause much trouble, especially if care is not taken to ensure the validity of the pointer value. Pointers should be created through the use of `malloc()` or related functions, the `new` operator, or by copying the value of one pointer to another. You can also use the `&` unary operator to return the address of an existing variable or function and assign that address to a pointer variable. Failure to properly initialize pointers may result in disastrous consequences for your program, and possibly your PC.

After pointer initialization is taken care of, the next issue is proper disposal of dynamically allocated objects. Use `free()` or `delete` to return the memory occupied by an object back to the heap memory system. After the object has been disposed of, the pointer should no longer be used. Unfortunately, `free()` does not reset the pointer value to `null` or some other recognized value. Instead, the pointer continues to point to the memory allocation that has been disposed. In many instances, the now-disposed memory may not be reassigned for other uses right away, and it still holds its original value. Consequently, your program may erroneously continue to use the pointer for quite some time because it still points at "valid" data. Only when the heap memory is reassigned to a new purpose does the data become invalid. Then your program crashes.

Here are some hints to help prevent this sort of error from occurring:

- If a pointer is assigned to another pointer, as soon as either pointer is disposed, both pointers become invalid.
- When disposing of pointers, you may wish to set the disposed pointer to `NULL`. The `free()` function frees up the memory that the pointer points to, but it does not change the memory address stored in the pointer. Alternatively, you can create your own `free()` routine to automatically set pointers to `NULL` after calling the library `free()` function. See "Pointer Problems and Memory Trashers" in Chapter 5 for an example.
- Insert pointer consistency checks in your code and halt the program if an invalid value is obtained. For example, you can ensure that pointers are non-null by using the `assert()` function (see Chapter 5, "Managing Memory"). Normally, this type of check code is used during program testing and is removed from the final version. Fatal runtime errors can

ruin a user's whole day, and they should not exist in properly tested software anyway (famous last words . . .). Use conditional compilation statements (`#if`) to selectively include your debugging code in test versions.

CHANGES TO GLOBAL VARIABLES

Many functions must access global data structures. Occasionally, one function may modify a global variable in a way that impacts another function. For example, consider a word processor that maintains a giant text array to keep track of text, and contains a `TextIndex` variable to indicate the cursor's location within the text data structure. One function may temporarily adjust `TextIndex` to point to some other location within the text structure. If it fails to return `TextIndex` to its original value, some other function may subsequently fail because `TextIndex` now has an invalid value. This problem can be difficult to solve because the effects of the defect show up somewhere other than where the actual defect is located.

Unintentional (and sometimes intentional) changes to global variables are called *side effects*. Programming practices such as encapsulation (that is, object-oriented programming) provide mechanisms to avoid side effects. Whether you choose to use traditional or OOP programming, you can restrict access to important data structures by requiring that operations on those data structures be handled by a core set of access routines. OOP's `private` declarations, or hiding data structures within the implementation of a module, provide ways to enforce encapsulation.

FAILURE TO FREE UP DYNAMICALLY ALLOCATED MEMORY

Any time memory is dynamically allocated, it must eventually be disposed. This is especially important when memory is dynamically allocated within a frequently called function and the dynamic allocation is assigned to a local pointer variable. On exiting the function, all dynamically allocated local pointers should be disposed. If the appropriate function—`free()`, `ffree()`, `delete`, or others—is not called, the memory allocated from the heap remains allocated for the duration of the program. At exit from the function, the value of the local

pointer variable is discarded. With the pointer now gone, there is no way to free up this memory later.

During testing and debugging, it may be helpful to display the current value of `coreleft()` or `farcoreleft()`, which are library functions that return an unsigned long value representing the total number of bytes currently in the free area of the heap. By frequently writing this value to some out-of-the-way place on the screen, you can keep an eye on the memory usage of your program. If free memory continually shrinks during program execution, your program will eventually halt with an out-of-memory error. An especially important check is to ensure that the amount of free memory at your program's conclusion is identical to the amount of free memory available when the program starts. If there is a difference in memory between the start and finish of your program, you have failed to discard all memory allocations.

If you suspect that certain areas of your program may be making memory allocations and—for whatever reason—failing to deallocate their memory, you may insert code to check the before and after values of `coreleft()` during program execution. Before calling a suspect function, check the value of `coreleft()`. On return, check `coreleft()` again and use this value to help you isolate the problem code.

TYPOGRAPHICAL ERRORS

Watch for typographical errors that transpose or omit symbols. For instance, it's easy to write incorrect conditional statements such as:

```
if (n = 1 ) ...
```

when you mean to write,

```
if (n == 1) ...
```

Misplaced semicolons may also generate incorrect code, as in the following example. This code is intended to encompass a section of code in a `while` loop, but instead it doesn't do much of anything:

```
#include <stdio.h>
void main( void )
{
   int i = 0;
   while (i++ < 10) ;
```

```
   {
   printf("%d\n", i);
  }
}
```

Another typing error results in this compilable but meaningless statement:

```
n - 2;
```

when you meant to write

```
n -= 2;
```

The Borland compiler will warn you (if you pay attention to warnings) that this code has no effect.

OFF-BY-1 ERRORS

*Off-by-1 error*s cause more irritating program defects than perhaps any other category of defects. The problem occurs when a calculated index value is off by 1. Instead of computing, say, 10, the calculated value is 9. This programming error is a pernicious problem requiring close attention to detail. To see how easy it is to encounter problems like this, consider the following code fragment that tries to extract the letters at array index 12, 13, 14, and 15 (in this zero-based array, this corresponds to the letters *M*, *N*, *O*, and *P*).

```
strcpy( alphabet, "ABCDEFGHIJKLMNOPQRSTUVWYXZ");
strncpy( s, &alphabet[12], 15 - 12 );
s[15 - 12] = '\0';
printf("%s\n", s );
```

The output from this code is

```
MNO
```

and not

```
MNOP
```

At first glance, the example code seems reasonable, beginning at `alphabet[12]` and extracting the letters from `alphabet[12]` to `alphabet[15]`. The problem is that the length determination is off by 1. Rather than `15 - 12`, you should use `15 - 12 + 1`. This is just like the problem encountered in determining the number of segments in a fence that has four posts: there are three segments, because the

fence has a post at each end. You need to be careful when calculating distances between two points.

This problem happens extremely frequently. Generally, when you are trying to compute the number of bytes or elements between a starting position (such as `StartPos`) and an ending position (such as `EndPos`), compute the number of elements as

```
EndPos - StartPos + 1
```

Similar to off-by-1 errors is the occasional mixing up of `<` with `<=` or `>` with `>=`. For example, if you mean to have a `for` loop run through values from 0 to 10, you must write

```
for( i=0; i<= 10; i++) ...
```

It is very easy to inadvertently write

```
for( i=0; i<10; i++) ...
```

In many instances, a problem like this can go unseen for quite some time. The solution is to ensure that the proper relational operator is used when testing for limits in conditional expressions.

CLOBBERING MEMORY AND OUT-OF-RANGE ERRORS

Related to the off-by-1 error is the memory trasher, which often occurs when an index or length value is incorrectly specified in an array index or to one of the byte block move functions such as `strcpy()` or `memcpy()`. Neither of these functions, nor array indexing, checks for out-of-range conditions; each can easily overwrite sections of memory. Listing 11.1 illustrates the problem as it occurs with arrays. This program outputs

```
p is not NULL!
```

because of the incorrect `for` loop in line 10. The condition `i<=10` should be either `i<=9` or `i<10`. As written, this `for` loop writes data beyond the end of the array `s`. Because of how the compiler stores variables, pointer `p` is located in memory immediately following `s` and is clobbered by the errant code. Subsequent use of `p` can overwrite other sections of memory because `p` now points to a random location in memory. As you can guess, this indirect damage can be difficult and time-consuming to trace back to its original source.

LISTING 11.1. AN EXAMPLE OF HOW ARRAY INDICING CAN CLOBBER A POINTER OR OTHER VARIABLE.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  void main(void)
 5  {
 6    int i;
 7    int * p = NULL;
 8    char s[10];
 9
10    for (i=0; i<=10; i++) s[i] = ' ';
11
12    if (p) puts("p is not NULL!\n");
13    else puts("P is NULL\n");
14
15  }
```

When trashed memory is discovered, finding the source can be difficult because finding the errant pointer does not necessarily lead to the code that caused it to be destroyed. Turbo Debugger can monitor variables during program execution (see "Usung Turbo Debugger" in this chapter). If the contents of a monitored variable change, Turbo Debugger immediately breaks the program's execution. Consequently, Turbo Debugger is an invaluable tool for locating memory trashers.

IGNORING SCOPING RULES

When functions define local variables that have the same names as external variables (those that are not defined within any function), it's easy to think you are setting the external variable, when in fact only the local value is set. For example, given the following code:

```
int Total;

void SumTotal( void )
{
  int Total, Sum;

  ...
  Total = Sum;
}

...
```

The assignment statement

```
Total = Sum;
```

affects only the local variable `Total` and not the external variable `Total`. Because both the local and external variables have been given the same name, it's easy to confuse the variables.

Another simple problem occurs when you inadvertently use an external variable within a function, thinking that the variable is locally defined. Here's an example of how this problem can ruin your whole day:

```
int i;

void P(void)
{
  for( i=1; i<=100; i++) { ... };
};

void main(void)
{
  for( i=1; i<=10; i++ ) P();
}
```

In this short section of code, the problem is obvious. But add a few hundred C or C++ statements and the problem may no longer be so easy to spot.

UNDEFINED FUNCTIONS

Within the body of a C or C++ function, your code must return a value for typed functions, for example:

```
int f(void)
{
  return 1;
}
```

Failure to return a value is an error that the compiler will catch for you. Sometimes, however, the function's `return` statement is hidden inside a loop or inside a conditional statement and is never executed during the course of the function's execution. For example, in the function `ComputeInvestmentYield()` that follows, the value of `result` remains undefined if `n` is equal to or less than zero.

```
float ComputeInvestmentYield( int n, float rates )
{
  float result;
  if (n > 0)
  {
    // Calculate investment yield, store answer in result
    ...
    return result;
  }
}
```

The compiler issues a warning when it sees a function that does not have an explicit `return` statement. The program will still run, albeit possibly incorrectly. For this reason, it is very important that you study the warning messages issued by the compiler and be certain you understand why each warning message is issued. Although you can safely ignore most warnings, there are many, as in this example, that you should heed. When a function returns seemingly random or wildly incorrect values, you should suspect that the `return` statement is not being executed.

Still another area related to undefined function results is the incorrect use of the `&` operator to return the address of a local variable. For example, take a look at the following short program:

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

char * f()
{
  char s[80];
  strcpy( s, "Goodbye world..." );
  return &s;
}

void main( void )
{
  puts( f );
}
```

When `f()` returns the address of its local character array `s`, it is returning an invalid address. When `f()` is no longer in scope, `s` is undefined and the pointer received by the caller is good for causing more memory trash.

EXPRESSION ERRORS

Occasionally, an expression produces the wrong result even though it appears correct each time you look at the code. The problem may be that the expression's order of evaluation is different from what you were expecting. As you already know, multiplication and division come before addition and subtraction. But with so many possible combinations of C expression operators, there are many ways that evaluation order (also known as *operator precedence*) can produce an unexpected answer. If you can't keep the operator precedence straight, use parentheses to force the desired evaluation order.

CHECK ALL RETURNED ERROR CODES

To ensure that software is robust, always check returned error return codes. Some functions, such as `malloc()`, return `NULL` if they are unable to allocate memory. Other functions set the global `errno` variable. For most personal programming, you can get away with ignoring error codes; when you are producing professional-quality software, however, you must detect and handle gracefully every possible error condition.

BOUNDARY CONDITIONS

Programs must be capable of operating near the limits of their memory requirements. When those memory requirements are exceeded—whether it be from lack of RAM or lack of disk space—the program must not hang or crash. See Chapter 5, "Managing Memory," for more information about the use of dynamic memory allocations and trapping heap memory errors.

Running out of disk space is fairly rare these days, with 100-megabyte and larger hard disks, but it can happen. Programs that copy data to floppy disks frequently encounter out-of-disk-space errors. Your programs must check the error codes when you use functions such as `fwrite()`, `write()`, `close()`, and so on.

On rare occasions, an application can attempt to open more files than DOS will allow. The config.sys file should contain a statement such as

```
FILES = 30
```

that sets FILES equal to the number of file handles that DOS can make available simultaneously. Although each DOS process is always limited to a maximum of 15 files, the use of TSRs may cause applications to obtain a "Too many open files" error prior to reaching the maximum of 15 files. Be certain to set FILES to a suitable value. If your application will be used by others, be sure to catch this error and display an appropriate message instructing the user to fix the problem by increasing the value of the DOS FILES variable.

DEBUGGING TECHNIQUES

This section describes two separate debugging tools:

- The IDE's built-in debugger for source-level debugging, including setting breakpoints, using single-step execution, and displaying and changing variables during program execution.
- Turbo Debugger 3.0, for complete control of your program's execution at either the source or assembly language level.

THE IDE DEBUGGER

The IDE's built-in or *integrated* debugger is the most convenient of the three debugging approaches. Using the IDE, you can set *breakpoints* on one or more lines in your program. When one of the breakpoints is encountered during program execution, the program's execution is temporarily interrupted, but not stopped. At this point, you can display the value of global or local variables, even change their values if you wish, and then continue execution of the program. Execution can be single-stepped through the program, advancing one source line and then pausing. The single-step mechanism has a special feature to jump over functions or function calls. This enables you to avoid tracing into a function or functions that are known to be working, saving a considerable amount of time during the debugging process.

COMPILING FOR THE IDE DEBUGGER

The IDE default settings set the compiler for use of the integrated debugger. These options may also be set manually, and might need to be if you make adjustments to the items on the Options pull-down menu. On the IDE's Options menu, choose the Debugger... selection. On the Debugger options dialog box, under the heading Source Debugging, make sure that the On choice is enabled. Under the Display Swapping heading, enable the Smart selection so that the debugger automatically and optimally switches between the debugger screen and your program's output screen.

With Smart display swapping turned on, the debugger will try to restrict switching to the program's displayable output screen so that switches occur only after output has actually occurred. This helps to minimize "screen flicker" caused by rapidly switching between the IDE screen and your program's output.

Sometimes the debugger may not make the right decision as to which screen to display, failing to update the program output screen. If this occurs, choose the Always option. Always forces the debugger to always switch to the program screen when the programs runs, and back to the debugger when the debugger is in use. If your program does not produce any screen output (such as a program that processes file data), you may want to choose None.

When your program is under control of the debugger, the amount of memory available for the heap (and dynamic memory allocations) is, by default, limited to 64K. If your application needs more, set the amount in the Program Heap Size field of the Debugger options dialog.

Program optimization also influences your ability to access debug information. For initial debugging, I recommend that you select the Default button in the Options | Compiler | Optimizations... dialog box. When the compiler is set to optimize for speed or program size, it may optimize away some of your program code or variables because, through optimization, they may not be needed. Consequently, you may not be able to check the values of source-level variables because the compiler has managed to eliminate them while still producing code that performs your desired task.

In a related vein, the use of C++ inline functions can make debugging difficult. For this reason, you can elect to disable inline functions by choosing the Out-of-line inline function option of the Options | Compiler... | C++

Options... dialog box. When this is selected, inline functions are compiled like regular functions and are called at their points of invocation.

After these options are enabled, subsequent compiles will generate the appropriate code and symbol table information needed to use the integrated debugger.

USING THE INTEGRATED DEBUGGER

Putting the integrated debugger to work is as easy as positioning the cursor on a line where you want execution to pause, and then pressing the Ctrl-F8 key combination (or using the Debug | Toggle breakpoint menu option). When you select Run | Run, the program executes until it encounters the specified line, stopping and highlighting the breakpoint line with a solid cursor all the way across the edit window.

The text in the next several sections refers to IDE keystrokes, such as Alt-F5 or Ctrl-F4, which are used as shortcuts to activate functions available from the pull-down menus. However, because you can reconfigure the IDE's command keystroke assignments, the actual keystrokes you see on the pull-down menus in your installation may be different. The keystrokes referenced in this chapter assume that you are using the Alternate IDE keystroke set which is the default keystroke set just after Borland C++ has been installed.

To select a different set of keystrokes, choose Options | Environment... | Preferences.... In the Preferences dialog box, under the Command Set heading, you may choose CUA, Alternate, or Native. The CUA selection provides commonality with Windows applications and is useful if you are accustomed to operating Windows applications. The Alternate command set is the normal or default command set used the the DOS implementation of the Borland C++ IDE. When you select the Native option, the IDE automatically selects the CUA keystrokes when running

continues

continued

Borland C++ for Windows, and selects the Alternate or standard keystrokes when running the Borland C++ DOS-hosted IDE. For this reason, the Native option is the default of mode of operation when you first begin using the IDE.

At this point, you can single-step through the program or display the values of variables. To execute single statements, press either F7 (Run | Trace into) or F8 (Run | Step over). The difference between Trace into and Step over is that Trace into walks right on into a function or function call and begins tracing the function or function's statements. Step over, on the other hand, executes the function or function call, but continues to the immediately following statement before pausing again, ignoring the innards of the function or function call.

When the debugger has halted at a program statement, in its default mode of operation, the screen switches to display the source code and other debug windows. To see the program's own output, press Alt-F5 (Window | User screen). This temporarily displays the last output from your program; press any key to return to the IDE screen.

DEBUGGER WINDOWS

The IDE can display several windows of debug information. These windows include the Watch window for viewing the value of variables during program execution, the Registers window to display the CPU register values, and the Call Stack (Ctrl-F3) window for tracking the list of currently active functions. These windows are selected during debugging using the Window pull-down menu, or by selecting Call Stack from the Debug menu. If the screen becomes too cluttered with windows, the Tile option (Window | Tile) may improve the screen appearance by tiling the visible windows.

THE WATCH WINDOW

The values of variables are displayed in the Watch window. The Watch window gets its name from enabling you, the programmer, to watch the values during program execution. Figure 11.1 shows a sample Watch window during program execution.

Figure 11.1. An example of the Watch window in use to view a program's variables during execution.

To add a variable to the Watch window, press Ctrl-F7 or use the menu item Debug | Watches... | Add watch... and then type the name of the variable you wish to see (see Figure 11.2).

Although you can add any variable name at any time, the values of local, external, and object private variables cannot be displayed until the program's execution enters the scope in which the symbols are defined. Structure components must be specified using the full `structure.component` notation. Optionally, you can type only the structure name, and the Watch window displays all the record's fields, like this:

```
DataRecord:("Ed", "555-4307", 32, 0)
```

Figure 11.2 The Add Watch variable dialog box.

The Watch window displays as a small window overlaying your program source code. You may optionally drag the window to a new location using the mouse, hide it by pressing F6 (Window | Next) to bring the next desktop window to the top, or bring the previous desktop window to the top by pressing Shift-F6 (Window | Previous).

Watch variables remain in effect as long as you remain in the IDE. To remove a no-longer-needed variable from the Watch window, use the cursor or mouse to highlight the specific variable, and press the delete key or choose Debug | Watches | Delete Watch. To delete all watch variables, select Debug | Watches | Remove all watches.

If you mistype the name of a variable, you can change it by highlighting the particular variable and choosing Debug | Watches | Edit watch.... This displays a small dialog box in which you can correct the spelling, fix a typographical error, or even type a new variable name altogether.

CHANGING THE VALUE OF VARIABLES

As an alternative to watches, you can also examine the value of an individual variable using the Debug | Evaluate/Modify... menu selection. Evaluate/Modify displays an Evaluate and Modify dialog. If you position the cursor over a variable name at the time the dialog is activated, that variable appears in the Expression field of the dialog, and its current value displays in the Result field. The Expression field may contain individual variables or complex expressions. When you press the Enter key, the value of the expression is computed and displayed in the Result field. If there is an error in the expression, the error message text is displayed in the Result field.

To change the value of a variable, type the new value in the New Value field and press the Modify button. The variable is updated immediately.

A SUGGESTION

Evaluate/Modify can be used for more than just checking the value of variables. Because you can enter a complex expression, you can use this dialog to interactively check the effect of various casting operators on an expression. For example, Listing 11.2 shows a function that is called by the library routine `qsort()`. `qsort()` sorts blocks of data into either ascending or descending order, depending on a sort key comparison routine that you provide.

The parameters to the comparison functions are pointers to `void`. You must recast them to the data type that is appropriate for your comparison. For my application, I needed the complex recast shown in lines 9–13. Needless to say, I didn't get this correct on my first attempt. Instead, I used the Evaluate/Modify expression field to experiment with different recast expressions until I achieved the result I wanted. In the expression field, I typed:

```
((struct ttable_entry*) * (int *) a)->count
```

When I saw the correct value for `count`, I knew I had found the proper sequence of casting operators.

LISTING 11.2. A FUNCTION THAT WAS DEBUGGED USING THE EVALUATE/MODIFY DIALOG.

```
 1  int compare_func(const void *a, const void *b)
 2  {
 3    if ((* (int *)a==NULL) && (* (int *)b == NULL)) return 0;
 4
 5    if ((* (int *)a==NULL) && (* (int *)b !=NULL)) return -1;
 6
 7    if ((* (int *)b==NULL) && (* (int *)a != NULL)) return 1;
 8
 9    if   ( ((struct ttable_entry*) * (int *) a)->count <
10    ((struct ttable_entry*) * (int *) b)->count)   return 1;
11    else
12    if   ( ((struct ttable_entry*) * (int *) a)->count ==
13    ((struct ttable_entry*) * (int *) b)->count)   return 0;
14    else
15      return -1;
16
17  }
```

USING BREAKPOINTS

You can set and clear breakpoints in several ways, including toggling unconditional breakpoints on or off at the cursor's location, executing the program to the cursor's current location, or setting a conditional breakpoint.

Before starting execution of the program, you can quickly set breakpoints at a variety of locations by moving the cursor to the desired line and pressing Ctrl-F8 (Debug | Toggle Breakpoint). This sets a breakpoint at the beginning of the line on which the cursor is located. You can set multiple breakpoints in this manner. Each line containing a breakpoint is displayed with a bright red highlight bar (if you are using a color display).

The Toggle Breakpoints command also turns off existing breakpoints. To do this, move the cursor to the line containing the breakpoint and press Ctrl-F8.

To begin program execution, choose the Run command from the Run menu in the normal manner. When the program stops at a breakpoint location, you can use the debugger's features to set other breakpoints, or examine or modify variables. To resume running the program, select the Run option from the Run menu (or choose the Go to Cursor option as described in the next paragraph). If for some reason the program should be restarted from the beginning, choose the Run menu's Program Reset option. This terminates the current execution of the program and resets all the system parameters so that the next time Run is invoked, the program will begin running from the start of the program.

If you need to set just one breakpoint, move the cursor to the line where execution should be stopped and then press F4 (Run | Go to cursor). The program will begin execution and will run until encountering the source line that contained the cursor. Once execution has stopped, you can use F4 (Run | Go to cursor) to continue and again stop at a specific line.

The most comprehensive breakpoint facility is provided under the Debug menu's Breakpoints selection. This displays a dialog box, shown in Figure 11.3, containing a list of all currently set breakpoints in the program and related units. From this dialog box, you can edit the attributes associated with a breakpoint, delete an existing breakpoint, view the source line containing the breakpoint, or clear all currently set breakpoints.

Figure 11.3. The Breakpoints dialog box.

To edit a breakpoint, move the cursor to the desired breakpoint and choose the Edit button. Edit displays the dialog shown in Figure 11.4, at which you can specify breakpoint attributes to make the breakpoint occur when a specific condition occurs. The Edit Breakpoint dialog contains four fields: Condition, Pass, Filename, and Line number. The last two fields specify the file and line number where the breakpoint is located.

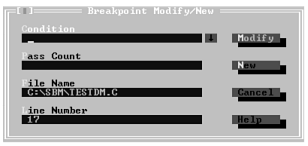


Figure 11.4. The Breakpoint Modify/New dialog box that is displayed when you edit breakpoints.

The Condition and Pass fields are used to restrict the breakpoint to taking effect only when certain conditions are met. To understand how this works, consider the following sample program:

```
#include <stdio.h>
void main( void )
{
  int i;
  for( i=1; i<=50; i++ )
    printf("%d\n", i*2 );
}
```

Pretend that this program is a bit more complicated, and that it runs through some fairly complex calculations before displaying the result. Let's assume that something has gone wrong in our calculation, but the problem shows up only when the calculation results in certain values. To debug through this sequence of statements, you could set a breakpoint at the start of the loop and repeatedly restart the program after hitting each breakpoint until the calculated value is hit. That approach would be quite tedious, and fortunately, it is not needed. Instead, you can type the calculation into the Condition field as a relational expression. When the condition becomes true, the program's execution will halt. For example, suppose you want the preceding program to halt when `i*2` equals 24. To make this happen, type the following in the Conditions field:

```
i*2 = 24
```

and then run the program. When the condition `i*2` equals 24 is met, the breakpoint is hit and the program stops running.

In this particular example, there is another way to stop the program at this point. The Pass field specifies an iteration factor for the breakpoint. By setting Pass to 12 (for example), the breakpoint is skipped for the first 12 times that it is encountered. On the next attempt to pass through the breakpoint, the program stops.

OTHER BREAKPOINT FEATURES

To delete a breakpoint using the Debug | Breakpoints list of breakpoints, move the highlight bar to the breakpoint to be deleted and choose the Delete button.

To immediately jump to a breakpoint's location in the source, move the highlight bar to the breakpoint of interest and press the View button. The edit cursor jumps to the program statement containing that breakpoint.

DEBUGGING THE OLD-FASHIONED WAY

The use of an interactive debugger provides incredible debugging power (also see the discussion of Turbo Debugger in the next section). Occasionally, however, an old-fashioned debugging tool can prove handy. In the old days, before interactive terminals and PCs, programmers submitted their decks of punched cards to the deity known as the Computer Center. Some time later, a stack of computer printouts would appear containing the results of their program's execution. Debugging a program in the batch environment required a different approach to debugging—embedding special debug statements inside the program itself to print the contents of variables during the program's execution. After the program had run, the program's *execution trace* was examined to see what really went on inside the program.

In spite of modern interactive debuggers, there are times that a dump of a program's execution history can be useful to your debugging efforts. An easy way to do this is to add various `printf()` statements inside the source code that are used only during the debug phase.

The problem with adding `printf()` statements indiscriminately is that they are time-consuming to remove when they are no longer needed. An improvement is to use Borland C++'s conditional compilation directives (such as `#if`) to conditionally include debug code during test and checkout, but automatically to eliminate it when you are compiling the final version. This also has the advantage that when problems are encountered later (and they always are), the debug code can be switched back on just by setting a conditional compilation symbol and recompiling the program.

For example, you could embed in your source code the following:

```
#define Debug
...
#ifdef Debug
printf("Inside AddRecord, RecNum=%d\n", RecNum);
#endif
```

By removing the definition of `Debug`, the debug code is instantly removed from the source. You can optionally define constants using the Defines field of the IDE's Options | Compiler | Code Generation... dialog box, or using the `-D` command-line compiler option (for example, `bcc -DDebug=1`).

Another excellent debugging tool is the `assert()` macro, which is defined in the standard header file assert.h. `assert()` is defined as:

```
void assert( int test );
```

When `assert()` is called, replace `test` with an expression that evaluates to an integer result, such as a relational test. If the value of `test` is zero, `assert()` aborts your program and prints a message detailing where the failure occurred. By defining the macro symbol `#define NODEBUG` on the line before the `#include <assert.h>`, all subsequent `assert()` references are converted to comments in your source. In this way, the assertion is conditionally compiled into your program so that you can include it during testing but quickly remove it for compiling the final version of the program.

`assert()` is an excellent tool to help catch program errors early. For much more information about the use of `assert()`, as well as a sample program that uses `assert()`, see "Pointer Problems and Memory Trashers" in Chapter 5.

USING TURBO DEBUGGER

Turbo Debugger is a stand-alone debugger (included with the Professional editions of Borland products and also sold separately) providing Borland C++, assembly, C, and C++ statement-level debugging, as well as machine code debugging and access to the CPU's registers, stack, and other critical memory areas. Turbo Debugger provides all the features of the IDE's integrated debugger, but has a wide variety of additional features, including keystroke macros, compatibility with EMS for handling larger programs, 80836 virtual debugging for loading applications in their own 640K, access to a built-in assembler and disassembler, and even remote debugging, that is, interfacing to and debugging a program running on a separate computer. Turbo Debugger is excellent for debugging programs that are too large to be run within the Integrated Development Environment.

The Turbo Debugger package includes a group of utility programs to aid in debugging. These include a file transfer utility called `TDRF`, `TDSTRIP` to delete the debugging information created by the compilers (which reduces the need to recompile or relink executable files), a disassembler, and other programs.

This section provides information about Turbo Debugger and instructions to get you quickly up to speed debugging Borland C++ programs.

COMPILING FOR TURBO DEBUGGER COMPATIBILITY

The IDE default settings set the compiler for use of the integrated debugger. To use Turbo Debugger with programs compiled from within the IDE, you should set several options. To start, use the Options | Debugger selection to display the debugging options. On the Debugger options dialog box, under the heading Source Debugging, select the Standalone option.

Because memory may be limited during debugging, the debugger may restrict your total dynamic memory size (the heap) to just 64K bytes. If your application requires more memory, set a larger heap size in the Program Heap Size field.

Program optimization also influences your ability to access debug information. For initial debugging, I recommend that you select the Default button in the Options | Compiler | Optimizations... dialog box. When the compiler is

set to optimize for speed or program size, it may optimize away some of your program code or variables because, through optimization, they may not be needed. Consequently, you may not be able to check the values of source-level variables because the compiler has managed to eliminate them while still producing code that performs your desired task.

In a related vein, the use of C++ inline functions can make debugging difficult. For this reason, you can elect to disable inline functions by choosing the "Out-of-line inline function" option of the Options | Compiler... | C++ Options... dialog box. When this is selected, inline functions are compiled like regular functions and are called at their points of invocation.

After these options are enabled, subsequent compiles will generate the appropriate code and symbol table information needed to use Turbo Debugger. You can launch the Turbo Debugger directory from within Borland C++. Press Alt-space bar to display Borland's Transfer menu. By default, Turbo Debugger has been installed on this menu and you may select it to run the Debugger directly from the IDE.

COMPILING WITH THE COMMAND-LINE COMPILER

To compile Turbo Debugger-compatible programs using the command-line compiler, you must specify the `-v` switch. This causes the compiler and linker to insert debugging information into generated .obj files. During debugging, the effect of the `-v` switch causes C++ inline functions to be treated like normal functions. This can make debugging easier. If you would like C++ inline functions to remain as inline functions, however, you should also add the `-vi` switch. `-vi` turns inline expansion on; `-vi-` turns inline expansion off. You may also wish to disable optimizations using the `-Od` switch.

STARTING TURBO DEBUGGER

Assuming that Turbo Debugger can be reached through your PATH statement, start the debugger by typing

```
TD program.EXE optional program parameters
```

or by typing

```
TD
```

and then choosing the File menu's Open command to load an executable file and initialize the debugger.

The corresponding source for the executable program is loaded into the Module window, and the cursor is placed on the first line of the program's `main()` function, as shown in Figure 11.5.

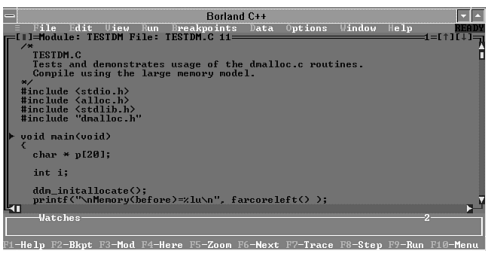


Figure 11.5. Turbo Debugger as it begins a debugging session.

If you have neither seen nor used a debugger before, your first acquaintance with Turbo Debugger may feel like you've just been dropped into the pilot's seat of a Boeing 767 airliner. Turbo Debugger presents much information about your program and its execution, and enables you to access a wide variety of breakpoint options, both for code and data. Fundamentally, it's not much different than the IDE's integrated debugger except that there are more debugging tools for your debugging pleasure.

The basic concepts are similar: Watch windows to examine the value of variables, an Evaluate/Modify option like the IDE's for altering variables, and several breakpoint options. Turbo Debugger offers many additional features as enhancements to the basic functions, plus more options for overall control over the debugging process.

PRESSING CTRL-BREAK

Occasionally (well, actually pretty often), your program takes an execution path different than the one you are expecting, and it never encoun-

ters a breakpoint. You can manually halt the program by pressing Ctrl-Break to return operation to the control of the debugger, unless interrupts have been disabled or your program has truly launched into hyperspace, crashing the system. When the program stops, the address of the Ctrl-Break interrupt may be anywhere. Don't be surprised if Ctrl-Break takes effect while the CPU is executing DOS or BIOS code and the program's address is somewhere outside the scope of your program.

THE WATCH WINDOW

By default, Turbo Debugger displays the Watch window just below the Module window. If it's not visible, choose Watches from the View menu (see Figure 11.6). To add a variable to the watch window, press Ctrl-F7 (or Data | Add watch...) and type the name of the desired variable in the data-entry dialog box. Alternatively, move the cursor to a variable name and press Ctrl-F7. The variable name is automatically entered in the variable name field of the Add watch dialog. Most of the rules concerning the use of watches are the same as those in the integrated debugger.

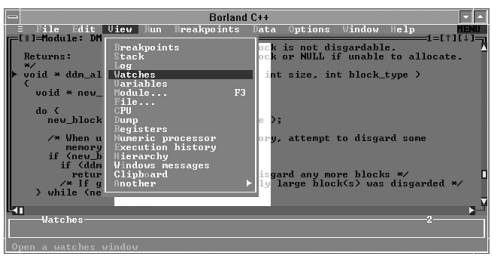


Figure 11.6. Turbo Debugger's View menu and the Watches window.

INSPECTOR WINDOWS

The second way to examine or change a variable is through an *inspector window*, available on the Data menu. The inspector window displays the variable name, its memory address, its contents, and its type (such as `int`, `unsigned int`, `char *`, and so on). When the inspector window displays a structure or object, it shows each of the components or fields by name.

To change the value of an item in the inspector window, move the highlight bar to the desired item and press Alt-F10 to open the inspector window's local menu, which is shown in Figure 11.7. From this menu, select Change and type a new value for the highlighted variable.

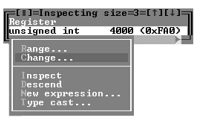


Figure 11.7. Available options when you are inspecting a data value.

If you are viewing an array, use the local menu to select the Range option and choose which elements of the array to view. In the Range dialog box, you can type a starting index and the number of elements to observe. For example

```
10, 5
```

instructs the inspector to show `5` elements beginning at index position `10` in the array. Alternatively, you can press the PgUp, PgDn, and cursor-movement keys, or click the scroll bar with the mouse, to move to a new position in the array.

Inspector windows can also examine functions. For these items, the inspector displays the address of the function, information about the parameters, and the return result data type.

EVALUATE/MODIFY

Turbo Debugger provides several ways to examine and alter variables. Evaluate/Modify, on the Data menu, operates essentially the same as the IDE's integrated debugger's Evaluate/Modify window. One added feature is that the expressions typed into the Expression field of the dialog box can be written in C, Pascal, or assembler, depending on the current setting of the Debugger's language setting (see the Options menu, Language setting). By changing the Language setting (the default setting of Source tracks the language of the source program), you change the syntax allowed in the Expression window.

VIEWING ALL VARIABLES

By selecting the View menu's Variables window, you can see and alter all of the program's variables. The Variables window, shown in Figure 11.8, displays a list of all accessible symbols, including variables and functions, used in the program.

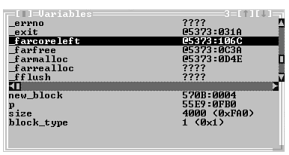


Figure 11.8. The Variables window.

Each variable or function is shown in the list box, with its value or address on the right. From this window, you can assign a new value to a variable by moving the highlight bar to the desired variable and pressing Alt-F10 to display the variable's local menu. Two of the options are Inspect and Change. Choose Change and type a new value for the variable in the dialog box. The new value is immediately assigned to the selected variable. Choosing Inspect opens an inspector window for the selected variable.

THE VIEW | HIERARCHY COMMAND

For object-oriented programming, the View | Hierarchy command displays all objects, showing who is descended from whom. Press Alt-F10 to display Hierarchy's local menu, from which you can inspect any object in the list and display each object's fields and methods.

CONTROLLING PROGRAM EXECUTION

Turbo Debugger provides several execution options to assist with tracing program execution. These options are available on the Run and Breakpoints menus.

On the Run menu, the options Run, Go to cursor, Trace into, Step over, and Program reset correspond to the identically named functions in the IDE's integrated debugger. Several other features enable more precise control over the debug execution.

Execute to... displays a dialog box, prompting for an address up to which the program should run and then stop. By typing the name of a function, you set a breakpoint at the beginning of the function. Execution begins and continues until it encounters that breakpoint. (The Execute to... function does not recognize statement labels, only functions.)

Use Until Return to execute the remainder of the current function, through the RET return opcode, coming to a stop on the statement following the function's call. For example, while you are single-stepping through a function, you may conclude that you've seen enough. To resume program execution back to the statement that called the function, select Until Return. Another use is when you inadvertently choose Trace into when you really meant to choose Step over. Finding yourself in the midst of long function, you can select Until Return to get the program back on track.

Animate... is sort of an automatic Trace into key presser. When you select Animate, a dialog asks for a delay factor; the default is three tenths of a second. While under control of the Animate function, the debugger single-steps

through your program, pausing at each statement for the specified delay factor. This lets you watch your program execute statement by statement, and is a bit easier on the fingers than pressing F7 (Trace into) a zillion times. Press any keyboard key to stop the animation and return control to the debugger.

Finally, Back trace is an ingenious function that lets you run your program backward. Any instructions executed as a result of single-stepping through the program (using F7 or Trace into, or Alt-F7 or Instruction Trace) can be "undone" by single-stepping the program in the reverse direction. Back trace (Alt-F4) uses the single-step execution history maintained by Turbo Debugger and works only when used after a sequence of single-step functions. Certain restrictions do apply: you cannot back up into an interrupt, nor can you back up beyond a function call that was stepped over using F8 or Step over.

To view the instructions stored in the execution history, select the View menu's Execution history option. The contents of the execution history are displayed in a separate window. Each time the program is run (other than single-stepping), the execution history is cleared.

BREAKPOINTS

Turbo Debugger provides a variety of unconditional and conditional breakpoints. Conditional breakpoints can be set to break program execution depending on the value of expressions, a pass count, or when the values of particular memory areas change.

Some of the breakpoint options are nearly identical to the IDE's integrated debugger. For instance, you can toggle breakpoints on or off by moving the cursor to the appropriate source line in the Module window and pressing F2 or selecting Breakpoints | Toggle.

SETTING BREAKPOINT OPTIONS

Breakpoints | At... presents a "one-stop shopping" dialog box for setting and controlling all types of breakpoints (see Figure 11.9). The address field

nominally contains the module name and line number where the cursor is located when this dialog is activated. You can edit this field to specify a different module, a new line number, or a function or function name.

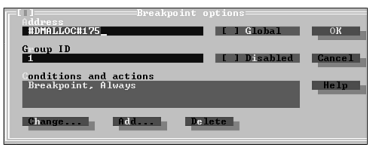


Figure 11.9. The Breakpoint options dialog box.

In Turbo Debugger, when a breakpoint is hit, one of three possible actions may occur: *Break* program execution, *Execute* an expression, or *Log* the breakpoint in Turbo Debugger's Log window (available as the Log option on the View menu). To select an appropriate action, choose the Change button. This displays the Conditions and actions dialog box shown in Figure 11.10.

Figure 11.10. The Conditions and actions dialog box, which controls the use of breakpoints.

Choosing a radio button beneath the Action heading selects the appropriate action. If either Execute or Log is chosen, you can enter an expression in the Action expression edit field.

USING THE BREAKPOINTS GLOBAL OPTION

Enable the Global breakpoints setting if the intended result of your breakpoint is something other than stopping program execution. For example, to trap or log the occurrence of a change in a variable's content,

Turbo Debugger must check after each instruction is executed. Such a "breakpoint" is not associated with a particular statement, but with the memory location owned by the variable. By setting the Global option, the "breakpoint" is made global to the program's operation.

When using the Breakpoints | At... dialog to set conditional breakpoints to trap or log memory changes or expressions being true, you should set the Global option. Alternately, you may set global breakpoints directly using the Breakpoints | Change memory global or Breakpoints | Expression true global selections.

INSERTING EXECUTABLE EXPRESSIONS

An Execute expression consists of an arithmetic expression to be executed just before the line containing the breakpoint and acts like a tiny code splice inserted into your existing program. Generally, such an expression assigns a value to a variable, letting you change or correct a program error. By inserting this code fragment, you can temporarily fix a problem and continue with the debug session. In another manner, you can test the program's ability to cope with out-of-range values by inserting test data during the debug session.

A Log expression consists of an expression or variable name. When the breakpoint is encountered, program execution does not halt, but instead the value of the Log expression is written to the Log window, together with the module name and statement location. Choose View | Log to examine the logged trace information. To clear the current log, while viewing the log, press Alt-F10 to display the Log window's local menu and choose Erase log.

Beneath the Condition heading, the condition causing the breakpoint to occur is set. The default is to make the breakpoint active each time it is encountered, hence the Always selection is set automatically. Like the IDE's integrated debugger, however, breakpoints can be selectively set. To make the breakpoint active based on the value of an expression, select the Expression true radio button and type the expression into the Condition expression field, for example:

```
index = 10
```

If the Changed memory condition is made active, you can enter a variable name in the Condition expression window. Whenever this variable changes, the program will break its execution.

CHANGED MEMORY GLOBAL...

Breakpoints | Changed Memory Global... is the easy way to set a memory watcher. Just type the name of the variable to be monitored and Turbo Debugger does the rest. If the variable changes during program execution, the program will break to the debugger.

EXPRESSION TRUE GLOBAL...

Breakpoints | Expression True Global... is an easy way to set a conditional breakpoint to occur when a expression evaluates to true. Type the expression into the dialog box field, remembering to enter a relational expression that can be evaluated to either true or false, for example:

```
index == 10
```

VIEWING BREAKPOINTS

Conditional and unconditional breakpoints that are associated with program statements are highlighted in bright red (on a color display) in the Module window. The complete list of all breakpoints is visible by choosing the View | Breakpoints window. From this window, breakpoints can be added, edited, or deleted. Press Alt-F10 to display the local Breakpoint window menu and select the appropriate function from the menu.

TURBO DEBUGGER AND ASSEMBLY LANGUAGE PROGRAMS

If you program in assembler or use external assembly language routines linked to your Borland C++ programs, the use of Turbo Debugger is essential because Turbo Debugger can single-step through programs at the assembly language source level. Turbo Debugger also includes a built-in disassembler so that you

can disassemble your C or C++ programs and single-step through the machine code generated by the compiler.

To see your program in its disassembled form, select View | CPU. This displays Borland C++ lines interspersed with the resulting machine code disassembly, and also the current status of CPU registers and status bits. Figure 11.11 shows an example of the CPU display. All of the normal debugging features are available in the CPU window, including single-stepping, breakpoints, logging of data, and so on.

PROTECTED-MODE DEBUGGING ON THE 80286

If you are developing and testing Borland C++ applications on either an 80286- or 80386-based system and the system has at least 640K of EMS available, you can make more memory available to the application being debugged. Two separate methods are provided: for the 80286 (and also the 80386), a special protected mode of execution is available that loads Turbo Debugger into high memory; for the 80386 CPU, programs can be debugged in a virtual 8086 environment (see the next section).

To access the protected-mode debugger, you run `TD286` instead of `TD`. Before launching `TD286` for the first time, the debugger must be configured for your system. The configuration is done automatically by running `TD286INS` and pressing the space bar in response to the prompts. If `TD286INS` hangs, this is considered okay. Reboot the system and run `TD286INS` again. The "hang" is all part of its configuration process (which makes it one of the few programs that can crash and actually get away with it).

`TD286` is essentially equivalent to `TD`, except that less memory is available for shelling to DOS, if you happen to use that feature.

VIRTUAL DEBUGGING ON THE 80386

If your PC uses an 80386 processor, Turbo Debugger can run your applications in virtual 8086 mode. *Virtual debugging* runs Turbo Debugger out of extended memory and loads your application into a virtual 8086 memory area, enabling your program to have total access to DOS's 640K low memory area. This

enables Turbo Debugger to debug programs that would otherwise not fit in memory. Because the application is running on a virtualized 8086, Turbo Debugger can also monitor IN and OUT I/O instructions with little degradation in performance.

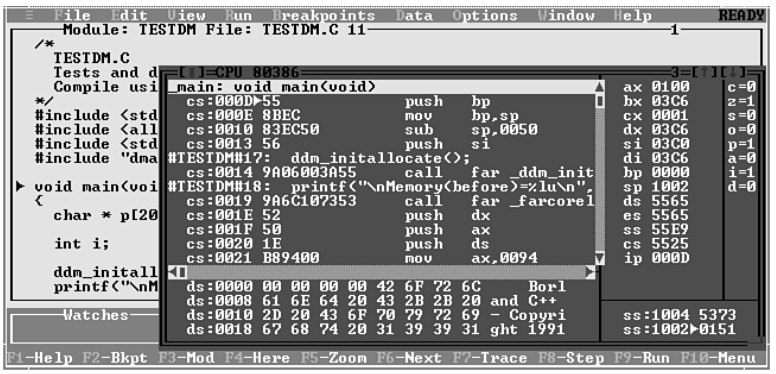


Figure 11.11. The CPU disassembly operation mode.

To use the virtual mode capability of the 80386, you must add a special device driver to config.sys. TDH386.SYS provides support to operate the debugger in the 80386's virtual machine mode. Assuming that Turbo Debugger is installed in the default directory \borlandc\bin, add the following DEVICE statement to CONFIG.SYS:

```
DEVICE=c:\borlandc\bin\TDH386.SYS
```

An optional single parameter for TDH386.SYS overrides the default of 256 bytes for the DOS environment variables. By appending -e*nnnn*, where *nnnn* is the number of bytes to allocate to the DOS environment variables, the virtual debugger allocates the number of bytes specified.

> IMPORTANT: PROGRAMS THAT CONFLICT WITH VIRTUAL DEBUGGING
>
> You cannot run the virtual debugger in tandem with other applications that are also using both virtual and protected CPU modes. These applications include Windows and DesqView, DR DOS's EMM386.SYS,

and other utilities. If the virtual debugger detects a conflict, it will display the following message when you try to launch the debugger:

```
Cannot run TD386: processor is already in V8086 mode
```

To play it safe, disable all other applications that use extended memory, including disk cache utilities and so on.

STARTING THE VIRTUAL DEBUGGER

To run the virtual debugger, type

```
TD386
```

or

```
TD386 programname
```

If you forget that you're trying to run the virtual debugger and type `TD` instead of `TD386`, the normal Turbo Debugger will run, and you won't get any of the advantages of running in virtual mode. It's an easy mistake to make (and one that has confused me a few times).

Choose Get Info... from the File menu to see the amount of memory available to your program. The available memory should be the same as when only DOS has been loaded.

USING TURBO DEBUGGER MACROS

Turbo Debugger's built-in macro system can boost your debugging productivity. You use it to save a sequence of keystrokes that can be played back at a later time. By doing this, you can save yourself the trouble of entering a keystroke sequence that you must use repeatedly. For example, when you are testing a new function, you may have to set a breakpoint, then press F8 to single-step down through ten source lines, and then press F7 to trace into a certain routine. Each time you run through this section of code, you'll have to repeat the preceding sequence of steps. This type of operation is perfect for recording into a macro.

To create a macro, select Options | Macros. This displays a submenu (see Figure 11.12) from which you may choose to create a new macro, stop recording a macro, remove an individual macro, or delete all macros. Select Create... to begin recording a new macro definition (you may also access this function using the Alt-= hot key).

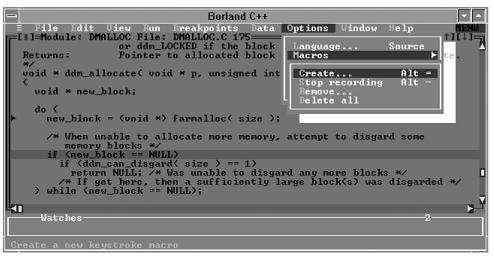


Figure 11.12. The Macros submenu.

Create... displays a dialog where you can select a keystroke to associate with the macro definition. You can use any key or valid key combination (such as Alt-F2 or Shift-F3) for your macro definition. You can even use keystrokes that are part of the Turbo Debugger command set. For instance, if you type Alt-B, you will no longer be able to access the Breakpoint menu by using the Alt-B keystroke. In case you do inadvertently redefine a keystroke, you can easily delete the macro later and restore Turbo Debugger to its normal usage.

After you have typed your macro key or keys, Turbo Debugger begins recording all subsequent keystrokes. You can single-step and access all Turbo Debugger functions. When you have finished recording your macro, choose Options | Macros | Stop recording to save the new macro.

You can play back a macro by pressing the macro's hot key. The keystrokes will play back into Turbo Debugger, automatically controlling its actions. Be sure that you invoke your macro at the same point from which you originally recorded it. For instance, if your macro expects to run the program through a sequence of debugging steps beginning from the first line of the `main()` function, you should reset the program (using Run | Program reset) to position back to the start of the program.

You can delete any macros by selecting Options | Macros | Remove... and then choosing the macro to delete from the list box that is displayed. To remove all macro definitions, choose Options | Macros | Delete all.

DEBUGGING TSRS

You can learn how to create your own terminate-and-stay-resident (or TSR) pop-up programs in Chapter 15, "How to Write a TSR." To debug a TSR requires the use of Turbo Debugger; you cannot use the built-in debugger provided in the Borland C++ IDE to debug the RAM-resident portion of the TSR. You *can* use the IDE debugger to debug the portion of the TSR program that executes and makes the program memory-resident. When the program is resident, however, you will no longer have any control of the program unless you use Turbo Debugger. Debugging the portion of the TSR that executes and terminates with the `keep()` function is the same as debugging any other C or C++ application. Only when you get to the memory-resident portion of the program does debugging require a slightly different mechanism.

Turbo Debugger provides special features specifically for debugging TSR applications, making the process quite simple and effective. To debug the memory-resident TSR, follow these steps:

1. Run Turbo Debugger and load the TSR, just as you do when you use Turbo Debugger on any other application. Presumably, you have compiled the program to include all the usual debug and symbol table information.
2. Run the TSR like a normal application. The TSR will execute and make itself memory-resident.
3. Set breakpoints, as desired, in the memory-resident portion of the TSR. Do this as you would for any other application.
4. Select the Resident command from the File menu. This command makes Turbo Debugger itself become a memory-resident TSR.

5. Activate your TSR through the keyboard or whatever interrupt it responds to. When the TSR encounters a breakpoint, Turbo Debugger reappears on-screen and takes control. You can now single-step and perform traditional debugging functions.
6. If you are unable to activate your TSR from the DOS command line, press Ctrl-Break twice. Turbo Debugger will resume control over the system.

A particularly nice feature of using Turbo Debugger is that it saves the state of the CPU prior to execution. If your TSR resets various interrupts and generally makes a mess of things, you can often just exit from the Debugger and the system will be safely restored. This is much easier than having to reboot the system to get rid of the damage caused by an errant TSR, or perhaps a new TSR that has an "uninstall" feature that is not yet working properly.

DEBUGGING TURBO VISION APPLICATIONS

Event-driven software, including both Turbo Vision and ObjectWindows, is different from traditional programming in that program flow moves in a nonsequential manner. The discussion here centers on Turbo Vision, but the concepts apply equally to ObjectWindows applications (for which you will probably use Turbo Debugger for Windows, described in the next section). For interactive software, much time is spent inside Turbo Vision, waiting for events, and then passing them along to the appropriate event handler. Trying to debug such software by single-stepping through code sections does not work well. When you attack a stubborn defect in your Turbo Vision applications, use the following suggestions in conjunction with checking for traditional, non-OOP related problems, such as using or disposing of invalid pointers, incorrect operation about boundary conditions, off-by-1 errors, and so on. Make sure that you create and use constructor methods, as required, and that destructor methods are called to dispose of data structures maintained by each object. Also remember that each destructor should normally call its parent's destructor method too, immediately after cleaning up after itself.

In general, for debugging Turbo Vision applications, you should set breakpoints at various places in the code. Placing a breakpoint in the event handler

enables you to identify the actual events that are being generated. If you do not see the event you are expecting, you should test for some of the problems outlined in the next few paragraphs.

First, ensure that if you call an ancestor's `HandleEvent` method that you call the `HandleEvent` method before or after your event processing as needed to override or supplement the ancestor's functionality. If your event handler calls the ancestor's `HandleEvent` method before doing its own event handling, the ancestor may completely handle and clear the event. The solution is to trap and process the event before handing it off to the ancestor.

Also check for duplicated `cmXXXX` command constants. In some cases, this can result in the event being swallowed up by some other view because each view may think it is getting a unique command code. Another place to look is the `TView.EventMask` variable. Each window is a descendent of `TView`. Setting `TView.EventMask` to `$FFFF` allows all events to be recognized by the given view, whereas setting `TView.EventMask` to 0 filters out all events. In the latter case, the view receives no events. Consequently, depending on the setting of the `EventMask` variable, your view may be excluding events that you expect to see.

TURBO DEBUGGER FOR WINDOWS

When you use Borland C++ for Windows to edit, compile, and execute your Microsoft Windows applications, you do not have access to a built-in debugger like that available in DOS. Instead, when you select the Run menu's Debugger command, Borland C++ for Windows launches Turbo Debugger for Windows, a separate debugging application that is similar to Turbo Debugger for DOS. Turbo Debugger runs only in Windows Standard and Enhanced mode, requiring at least an 80286 CPU and one megabyte of memory (with more recommended).

Turbo Debugger for Windows (TDW) runs as a full-screen DOS application under Windows. You cannot run TDW at the DOS command line or within a DOS box; TDW is itself a Windows application that just happens to look like a DOS application. While Turbo Debugger for Windows is active, you cannot

switch out of TDW to run another Windows task. This means that the Alt-Enter and Alt-Tab keystrokes—for turning a full-screen DOS application into a windowed application, or for task switching, respectively—cannot be used with TDW. However, while your Windows application is executing, even if under the control of TDW, you can run other applications, switch tasks, and even minimize your application's window. The restriction applies only when the TDW user interface is visible on-screen.

USING TURBO DEBUGGER FOR WINDOWS

You can run Turbo Debugger for Windows directly from within Borland C++ for Windows. Choose the Debugger option from the Run menu. This is the easiest and most common way to launch TDW. Alternately, you may run TDW from the Windows Program Manager, using File | Run, but this is seldom how you will access TDW.

The actual debugging commands available in TDW are pretty much the same as those in Turbo Debugger with a few extra features to support Windows. For instance, while your application is running, you can break out of the application and return to TDW by pressing Ctrl-Alt-SysRq (this is equivalent to pressing Ctrl-Break in Turbo Debugger-controlled applications). Another unique feature lets you watch the messages that Windows passes to your program. This feature is described in the following section. Some features, such as the DOS Shell and the Resident command option for debugging TSRs, are not available in TDW.

To ensure that dynamic resource allocations are properly cleaned up, you should terminate your Windows applications using its File | Exit command (or another program termination instruction such as the System menu's Close command). If you do not terminate the Windows application using a standard method, it is quite possible that unreclaimed resources will remain allocated. This can cause trouble if you run your program again, and can leave resources unavailable for other programs.

WATCHING MESSAGES

You can examine the messages that Windows sends to your application by using the View | Windows messages command. This function displays a window

showing three areas: a list of windows to monitor for messages, a list of message types (or message classes) that it will intercept, and the content of the message parameters used by Windows (the `wParam` and `lParam` variables). To select a window to view, activate the Windows message's local menu by pressing Alt-F10. From this submenu, you can choose to Add or Remove a window from the list.

Before you can intercept messages, you must add a window to the intercept list by entering the name of a window (or dialog or dialog controls, which are also windows). Use the *Add window or handle identifier to watch* dialog to enter the name of the function that processes messages for the desired window (such as `WndProc` or another function). If your application is written in ObjectWindows, you need to configure the TDW using `TDINST` (see the section "Turbo Debugger for Windows Command-Line Options"). After the ObjectWindows option is set, you can directly reference ObjectWindows objects.

To add an ObjectWindows object, it is easiest if you set a breakpoint in the routine that creates the window you wish to examine. Halt the program on the line following where the window is initialized and then use the Windows message's submenu to add the window variable. For example, to log messages to the two windows initialized in the code sample in Listing 11.3 (from an ObjectWindows application), you should stop on line 4 (for `MainWindow`) or line 5 (for either `MainWindow` or `TheWindow`). On the line after the window you wish to add, add either variable to the message log.

LISTING 11.3. A SAMPLE OBJECTWINDOWS INITIALIZATION ROUTINE.

```
1   void TRadioApp::InitMainWindow()
2  {
3  MainWindow = new THamWindow(NULL, Name);
4   TheWindow = (PTWindow) MainWindow;
5 }
```

Move the cursor to the message class subwindow of the message viewer and press Alt-F10. A menu pops up from which you can select the message type, or all messages. I recommend that you select a particular class of messages rather than all messages. When all messages are intercepted, your application's speed will be somewhat slower than a sloth on a cold day. If you wish to intercept more than one message class, you need to perform a little trick, because the message

intercept tracks messages from only one message class at a time. For example, if you want to trap both System and Mouse messages, add your window (say, `MainWindow`, in the preceding example) *twice*. That is, add `MainWindow` to the message tracking list twice and set one instance to System messages and the other to Mouse messages.

The message interceptor can also track a specific message, such as `WM_PAINT`. This requires that you type the message identifier into a field in the dialog box. Using the message class submenu and the Add... dialog, you can also elect to break your program's execution upon receipt of a specific message. This can prove invaluable when you are trying to isolate a problem.

Next, run your application. You will notice a performance degradation as the internal Windows messages are checked and logged. When your program stops at a breakpoint or finishes its execution, you can look at the messages that were transmitted to the various views using the View | Windows message windows.

USING WINSIGHT

Another way you can watch the message traffic inside your application and other applications is by using the Winsight utility program. Winsight, like a government intelligence agency, spies on Windows messages from any active window that you select. Winsight is not part of Turbo Debugger for Windows, but is instead a completely separate utility that you can launch before or during your program's execution. Winsight's Windows icon is displayed in the Borland C++ group of the Program Manager.

When Winsight is executing, it traces all or selected messages to designated windows. Figure 11.13 shows sample output. Winsight, unlike TDW, enables you to easily trap all messages, individual message classes, or selected message classes. I will not detail much of Winsight because it's very easy to use. Almost everything you need to know you can find by experimenting or using Winsight's help messages. Use Spy | Find Window, and then a mouse click on the window whose messages you wish to intercept. Select Message | Options... to choose the type of messages you wish to intercept. Using the Options dialog, you can elect also to have the trace logged to a disk file.

TURBO DEBUGGER FOR WINDOWS COMMAND-LINE OPTIONS

TDW has a number of command-line options, which is especially interesting because you cannot run it at the command line! What you can do is use the Run | Debugger arguments menu selection to set command-line options (or use the File | Properties... command in the Windows Program Manager). These command-line options are shown in Table 11.1.

TABLE 11.1. COMMAND-LINE OPTIONS TO TURBO DEBUGGER FOR WINDOWS.

Option	*Usage*
`-?`, `-h`	Display TDW command-line options
`-c`*`filename`*	Use *`filename`* as the configuration file.
`-do`	Display TDW on a secondary monochrome display.
`-ds`	Swap screens using page swapping.
`-l`	Begin debugging in assembly language, showing DLL startup code.
`-p`	Default setting; select use with a mouse.
`-sc`	Ignore the case of symbol names.
`-sd`*`dir1`*`;`*`dir2`*	Set one or more default directories to locate the source files.
`-t`*`directory`*	Set the default starting directory.

Before you set up command-line options, however, you may want to use the Turbo Debugger installation program TDINST. You can use this program with TDW by adding the `-w` command-line option like this:

```
C:\BORLANDC\BIN> TDINST -w
```

In this form, you can make installation changes to TDW. You should use TDINST to configure TDW to work with ObjectWindows if you are using the ObjectWindows class libraries. When you run TDINST, select the Options | Source debugging selection, and enable the OWL windows messages on the dialog box (see Figure 11.14). Then choose Save | Modify TDW.EXE to save your changes.

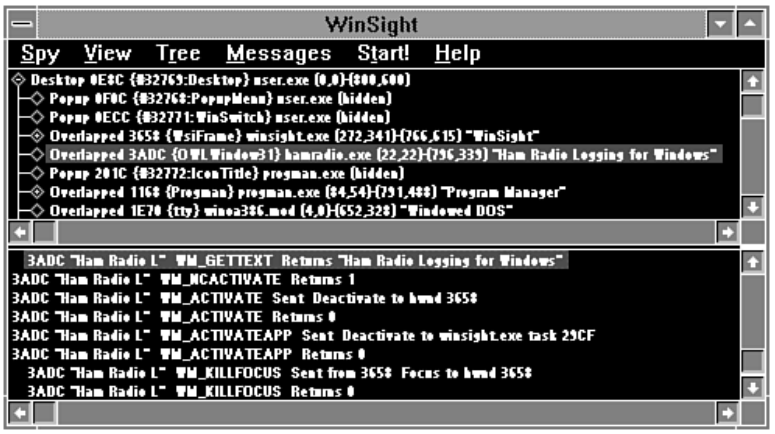


Figure 11.13. The Winsight message intercept program.

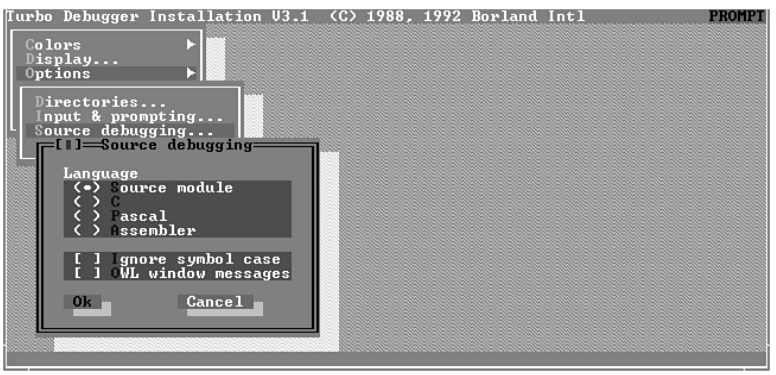


Figure 11.14. The Source debugging dialog box.

OTHER DEBUGGING FEATURES

Turbo Debugger (and Turbo Debugger for Windows) programs have additional features that may prove useful to you. When debugging using the View | CPU mode of operation, you see machine-level or assembly language instructions. You may optionally insert new assembly instructions by pressing Alt-F10 to

display the CPU window's local submenu. From this menu, choose the Assemble... menu item. You can then use this dialog to enter individual assembly instructions that will be temporarily inserted into your program's instruction team. These instructions do not become a permanent part of your code file, but they can be used as a quick way to test programming ideas.

If you have more than one computer available, you can use the remote debugging facilities to run Turbo Debugger (or TDW) on one computer while the program under test is executed on another system. The two computers communicate between each other over a serial cable link or via a local area network. Remote debugging can be useful when you are unable to run both Turbo Debugger and your application on the same system due to a shortage of memory. If you can, it is usually much simpler to use EMS memory on a single system than to perform remote debugging. The programs TDREMOTE and WREMOTE are the primary programs used for remote debugging.

Lastly, when you debug applications on a remote system, you will have to transfer files back and forth. You may be able to do that over a network or via a floppy disk, or you may be able to use the TDRF remote file transfer utility included with Borland C++.

This concludes the overview of debugging methods and tools. The Borland C++ package has quite a few features to help you build reliable programs (such as pretested libraries and class libraries). When things go wrong, you can use the built-in debugger, Turbo Debugger, or Turbo Debugger for Windows to isolate and identify defective code.

CHAPTER 12

PROGRAM OPTIMIZATION AND TURBO PROFILER

If you've ever gone shopping for a new home, you know the lowest priced house meeting your needs always costs about 20% more than you can afford. Computer programs and execution speed are kind of like homes and home prices. Completed programs always run too slow and use about 20% (or 30% or 40% . . .) more memory than is available. A Murphy's law of programming states "Computer programs always grow to exceed the available memory," with the obvious corollary that "No matter how much memory is available, programs always need more."

Compiling for Turbo Profiler compatibility

Selecting program areas to profile

Improving the program

Using better algorithms

Using fixed-point arithmetic in place of `float` *data-types*

This chapter examines techniques for analyzing and improving performance, plus suggestions to reduce memory requirements. In real-life programming, however, peak performance and reduced memory requirements are often traded off: The faster the program, the more memory it requires. However, many fundamental issues, once addressed, improve both speed and memory requirements. This chapter examines program optimization, use of Turbo Profiler to locate ideal optimization points, and memory reduction.

PROGRAM OPTIMIZATION

To make many programs run faster, employ simple improvements. By closely examining program source, you can sometimes translate simple improvements into significant speed-ups. The trick is to identify where such improvements can have the most profound effect. You do little good speeding up a routine the program barely uses. Instead, target improvements to small sections of a program or subroutine where execution spends most of its time.

Surprisingly, perhaps, many programs spend a large percentage of time confined to narrow sections of code. For example, a program compiler, which translates a source program into machine code, might spend the largest percentage of its execution reading characters from the program source. In one Pascal compiler that I looked at, changing just three lines of code in the compiler's lexical analyzer resulted in a 10% speedup of the entire compiler!

Obviously, the key to program optimization is identifying which few statements, out of perhaps thousands of source statements, are the underlying bottleneck. Here is where the Turbo Profiler comes into use. Turbo Profiler takes a picture of a running program's *execution profile*, providing statistics indicating which sections of the program use the most CPU clock cycles. Based on Turbo Profiler's output, you identify the program locations that can most benefit from optimization techniques.

You can use Turbo Profiler to compare different algorithms and different implementations. As a side effect of Turbo Profiler's output, you also can use the Profiler to help with program testing. With Turbo Profiler, make a record of each time a particular program statement is executed. Using this record, determine if sufficient tests have been designed to ensure that every program

statement is executed at least once. If not, you have a clue that the tests are inadequate, the program logic is flawed, or unneeded code is in the program.

For 80386/80486 CPU system owners, Turbo Profiler, like Turbo Debugger, can run your program in virtual 8086 mode, providing substantially more memory to profile your application. While not described here, the Turbo Profiler also can profile a program running on a remote PC. See the *Turbo Profiler Users' Guide* for instructions.

USING THE TURBO PROFILER

Assuming Turbo Profiler is installed in the default directory, `\borlandc\bin`, run Turbo Profiler by typing

```
TPROF programname
```

or

```
TPROF
```

and then choose the File | Open command to access the program. Figure 12.1 shows the Turbo Profiler screen after loading Profile1.exe, the sample program in Listing 12.1. This program is subjected to profile analysis in this chapter.

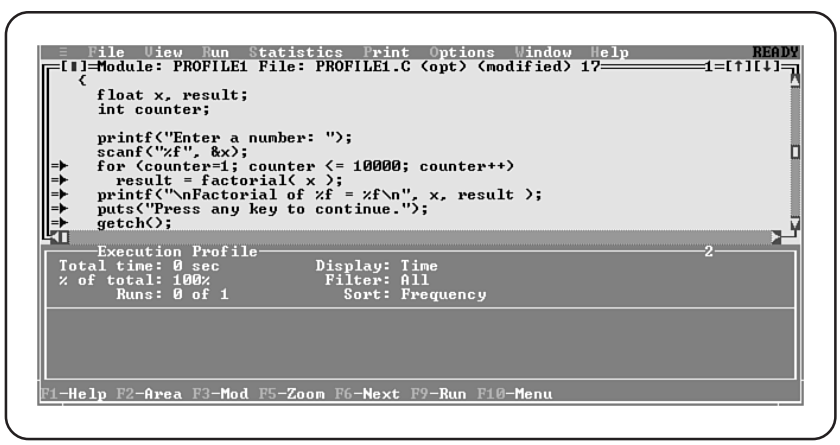


Figure 12.1. The Turbo Profiler screen.

LISTING 12.1. AN EXAMPLE PROGRAM TO BE TESTED WITH THE PROFILER.

```
 1  /* PROFILE1.C
 2     Demonstration program for use with Turbo Profiler.
 3  */
 4  #include <stdio.h>
 5  #include <conio.h>
 6
 7  float factorial ( float n )
 8  {
 9    if (n == 1.0)
10      return 1.0;
11    else
12      return n * factorial( n - 1.0 );
13  }
14
15  void main( void )
16  {
17    float x, result;
18    int counter;
19
20    printf("Enter a number: ");
21    scanf("%f", &x);
22    for (counter=1; counter <= 10000; counter++)
23      result = factorial( x );
24    printf("\nFactorial of %f = %f\n", x, result );
25    puts("Press any key to continue.");
26    getch();
27  }
```

COMPILING FOR TURBO PROFILER COMPATIBILITY

The Turbo Profiler uses the same information required by the debugger. Therefore, to use the Turbo Profiler, you must compile your program with the debugger symbol table information enabled (see "Compiling for Turbo Debugger Compatibility" and "Compiling with the Command line Compiler" in Chapter 11, "Debugging Techniques"). Set the compiler's optimization features as you desire for your final release (select Options | Compiler | Optimizations...).

If you inadvertently compile with the integrated debug information rather than the Standalone debug information, Turbo Profiler is unable to access your

program's source code during the profiling process. Source access is pretty much required to make sense of the Turbo Profiler's output, so be certain to set this option as required.

As this book goes to press, a few minor features of the Turbo Profiler may not work properly unless your program is compiled using the large memory model. Future versions of the Turbo Profiler will fix these problems, but for now I recommend that you compile programs for profile analysis using the large memory model.

SELECTING PROGRAM AREAS TO PROFILE

The first step in analysis is to identify the sections or areas of your program to be analyzed. Identification consists of marking only those lines, or functions, to be subjected to analysis. Generally, you do not include all program statements during a single profile run. Not only does this slow down program execution, but also inclusion of input statements, such as `scanf()`, can easily skew the results, because the time it takes to type in a response can be hundreds or thousands of times longer than subsequent calculations. The inclusion of a `scanf()` results in Turbo Profiler telling you that your program spends most of its time doing `scanf()`s! This is probably not the result you are seeking.

IMPORTANT: OBTAINING ACCURATE MEASUREMENTS

In the sample program Profile1, note that the call to factorial() is repeated 10,000 times. If you measure a single call to factorial(), the routine runs so fast that the Profiler's timing measurement is inaccurate. For this reason, when using the Profiler on short bits of code, you might need to insert a looping mechanism to repeat the item enough times to ensure the Profiler's measurement is reasonably accurate. If you are comparing two nearly identical implementations, be certain you compare matching items. It's easy to select a slightly different set of lines in one program than in another.

continues

continued

Before setting the program areas to be profiled, the safe choice is to remove all lines from analysis and then put back only those to be examined. To do this, make the Module window the active window by clicking the mouse in the Module window or pressing F6 to move to the Module window. Press Alt-F10 to display the Module window's local menu (see Figure 12.2). Select Remove Areas to display a second menu (see Figure 12.3). Choose All Areas from this menu to remove profiling from all program locations.

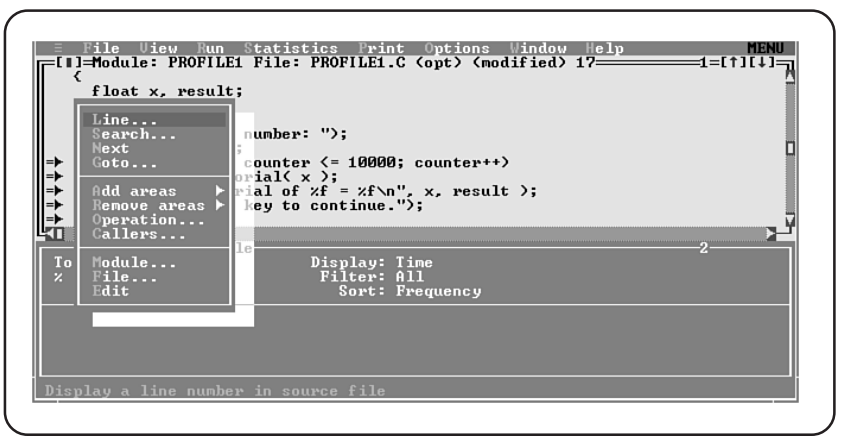


Figure 12.2. The Module window's local menu.

To analyze Profile1, move the cursor to the start of the `factorial()` function, press Alt-F10, and select Add Areas. From the submenu, choose Lines in routine, which marks all the lines in the `factorial()` function. To select individual lines, move the cursor to the desired line and press F2.

Turbo Profiler provides several convenient methods of marking areas for profile analysis. To select all the lines within a procedure or function, move the source cursor to some point within the procedure or function. Press Alt-F10 and select Add Areas. From the submenu, choose Lines in routine, which marks all the lines in the function.

To select all procedures within a module, select Add areas and choose Routines in module, which marks every function in the program for statistics

gathering. (Runtime statistics are gathered for the entire function, not for individual lines.) You can select a single function by moving the cursor to any line inside the function and choosing Current routine (or pressing Alt-F2). This procedure selects the function, but not the lines inside the function.

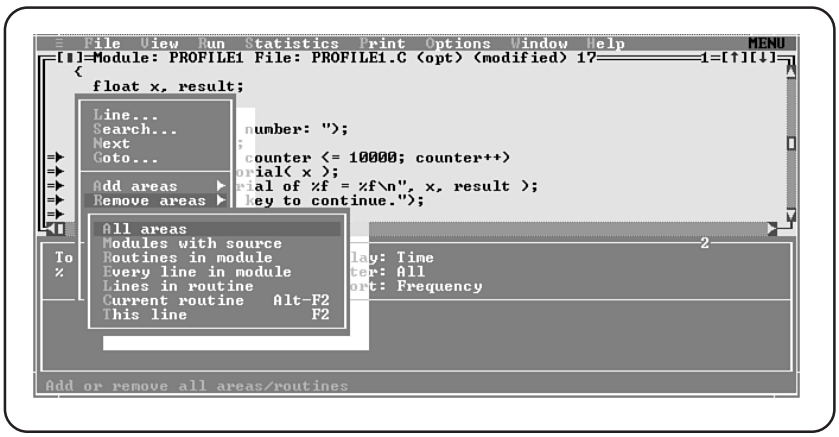


Figure 12.3. The submenu for Remove areas.

To execute the program, select the Run menu's Run option (or press F9). Enter an appropriate number for factorial computation (I used 6 for these examples). With profiling in effect, execution takes some time. Depending on the speed of your computer, you might have to wait several minutes. If you get worried, pressing Ctrl-Break halts the program under review and returns control to the Profiler. After the program runs, control returns to the Profiler and displays the results in the Execution Profile window, just below the Module window (see Figure 12.4).

The number of seconds and percentage of execution time spent on each line is shown within the Execution Profile window. `#PROFILE1#12` refers to line 12 of module `PROFILE1`, which is the statement containing

```
return n * factorial( n - 1.0 );
```

According to the Profiler, this statement takes up 42% of the program execution time (this can be a much larger value depending on the initial value of `n`). Interestingly, the simple `if( n == 1.0)` statement in line 9 also takes up 42% of the execution time. Improving these statements has the greatest impact on overall program performance.

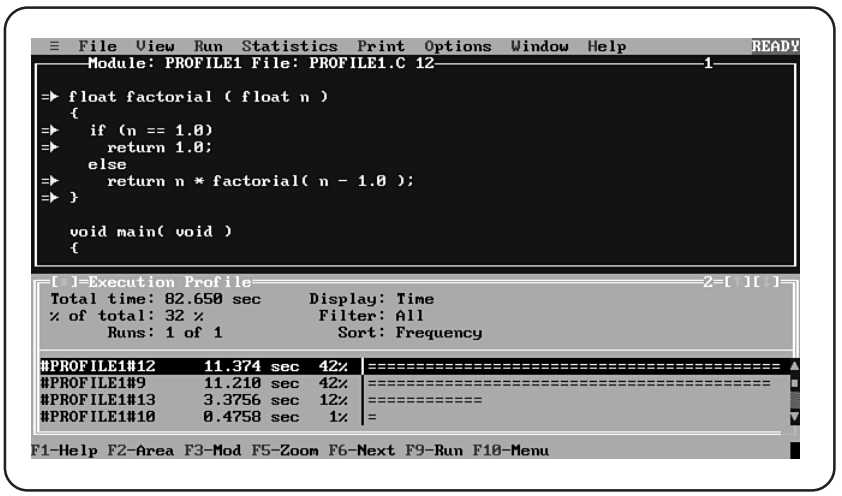


Figure 12.4. The results of Turbo Profiler's analysis of Profile1.c.

To switch back and forth between the Module and Execution Profile windows, press the F6 key, click in the desired window using the mouse, or choose the desired window from the Window pull-down menu. When scrolling through the Module window, the Execution Profile window automatically adjusts to display the data corresponding to the source line where the cursor is located.

IMPROVING THE PROGRAM

With the results of the first profile run, identify an area where an improvement has great potential to speed up the program significantly. Speed improvements usually come in one or two flavors:

- *Improved, more efficient implementations.* The existing code is tweaked so that it runs quicker. The next section addresses ideas for improving the implementation.
- *Better algorithms.* Often, a problem can be solved more than one way. By selecting a better algorithm, you may improve the program's execution speed.

For the `factorial()` function, first consider improving the basic implementation. Whether your PC system contains an 80x87 math coprocessor, floating-point real arithmetic almost always is slower than integer arithmetic. Consequently, changing the data type from `float` to an integer format can speed up program execution, provided that the integer type can handle the required range of values. To see the effect this might have, change the `TData` type assignment from `float` to `long` (see Listing 12.2) and change `%f` to `%ul` (shown in lines 23 and 26).

LISTING 12.2. THE PROFILE1 PROGRAM, MODIFIED TO USE LONG DATA TYPES INSTEAD OF `float` DATA TYPES.

```
 1  /* PROFILE2.C
 2     Demonstration program for use with Turbo Profiler.
 3  */
 4  #include <stdio.h>
 5  #include <conio.h>
 6
 7  typedef long TData;
 8
 9  float factorial ( TData n )
10  {
11    if (n == 1)
12      return 1;
13    else
14      return n * factorial( n - 1 );
15  }
16
17  void main( void )
18  {
19    TData x, result;
20    int counter;
21
22    printf("Enter a number: ");
23    scanf("%lu", &x);
24    for (counter=1; counter <= 10000; counter++)
25      result = factorial( x );
26    printf("\nFactorial of %lu = %lu\n", x, result );
27    puts("Press any key to continue.");
28    getch();
29  }
```

After compiling, load and run the modified program Profile2 in Turbo Profiler. Press Alt-F10 to display the Module window's local menu. Select

Remove areas and then choose All areas from the submenu. Move the cursor to the statement after the `for` loop, where `factorial()` is assigned to `result`. Press `F2` to mark this line. Then run the program. The Execution Profile window displays the time used to execute these calls to `factorial()`. The result is shown in Figure 12.5.

By comparing the result of Profile2 to Profile1, you see that the use of `long` data types for the computation provides an almost fivefold improvement in execution speed. The actual times vary, of course, depending on the type CPU in use, CPU clock speed, and other factors.

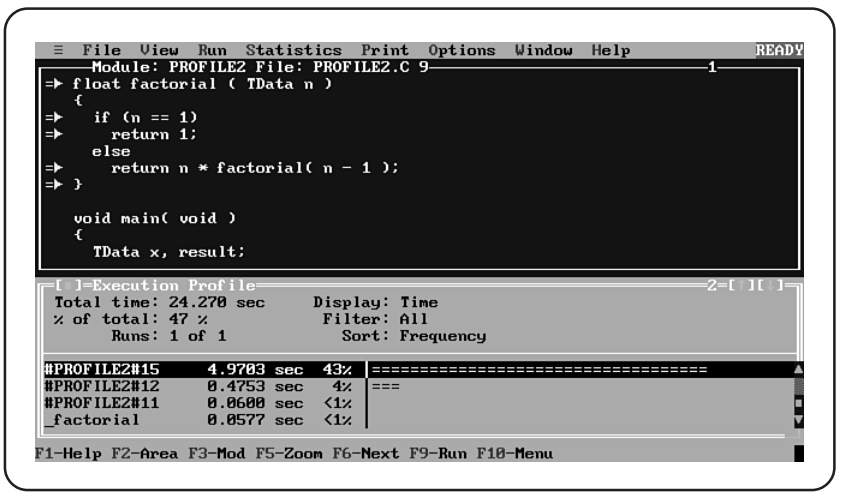


Figure 12.5. The time analysis of Profile2, showing that the use of `long` data types speeded the program's execution by almost a factor of five.

IMPORTANT: THE TOTAL TIME FIELD

The Total time field at the top of the Execution Profile window contains the total execution time of the program, not the sum of the statements under review. If your program contains an input statement, the time spent typing a response is included in the Total time field. Consequently, the

Total time field is highly inaccurate for many applications. Instead, refer to the times shown next to the specific statements in the Execution Profile window.

Another potential improvement is to look at alternative implementations of the factorial algorithm. Computing a factorial does not require a recursive algorithm; therefore, the overhead of the recursive function call may be eliminated. Listing 12.3 shows a new implementation that uses a simple `for` loop rather than a recursive function call. When Profile3 is loaded and profiled in Turbo Profiler, the total execution time is about *50 times* faster than the original Profile1.c program. The timing information is shown in Figure 12.6.

LISTING 12.3. A NONRECURSIVE IMPLEMENTATION OF `factorial()`.

```
 1  /* PROFILE3.C
 2  */
 3  #include <stdio.h>
 4  #include <conio.h>
 5  #include <math.h>
 6
 7  typedef long TData;
 8
 9  float factorial ( TData n )
10  {
11    TData result;
12    long count;
13
14    result = 1.0;
15    for( count=2; count <= n ; count++)
16      result = result * count;
17    return result;
18  };
19
20
21  void main( void )
22  {
23    TData x, result;
24    int counter;
25
26    printf("Enter a number: ");
27    scanf("%lu", &x);
```

continues

LISTING 12.3. CONTINUED

```
28    for (counter=1; counter <= 10000; counter++)
29      result = factorial( x );
30    printf("\nFactorial of %lu = %lu\n", x, result );
31    puts("Press any key to continue.");
32    getch();
33  }
```

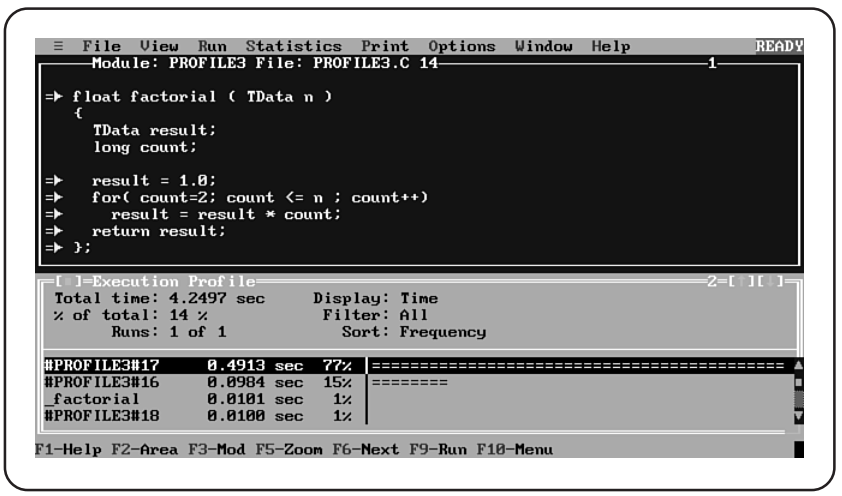


Figure 12.6. The timing results of the nonrecursive implementation of `factorial()`.

STATISTICS PROVIDED BY TURBO PROFILER

Besides determining which routines require the greatest percent of execution time, Turbo Profiler can examine program execution in other ways. These other analysis options are selected on the Execution Profile's local menu. To access these options, move to the Execution Profile window, press Alt-F10, and choose Display from the pop-up menu. This operation presents a Display Options dialog box (see Figure 12.7).

Use the Display Options dialog box to configure the type data shown in the Execution Profile window. These options are as follows:

- *Time*. Turbo Profiler displays the percent of time spent in each area being profiled.
- *Counts*. Execution counts are provided to count the number of times that a particular section of code is executed. A time-consuming operation is not necessarily one that is often executed. After all, a single statement might call a routine that runs for seconds.
- *Both*. When this option is in effect, the Execution Profile window displays both time and count information. Figure 12.8 shows the Execution Profile output when displaying both Time and Count information.
- *Per call*. Turbo Profiler can provide the average time required to run the functions under analysis. Keep in mind that, depending on the content of the function plus changes in the parameters, a given function can vary greatly in its execution time. The `factorial()` function is a good example: `factorial()` takes much longer for large values of `n` than for small values of `n`.
- *Longest*. This selection displays the longest time used by the area being profiled.
- *Modules*. Displays the total amount of time spent in each module.

Figure 12.7. The Execution Profile's display options.

TURBO PROFILER OUTPUT OPTIONS

Turbo Profiler provides a number of options for printing and saving the analysis information, available from the Print pull-down menu. Use the Print menu to copy the Execution Profile data to a printer or a file. Select file output by choosing Print | Options... and selecting the File radio button.

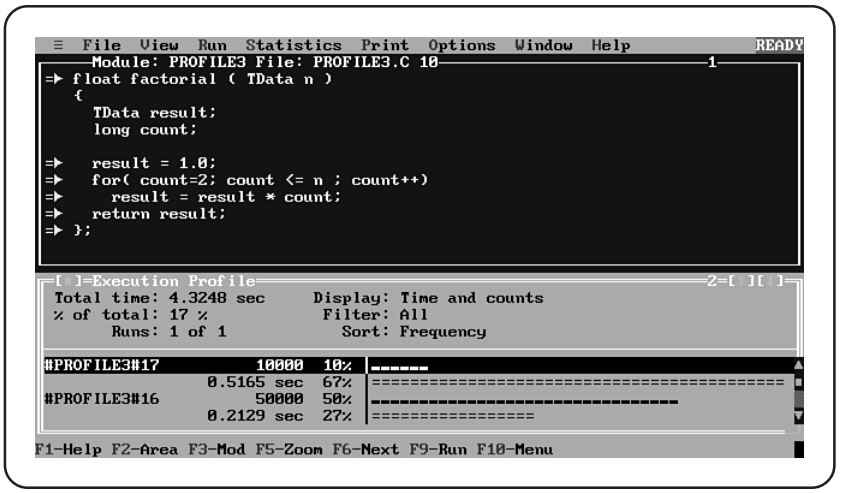


Figure 12.8. Displaying both time and count analysis data in the Execution Profile window.

The Statistics | Save... and Statistics | Restore... options save and restore the collected profile statistics. Use this feature to compare program execution before and after making implementation or algorithm improvements or for accumulating statistics over several program runs.

ACTIVE VERSUS PASSIVE PROFILING

Turbo Profiler has two modes of operation, *active* and *passive*. In active mode, Turbo Profiler collects execution statistics when it executes each and every program source line marked for profiling. Consequently, active mode significantly slows down overall program execution for the areas under analysis.

To reduce the impact of the profiling process, Turbo Profiler provides a second operational mode called *passive analysis*. In this mode, Turbo Profiler does not collect statistics for each marked line. Instead, the Profiler uses system clock ticks to generate profiling interrupts. At each clock tick, the Profiler evaluates the content of the `CS:IP` registers to determine where your program is currently executing. (If `CS:IP` points outside the area under analysis, the value is

disregarded.) Over time, this process produces a statistical sample of your program's operation. If your program executes for a sufficient length of time, a statistical sample usually provides an accurate representation of the percentage of time spent in each program section.

The big advantage to passive analysis is that your program runs at nearly full speed. A disadvantage is that line count statistics and some other values are not kept when monitoring in passive mode. Also, if the program's execution of the area being examined is too quick, the data sample collected may not be representative of the program's actual execution. For this reason, passive analysis works best for program sections executed many times. If necessary, Turbo Profiler can automatically run your program over and over by specifying a Run count value in the Profiling options dialog box. When the Run Count is greater than 1 (with a maximum of 20 allowed), the resulting statistics from each execution are added or *accumulated* together.

To select active versus passive program analysis, choose Profiling options... from the Statistics menu. In the Profiling options dialog box (see Figure 12.9), select either the Active or Passive radio button. Specify a run count value in the Run count field.

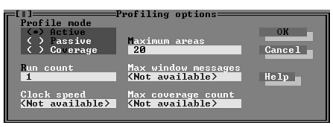


Figure 12.9. The Profiling options dialog box.

The Profiling options dialog also has an option to perform *coverage analysis*. Enabling the coverage analysis mode replaces the Execution Profile window by the Coverage window. This function determines which areas of your program are encountered by a run through the code. After you execute your program, the default display of the Coverage window shows those lines that have *not* been executed. Use this analysis information to determine which areas of your code might be superfluous or, perhaps, are not executed because of a faulty conditional test. You can run the program more than once to accumulate information on each pass through the program, changing program options on each run. The Coverage test is especially helpful when testing your program's operation. If you have code left unexecuted, your program may be defective. It's possible that the internal program logic is faulty and should be fixed.

OPTIMIZATION TRICKS

Remember, the first step in profiling is to identify the general areas of your program that use most of the execution time. Refine the profiling by narrowing down the scope until you reach areas that are likely candidates for improvements.

Start your program analysis by using Turbo Profiler to identify the routines with which you spend the most time. To set up Turbo Profiler to analyze the program by function, do the following:

1. Move to the Module window and press Alt-F10.
2. Choose Remove areas and then select All areas on the submenu.
3. Choose Add areas and then select Routines in module.

Run your program and review the Execution Profile to determine which procedures and functions are the most time-consuming.

Turbo Profiler limits how many sections or areas you can analyze at one time, so you may need to limit your profile analysis to a subset of all the functions in your module. (Increase the default maximum by changing the Maximum Areas field in the Statistics | Profiling options dialog box.) Further, the more areas you analyze at once, the slower your application runs while under control of Turbo Profiler. Hence, it's essential to focus the analysis to targeted areas.

Once you identify time-consuming functions, the next step is to mark the lines within functions needing closer analysis.

1. Move to the Module window and press Alt-F10.
2. Choose Remove areas and then select All areas on the submenu.
3. Press Esc to leave the local menu.
4. Move the cursor to each line in the program source that you wish to have profiled. Press F2 to mark the line. Repeat as needed to mark the necessary lines. To select all the lines in a procedure or function, move the cursor to the start of the routine and press Alt-F10. Select Add areas and then choose Lines in routine.

Keep refining the analysis until you identify the key program areas. Once you locate problem spots, determine possible code improvements. This section contains a number of suggestions for improving program performance.

CLEANING UP LOOP STATEMENTS

Loops contain code sections you execute repeatedly. As a result, your programs tend to spend much time within the `for`, `while`, or `do-while` looping constructs. Improvements made inside a loop, even small ones, magnify by the number of times the loop is executed.

The key to improving a loop, as with any improvement, is to reduce the amount of code that must be executed. Identify calculations that do not change inside the loop; move these outside and assign to a temporary variable for use inside the loop. For example, the `for` loop shown in this code section performs an expensive trigonometric calculation during each execution through the loop:

```
for( item = 1; item <= MaxValues; item++ )
{
  X = floor( 1.1 * Radius[item] * cos ( AngleInRadians ) );
  Y = - floor( 1.1 * Radius[item] * sin ( AngleInRadians ) * (Xasp/Yasp));
  /* Compute (EndX, EndY) 40% further out than (X,Y) */
  EndX = floor(X * 1.4);
  EndY = floor(Y * 1.4);
  line (X, Y, EndX, EndY );
};
```

By rewriting this code fragment, the time-consuming calculation can be moved outside the loop and stored in a temporary variable. Often, the compiler's optimization features catch this type of duplicate code and create a temporary variable automatically.

```
TempMultX = 1.1 * cos (AngleInRadians);
TempMultY = 1.1 * sin(AngleInRadians) * (Xasp/Yasp);
for( item = 1; item <= MaxValues; item ++ )
{
  X = floor( TempMultX * Radius[Radius] );
  Y = - floor( TempMultY * Radius[Radius] )
  /* Compute (EndX, EndY) 40% further out than (X,Y) */
  EndX = floor(X * 1.4);
  EndY = floor(Y * 1.4);
  line (X, Y, EndX, EndY );
};
```

In rare instances, you might make the loop control variable a register variable. Normally, it's best to let the compiler's optimization strategy determine which values to place in registers. (The 80x86 CPUs don't have a lot of registers to spare.) For especially tight situations in straightforward code, you might want

to experiment with the use of the `register` keyword on the control variable. Use the Turbo Profiler to measure the performance and determine if the use of the register variable is useful.

Duplicate calculations also should be identified. Avoid repeated calculations of values that you can place in a temporary location. For example,

```
if  ((EndPos-StartPos+1) < MaxAllowed)
   copy( S, StartPos, EndPos-StartPos + 1 );
```

Optimize this statement by writing the following:

```
amount = EndPos - StartPos + 1;
if  (amount < MaxAllowed)
   Copy( S, StartPos, amount );
```

TEST FOR THE MOST LIKELY OUTCOMES FIRST

In conditional expressions, and especially in `while` loops, keep the number of items in the conditional expression to a minimum. Structure each condition to take advantage of short-circuit expression evaluation technology. A short-circuited expression is one in which you can determine the outcome before evaluating the entire expression. When the compiler optimizes the generated machine instructions, it can recognize this situation and produce more efficient code, for example, in the following conditional expression.

```
while( prime ¦¦ (I < MaxNumber)
{
    ...
    prime = (I % J) == 0;
};
```

If `prime` is nonzero, the code for `(I < MaxNumber)` is not evaluated. As soon as `prime` is nonzero, the outcome of the overall expression is known, and usually you do not need to continue evaluating the remainder of the statement. The exception to this rule is when the subsequent expression calls a function that has a side effect such as setting a global flag.

You also should apply testing for the most likely condition first to `if-then` conditionals anywhere inside the loop.

SET COMPILER OPTIONS FOR MOST EFFICIENT EXECUTION

A large number of the compiler's options affect the efficiency of generated code. How these options are set can greatly influence the execution time of compiled programs. Refer to "Generating the Fastest and the Smallest Code" in Chapter 2, "Power Features of the IDE and Borland C++."

REPLACE FUNCTION CALLS WITH LOOKUP TABLES

If possible, replace function calls with lookup tables, particularly when the calculation is time-consuming. Consider the following evaluation:

```
Y = sin (Angle);
```

If `Angle` is in degrees, you can create a precomputed table of values to store the sine of each angle from, say, 1 to 90. The assignment then can index into the array as

```
Y = SinT [Angle];
```

and avoid an expensive trigonometric calculation. Listing 12.4 provides an example implementation. The table lookup, in this example, executes nearly 170 times faster than calling the `sin()` function! This operation runs even faster when using `long` values to represent fixed point numbers. (See "Use Fixed Point Arithmetic in Place of *float* Data Types" later in this chapter.)

LISTING 12.4. AN EXAMPLE OF A TABLE LOOKUP TO REPLACE AN EXPENSIVE CALCULATION.

```
1  /* SINT.C
2  Example using a table of sine values.
3  */
4  #include <stdio.h>
5
6  float SinT[91]       = { 0.0,
7  1.7452406437E-02,  3.4899496702E-02,  5.2335956243E-02,  6.9756473744E-02,
```

continues

```
 8  8.7155742748E-02,  1.0452846327E-01,  1.2186934340E-01,  1.3917310096E-01,
```

LISTING 12.4. CONTINUED

```
 9  1.5643446504E-01,  1.7364817767E-01,  1.9080899538E-01,  2.0791169082E-01,
10  2.2495105434E-01,  2.4192189560E-01,  2.5881904510E-01,  2.7563735582E-01,
11  2.9237170472E-01,  3.0901699437E-01,  3.2556815446E-01,  3.4202014333E-01,
12  3.5836794955E-01,  3.7460659342E-01,  3.9073112849E-01,  4.0673664308E-01,
13  4.2261826174E-01,  4.3837114679E-01,  4.5399049974E-01,  4.6947156279E-01,
14  4.8480962025E-01,  5.0000000000E-01,  5.1503807491E-01,  5.2991926423E-01,
15  5.4463903501E-01,  5.5919290347E-01,  5.7357643635E-01,  5.8778525229E-01,
16  6.0181502315E-01,  6.1566147533E-01,  6.2932039105E-01,  6.4278760969E-01,
17  6.5605902899E-01,  6.6913060636E-01,  6.8199836006E-01,  6.9465837046E-01,
18  7.0710678119E-01,  7.1933980034E-01,  7.3135370162E-01,  7.4314482548E-01,
19  7.5470958022E-01,  7.6604444312E-01,  7.7714596146E-01,  7.8801075361E-01,
20  7.9863551005E-01,  8.0901699437E-01,  8.1915204429E-01,  8.2903757255E-01,
21  8.3867056795E-01,  8.4804809616E-01,  8.5716730070E-01,  8.6602540378E-01,
22  8.7461970714E-01,  8.8294759286E-01,  8.9100652419E-01,  8.9879404630E-01,
23  9.0630778704E-01,  9.1354545764E-01,  9.2050485345E-01,  9.2718385457E-01,
24  9.3358042650E-01,  9.3969262078E-01,  9.4551857560E-01,  9.5105651630E-01,
25  9.5630475596E-01,  9.6126169594E-01,  9.6592582629E-01,  9.7029572628E-01,
26  9.7437006479E-01,  9.7814760073E-01,  9.8162718345E-01,  9.8480775301E-01,
27  9.8768834059E-01,  9.9026806874E-01,  9.9254615164E-01,  9.9452189537E-01,
28  9.9619469809E-01,  9.9756405026E-01,  9.9862953475E-01,  9.9939082702E-01,
29  9.9984769515E-01,  9.9999999999E-01 };
30
31  void main( void )
32  {
33    float Y;
34    int i;
35    for( i=1; i<=5000; i++)
36       Y = SinT[45];
37    printf("Sin(45)=%f\n", SinT[45] );
38  }
```

DON'T BE AFRAID OF *GOTO*

Many programmers religiously avoid using `goto` statements, relying instead on setting flags which are then tested in conditional expressions such as `while` loops. Often, with a single forward jumping `goto` to escape from a deeply nested looping structure, you can improve program performance and sometimes improve program readability as well.

USE BETTER ALGORITHMS

Though obvious, seeking a better algorithm is one alternative often overlooked. Consider the example of a simple prime number program shown in Listing 12.5. This program uses a simple method of dividing the number being checked for primality by values less than the candidate number, which is not the best algorithm, nor is it a particularly good implementation. A variety of changes could improve the program. For example, in line 25, you only need to increase the divisor up to the square root of `NumToCheck`.

LISTING 12.5. A SIMPLE PROGRAM TO CALCULATE A SET OF PRIME NUMBERS.

```
1  /* PRIME1.C
2    One method of calculating prime numbers.  Prime2 illustrates how
3    using a goto statement can improve the performance of this program.
4    Prime3 shows how using an entirely different algorithm can improve
5    performance.
6  */
7  #include <stdio.h>
8  #include <conio.h>
9
10  #define MaxPrimes 4000
11  #define False 0
12  #define True 1
13
14  void main( void )
15  {
16    unsigned NumToCheck, Divisor;
17    int Prime;
18
19    printf("\nPrimes from 3 to %d\n", MaxPrimes);
20    NumToCheck = 3;
21    while( NumToCheck <= MaxPrimes)
22    {
23      Divisor = 2;
24      Prime = False;
25      while( !Prime && (Divisor < NumToCheck))
26      {
27        Prime = (NumToCheck % Divisor) == 0;
28        Divisor++;
29      };
30      if (!Prime) printf("%d  ", NumToCheck);
31      NumToCheck += 2;
32    };
33    getch();
34  }
```

You can tweak the code in Listing 12.5 a number of ways to improve its execution speed. However, these tweaks produce only incremental improvements. By changing the approach entirely, I was able to write a quick program that runs about five times faster than the solution in the previous section.

This new program, shown in Listing 12.6, avoids division, an expensive operation, even for integer or word values. Instead, the program creates a map with one element set for each potential prime number. Then, beginning with the value 3, the program clears each multiple of the first prime number, because multiples of a prime number cannot be prime. Starting with 3, this clears `Primes[6]`, `Primes[9]`, `Primes[12]`, `Primes[15]`, `Primes[18]`, `Primes[21]`, and so on.

The next prime value, 5, clears the entries at `Primes[10]`, `Primes[15]`, `Primes[20]`, `Primes[25]`, and so on. After this comes 7 and then 9. For 9, the program checks to see that `Primes[9]` is not already primed, because it clears as a multiple of the first prime number, three. Hence, no processing is necessary; the program advances to 11, and so on.

The overall result is that the calculation of prime numbers using the bitmap approach is five times faster than when using conventional arithmetic! No amount of code twiddling of the previous prime number programs could achieve this performance improvement.

LISTING 12.6. A PRIME NUMBER PROGRAM THAT AVOIDS DIVISION ARITHMETIC.

```
1  /* PRIME2.C
2     Uses an array to calculate a set of prime numbers.
3  */
4  #include <stdio.h>
5  #include <conio.h>
6  #include <mem.h>
7
8  #define MaxPrimes 4000
9  #define False 0
10
11 void main( void )
12 {
13   unsigned char Primes[MaxPrimes];
14   unsigned NumToCheck, I;
15
16   memset( Primes, 1, sizeof(Primes) );
17
18   NumToCheck = 3;
```

```
19    while( NumToCheck < MaxPrimes)
20    {
21      if  (Primes[NumToCheck])
22      {
23        I = NumToCheck + NumToCheck;
24        while( I < MaxPrimes )
25        {
26    Primes[I] = False;
27    I += NumToCheck;
28        };
29      };
30      NumToCheck += 2;
31    };
32
33    I = 3;
34    while( I < MaxPrimes )
35    {
36      if (Primes[I]) printf("%d  ", I);
37      I += 2;
38    };
39    getch();
40  }
```

Here's another example, albeit somewhat contrived and probably not encountered in everyday life. Consider a list of, say, 101 numbers. The list contains all the numbers from 1 to 100 and duplicates one of the numbers somewhere in the list. How can you identify the duplicate number? You might give some thought to this before reading on.

Listing 12.7 shows one solution. Again, using a bitmap-like array, scan through the list of numbers. For each number in the list, set the corresponding entry in the bitmap. If, when setting a bitmap entry, you find the entry is already set, you promptly locate the duplicate number. Overall, this solution is fairly good and is useful if you have sufficient memory for the bitmap.

LISTING 12.7. USING A BITMAP TO LOCATE A DUPLICATE NUMBER IN A SET OF CONSECUTIVE NUMBERS.

```
1  /* DUPL1.C
2    One approach to finding a duplicated number in
3    a list of consecutive numbers.
4  */
5  #include <stdio.h>
```

continues

LISTING 12.7. CONTINUED

```
 6  #include <stdlib.h>
 7  #include <time.h>
 8  #include <conio.h>
 9  #include <math.h>
10
11  #define MaxNumbers 1500
12  #define False 0
13  #define True 1
14
15  float Values[MaxNumbers+1];
16
17  void SwapNums ( int I, int J )
18  {
19    float Temp;
20    Temp = Values[I];
21    Values[I] = Values[J];
22    Values[J] = Temp;
23  };
24
25  void main( void )
26  {
27    int Sum;
28    int I;
29    unsigned char BitMap[MaxNumbers+1];
30
31    randomize();
32    /* Set up a list of numbers from to 1 to 100 */
33    for(  = 1; I <= MaxNumbers; I++ ) Values[I] = I;
34
35    /* Let the duplicated value be the mid-point of the list */
36    Values[MaxNumbers+1] = MaxNumbers / 2;
37
38    /* Scramble the numbers in the list so they are in random order */
39    for( I = 1; I <= MaxNumbers * 2; I++ )
40      SwapNums( random(MaxNumbers)+1, random(MaxNumbers)+1 );
41
42    printf("\nHere's the scrambled list:\n");
43    for( I=1; I<=MaxNumbers; I++ )
44    {
45      printf("%f  ", Values[I]);
46      if ((I % 10) == 0)
47        printf("\n");
48    };
49    printf("\n");
50
51    /* Now, find the number that is duplicated */
```

```
52    for( I = 1; I<= MaxNumbers+1; I++ )
53      BitMap[I] = False;
54
55    for( I = 1; I <= MaxNumbers+1; I++ )
56    {
57      if (BitMap[floor(Values[I])])
58      {
59        printf("The Duplicate number is %f \n", Values[I]);
60        goto AllDone;
61      }
62      else
63        BitMap[floor(Values[I])] = True;
64    };
65  AllDone:
66    getch();
67  }
```

But you can solve this problem an entirely different way, relying on the following well-known formula to compute the sum of a list of *n* numbers:

$$sum = \frac{n \times (n - 1)}{2}$$

Given a list of 101 numbers, where the values include every number from one to 100 plus one duplicate value, you can determine the unknown duplicate value, *x*, by writing the following:

$$sum = \frac{101 \times (101 - 1)}{2} + x$$

The actual sum of the list of numbers is easily calculated by running through the total list and adding up the values. With *sum* and *n* both known, the duplicate value is easily computed as

$$x = sum - (n \times (n - 1) / 2)$$

Listing 12.8 shows the code for this implementation. The guts of the algorithm are in lines 52–54. This approach runs more than twice as fast as that given in Listing 12.7.

LISTING 12.8. FINDING A DUPLICATE NUMBER ARITHMETICALLY.

```
/* DUPL2.C
  A much faster approach to finding a duplicated number in
  a list of consecutive numbers.
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <math.h>

#define MaxNumbers 1500

float Values[MaxNumbers+1];

void SwapNums ( int I, int J )
{
  float Temp;
  Temp = Values[I];
  Values[I] = Values[J];
  Values[J] = Temp;
};

void main( void )
{
  int Sum;
  int I;

  randomize();
  /* Set up a list of numbers from to 1 to 100 */
  for( I = 1; I <= MaxNumbers; I++ ) Values[I] = I;

  /* Let the duplicated value be the mid-point of the list */
  Values[MaxNumbers+1] = MaxNumbers / 2;

  /* Scramble the numbers in the list so they are in random order */
  for( I = 1; I <= MaxNumbers * 2; I++ )
    SwapNums( random(MaxNumbers)+1, random(MaxNumbers)+1 );

  printf("\nHere's the scrambled list:\n");
  for( I=1; I<=MaxNumbers; I++ )
  {
    printf("%f  ", Values[I]);
    if ((I % 10) == 0)
      printf("\n");
  };
  printf("\n");

```

```
48    /* Now, find the number that is duplicated */
49    Sum = 0.0;
50    for( I = 1; I <= MaxNumbers+1; I++ )
51      Sum = Sum + Values[I];
52
53    printf("The Duplicate number is %f\n",
54         Sum - (MaxNumbers * (MaxNumbers + 1)) / 2.0 );
55
56    getch();
57  }
```

When evaluating implementations and algorithms, try to think as creatively as possible. Consult books on data structures and algorithms. While contrived problems, like the one just described, often don't seem applicable to the real world, some of these textbook problems are surprisingly similar to real situations. Stripping away all the accoutrements and getting to the core of the problem is the real difficulty.

USE PASS-BY-ADDRESS PARAMETERS INSTEAD OF VALUE PARAMETERS

Calling a function places a copy of each parameter on the stack. You always pass some items, such as arrays, by address. For large items like structures, copying the entire structure to the stack is both time-consuming and memory intensive. If possible, pass these items by address. In the called function, define the parameter as a pointer to the structure rather than the structure itself. Address parameters are either two bytes or four bytes (depending on the memory model in use), probably much fewer than the bytes used by most structures.

CONSIDER ASSEMBLY LANGUAGE

Once the Profiler helps you identify key areas for improvement, think about rewriting short sections of code in assembly language. Compilers usually do not generate code that is as efficient as handwritten optimized assembly language.

By writing in assembly language, you precisely control the values you keep in CPU registers, providing the fastest possible access to your data. For highest performance, refer to a CPU handbook to determine the number of clock cycles used by various CPU instructions. By carefully selecting the best instructions, you can reduce the number of clock cycles required for a given operation. This type of programming is especially important for low-level systems software frequently used by other applications. In some cases, as in writing serial port drivers (see Chapter 16, "High Speed Serial Communications"), top performance is a necessity for transmitting data at the maximum possible speed.

The Borland compilers provide easy access to assembly language through the built-in Assembler. Alternatively, you can use Turbo Assembler to write either individual routines or entire code modules for later linking with your C or C++ code.

USE FIXED POINT ARITHMETIC IN PLACE OF *FLOAT* DATA TYPES

You often can replace floating point calculations with less time-consuming fixed point computations. Borland C++ does not specifically support fixed point data types and expressions, but they are easy to simulate (for limited accuracy) using the `long` data type. `long` constants can range from –2,147,483,648 to 2,147,483,647 provide roughly nine decimal digits accuracy. By assuming a particular location for a decimal point, you can use a `long` for fixed point calculations. For example, storing the value of pi to four decimal places of accuracy requires you to multiply `pi` by 10000, giving 31415. Remembering the decimal place is located between the fourth and fifth digits from the right of the number, you can use this value in subsequent calculations. Listing 12.9 illustrates this technique. This short program, written explicitly for the Profiler (hence, the `for` loop in lines 19–20), executes about twice as fast as `float` and about 12 times faster than a version using `double` data types. The penalties are that the range of numbers is limited and you easily encounter overflow errors when performing multiplication.

LISTING 12.9. DEMONSTRATION OF FIXED POINT OPERATIONS.

```
/*
   REAL2.C
*/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

#define PI 3.1415935
#define accuracy 10000

void main( void )
{
  long X, Y;
  unsigned I;
  ldiv_t converted;

  X = PI * accuracy;
  for( I= 1; I<= 5000; I++)
    Y = X * 5;
  converted = ldiv( Y, accuracy );
  printf("%ld.%ld\n", converted.quot, converted.rem );
  printf("Press Enter to continue.");
  getch();
}
```

To simplify use of the long data type for fixed point numeric representation, create a class named `fixedpoint` and overload the various operators to provide `fixedpoint` addition, subtraction, multiplication, and so forth. You also can overload the various math functions. Listing 12.10 shows one way you might write a `fixedpoint` class. The `main()` function demonstrates a few examples of how the `fixedpoint` class can be used. The `fixedpoint` type executes basic arithmetic functions about 12 times faster than the `double` data type.

LISTING 12.10. THE `fixedpoint` CLASS SHOWS ONE WAY OF IMPLEMENTING FIXED-POINT OPERATIONS IN A CLEAN FASHION.

```
// FIXEDPT.CPP
// Demonstration of how a fixed point data class
// might be implemented.
#include <iostream.h>
#include <stdlib.h>
```

continues

LISTING 12.10. CONTINUED

```
#include <math.h>

#define ACCURACY 100
class fixedpoint {
  long value;
public:
  fixedpoint() { value = 0; };
  fixedpoint(double x);
  fixedpoint( fixedpoint &other);
  fixedpoint operator+(fixedpoint &b);
  fixedpoint operator-(fixedpoint &b);
  fixedpoint operator*(fixedpoint &b);
  fixedpoint operator/(fixedpoint &b);
  fixedpoint& operator=(double x);
  friend double Double( fixedpoint &x);
  friend fixedpoint Fixedpt( double x );
  friend fixedpoint abs(fixedpoint &x);
  friend fixedpoint sin(fixedpoint &x);
};

fixedpoint::fixedpoint( double x )
{
  // The use of the ceil() function for rounding up is needed
  // because the translation from double to long suffers from
  // a conversion error which can cause value to be off by a
  // slight amount.
  value = ceil(x * ACCURACY);
};

fixedpoint:: fixedpoint( fixedpoint &other)
{
  value = other.value;
};

fixedpoint fixedpoint::operator+(fixedpoint &b)
{
  fixedpoint temp;
  temp = *this;
  temp.value = temp.value + b.value;
  return temp;
}

fixedpoint fixedpoint::operator-(fixedpoint &b)
{
  fixedpoint temp;
  temp = *this;
  temp.value = temp.value - b.value;
```

```
  return temp;
}

fixedpoint fixedpoint::operator*(fixedpoint &b)
{
  fixedpoint temp;
  temp = *this;
  temp.value = (temp.value * b.value) / ACCURACY;
  return temp;
}

fixedpoint fixedpoint::operator/(fixedpoint &b)
{
  fixedpoint temp;
  temp = *this;
  temp.value = (ACCURACY * temp.value) / b.value;
  return temp;
}

fixedpoint& fixedpoint::operator=(double x)
{
  *this = Fixedpt( x );
  return *this;
};


//=======================================
// Implementation of friend functions follow

double Double( fixedpoint &x)
// Convert a fixed point value to a floating point value
{
  ldiv_t converted;
  converted = ldiv( x.value, ACCURACY );
  return converted.quot + (converted.rem+0.0) / ACCURACY;
}

fixedpoint Fixedpt( double x )
{
  fixedpoint temp;
  temp.value = x * ACCURACY;
  return temp;
};

fixedpoint abs(fixedpoint &x)
{
  if (x.value<0)
    x.value = -x.value;
```

continues

LISTING 12.10. CONTINUED

```
101    return x;
102  };
103
104  fixedpoint sin(fixedpoint &x)
105  {
106    return Fixedpt( sin( Double( x ) ) );
107  };
108
109  //=========================================
110  // Demonstration of how the class is used.
111  void main(void)
112  {
113    fixedpoint a(1.52), b(3.88), c, d;
114
115    cout << "a=" << Double(a) << "  b=" << Double(b) << "\n";
116    cout << "Arithmetic:  a + b= " << Double( a + b ) << "\n";
117    cout << "             (a*b)/b=" << Double( (a*b)/b) << "\n";
118
119    // demo assignment statements
120    c=7.5;
121    d=-3.0;
122
123    // demo overloaded function
124    cout << "abs( c * d )=" << Double( abs( c * d )) << "\n";
125
126  };
```

INCREASE FILE I/O BUFFERS

Always read and write data to disk using the largest file access buffer that makes sense for your program. Sometimes, when performing sequential and random access to disk files, you may want to read not only the desired record, but also the immediately following record, assuming it as the next record required. You can allocate and associate a larger file buffer with an open file by calling the `setvbuf()` function (see Listing 5.6 in Chapter 5, "Managing Memory").

MEMORY REDUCTION

Reducing a program's memory requirements is another vexing problem that strikes all programs sooner or later. Eventually, it seems, all programs expand to fit the maximum amount of available memory. Once upon a time, each application had all of the PC memory left after DOS was loaded. But then came new versions of DOS that consumed more memory, network drivers that ate up 100K or more memory, assorted TSRs, and the need to run DOS applications in the Windows or OS/2 DOS box. Programs once with a comfortable safety margin now were too large to run on most PCs.

Memory reduction involves two components: reducing and changing the memory required by the executable program code and reducing the memory used for data during program execution. Some suggestions in the previous section also help reduce code requirements. The tips that follow help you reduce the memory required by the program's data.

USE LOCAL AND DYNAMIC VARIABLES

When you define a variable within a module (not inside a function), a static variable anywhere, or a character constant, the variable or constant occupies memory for the entire duration of the program's execution. By comparison, you allocate auto variables, those defined within functions, on the stack at the time the function is called. When exiting the function, you release the memory used by the local variable back to the system. As a result, local variables occupy memory only temporarily. The memory, when not in use, can be used for other purposes elsewhere in the program.

Programs that require an auto variable for use across several function calls, similar to a global value, can use a static pointer to access the temporary data structure. When no longer needed, the structure may be disposed. Chapter 5, "Managing Memory" describes memory layouts and the use of dynamic memory allocations.

RECYCLE MEMORY

Another way to use memory efficiently is to recycle it. That is, apply it to more than one purpose. For example, consider a file buffer used for text file access. Ideally, you allocate space for the file buffer only when it is needed, but sometimes you can recycle this space directly. For instance, when the file is not open, use this buffer for other purposes, such as a buffer for text processing, concatenating error message strings, or whatever your application requires. When evaluating your program, be creative in analyzing memory requirements. The more data items potentially sharing the same data space, the more memory potentially saved.

Use the union (like a structure) to recycle memory. For example, instead of four separate 255-byte character arrays, consider using the following union structure to share the memory requirements:

```
union TMessages {
  char LineBuffer[255];
  char ErrMsgBuffer[255];
  char PromptLine[255];
  char OutputLine[255];
};
```

Even with four separate 255-byte-long arrays, the total space occupied by this structure is 255 bytes. Remember, in a union, the compiler overlays each data element so that each is stored at the same location. Of course, this procedure works only when the buffers can be used independently.

CHAPTER 13

USING BORLAND C++ WITH OTHER PRODUCTS

Borland C++ normally is used as a stand-alone development environment. You can, however, use Borland C++ to export routines to Turbo Pascal programs and to link in functions that have been coded in Turbo Assembler. In another vein, you can convert source code written in Microsoft C/C++ 7 to Borland C++ format, or vice versa. Although C and C++ are standardized programming languages, the Borland and Microsoft products have separate library routines and a few rough edges where conversions are made a little more difficult. In this chapter you learn how to interface to Turbo Pascal and Turbo Assembler and how to write C code that can be translated to other development environments if needed.

Exporting routines to Turbo Pascal

Writing portable C and C++ code

Using assembly language

EXPORTING ROUTINES TO TURBO PASCAL

Many programmers recycle their program code from one language to another. For this reason, a number of Turbo Pascal 6.0 programmers convert routines they have written in C into Pascal. In limited cases, it is possible to link .obj files created using Borland C++ directly into Turbo Pascal programs, saving you the trouble of translating the source code. As with all good things, you must deal with some limitations and hassles before it can all work.

The mechanism for importing C code into a Turbo Pascal program is identical to that used for importing routines written using Turbo Assembler. Listing 13.1 shows a small Turbo Pascal program to sum the values in an integer array. Line 16 defines the function `Sum` as an externally provided function. Note carefully the use of the `far` and `external` keywords—you should always declare your external Pascal routines as far procedures. Line 17 contains a compiler directive that causes the code found in the specified .obj file to be linked into the Turbo Pascal program. This interface is provided primarily for access to Turbo Assembler routines, but it works just fine for access to Borland C++ generated .obj files too.

LISTING 13.1. A SHORT TURBO PASCAL PROGRAM THAT CALLS A FUNCTION WRITTEN IN C.

```
 1  program DemoBCC;
 2  { Demonstrates how to use a program written in C or C++
 3     within a Turbo Pascal 6.0 program. }
 4
 5  const
 6    ArraySize = 20;
 7
 8  type
 9    TArray = Array[0..ArraySize] of Integer;
10
11  var
12    Values: TArray;
13    Result: Integer;
14    I     : Integer;
15
16  function Sum ( N: Integer; var Values: TArray): Integer; far; external;
17  {$L \tp\tv\SUM.OBJ}
18
19  begin
```

```
20    for I := 0 to ArraySize do
21      Values[I] := I;
22
23    Result := Sum ( ArraySize, Values );
24
25    Writeln( Result );
26    Readln;
27  end.
```

The C implementation of the `sum` function is shown in Listing 13.2. (This routine should be compiled as a large memory model program.) The keyword `pascal` (`_pascal` is also permitted) in line 2 tells the C compiler to use the Pascal calling conventions for the function. C and Pascal normally use incompatible calling conventions, but reference to the `pascal` keyword causes the C compiler to adhere to the Pascal function parameter format. In the Turbo Pascal code, the function name is written `Sum`, and in C the function name is lowercase `sum`, although it makes no difference. In Pascal all identifiers are converted to uppercase, so for `pascal` routines, there is no problem with mixing the case.

LISTING 13.2. A SAMPLE FUNCTION, SUM.C, CALLED BY THE TURBO PASCAL PROGRAM IN THE PRECEEDING LISTING.

```
1  /* sum.c */
2  int pascal sum( int n, int * values )
3  {
4    int i;
5    int tempsum = 0;
6    for (i=1; i<=n; i++)
7      tempsum += *(values+i);
8    return tempsum;
9  }
```

A key difference in Pascal versus C is that C has no equivalent for the `var` parameter declaration. It so happens that a Pascal `var` parameter is identical to a pointer parameter in C (see line 2). Hence, the declaration `int * values` becomes a pointer to the `var` array type. For efficiency, you should always pass arrays as Pascal `var` parameters, even though Pascal allows pass-by-value (which copies the entire array onto the stack prior to the function call).

Using simple integer- or character-based C functions is straightforward. However, a few "gotchas" can cause problems when you try to do much more than that in your C code. A big problem is that much coding in C requires that you access the C library. Turbo Pascal cannot link in C library code. You can work around this limitation, but the amount of work involved might become prohibitive for all but the simplest routines. Listing 13.3 shows a C function that uses the library functions `strcpy()` and `strlen()`. To reference these routines, you can `#include <string.h>` or declare them directly in your source file, as shown in the listing.

Using this code in your Turbo Pascal program requires a clever trick. Because Turbo Pascal cannot access the C library containing the code for `strcopy()` and `strlen()`, you must manually remove those routines from the C library and then shoehorn them into your Pascal code. You extract a library routine using the `tlib` library maintenance utility provided by Borland. To extract these two routines, change your active directory to that containing the large memory model C library. For example,

```
C:\> cd \borlandc\lib
```

Next, type the `tlib` command to extract the functions from the large memory model library, cl.lib:

```
C:\borlandc\lib> tlib cl.lib *strcpy *strlen
```

This produces two .obj files, strcpy.obj and strlen.obj. You can leave these files in the library directory or you can copy them to the directory you are using for your Pascal development.

Next, you must compile your Turbo Pascal code. Listing 13.4 shows the sample Pascal program that uses the C code. Lines 10–13 import the .obj files into the Pascal program. Even though the Turbo Pascal program does not reference `strcpy` or `strlen`, their code must be brought in so that the external symbols in setstrin.c are correctly linked.

Using this technique, you *can* call C library functions. In this case, it's a simple matter to extract the routines from the library and import them into the Pascal program. In some cases, one routine may itself use other routines in the C library. In these situations, you must identify the correct library functions (the link step displays an error message for each routine it cannot find) and import them into the Turbo Pascal program. For sizeable programs, this can be quite difficult, and you should realize that it might be simpler to translate your

original C source code into Pascal. Real arithmetic can be used in the C source, but again, to perform nearly any useful operation using the `float` or `double` data types, the C compiler generates code to call library helper routines. You must catch all the linker error messages, pull the appropriate object files out of the library, and then manually insert the .obj files into the Turbo Pascal source.

You can easily reference Pascal strings. Remember that in Pascal, the first byte of a string contains the length byte. In C, the last byte of the string contains a null character. Listing 13.3 shows a C function that uses the library functions `strcpy()` and `strlen()`. Lines 6 and 7 show how to set a value into a Pascal string. The `var` parameter defined in Pascal matches the `char * thestring` parameter in the C code. Listing 13.4 shows a Pascal program that calls the C routine shown in Listing 13.3. Note the use of the C library functions `strcpy()` and `strlen()` (see lines 11 and 12).

LISTING 13.3. A C FUNCTION, SETSTRIN.C, THAT CALLS C LIBRARY FUNCTIONS.

```
 1  extern char * strcpy( char *dest, const char *src);
 2  extern unsigned strlen( const char * );
 3  int pascal setstring( char * thestring )
 4  {
 5    /* Set byte 0 (Pascal length byte) to string length */
 6    thestring[0] = strlen("This is a test");
 7    /* Then copy the data beginning at byte #1 */
 8    strcpy( &thestring[1], "This is a test");
 9    return strlen(thestring)-1; /* Does not include length byte */
10  }
```

LISTING 13.4. A PASCAL PROGRAM THAT CALLS THE C ROUTINE SHOWN IN LISTING 13.3.

```
 1  program DemoStr;
 2  { Demonstrates how to use a program written in C or C++ within
 3    a Turbo Pascal 6.0 program. }
 4
 5  var
 6    S : String;
 7    Result : Integer;
 8
 9  function setstring (var S : STRING): Integer; far; external;
10  {$L \tp\tv\setstrin.obj}
```

continues

```
11  {$L \tp\tv\strcpy.obj}
```

LISTING 13.4. CONTINUED

```
12  {$L \tp\tv\strlen.obj}
13
14  begin
15    Result := SetString ( S );
16
17    Writeln( 'Length=',Result, 'String=', S, '.' );
18    Readln;
19  end.
```

WRITING PORTABLE C AND C++ CODE

Many software development teams have chosen the C programming language because of its presumed portability between compilers and operating systems. Realistically, few C programs written today are portable "as is." It is true that if your programming is restricted to a subset of language features (simple character I/O, no graphics, no screen manipulations to position the cursor, and so on), your code will be reasonably portable to other systems. But in the real world, most of your software must use nonportable features. Merely to position the cursor on the screen is to use a nonportable feature of Borland C++. Can you imagine writing software that treats the PC screen as a glass teletype? That primitive user interface is what you will have if you try to write 100-percent-portable C code. As a result, most software written today is not painlessly portable. Indeed, with today's complex programming environments, such as Windows, X-Windows, and the Macintosh, casually sharing C code across platforms is difficult.

If you use Turbo Vision, ObjectWindows, or EasyWin, for instance, you won't be able to port your software directly to another operating system (unless Borland introduces, for example, a UNIX or a Macintosh version of ObjectWindows). If you write directly to the Microsoft Windows API (and avoid using ObjectWindows), you will have code that is roughly portable between Borland C++ and Microsoft C/C++ 7—but at the cost of using the confusing and error-prone Windows API and at the loss of the productivity

benefits you gain by using ObjectWindows. If your goal is to write a single source application that can run under DOS, Windows, the Macintosh, and UNIX, you should investigate third-party class libraries that provide a high degree of uniformity across all the platforms. (About 8 to 10 such third-party libraries are available.)

C AND C++ LANGUAGE ISSUES

Although the syntax of C and C++ is quite standardized, each vendor is at different stages of implementation compliance. The ANSI C and the AT&T C++ specifications define the basic language structure. But because the specifications are evolving, at any given time, every vendor's compiler might be slightly out of sync with the current specification document. Consequently, you might become reliant on a standard feature in one implementation that is not yet available in another vendor's implementation. For instance, when Microsoft introduced Microsoft C/C++ 7, they identified a number of standard language constructs that did not work in Borland C++ 3.0, even though Microsoft C/C++ 7 is itself defective in a number of areas (Microsoft C/C++ 7 does not implement templates, for instance). Meanwhile, Borland introduced Borland C++ 3.1, featuring improved compatibility with the latest specification document. The problem is that the specification document is a moving target, and vendors can upgrade their product only every so often. As a result, until the C and C++ languages cease to evolve, no one will ever be 100-percent compatible for very long.

GENERAL GUIDELINES

Except for the comments just noted, the C and C++ language syntax is fairly well standardized. You will run into few problems converting the raw language elements of a Borland C++ program into a compatible source code for use on another operating system or compiler. Where you will have trouble is in the use of library routines. Between the two most popular C/C++ compilers in the PC world, the Borland and Microsoft C++ libraries have many significant differences. At the least, they often provide equivalent functions but have different function names. At the worst, there might not be any direct equivalents between functions in their two libraries. Microsoft C/C++ 7, for instance, provides virtual memory allocation routines; Borland C++ 3.1 has no equiva-

lent for this operation.

If you intend to write a single source file that can be compiled on two separate compilers, you should use conditional compilation directives. Include the source code differences that are required for the different compilers, but compile the different sections only when needed. In Borland C++, you can check the predefined macro constants `__BORLANDC__` or `__BCPLUSPLUS__`. The `__BORLANDC__` manifest constant is automatically defined for any application compiled using Borland's C or C++ compilers. The `__BCPLUSPLUS__` constant is defined only when compiling C++ source code. The Microsoft compiler defines `_MSC_VER`. You can use this symbol to determine when the source is being compiled under Microsoft C. Both compilers define `__cplusplus__` to let your source know that C++ compilation mode is currently in effect.

DATA TYPES

The C language does not specify any standard sizes for its basic variable types of `char`, `int`, `unsigned int`, `short`, `long`, `float`, `double`, and `long double`. For this reason, you cannot depend (as we have all done since the creation of the PC) on an `int` to be a 16-bit integer. Indeed, 32-bit C compilers are now available that work with the 32-bit integers of the 80386 and higher microprocessors. When you must know the byte size of a data type (and you actually shouldn't even depend on a byte to be 8 bits), use `sizeof()`.

Do not rely on preconceived notions of maximum and minimum data values for each data type. In 16-bit arithmetic, an `int` type may range from –32,878 to +32,767. In a 32-bit C compiler, the `int` type range is considerably larger. Unfortunately, Borland C++ does not provide constants that specify the compiler's maximum and minimum data range (some other compilers do provide manifest constants). You might want to create your own based on the ranges shown in Table 13.1.

Type	*Size in bytes*	*Range*
`char`	1	–128 to 127
`unsigned char`	1	0 to 255

`enum`	2	–32,768 to 32,767

Type	*Size in bytes*	*Range*
`int`	2	–32,768 to 32,767
`short int`	2	–32,768 to 32,767
`unsigned int`	2	0 to 65,535
`long`	4	–2,147,483,648 to 2,147,483,647
`unsigned long`	4	0 to 4,294,967,295
`float`	4	3.4×10^{-38} to $3.4 \times 10^{+38}$
`double`	8	1.7×10^{-308} to $1.7 \times 10^{+308}$
`long double`	10	3.4×10^{-4932} to $1.1 \times 10^{+4932}$

When providing your own file buffer to a file function, pass the buffer's size using the `sizeof()` function. Do not rely on a hard-coded constant or a manifest constant to correctly determine the buffer's size.

If your program must perform bit-level manipulations, you might want to hide these manipulations using a macro definition. For instance, to set a bit within an identifier, you can use this macro:

```
#define setbit(x, y) x = x ¦ y
```

To set the fifth bit of a bitfield variable, you would type

```
setbit( bitmask, 32 );
```

As shown, you use the constant 32 because 2 raised to the fifth power is 32. You also could use a hexadecimal constant. By using this macro definition, you can easily make changes to accommodate nearly any compiler.

Bit-field structures are distinctly nonportable. A bit-field structure provides high-level language access to bit layouts. A sample bit-field structure definition might be as follows:

```
struct status_bits {
  read_only : 1;
  write_only : 1;
  locked : 1;
  num_locks : 4;
  blocks : 9;
```

```
} result_code;
```

This structure assigns one bit to the `read_only`, `write_only`, and `locked` fields, four bits to the `num_locks` field, and nine bits to the `blocks` field. The Borland C++ compiler assigns bit fields such that the first bit field appears in the lowest numbered bits of a word, the second field appears in the next highest numbered bits, and so on (remember that at the bit level, bits are numbered from right to left). There is no guarantee that other compilers will adhere to this definition of a bit field. Other compilers might assign bit fields from left to right. If the bits span more than one 16-bit word, the compiler might split the bits across two words or restrict the bits to fall entirely within one word.

To access the low byte or the high byte of an integer, define a macro. Each of the following macros extracts the appropriate byte. The second set, however, generates less code. Note carefully the use of parentheses around the symbol `x` within the macro expressions. The parentheses assure that when an expression is used when calling one of the macros, the expression will be fully evaluated before extracting the selected byte.

```
#define LOBYTE(x) ((x) & 255)
#define HIBYTE(x) (((x) >> 8) & 255)
```

or

```
#define LOBYTE(x) ((unsigned char) (x))
#define HIBYTE(x) ((unsigned char) ((x) >> 8))
```

LIBRARY FUNCTIONS

As you select library functions to incorporate into your program, you can elect to use functions that are ANSI C or UNIX compatible. The Borland C++ Library Reference indicates for each library function whether the function is generally compatible with DOS, UNIX, Windows, ANSI C, or C++. Because you restrict your use of the library to the ANSI C or UNIX functions, your code will be highly portable across differing C compilers and operating systems. But realistically, it is difficult to write an interesting, state-of-the-art user interface using the standardized functions. Most programs use the enhanced library features (especially for graphics and screen I/O) available from the Borland Library.

If you do use nonstandard routines, you might be able to improve your code's

portability by creating a layer of code to hide the underlying function. When you port the code, just change the layer rather than fix every function call in your entire program. An example makes this concept clear. In Borland C++, you can position the text cursor on the screen (within the active text window) by calling `gotoxy(int x, int y)`. This positions the cursor to column *x* of row *y*. In Microsoft C/C++ 7, you must call `_settextposition (y, x)`. Note the reversal of the *x* and *y* parameters. In this example, you can hide the underlying function call by accessing either `gotoxy()` or `_settextposition()` indirectly through another function (or macro definition). Instead of writing in Borland C++

```
gotoxy( 10, 20 );
```

you could create a new function, `positioncursor()`, that calls `gotoxy()` in the Borland implementation or `_sextextposition()` in the Microsoft implementation. By doing so, you hide the underlying target environment. Converting your source from one compiler to the other requires only that you change the single function.

You can implement `positioncursor()` as a function or as a macro. Here's the function implemented for use in Borland C++:

```
void positioncursor( short x, short y )
/* Position the cursor to (x, y) */
{
  gotoxy(x,y);
}
```

Some conversions are sufficiently complicated that the function method will be preferred. For simple functions, such as `gotoxy()`, a macro is more efficient, eliminating an extra runtime function call:

```
#define positioncursor(x,y) gotoxy((x),(y))
```

To compile this code in Microsoft C/C++, change either the function or the macro definition (whichever you use) to call `_settextposition()`.

When you create an interface layer to hide the underlying implementation, the routines you need to hide usually are related to the user interface. In particular, the screen I/O and graphics functions of Borland C++ are not portable to other implementations of C. All the graphics functions in Borland C++ are implemented differently in Microsoft C/C++, although the two products have similar functionality in their libraries. Microsoft C/C++ does not have any of the text-oriented functions such as `textcolor()` and `textbackground()`.

Your best bet when you know that you will be converting to or from a

particular vendor's compiler is to observe compatibility information provided in Borland's Library Reference. If the function is shown as DOS-only, it probably is not available in the other compiler. Before using such functions, look at the other vendor's library reference. If you can find another function that performs roughly the same function, use a macro or a function to act as an interface layer. If you cannot find a matching function, you still might want to use an interface layer. In the other environment, you just might find yourself writing a good deal more code to make up for the incompatibility.

SPECIAL ISSUES CONCERNING MICROSOFT C/C++ AND BORLAND C++

The Make files used to build projects in Borland C++ and Microsoft C++ are not compatible with one another. Borland's MAKE utility, beginning with Borland C++ 3.0, does include a command-line option, `-N`, to improve compatibility with Microsoft's NMAKE-compatible make files. When the `-N` option is in effect, Borland's MAKE recognizes most of the features of NMAKE, making the conversion from Microsoft C easier than before.

I have converted a significant amount of C++ code back and forth between Borland C++ 3.0 (prior to the release of 3.1) and Microsoft C/C++ 7.0. The C++ language implementations of these two compilers are quite good. Except for the lack of template support in Microsoft C/C++ 7, I have had few problems recompiling the sources, except for some of the library issues noted earlier. One minor problem I encountered involves the use of static data members within classes. In Borland C++, you can write code such as the following (this example is from *Using Borland C++* and *Using Microsoft C/C++*, both published by Que Corporation):

```
class star {
public:
  static char name[21];
  star() { strcpy( name, "NONAME" ); }
  star( char *sname ) { strcpy( name, sname ); }
  void tellname() { printf( "%s\n", name ); }
};
```

The data member `name` is statically defined. Each instance of this class references the same copy of the `name` member. This technique can be used to create private data within a class that may be shared between instances of the

same class.

Microsoft C/C++ also supports static data members, but not identically to Borland C++. In Microsoft C/C++, you must add an additional declaration for the static data member outside the class definition. The extra declaration is shown after the class definition in this revised listing:

```
class star {
public:
  static char name[21];
  star() { strcpy( name, "NONAME" ); }
  star( char *sname ) { strcpy( name, sname ); }
  void tellname() { printf( "%s\n", name ); }
};

char star::name[21];
```

Both Microsoft C/C++ and Borland C++ support the same memory model sizes—tiny, small, compact, medium, large, and huge. If your code needs to know the exact memory layout of these memory models, however, you need to account for significant differences in how the large memory model is implemented between these compilers.

Borland C++ does not support Microsoft's based pointers (hence, it does not support the `_based` keyword), nor does Borland C++ support the `_self` and `_segname` keywords. Borland C++ does not support the `_fortran` keyword for selecting the Fortran language calling conventions, so you should use `_pascal` instead.

HEADER FILES

In Borland C++, you can `#include` either alloc.h or malloc.h. Microsoft C/C++ does not define alloc.h, so you must specify malloc.h when compiling under Microsoft C. Similarly, you must use direct.h for Borland's dir.h, and memory.h for Borland's mem.h header files. For direct compatibility with Microsoft, Borland enables you to use any of the header filenames just described.

USING ASSEMBLY LANGUAGE

Assembly language programming means programming the PC in its native 80x86 processor instructions. Through assembly language programming, you can write hand-optimized code to perform faster operations than may be possible when writing in the high-level language of Borland C++. Further, assembly language provides total control over the CPU, which is essential for writing certain kinds of system software and is often needed for controlling certain types of external devices.

This section provides an introduction to assembly language programming and specifically covers interfacing to Borland C++. The focus in this chapter is on the use of the Borland C++ assembly language interface features, including the `asm` statement, and an introduction to the use of Turbo Assembler. This chapter is not a tutorial on programming the 80x86 family of microprocessors. However, if you are familiar with programming in any assembler language and have access to an 80x86 processor handbook, you should be able to write your own assembly procedures using the material provided here.

For many applications, the power of nitty-gritty assembly coding is essential. For instance, in order for graphics routines to generate complex drawings at high speed, their bit manipulation routines must be written in assembly language. Similarly, library routines such as `memcpy()` and `strcpy()` can take advantage of special machine instructions for moving entire blocks of characters at high speed, far faster than can be accomplished by using a `for` loop to copy bytes from one area of memory to another. Another type of routine, the interrupt handler, may be written in either Borland C++ or assembly language, depending on the type of interrupt being processed and the frequency with which the interrupt is generated. For instance, a serial port interrupt handler must be able to process approximately 2,000 individual interrupts per second (at 19,200 bps) on a CPU as slow as a 4.77 MHz 8088 processor chip. Through programming tricks, it's even possible to access a serial port at speeds of 115,200 bps (see Chapter 16, "High-Speed Serial Communications"). Such coding demands the use of high-speed, hand-optimized, assembly-level programming.

This chapter provides a broad overview of 80x86 assembly language programming, with an emphasis on the features of the Borland C++ programming environment that facilitate assembly programming. For any significant assembly language programming, you should have a copy of the appropriate CPU processor handbook. The processor handbooks detail the complete list of

registers, instructions, memory addressing modes, and instruction timing details that are needed to fully exploit the capabilities of assembly language programming. Processor handbooks are published by the manufacturers of the CPU chips, and a variety of programming guides and other handbooks are widely available at technical bookstores. I use the *80386 Microprocessor Handbook* by Chris H. Pappas and William H. Murray III, and *The 8086 Book* by Russell Rector and George Alexy, both published by Osborne/McGraw-Hill.

To really learn the nuts and bolts of assembly language programming, or speciality areas such as device drivers or operating system support, you should consult books on assembly language programming and DOS internals. An excellent DOS reference is *DOS Programmer's Reference*, published by Que Corporation.

A VERY BRIEF 80X86 PROCESSOR INSTRUCTION SET OVERVIEW

Assembly language programming requires a basic understanding of the CPU's instruction set, memory configuration, and its registers. See Chapter 5, "Managing Memory," for a quick overview of the 80x86 processor architecture.

The 80x86 architecture defines a large number of machine instructions. These instructions provide for moving data values to or from memory and registers, basic arithmetic and logical operations, comparing values, calling subroutines, and a variety of other instruction types. Refer to a processor handbook for a complete list of instructions and their operands.

The purpose of the assembler is to convert symbolic assembly language instructions such as

```
MOV     AL, 5
```

into the specific machine code bytes that instruct the processor to move the value 5 into the AL register. Each assembly language statement consists of an optional statement label, followed by the instruction and operands, followed by an optional comment, as shown by this syntax:

```
label:     instruction operands     ; source comment
```

The exact format of a line of assembly language source depends on which assembly language development tool is used—the `inline` or `asm` statement or

Turbo Assembler.

Here are examples of assembly language statements as they are written in Turbo Assembler assembly language.

```
     MOV     AL, 5          ; Move the constant 5 into the AL register
     MOV     DX, Buffer     ; Move the address of 'Buffer' into DX
     INC     AL             ; Increment the contents of AL by 1
     JMP     Abort          ; Jump to the Abort label
Fetch:                      ; Defines an assembly language 'label'
     CMP     AL, 2          ; Compares AL to the constant 2
     JE      Abort          ; If equal, then jump out of here
...
Abort:
....
```

An assembly language statement may optionally be prefaced with a statement label, which is an identifier ending with a colon (:). These labels are used to identify statements referenced in unconditional and conditional jump instructions.

Each assembly language instruction consists of the instruction name, such as MOV or INC, and, where required, a set of instruction operands. The operands are the data that is operated on by the instruction. All text appearing after a semicolon (;) and extending to the end of the line is treated as a source comment. The assembly language syntax available to the built-in assembler is similar but not identical to that used in Turbo Assembler.

THE BUILT-IN ASSEMBLER

Because assembly language code often is used sparingly, you can write short routines in assembly language using the Borland C++ built-in assembler. This assembler enables you to embed assembly language statements directly within your C code, providing you with the best of C, C++, and assembly language programming.

The `asm` statement is a full-function assembler built into the Borland C++ compiler, and it can be used anywhere within a C or C++ program that a C or C++ statement is permitted. This means that you can rewrite individual statements, portions of a function, or a function's implementation in assembly language.

The `asm` statement provides all the standard assembler features, including full instruction set support, statement labels, source comments, and data defini-

tions. Because the `asm` statement is itself a C statement, assembly language may be incorporated into your program at any point that a statement may be entered. Plus, because `asm` source code is contained within Borland C++ source files, `asm` source has full access to your C variables, labels, and functions.

USING THE BUILT-IN ASSEMBLER

To use the built-in assembler, place the keyword `asm` into your source, followed by an assembly language statement or a group of statements surrounded by brackets (`{ }`). Each built-in assembly language statement consists of an optional statement label, an optional instruction prefix, and the processor instruction (or *opcode*), followed by the opcode's *operands* (registers, constants, and identifiers), like this:

```
Prefix Opcode     Operands
```

Statement labels, which normally appear at the beginning of a Turbo Assembler statement, are not recognized in the built-in assembler. Comments may be placed within standard comment delimiters (`/*` and `*/`). The only restriction on comments is that they must appear only between assembly statements and not in the middle of an assembly statement. This means that (in the preceding code line) a comment may appear after `Operands` but not between `Opcode` and `Operands`. Each assembler statement is separated by an end-of-line character, a semicolon, or a comment.

Listing 13.5 illustrates the use of the `asm` statement to implement a simple function, `increment`, that increments its value parameter by 1 and returns the result in the function identifier.

LISTING 13.5. A SIMPLE DEMONSTRATION OF BUILT-IN ASSEMBLY LANGUAGE.

```
 1  /* ASMDEMO1.C */
 2  #include <stdio.h>
 3
 4  unsigned int increment( unsigned int x )
 5  {
 6    asm {
 7      inc    x            /* Increment the local x parameter */
 8      mov    ax, x        /* Return value of x */
 9      };
10  };
11
```

```
12  void main( void )
13  {
14     unsigned var1;
15     var1 = 1000;
16     var1 = increment( var1 );
17     printf("var1=%d\n", var1 );
18  }
```

Inside the `asm` statement, the local parameter value `x` is directly accessed by writing

```
inc         x
```

The assembler automatically translates this into the `INC` processor instruction, followed by the memory address of `x`. Because `x` is a local function parameter, `x` is stored on the stack, and the assembler generates the appropriate code to access `x` with respect to the current stack frame. The result of `x` incremented by 1 is moved to the AX register for return. For simple 1-, 2-, or 4-byte return results, you can place the function's return value in registers. For a 1-byte result, place the value in AL; for a 2-byte result, place the value in AX; and for a 4-byte result, place the value in AX and DX. For structures that are 3 bytes in size, or 5 bytes and larger, the compiler generates a hidden function parameter containing the address of a memory location where the result should be returned. To keep things simple for yourself, return only simple 1-, 2-, or 4-byte values.

HOW PROCEDURES AND FUNCTIONS ARE CALLED

When a Borland C++ program calls a function, it pushes parameter values onto the stack, and then issues the assembly language `CALL` instruction. For near calls, `CALL` pushes the IP register to the stack. For far calls, `CALL` pushes both the CS and IP registers. Upon entry to the function, the first instructions push the BP register onto the stack, allocate space on the stack for local variables by subtracting the required space from SP (remember that stacks grow downward in memory), and copy SP to BP, like this:

```
push     bp            ; Save BP from caller, onto the stack
mov      bp, sp        ; Copy SP to BP so that BP points to local
                       ; "frame"
```

```
sub      sp,  n        ; Subtract n from SP, where n = size of
                       ; memory needed for local vars
```

The collection of stack data, including parameters, local variables, and saved copies of BP, constitutes the procedure's *stack frame*. Parameters are pushed in the reverse order that they appear within the function's definition. This means that the last parameter is the parameter farthest back in the stack, and the first parameter is the parameter closest to the top of the stack.

When SP is copied to BP, BP becomes the *frame pointer*. Parameter values are accessed by adding an offset to BP. For a near procedure or a function, the last parameter pushed onto the stack is accessed with an instruction, like this:

```
MOV     AX, [BP+4]     ; For a near procedure call
```

The parameter before the last one is at `[BP+6]`, and so on. (The size of the offset value depends, of course, on the size of the parameter value.)

Far procedures have an additional two bytes on the stack for storing the CS register; hence, to access the last parameter of a far call, you must write code similar to this:

```
MOV     AX, [BP+6]     ; For a far procedure call
```

The parameter before the last one is at `[BP+8]`, and so on, depending on the size of the parameter value.

Fortunately, most of the time you do not need to determine how many bytes to add or subtract to BP. The built-in assembler allows symbolic references so that you write only the parameter or variable name and the assembler automatically generates the appropriate offset. See the sections "Accessing Global Variables" and "Local Variables in Functions" below. When using Turbo Assembler, you can use the `ARG` directive to associate symbols with parameters (see the section "A Sample TASM Program" later in this chapter).

At the end of a function, special code undoes the actions of the procedure's initial instructions, like this:

```
mov     sp, bp     ; Because BP was saved prior to allocating
                   ; locals, this "throws
                   ; away" all the local variable space
pop     bp         ; Restore the BP register
ret     n          ; Pop the IP, and if needed CS, registers.
                   ; Then, subtract n bytes, where n
                   ; is the number of bytes used by the
                   ; parameter list.
```

ACCESSING GLOBAL VARIABLES

Within assembly language statements, reference may be made to public variable declarations. In effect, this enables you to use C for the data declarations, including macro symbols, simple variables, structures, and other functions used by your assembly code. Identifiers are accessed by assembler statements through reference to the identifiers, as shown in Listing 13.6.

LISTING 13.6. EXAMPLE SHOWING HOW VARIABLES MAY BE REFERENCED BY BUILT-IN ASSEMBLY CODE.

```
 1  /* ASMDEMO2.C */
 2  #include <stdio.h>
 3
 4  unsigned total, value;
 5
 6  void main( void )
 7  {
 8    /* Sums the entered values, producing the sum in total */
 9    total = 0;
10    do {
11      printf("Enter value:  ");
12      scanf("%d", &value );
13      asm {
14         mov      ax, total
15         add      ax, value
16         mov      total, ax
17      };
18      printf("Total=%d\n", total);
19    } while (value != 0);
20  }
```

The statement

```
mov     ax, total
```

moves the value stored at `total` to the AX register. Next, the contents of the `value` variable are added to AX:

```
add     ax, Value
```

The result of `total` + `value` is then stored back to the `total` variable with the instruction

```
mov     total, ax
```

The assembler automatically translates references to `value` and `total` into appropriate memory addresses or stack offsets.

DISTINGUISHING BETWEEN VALUES AND ADDRESSES

As used in the preceding section, the statement

```
mov     ax, total
```

moves the value or content of variable `total` into the AX register. The assembler has a special keyword, offset, used to obtain the address of a variable. For example, the following statement moves the address of `total` (not its current `value`) into BX:

```
mov     bx, offset total
```

Because public and static variables (in all memory models but huge) are stored in the data segment pointed to by the DS register, all global variables and typed constants are addressed as offsets from the DS register. Therefore, obtaining the value of `total` using this code structure requires an additional instruction:

```
mov     bx, offset total     /* Store total's offset from DS
                             /* register to BX */
mov     ax, ds:[bx]          /* Load value at [DS:BX] into AX */
```

The last instruction accesses the memory word pointed to by `DS:BX` and loads that value into the AX register.

At time of assembly, you can perform arithmetic on the offset address, as shown in the following sample code that computes the location of `value` based on the address of `total`, and uses that calculated address to copy the content of `value` to a new variable named `temp`:

```
asm {
  /* Offset total+2 is the address of total plus two bytes
     This gives the address of value. */
   mov      bx, offset total+2
   mov      ax, ds:[bx];
   mov      temp, ax
   mov      ax, total
   add      ax, value
   mov      total, ax
};
```

In this example, the content of the `value` variable is accessed indirectly.

Because the variables `value` and `total` are declared one after the other and because the compiler set aside 2 bytes of memory for each of these variables, one after the other in the data segment, you can use one to find the other. Because `value` is stored adjacent to `total`, you can access `Value` by computing

```
mov     bx, offset total + 2
```

so that `BX` contains the address of `value`.

THE DIFFERENCE BETWEEN CONSTANTS AND VARIABLES

Manifest constant identifiers (or macros) may be referenced in assembler statements. As in C, the use of a constant identifier is equivalent to using the constant to which it is equated, meaning that the statements

```
#define MaxElements 10
asm
   mov     ax, MaxElements
   ...
end;
```

and

```
asm
   mov     ax, 10
   ...
end;
```

are equivalent. Each loads the constant, decimal value 10, into the AX register.

If you want to load the *value* stored at memory location `DS:10`, you must write

```
mov     ax, [10]
```

or

```
mov     ax, [MaxElements]
```

LOCAL VARIABLES IN FUNCTIONS

You reference variables that are local to a function the same way you reference public variables, as shown in this code section:

```
void sample(void) {
  unsigned i, answer;
```

```
  ...
  asm {
    mov     ax, i     /* Fetch value of local variable i /*
    inc     ax
    mov     answer, ax
  };
  ...
};
```

ACCESSING PASS-BY-VALUE AND PASS-BY-REFERENCE PARAMETERS

Parameters that are passed by value are used identically to any other local variable, as described in the preceding section. Parameters passed by reference (as a pointer) are referenced somewhat differently because a pointer is not a value but a 4-byte segment:offset memory address (in the large data memory models; it is a 2-byte offset address in the small data models). Accessing what the pointer points to is a bit trickier because the 4-byte segment:offset pair must be loaded into both segment and index registers. The resulting segment:offset pair is then used as a pointer to read or write the value stored at that location. Listing 13.7 is a sample program that uses a pointer parameter in its `AddValues()` function (see lines 7–15). This assembler code assumes that the function parameters are passed using far pointers. Therefore, the example works only when compiled using a large data memory model.

LISTING 13.7. A DEMONSTRATION OF ACCESS TO A POINTER PARAMETER.

```
 1  /* ASMDEMO3.C
 2   Demonstration of accessing pointer and value parameters.
 3   Compile using the large memory model.
 4  */
 5  #include <stdio.h>
 6
 7  void AddValues( unsigned * Total, unsigned Value )
 8  {
 9    asm {
10      les di, DWORD PTR Total  /* Put address of Total into ES:DI */
11      mov ax, ES:[DI]          /* Fetch value at that address */
12      add ax, Value            /* Add the contents of Value to Total */
13      mov ES:[DI], ax          /* Store result back to address */
14    };
15  };
16
17  void main(void)
18  {
19    unsigned Total;
```

```
20    unsigned Value;
21    /* Sums the entered values, producing a result in Total */
22    Total = 0;
23    do {
24      printf("Enter value (0 when done): ");
25      scanf("%d", &Value);
26      AddValues( &Total, Value );
27      printf("Total=%d\n", Total);
28    } while ( Value != 0);
29  }
```

In line 10, the assembler instruction

```
les di, DWORD PTR Total
```

loads the segment portion of the address of `Total` into the ES register, and the offset portion into the DI register. The `MOV` opcode copies the value located at the memory address contained in `ES:[DI]` into AX. After the contents of `Value` are added to AX, the result is stored back to the memory address of `Total` using the address in `ES:[DI]`.

ACCESSING STRUCTURES

You access components of a structure exactly the same as when referenced in C statements, using the period (.) to access individual members of the structure. Listing 13.8 shows an example of how to access fields of a structure. To use this feature, you must select the Options | Compiler | Code generation... dialog box and enable the Compile via assembler option. Borland's documentation implies that the built-in assembler can access structure fields using this standard notation, but this does not appear to be true. Enabling the Compile via assembler option causes the compiler to use Turbo Assembler instead of the built-in assembler.

LISTING 13.8. SAMPLE ASSEMBLY CODE THAT USES A STRUCT TYPE.

```
1  /* ASMDEMO4.C
2     Demonstrates referencing components of a structure.
3  */
4  #include <stdio.h>
5
6
```

```
 7  void main( void )
 8  {
 9  struct {
10      unsigned Total;
11      unsigned Value;
12    } Results;
13
14    /* Sums the entered values, producing the sum in Total */
15    Results.Total = 0;
16    do {
17      printf("Enter value: ");
18      scanf( "%d", &Results.Value);
19      asm {
20        mov      ax, Results.Total
21        add      ax, Results.Value
22        mov      Results.Total, ax
23      };
24      printf("Total=%d\n", Results.Total);
25    } while (Results.Value != 0);
26  }
```

STATEMENT LABELS

The built-in assembler works with C statement labels only; you may not define labels within the body of the assembly source code. Generally, this is not much of a problem because the assembly language code can jump to any C source code label. A typical C label (used with the `goto` statement) may be the target of a `jmp` or a conditional jump machine instruction. Here's an example:

```
asm {
    ...
    cmp     ax, 0                /* Check of non-zero AX register */
    jne       ErrorCondition     /* Jump to error handler if not
                                 /* zero */
    ...
};
...
ErrorCondition:                 /* Somewhere in the C source, */
                                /* same function */
```

JUMP INSTRUCTIONS

The built-in assembler optimizes code generated for all types of jump instruc-

tions, including `jmp` and the conditional jump instructions, with the restriction that all jumps must be to locations within the function where the assembly code is being used. Whenever possible, the assembler uses a short jump instruction (consisting of two code bytes) for jumps within –128 bytes to +127 bytes of the current jump instruction. For jumps beyond this range, a long jump (consisting of three code bytes) is generated.

The branching instructions, `je` and `jne` for example, are always limited to jumping to within –128 to +127 bytes of the jump instruction.

ASSEMBLER EXPRESSIONS

The built-in assembler can perform limited arithmetic operations, such as adding constants to the address of symbols. For instance, in the statement

```
mov     ax, Label + 2
```

the assembler adds the constant 2 to the offset of `Label` and assembles this as a `MOV AX, <constant>` instruction. Constants can be written in base 2, 10, or 16, as follows:

Base	*Type*	*Add*	*Example*
2	Binary	Add trailing "B"	0000 0101B
10	Decimal	Use 0 to 9 only	12345
16	Hex	Add a trailing "H"	0AF16H

For hexadecimal constants, if the first digit of the constant is A through F, you must prefix the entire number with the number 0. If you omit the 0 prefix character, the assembler thinks that the number is an identifier such as AF16H instead of 0AF16H. The IDE editor highlights binary and hexadecimal constants in red because it thinks that they are invalid symbols; however, they

work just fine in the assembler and do not cause any problems.

Expressions within parentheses are evaluated first, as are memory addressing expressions within brackets ([and]), as in

```
mov     ax, [offset Label + 2]
```

Table 13.2 provides a list of the supported operators. Borland does not document any of these operators for use with the built-in assembler, so there is no guarantee that they will be supported in future editions of the compiler. However, common sense suggests that they all will be available in future C and C++ compiler products.

Operator	*Description*
`+, -`	When used before an individual symbol or constant, these are treated as unary operators. When used between symbols or constants, they perform addition and subtraction, respectively.
`*, /`	Used to multiply or divide the values of two constants.
`MOD`	Performs the modulus operation between two constant values. This operator is available only when compiling with the Compile via assembler option.
`SHL`	`x SHL y` shifts the constant value `y` to the left by `x` (constant) bits.
`SHR`	`x SHR y` shifts the constant value `y` to the right by `x` (constant) bits.
`NOT`	Inverts each bit setting in the constant.
`AND`	`x AND y` performs a bitwise `AND` between the constant values `x` and `y`.
`OR`	`x OR y` performs a bitwise `OR` of the constant values `x` and `y`.
`XOR`	`x XOR y` performs a bitwise exclusive-or of the constant values `x` and `y`.
`HIGH x`	Returns, as a byte value, the high-order byte of constant `x`. This operator is available only when you use the Compile

continues

	via assembler option.
`LOW x`	Returns, as a byte value, the low-order byte of constant `x`. This operator is available only when you use the Compile via assembler option.
`OFFSET`	Returns the offset component of an identifier's segment:offset memory address.
`TYPE x`	Returns the size of `x` in bytes. To the assembler, the only "type" associated with an identifier is the identifier's memory requirement. This operator is available only when you use the Compile via assembler option.

Operator	*Description*

`PTR`	`x PTR y` uses `x` as a "type" and the expression `y` as a memory address. `x` can be one of the following keywords: `BYTE`, `WORD`, `DWORD`, `QWORD`, `TBYTE`, plus `NEAR` or `FAR`. These keywords, when used with `PTR`, tell the assembler which specific opcode to assemble. For example, the statement
	`mov dl, BigArray`
	might not result in what you want, particularly if `BigArray` is a word data type. Instead, you must force the type declaration by writing
	`mov dl, byte ptr BigArray`

TURBO ASSEMBLER

Turbo Assembler is a full-featured stand-alone assembler for the creation of assembler-written programs or functions. Although you can create entire assembler-written programs with Turbo Assembler, you probably will use

Turbo Assembler to write functions that may be called from inside your Borland C++ programs.

In Borland C++, routines declared as "external" are linked from externally provided object files (or libraries), where the object files often are created from assembly-language source code that has been assembled by Turbo Assembler. This section provides information on using Turbo Assembler for the creation and linking of external routines called from Borland C++. Turbo Assembler has many more features than are covered here, however. If you want to use Turbo Assembler for other applications or to make use of other features it can provide, see Borland's Turbo Assembler user's guide and Turbo Assembler reference manual.

TURBO ASSEMBLER BASICS

To use Turbo Assembler, you must write the assembly-language source program using a text editor or the Borland C++ IDE editor and save your source code to a file ending in .asm (the default extension recognized by Turbo Assembler). Your assembly language program is assembled using TASM, the Turbo Assembler program, to convert the source code into a Borland C++ compatible object (.obj) file. You must add the .obj file to the IDE's project file or specify the name of the .obj on the command line used to call the command-line compiler.

TURBO ASSEMBLER STATEMENTS

In many respects, Borland C++'s built-in assembler is similar to Turbo Assembler, but there are important distinctions. The basic format for an assembly language statement remains the same:

```
Label:    <Prefix> Opcode    Operands    ; Source comments
```

Note that Turbo Assembler source comments are always preceded by a semicolon and continue to the end-of-line character. Borland C++'s comment syntax (`/*` and `*/`) is not recognized by Turbo Assembler. Turbo Assembler also provides hundreds of additional symbols and assembler directives that are not included in the built-in assembler.

A SAMPLE TASM PROGRAM

Listing 13.9 contains a sample C program calling a function, `sum`, that is written

continues

in assembly language. `sum` is a simple function that sums the values in an array. The values are specified in the variable parameter `values`, which is an array of `int`, and `n` is the number of elements in the array. (For those of you who are wide-awake after your morning espresso, yes, there is a simpler way to sum the values used in this demonstration program, by computing `N * (N + 1 ) div 2`. However, `sum` is a general-purpose routine that can sum arrays containing nonconsecutive values.)

LISTING 13.9. A SAMPLE C PROGRAM THAT CALLS AN EXTERNAL ASSEMBLY LANGUAGE FUNCTION.

```
/* DEMOTASM.C
   Demonstrates interface to a program written in TASM.
   Compile as bcc -ml demotasm.c sum.obj
```

LISTING 13.9. CONTINUED

```
   where sum.obj is created by tasm.
*/
#include <stdio.h>

#define ARRAYSIZE 20

extern int sum( int n, int * values );

void main (void )
{
  unsigned i;
  int values[ARRAYSIZE];
  unsigned result;

  for( i = 0; i < ARRAYSIZE; i++ ) values[i] = i+1;

  result = sum ( ARRAYSIZE, values );

  printf("Result=%d\n", result );
}
```

Within the C program, the function `sum` has only a header because its implementation will come from an assembly language routine. When calling assembly language routines from C++ code, you might want to preface the external function header with the extern "C" directive, as in this example:

```
extern "C" int sum( int n, int * values );
```

This ensures that the compiler uses the simpler C calling conventions rather than C++, even if your source code is written in C++. In particular, the name mangling performed by C++ causes the function name `sum` to be expanded with additional characters. By specifying the extern "C" directive, you ensure that the function name will be translated directly, without name mangling.

Listing 13.10 contains the assembly-language source code. Generally, this source code resembles the source code format used by the built-in assembler. However, Turbo Assembler is a separate program for creating stand-alone assembly language programs, as well as routines that are compatible with a variety of programming languages, including Borland's C++ products. Therefore, Turbo Assembler programs must include instructions telling the assembler to generate Borland C++ compatible code, and additional assembler directives to provide access to Borland C++ function identifiers.

LISTING 13.10. A SAMPLE ASSEMBLER PROGRAM TO BE LINKED TO THE DEMOTASM.C PROGRAM.

```
 1  ; sum.asm
 2  ; assemble as: TASM /ml sum.asm;
 3  .MODEL large
 4  PUBLIC _sum
 5  .CODE
 6  ; int sum( int n, int * values)
 7  _sum  PROC
 8  ARG n:word, values:far ptr
 9   push bp
10   mov  bp, sp
11   mov  cx, n             ; CX holds number of array elements
12   les  di, values        ; ES:DI points to values array
13   xor  ax, ax            ; Set AX = 0
14  countem:
15   mov  bx, es:[di]       ; Fetch value at [di]
16   add  ax, bx
17   inc  di
18   inc  di
19   loop countem
20   pop  bp
21   ret
22  _sum  ENDP
23  END
```

The `.model` directive at the beginning of the file tells the assembler that this source code will be assembled using the large memory model. This selection affects how the assembler will access certain features. Knowing this, the assembler can perform some of its work automatically, work that would otherwise require additional effort from you, such as manually determining byte offsets to stacked parameters and creating entry and exit code for the procedure.

The statement

```
_sum  PROC
```

identifies `_sum` as the name of this far procedure. Note the use of the leading underscore. The C compiler automatically adds an underscore character to the beginning of all external functions. In order for the linker to match the name used in C (which is `sum`) with the assembler routine, you must manually add the underscore in the assembler source code. Also, because C is a case-sensitive language, and the assembler ordinarily is case-insensitive, a problem could arise in matching the names. The C function `sum` becomes `_sum`, but by default the assembler's symbol becomes `_SUM` and the names will not match. You can solve this problem by using the `/ml` switch to the TASM assembler. This preserves the case sensitivity of external symbols defined or accessed within the assembly language code.

Symbols must be exported from the assembler code. To make symbols visible to users of the object module, use the public directive. The statement

```
PUBLIC     _sum
```

causes this symbol to be made public and available to those who will use the resulting object module.

`ARG` is an assembler directive that enables programs to access the procedure's parameters in symbolic fashion. Each parameter is listed in the `ARG` directive, as shown in the sample code, with an optional type, in the same order as defined in the Borland C++ function declaration. If no type is specified, it is assumed to be `WORD`, or a type may be selected from one of the types listed as follows (or by specifying an assembler "structure," similar to a C struct type, but not described here):

`BYTE`	1 byte; 2 bytes for parameters
`CODEPTR`	2 bytes for near procedures, 4 bytes for far procedures

DATAPTR	Pointer to data. Automatically selects 2 bytes for near model, 4 bytes for far model
DWORD	4 bytes
FWORD	4-byte far pointer size
PWORD	4-byte far pointer size
QWORD	8-byte (quad word) size
TBYTE	10 bytes
WORD	2-byte size

Without the use of ARG, each reference to a parameter value must be manually translated to an offset with respect to the BP register, with increased chance for programmer errors. Assembly language procedures should always preserve the stack frame pointer, which is kept in the BP register. Each time you call a function, the return address and other information, such as parameter and local variables, is stored on the stack. The information that is collected at each function call is referred to as the stack frame. Programs access parameters and local variables as an offset from the BP register, which always points to the current function's stack frame. For this reason, the first thing the function does is to preserve the old value of BP and set BP to the current top of stack:

```
push    bp
mov     bp, sp
```

At the end of the function, the old value of BP is popped from the stack.

The heart of the routine copies the parameter value *n* to the CX register and sets up ES:DI to point to the start of the Values array. The statement

```
xor     ax, ax
```

is a quick way of setting AX to 0 by exclusive-oring the contents of AX with itself.

The label countem marks the start of the loop construct to scan through the array, adding each of the elements in values to the AX register. The loop opcode decrements the value in the CX register and, if nonzero, jumps to the label specified. If zero, the loop opcode falls through to the next instruction, which is the RET opcode to return to the caller. The function value, in this example, is returned in the AX register.

ASSEMBLING AND COMPILING

TASM is run from the DOS command line by typing TASM followed by optional assembler switches, the source filename, and optional object and listing filenames. You may also run TASM from the IDE using the IDE's Transfer menu. When run from inside the IDE, Turbo Assembler attempts to assemble the file in the active editor window. When building Borland C++ compatible object modules, you can invoke TASM from the command line like this:

```
TASM     /ml source;
```

The `/ml` command-line switch ensures that lowercase public symbols within the assembler source code will be preserved.

Compile the C source in the normal fashion, using either the IDE or the stand-alone compiler, bcc. If you use the IDE, you must add the TASM-produced .obj file to the project. This causes the .obj file to be linked into the program. If you are using bcc, you need only to name the .obj file on the command line. To compile and link the sample programs presented in this section, type these commands:

```
tasm /ml sum;
bcc -ml demotasm.c sum.obj
```

When you run TASM, you may optionally produce a listing file. If you add two trailing commas, like this,

```
TASM source,,
```

Turbo Assembler produces *source*.LST, containing your source code and the corresponding machine code that has been generated. This provides helpful information to assist with debugging your programs. For the sample program shown here, typing

```
tasm sum,,
```

produces sum.obj and sum.lst.

When using assembler programs, you often will find that the IDE's built-in debugging facilities (which operate only at the C or C++ statement level) are inadequate. In these situations, you need to use Borland's Turbo Debugger program, which is a stand-alone debugger providing both C and C++ statement-level debugging, as well as assembly and machine code statement debugging and access to the CPU's registers, stack, and other critical memory areas. Use of the Turbo Debugger is described in Chapter 11, "Debugging Techniques." If you are using the IDE to create Turbo Debugger-compatible programs, select Options | Debugger and set the Debugging option in the dialog box to Standalone. If using bcc, be certain to use the `-v` command-line switch.

CREATING SOFTWARE FOR THE INTERNATIONAL MARKETPLACE

BY CYNTHIA FINNEL-FRUTH AND BOB FRUTH

International markets for software have become increasingly lucrative in recent years. Many software developers based in North America have found additional markets for their software in Europe and Asia. However, for a variety of reasons, many software products have not been favorably received in international markets. In this chapter, we will identify some common pitfalls and often overlooked concepts that have hindered product acceptance.

System differences

Using resources for translation

Input and Output considerations

Character support, sorting, and searching

LOCALIZATION VERSUS TRANSLATION

Many software developers mistake the term *translation* for the more comprehensive terms *localizability* and *localization*. Translation is the process of changing a program's on-screen text from one language to another—for instance, from English to German. Localizability is a measure of a product's readiness for localization. Much up-front work in the base product is required to prepare it for localization. This up-front work includes collecting all program text in a single resource file or module, supporting the data formats required by all target markets, defining sorting sequences, and supporting different code pages and character sets. The process of localization takes place after this up-front work is complete. The more localizable a product is—that is, the more complete and thorough the up-front work in that product—the simpler the localization process. When a product is completely localizable, localization includes resetting the program's default settings and translating the program text. All other functionality has already been built into the base product, and so does not have to be added to or modified for localized versions. Products that are not as completely localizable may require significant reengineering and follow-up testing. The goal is to create a localized product that has the look and feel of a product created specifically for a localized market.

MORE THAN JUST CHANGING THE LANGUAGE

To create a truly localized product, you must change more than just the language. For example, most Europeans use the 24-hour clock, rather than the familiar 12-hour, A.M./P.M. format used in the United States. A time display of 11:32 P.M. is not as meaningful to a European as a display of 23:32. Both time displays share the same internal data representation within a program; only the format is different. If you were evaluating two different software products in order to make a purchase, and the two were identical except for data displays such as the time example above, you would choose the product that supported the more familiar format. Purchasers of software in international markets follow the same decision process, and will choose the product that allows them to see their data in the formats most familiar to them.

LOCALIZATION IS COUNTRY SPECIFIC

Products are localized for a specific country, not a specific language. The defaults for such settings as data formats should be appropriate for the intended country. This concept is best illustrated with a specific example. Both the United States and the United Kingdom share a common language, English, and so it is safe to assume that an English-language product would be appropriate for both markets. However, the average computer user in the U.S. displays time in the familiar A.M./P.M. format and dates in the form *mm/dd/yy*. The average computer user in the U.K. uses the 24-hour format for displaying time and shows dates in the form *dd/mm/yy*. In addition, some common English words, such as *color* and *colour*, are spelled differently on opposite sides of the Atlantic. Ideally, products should be available in several English language versions that reflect these differences. It is accepted practice, however, to produce a single English-language version, as long as different data formats and the like are supported. If a product has any features that use dictionaries, such as a spell checker, a dictionary appropriate to the targeted market should be included. Computer users in the U.K. will not accept *colour* flagged as a misspelling, just as U.S. users will expect *color* to be spelled c-o-l-o-r.

Though localization involves more than just the language, it is important to fulfill the requirements of the target market's language. For example, many languages include accented characters that must be supported in all locations of a program localized into these languages. Hebrew and Arabic languages are written right-to-left, which requires the program both to display the on-screen text right-to-left and allow the user to enter their data right-to-left. However, Hebrew specifically requires that numbers be entered and displayed left-to-right. Finally, programs must support double-byte character sets in order to support languages that require larger, completely different character sets. The Kanji character set, which is required for Japanese-language products, is one example of a double-byte character set. Special engineering is required to support these character sets.

SYSTEM DIFFERENCES

Personal computers in the U.S. differ from computers sold in other parts of the world in several significant ways. The most basic difference is the power supply and power cord. The U.S. standard is 110-volt electrical current and the familiar three-prong plug. Much of the rest of the world uses a 220-volt standard, with a variety of plug configurations. None of these power supply considerations should affect your software, but they are worth noting if you mention anything in your documentation about connecting a computer in your documentation.

CODE PAGES AND CHARACTER SETS

To facilitate the support and use of different national languages and character sets, support for multiple *code pages* was added for DOS 3.3. This new feature enabled a single version of DOS (with a localized user interface) to support the character sets used by languages other than English. A code page is the set of characters that can be entered at the DOS prompt. Multiple code pages are required because the standard character set (now known as code page 437) used in the DOS versions prior to 3.3 did not support all of the characters used in the various western European languages. Certain languages required the creation of specific versions of DOS that incorporated both a localized user interface and a modified character set. As a result, product localization before DOS 3.3 required a significant amount of reengineering to support these different language versions of DOS.

Version 3.3 and later versions of DOS recognize and use one of two code pages. These are the default code pages built into the system's hardware BIOS (the hardware code page), and an optional software code page loaded at boot time. When the system is booted, DOS loads the hardware code page into its memory. If the config.sys and autoexec.bat files then load a software code page, this software code page replaces the previously loaded hardware code page in memory. In most systems, the hardware code page is loaded and no software code page is used. For example, systems sold in the U.S. include U.S. code page 437 as the default hardware code page. Most U.S. users will not override this code page setting. Systems sold in international markets either default to a different hardware code page, or are used with DOS versions that load a different software code page at boot time.

The two most important code pages for DOS applications are the United States code page, 437, and code page 850, which is the multilingual code page designed to accommodate all language needs for western European and North American languages. All Windows programs should use the Windows code page, code page 1004. This code page is discussed separately below. Please note that characters 0 through 31 are not shown in the code page listings below, as they are control characters (Ctrl-A, Ctrl-B, and so on). Characters 32 through 126, the familiar U.S. ASCII character set, are the same in all code pages.

Code page 437, the United States code page, is shown in Figure 14.1. This code page is the hardware code page for systems sold in the U.S. It is safe to assume that most U.S. users will not override this code page setting.

032-	033-!	034-"	035-#	036-$	037-%	038-&	039-'
040-(	041-)	042-*	043-+	044-,	045--	046-.	047-/
048-0	049-1	050-2	051-3	052-4	053-5	054-6	055-7
056-8	057-9	058-:	059-;	060-<	061-=	062->	063-?
064-@	065-A	066-B	067-C	068-D	069-E	070-F	071-G
072-H	073-I	074-J	075-K	076-L	077-M	078-N	079-O
080-P	081-Q	082-R	083-S	084-T	085-U	086-V	087-W
088-X	089-Y	090-Z	091-[	092-\	093-]	094-^	095-_
096-`	097-a	098-b	099-c	100-d	101-e	102-f	103-g
104-h	105-i	106-j	107-k	108-l	109-m	110-n	111-o
112-p	113-q	114-r	115-s	116-t	117-u	118-v	119-w
120-x	121-y	122-z	123-{	124-|	125-}	126-~	127-⌂
128-Ç	129-ü	130-é	131-â	132-ä	133-à	134-å	135-ç
136-ê	137-ë	138-è	139-ï	140-î	141-ì	142-Ä	143-Å
144-É	145-æ	146-Æ	147-ô	148-ö	149-ò	150-û	151-ù
152-ÿ	153-Ö	154-Ü	155-¢	156-£	157-¥	158-₧	159-ƒ
160-á	161-í	162-ó	163-ú	164-ñ	165-Ñ	166-ª	167-º
168-¿	169-⌐	170-¬	171-½	172-¼	173-¡	174-«	175-»
176-▒	177-▓	178-▓	179-|	180-┤	181-╡	182-╢	183-╖
184-╕	185-╣	186-║	187-╗	188-╝	189-╜	190-╛	191-┐
192-└	193-┴	194-┬	195-├	196-─	197-┼	198-╞	199-╟
200-╚	201-╔	202-╩	203-╦	204-╠	205-═	206-╬	207-╧
208-╨	209-╤	210-╥	211-╙	212-╘	213-╒	214-╓	215-╫
216-╪	217-┘	218-┌	219-█	220-▄	221-▌	222-▐	223-▀
224-α	225-ß	226-Γ	227-π	228-Σ	229-σ	230-µ	231-τ
232-Φ	233-Θ	234-Ω	235-δ	236-∞	237-ø	238-ε	239-∩
240-≡	241-±	242-≥	243-≤	244-⌠	245-⌡	246-÷	247-≈
248-°	249-∙	250-·	251-√	252-ⁿ	253-²	254-■	255-a

Figure 14.1. Code Page 437, the U.S. code page.

The multilingual code page, code page 850, is shown in Figure 14.2. This code page includes most of the characters required by most western European languages.

Other code pages include 860 (Portugal), 863 (French Canadian), and 865 (Nordic). These code pages were designed to support the specific language needs of various countries, and are commonly used in these countries. The

characters they support, however, are included in code page 850, so if your software supports all of the characters in code page 850, supporting these other code pages should be straightforward. Code page 852, the Slavic code page, includes many characters not included in code page 850 that are required for supporting eastern European languages. If your target markets include eastern Europe, you will need to specifically support this code page. For further information on these code pages, please refer to your DOS documentation.

```
032-   033-!  034-"  035-#  036-$  037-%  038-&  039-'
040-(  041-)  042-*  043-+  044-,  045--  046-.  047-/
048-0  049-1  050-2  051-3  052-4  053-5  054-6  055-7
056-8  057-9  058-:  059-;  060-<  061-=  062->  063-?
064-@  065-A  066-B  067-C  068-D  069-E  070-F  071-G
072-H  073-I  074-J  075-K  076-L  077-M  078-N  079-O
080-P  081-Q  082-R  083-S  084-T  085-U  086-V  087-W
088-X  089-Y  090-Z  091-[  092-\  093-]  094-^  095-_
096-`  097-a  098-b  099-c  100-d  101-e  102-f  103-g
104-h  105-i  106-j  107-k  108-l  109-m  110-n  111-o
112-p  113-q  114-r  115-s  116-t  117-u  118-v  119-w
120-x  121-y  122-z  123-{  124-|  125-}  126-~  127-⌂
128-Ç  129-ü  130-é  131-â  132-ä  133-à  134-å  135-ç
136-ê  137-ë  138-è  139-ï  140-î  141-ì  142-Ä  143-Å
144-É  145-æ  146-Æ  147-ô  148-ö  149-ò  150-û  151-ù
152-ÿ  153-Ö  154-Ü  155-ø  156-£  157-Ø  158-×  159-ƒ
160-á  161-í  162-ó  163-ú  164-ñ  165-Ñ  166-ª  167-º
168-¿  169-®  170-¬  171-½  172-¼  173-¡  174-«  175-»
176-░  177-▒  178-▓  179-│  180-┤  181-Á  182-Â  183-À
184-©  185-╣  186-║  187-╗  188-╝  189-¢  190-┐  191-└
192-┴  193-┬  194-├  195-─  196-┼  197-ã  198-Ã  199-╚
200-╔  201-╩  202-╦  203-╠  204-═  205-╬  206-¤  207-ð
208-Ð  209-Ê  210-Ë  211-È  212-ı  213-Í  214-Î  215-Ï
216-┘  217-┌  218-█  219-▄  220-¦  221-Ì  222-▀  223-Ó
224-ß  225-Ô  226-Ò  227-õ  228-Õ  229-µ  230-þ  231-Þ
232-Ú  233-Û  234-Ù  235-ý  236-Ý  237-¯  238-´  239-
240-≡  241-±  242-‗  243-¾  244-¶  245-§  246-÷  247-¸
248-°  249-¨  250-·  251-¹  252-³  253-²  254-■  255-
```

Figure 14.2. Code Page 850, the multilingual code page.

In order to be localizable, a DOS application must execute successfully on different code pages. It is important to know which code pages are commonly used in the target markets for your software. These code pages should be properly detected and supported, and that support described in the documentation. If your software doesn't support the particular code page used in a target market, this limitation should be documented as well. However, limiting your software in this way will doubtlessly hinder its acceptance in that market. It is better to support multiple code pages in your software, so reengineering is not required for each localized version, saving valuable time and improving the marketability of your software.

As mentioned earlier, all Windows programs should support the Windows code page, code page 1004. This code page is used by 3.0 and later versions of Windows, including all local language versions. When the engineers at Microsoft designed code page 1004, they reorganized the extended portion of the character set (those characters above 127), positioning the various uppercase characters, the familiar 32 characters, below their lowercase counterparts. In addition, code page 1004 leaves out a number of characters that are not supported by Windows. These characters include the DOS line-drawing characters and the letters of the Greek alphabet. Microsoft has reserved the space created by not including these characters for future expansion. Windows code page 1004 is shown in Figure 14.3.

032-	033-!	034-"	035-#	036-$	037-%	038-&	039-'
040-(	041-)	042-*	043-+	044-,	045--	046-.	047-/
048-0	049-1	050-2	051-3	052-4	053-5	054-6	055-7
056-8	057-9	058-:	059-;	060-<	061-=	062->	063-?
064-@	065-A	066-B	067-C	068-D	069-E	070-F	071-G
072-H	073-I	074-J	075-K	076-L	077-M	078-N	079-O
080-P	081-Q	082-R	083-S	084-T	085-U	086-V	087-W
088-X	089-Y	090-Z	091-[	092-\	093-]	094-^	095-_
096-`	097-a	098-b	099-c	100-d	101-e	102-f	103-g
104-h	105-i	106-j	107-k	108-l	109-m	110-n	111-o
112-p	113-q	114-r	115-s	116-t	117-u	118-v	119-w
120-x	121-y	122-z	123-{	124-|	125-}	126-~	127-□
128-□	129-□	130-‚	131-ƒ	132-„	133-…	134-†	135-‡
136-ˆ	137-‰	138-Š	139-‹	140-Œ	141-□	142-□	143-□
144-□	145-‘	146-’	147-“	148-”	149-•	150-–	151-—
152-˜	153-™	154-š	155-›	156-œ	157-□	158-□	159-Ÿ
160-	161-¡	162-¢	163-£	164-¤	165-¥	166-¦	167-§
168-¨	169-©	170-ª	171-«	172-¬	173--	174-®	175-¯
176-°	177-±	178-²	179-³	180-´	181-µ	182-¶	183-·
184-¸	185-¹	186-º	187-»	188-¼	189-½	190-¾	191-¿
192-À	193-Á	194-Â	195-Ã	196-Ä	197-Å	198-Æ	199-Ç
200-È	201-É	202-Ê	203-Ë	204-Ì	205-Í	206-Î	207-Ï
208-Ð	209-Ñ	210-Ò	211-Ó	212-Ô	213-Õ	214-Ö	215-×
216-Ø	217-Ù	218-Ú	219-Û	220-Ü	221-Ý	222-Þ	223-ß
224-à	225-á	226-â	227-ã	228-ä	229-å	230-æ	231-ç
232-è	233-é	234-ê	235-ë	236-ì	237-í	238-î	239-ï
240-ð	241-ñ	242-ò	243-ó	244-ô	245-õ	246-ö	247-÷
248-ø	249-ù	250-ú	251-û	252-ü	253-ý	254-þ	255-ÿ

Figure 14.3. Code Page 1004, the code page used by Windows.

CHARACTER SETS AND FONTS

Rather than using the system code page for their character set, some products include their own, specific character set. This character set is usually the same regardless of which code page is loaded or which language version of the product is being executed. Lotus 1-2-3 for DOS, with its Lotus International

Character Set (commonly know as LICS), is an example of one such product. The major drawback for products with specific character sets is compatibility with other products. Importing and exporting non-native file formats is tricky, as characters must be mapped from one character set to another. In addition, you must include special handling for characters that are not supported in both character sets.

Specific character sets are not particularly recommended for Windows products. All Windows products should support code page 1004 and use its character set. Using a specific character set in a Windows product would severely limit its compatibility with other Windows products and might also hinder its acceptance by Windows users.

If your software supports fonts (such as TrueType or Adobe Type Manager), you should make sure that those fonts include all of the characters commonly used in your target markets. The most direct method for confirming this support is to make sure that the fonts in question include all of the characters in the code pages directly supported by your software.

KEYBOARDS AND KEYBOARD DRIVERS

Probably the most obvious difference between the personal computers sold in various parts of the world is the keyboards. The keycaps are the most visible difference, as the keyboards used in each country are tailored for the linguistic needs of that country's market. The keyboards include keys for those characters that are used in the individual languages, and do not include keys for characters that are not a part of the language. In addition to the specific keycaps, DOS provides a different keyboard driver for each country, and in some cases, multiple drivers for countries such as Switzerland that are multilingual. Many DOS manuals include sections discussing the various keyboard layouts and keyboard drivers. Changing keyboard drivers and code pages is discussed later in this chapter.

HOW TO ENTER CHARACTERS

All keyboard drivers recognize individual keystrokes, as well as the familiar Shift, Ctrl, and Alt key combinations. In addition, some keyboard drivers support two key combinations in order to allow users to enter diacritical marks

(or accents). In most cases, the user strikes and releases the diacritical mark, and then types the character. Due to the expanded character set, many keys are assigned more than one character. The second character assigned to a key is entered by holding down the "Alt Gr" key (always the right-hand Alt key), and then striking the key for the character desired. The Swiss keyboard provides an excellent example of multiple character keys, including three keys that have four characters assigned to them. One character is entered by simply striking the key, one requires the Shift key to be held down, the third requires the Alt Gr key be held down, and the fourth requires both the Shift and Alt Gr keys to be held down. Finally, it is important to note that some keyboards require that the Shift key be held down to enter some numbers. For this reason, we don't recommend you use number keys as universal speed keys.

In addition to entering characters by striking individual keys or keystroke combinations, characters may be entered by typing an *Alt-key sequence*. Alt-key sequences are the method provided by DOS and Windows for entering extended characters and any characters that do not have an individual key. To enter an Alt-key sequence, hold down the Alt key and type the appropriate three-key sequence on the numeric keypad. For example, to enter the British pound symbol (£) in either code page 437 or 850, hold down the Alt key and type `156` on the numeric keypad.

Windows recognizes both three- and four-key sequences. Typing a three-key sequence enters the designated character from the current system code page (437, 850, and so on). For the Windows code page, code page 1004, a four-key sequence is required to enter extended characters. The first character is always a zero. To enter the pound symbol in code page 1004, for example, you need to hold down the Alt key and type `0163` on the numeric keypad. If you leave out the `0`, typing just `163`, the character ú is entered instead.

HOW TO CHANGE THE KEYBOARD DRIVER AND CODE PAGE

Changing the keyboard driver and code page requires modifying both the config.sys and autoexec.bat files, and restarting the system. Table 14.1 below shows examples of the lines that need to be present in the config.sys and autoexec.bat files in order to set up a U.S. system for France, Portugal, and code page 850 for the United States.

TABLE 14.1. CONFIG.SYS AND AUTOEXEC.BAT MODIFICATIONS.

France (Code Page 437) - Changing the keyboard driver only:

config.sys
```
country=033,,c:\dos\country.sys
```

autoexec.bat
```
keyb fr,,c:\dos\keyboard.sys
```

Portugal (CP 860) - Changing both the keyboard driver and code page:

config.sys
```
country=351,860,c:\dos\country.sys
DEVICE=c:\dos\display.sys con=(ega,860,1)
```

autoexec.bat
```
cd\dos
nlsfunc
mode con cp prep=((860)c:\dos\ega.cpi)
keyb po,860,c:\dos\keyboard.sys
chcp 860
```

United States (CP 850) - Changing the code page only:

config.sys
```
country=001,,c:\dos\country.sys
DEVICE=c:\dos\display.sys con=(ega,437,1)
```

autoexec.bat
```
cd\dos
nlsfunc
mode con cp prep=((850)c:\dos\ega.cpi)
keyb us,,c:\dos\keyboard.sys
chcp 850
```

The commands shown in the above examples, including descriptions of the various parameters, are fully discussed in the DOS manual. Consult the sections "International Support" or "International System Configurations" for further details. The "DOS Reference" section may also be consulted for descriptions of the individual commands included in the examples above.

One of the commands shown in Table 14.1, the `country` command, deserves further discussion. The `country` command configures DOS to use country-specific conventions, including date and time formats, decimal separators, and case conversions. DOS also uses this command to identify the character set conventions and standard punctuation used in the specific country. Using the

`country` command in conjunction with the keyboard driver sets the correct country conventions.

Naturally, systems sold in France, Portugal, or any other local market will automatically load the correct keyboard driver and code page when the system is started. It is extremely important to test the base version of your software on all code pages and keyboard drivers that it supports. If the base version passes a thorough test, versions for local markets should require testing only on the code pages and keyboard drivers used in those markets.

OTHER HARDWARE CONSIDERATIONS—PRINTERS, SCREEN DISPLAYS, AND GRAPHICS HARDWARE

If possible, we strongly recommend that you certify your software on the most popular systems and peripherals sold in your target markets. As a rule, most of the hardware, such as memory chips, microprocessors, and graphics hardware and displays are standard throughout the world. However, as in the U.S., systems that claim to be 100% IBM compatible aren't always so. It is better to be safe than sorry, so plan on testing your software on these systems.

It is very important to support the most popular peripherals in your target market. For example, in the European laser printer market, Canon is the market leader in much the same way that Hewlett-Packard leads in the U.S. In general, printers and other output devices are not changed for international markets, although different parts of the world have different standard sizes of paper. Printers and other output devices typically support all commonly recognized international paper formats, in addition to the familiar 8½-by-11-inch paper used in the U.S.

LOCALIZED VERSIONS OF DOS AND WINDOWS

Users in local markets will typically use localized versions of DOS and Windows when these are available. The localized versions have the same features as the familiar U.S. versions, but with a couple of differences. As discussed earlier, the local version of DOS will automatically load the correct keyboard driver and code page for the local market when the system is started. In addition, both DOS and Windows will be set up for the formatting and other conventions appropriate for the individual country.

The most obvious difference is that all of the DOS and Windows text has been translated into the local language. Most English text expands in size when you translate it to another language, because most local versions of DOS take up more memory than the U.S. version. If your DOS program is already tight on memory, this increased memory requirement could be a potential source of problems. Finally, the installed hardware base in a particular market may require that your program support an earlier version of DOS. For example, the venerable PC XT and equivalent machines are the standard in some South American markets.

USING RESOURCES FOR TRANSLATION

The collections of data that makes up a program's user interface is often referred to as program *resources*. Program resources include all of a program's text, including menu items, dialog headings, prompts, error messages, and so forth. Resources also include accelerator or speed key definitions, help message text, bitmaps, and icons. By combining the user interface elements into as few files as possible, you can greatly simplify the translation and localization effort. Programs written for Microsoft Windows almost always use the *resource script* mechanism provided through the Resource Compiler or through the use of the Borland Resource Workshop resource editing program. These tools enable you to easily combine and localize country-specific issues in your Windows applications. DOS applications, on the other hand, must make do without a standard resource system. Techniques for using and managing resources in the translation process are described in the sections that follow.

WINDOWS VERSUS DOS

To facilitate the translation of program text, including menus, dialog boxes, and messages, it is advisable to collect all program text in as few files as possible. This process is relatively painless for writers of Windows applications, as Windows is structured around the use of resource files, which may be readily edited and built using the Windows tools provided. Unfortunately for programmers of DOS applications, DOS does not have a built-in resource architecture. DOS applications need to include their own resource implementation, with accompanying files of program text, or the program text can be

collected in one or more source code modules. Borland's Turbo Vision, with its implementation of resource objects, provides an additional alternative for DOS program development. Regardless, translation is a much simpler process when the program text is isolated from the functional code, and the program does not have to be relinked when the program text is translated.

WINDOWS: USING RESOURCE FILES TO IMPROVE LOCALIZABILITY

All Windows application programs should use Windows' resource architecture for storing all elements of the program's user interface and program text. The Windows Resource Compiler (RC) takes a resource script as input and adds the program elements described by that script to the program's executable (.exe) file. The Resource Compiler allows the developer to modify the program text without having to recompile and relink the entire program—a definite advantage when creating localized versions. Resource scripts allow the developer to create a single file that includes or refers to all of the resources used in an application. These resources can include menus, dialog boxes, accelerators, cursors, icons, bitmaps, fonts, string tables, and any resources of a type defined specifically for a particular application.

The syntax of resource scripts also allow developers to document the various resources inline, right where they are defined or referenced. Developers can use the editors provided by the Borland Resource Workshop to create the different types of resources included in the typical Windows application. During translation of the program strings, translators can also use this set of tools. For further information on the Borland Resource Workshop, the Windows Resource Compiler, and resource scripts, refer to the *Borland Resource Workshop User's Guide*, the *Borland C++ Tools and Utilities Guide* and Volume II of the Windows API, respectively.

String tables are a particularly powerful type of resource, as developers can readily define and store the character strings required by their program. These character strings may include error messages, formula or macro keywords, and various data strings such as month names or days of the week. We strongly suggest you group character strings into logical tables—for example, gather all error messages into a single table, or group together the month names (January, February, March, and so on). You then can reference individual strings by indexing into the appropriate table. By the way, if your program requires the use of both full names and their abbreviations, such as months or days of the

week, you should provide separate tables for these different sets of data. In some languages, the days of the week and month names are not abbreviated using the first three characters.

BITMAPS AND ICONS

To ease the translation process, we recommend that developers use as little text as possible in icons and bitmaps. Text in both of these user interface elements is very hard to translate, as it is difficult, if not impossible, to allow for the expansion (up to twice as long) of text when the text is translated. In addition, it is difficult to accurately translate text that is a part of a graphical representation. Translators may not have access to the right tools for editing icons and bitmaps. Translation proceeds more smoothly when icons and bitmaps are designed without text for the base product, and then don't need to be touched for any localized versions.

Developers should also beware of the pictorial images they include in bitmaps and icons. Images can have different meanings in different countries, or they may have no meaning at all. For example, a date icon that reads "Jan 12" might be very clear to most users in the U.S., but Dutch users will not understand this icon to mean a date—Jan is a very common name in the Netherlands. Symbolic images are not always safe either. An American or Canadian user will probably understand an icon containing an image of a baseball home plate, but this image will mean nothing to users who are not familiar with the game of baseball. It is best to stick with universally recognized icons, such as the familiar Windows icons.

SPEED KEYS AND ACCELERATORS

Speed keys and accelerators should be defined carefully and must not be hard-coded. They should be placed in a resource file so they can be translated into key combinations appropriate for individual languages. For example, Ctrl-S is an appropriate speed key for a Save command in the U.S., but would be confusing in other languages whose word for save does not begin with the letter *S*.

In defining and later translating accelerators and speed keys, it is important to match the conventions of the local markets and the versions of DOS and

Windows used in those markets. This is especially important for Windows programs, as each localized version of Windows will have its own defined set of accelerators. For speed keys, developers should confine themselves to single-stroke characters and key combinations. It is important to maintain this single-stroke rule when translating speed keys. Translating a speed key into a key combination that requires multiple keystrokes defeats the purpose of the speed key. Finally, as mentioned earlier, number keys should not be used as speed keys, since some keyboards require that the Shift key be held down to enter some numbers.

HELP TEXT

For defining, storing, and displaying help text, all Windows programs should use the Windows Help Compiler (HC) and Windows Help application (winhelp.exe). Help text may be edited (or translated) using any editor or word processor that supports Rich Text Format (RTF) file format. For further information on the Windows Help Compiler, refer to the *Borland C++ Tools and Utilities Guide*. The syntax for WinHelp is documented in Volume I of the Windows API.

DOS

DOS does not provide a built-in resource system. DOS applications need to include their own resource implementation, or the program text, including menus, dialogs, and messages, should be collected in a few program modules—preferably a single module. Borland's Turbo Vision can also be used as a basis for developing DOS programs. In most early DOS applications, the program text is sprinkled throughout the source code, which leads to lengthy and tedious translation efforts and much reengineering. Even though it may cost a nominal amount of time up front to centralize a program's text, the savings in the long run will more than make up for that cost.

If a DOS application cannot utilize Turbo Vision or some other resource implementation for defining and storing program text, then the program text should be defined in a single source module. When the program text is translated, only this module must be edited and compiled before the localized program version is linked. No other source modules need to be changed or recompiled. Within this module, the program text, including menus, prompts,

dialogs, data strings, and other text, can be organized into logical groups and tables and declared as public variables so they can be referenced externally by the rest of the program. It is important that the rest of the program be able to handle text that may be expanded in localized versions. Like any source code, the program text should be thoroughly documented. Completely documenting the usage of all program text and any limitations imposed on it (especially the maximum length) becomes particularly important when the text is translated. Incomplete documentation increases the risk of a poor or incorrect translation.

If your program does not contain much text or does not require a large system stack, all of your program text can probably be stored in the data segment. However, many DOS applications quickly exhaust the limited 64K data segment (DS). In such cases, consolidating the program text in a single module can readily allow the developer to overcome data segment overflow by storing program text in code segments. This process is most easily accomplished by defining the program text module in assembly language. Within this assembly language module, the program text is stored in various code segments and accessed by double-word pointers (32 bits, in the form *segment:offset*) defined in the data segment. Listing 14.1 shows an example of a table of English month names defined in the code segment `MonthNames`, as well as the double-word pointer defined in the data segment that is used to access this table and the data in it.

LISTING 14.1. SAMPLE CODE SEGMENT STORAGE OF PROGRAM TEXT.

```
 1  data SEGMENT PUBLIC WORD              ; begin data segment
 2  MonthPtr DW   MonthNames              ; pointer to segment
 3           DW   MonthNames:MonthOffset  ; and offset
 4  ENDS                                  ; end data segment
 5
 6  MonthNames SEGMENT PUBLIC WORD        ; begin code segment
 7  MonthOffset   DB   "January",0        ; Table of full month names
 8                DB   "February",0       ;        - in order
 9                DB   "March",0          ;        - ASCII zero strings
10                DB   "April",0          ;        - so terminate with 0
11                DB   "May",0
12                DB   "June",0
13                DB   "July",0
14                DB   "August",0
15                DB   "September",0
```

```
16                DB   "October",0
17                DB   "November",0
18                DB   "December",0
19  END                                  ; code segment MonthNames
```

Your C++ code should define this pointer as an `extern` far pointer to an array of `char` and access the table as `*yourpointer[monthnumber]`. Turbo Assembler provides various segmentation directives, including the `Segment`, `Ends`, `Group`, and `Assume` keywords for the express purpose of creating and accessing segments. Consult the Turbo Assembler 3.0 User's Guide for further details on these directives.

The cautions discussed above concerning bitmaps and icons in Windows programs also apply to DOS applications. Text in bitmaps and icons is no easier to translate in the DOS environment than it is in Windows. Misleading or incorrect pictorial images are not platform dependent. The earlier string table discussion also applies to DOS applications, as separate tables for abbreviations are required regardless of environment. In addition, DOS developers should heed the warnings regarding speed key and accelerator definition and translation. Finally, the help text for DOS programs should be stored in some easily edited format that allows for expansion. Assume that the help text will expand by as much as 50 percent when it is translated, and will overflow a fixed-sized help screen or poorly designed help system. Programs must allow for the expansion of help text, or face problems ranging from simple truncation of the text to program termination.

FORMATTING DATA

Computer users expect to view their data in formats familiar to them. The standard formats for data are different in various parts of the world. Increasingly, users want complete flexibility in formatting their data. They do not want to be limited to a single, hard-coded format, but prefer to choose from a selection of formats. In addition, hard-coding formats necessitates each local language version to be reengineered, which is time-consuming and costly. Hard-coding formats will also adversely affect compatibility between language versions.

The International portion of the Microsoft Windows Control Panel provides a thorough example of data format selection. In addition, the Control Panel allows you to easily see which formats are generally used in various countries. To see the formats used in a particular country, select the International icon from the Control Panel window, move the cursor to the Country pull-down list box, and use the up and down arrows to scroll through the list of countries until the desired one is highlighted.

If you are writing your program for the Windows environment, the Control Panel settings can be easily retrieved from the win.ini file using the Windows API calls (functions `GetProfileInt` and `GetProfileString`) provided for that purpose. The Control Panel's international settings are contained in the `[intl]` section of the win.ini file. An example of this section might look like this:

```
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
iDate=0
iTime=0
iTLZero=0
iCurrency=0
iCurrDigits=2
iNegCurr=0
iLzero=1
iDigits=2
iMeasure=1
s1159=AM
s2359=PM
sCurrency=$
sThousand=,
sDecimal=.
sDate=/
sTime=:
sList=,
sShortDate=M/d/yy
sLongDate=dddd, MMMM d, yyyy
```

If your program is a DOS application, data formats can be specified as part of a generalized formatting feature, or during program setup. To simplify localization, it is suggested that your program include a set of default settings that can be easily changed to the most common setting for each localized version.

NUMERIC FORMATS

Formats for numeric data consist of three elements, the decimal separator (or decimal point), thousands separator, and a leading zero. The decimal separator is either a period (.) or a comma (,). In general, English speaking countries use the period, and other countries use the comma. However, there are exceptions (Mexico and Switzerland use the period, French-speaking Canada uses the comma), so it is best not to generalize. The thousands separator may be a comma (,), space, period (.), or in Switzerland, an apostrophe (').The *leading zero* is a zero placed before the decimal point in fractional values (.5 versus 0.5, for example). In the U.S., the use of a leading zero is the recognized standard, but in other countries using a leading zero is incorrect.

SUPPORTING MULTIPLE CURRENCY SYMBOLS

If your application calculates monetary figures or otherwise supports monetary data, you've probably included support for the familiar U.S. dollar sign ($). However, the dollar sign is not an appropriate notation when data in other currencies, such as British pounds or German marks, is being displayed. Users in other countries expect to be able to use their own country's currency symbols. Table 14.2 lists the currency symbols for a number of countries. The proper format for each symbol, including decimal and thousands separators, is also given.

TABLE 14.2. SELECTED CURRENCY SYMBOLS.

Country *Notes*	*Symbol and Format*
Australia ASCII 36	$1,000.00
Brazil	Cr$ 1.000,00
Canada ASCII 36	$1,000.00

continues

TABLE 14.2. CONTINUED

Country *Notes*	*Symbol and Format*
Canada—French speaking ASCII 36	1 000,00 $
Denmark	Kr 1.000,00
France	1 000,00 F
Germany	1.000,00 DM
Italy	L. 1.000,00
Japan ASCII 157	¥123
Netherlands	F 1.000,00
Spain	1.000,00 Pts
Sweden	1 000,00 Kr
Switzerland	Fr 1'000.00
United Kingdom ASCII 156	£1,000.00
United States ASCII 36	$1,000.00

As can be seen from the table, currency symbols and formats vary widely from one country to another. Some of the symbols have a prefix format, which means that the symbol is displayed before the data. The U.S. dollar is a prefix symbol. Other symbols, such as the German mark or French franc, have a suffix format and are shown after the data value. In addition to the position of the currency symbol, some symbols are set off from the data by a space. This spacing is key, for a currency format is not considered correct if the required space is not included. Finally, any required punctuation must be included in the currency symbol.

If your program supports currency formats, you may want to consider supporting multiple currency symbols on either an individual data item (field) or a file-by-file basis. To facilitate the support of multiple currency symbols, your program could either permit users to type in their desired symbol or allow them to choose from a list of currency symbols. The International window of the Microsoft Windows Control Panel program combines both of these methods. Users are presented with a list of standard formats they may override by making changes in the text and list boxes provided.

The format used for displaying negative monetary values is not necessarily the same as the format used for displaying other negative data. For example, in the U.S. the format for negative data is a prefixed minus sign, for example, –1,000.00. However, negative monetary amounts may be formatted using either the prefixed hyphen or parentheses, for example, ($1,000,00). In addition, with a couple of notable exceptions, the currency symbol is displayed within the negation notation (for example, –$1,000.00). The exceptions are Denmark and Switzerland—in both of these countries, the minus sign (–) replaces the space between the currency symbol and the value (for example, Fr–1'000.00).

DATE FORMATS

Date data is usually represented by a short format, although a long format may also be supported. Nearly all programs that handle date data support the short format. In the United States, the short format is notated as the familiar *mm/dd/yy*, where *mm* is the month, *dd* the day and *yy* the year (4/19/86, for example). Other common short formats are *dd/mm/yy* and *yy/mm/dd*. The date separator is most often a slash (/), but hyphens (-) and periods (.) are also commonly used. For the sake of program usability, it is recommended that programs support all three formats and all three separators and allow users to choose the ones they want. Users will also appreciate being permitted to set optional leading zeros for the day and month, and either two- or four-digit years. In some countries, the short data format is considered incorrect if the leading zero isn't present or if only a two-digit year is displayed.

For the long date format, the day of the week may be added, usually before the rest of the date, the name of the month is spelled out, the date separators may be changed or dropped, and the year is expressed as a four-digit number.

In the U.S., *4/19/86* becomes *Monday, April 19, 1986*. Note that the order of the day and month is generally maintained from the short format to the long format, although this is not always the case. For some countries, the long date format includes some additional words. In Spain, for example, *19/04/86* becomes *"Sábado 19 de Abril de 1986."* Naturally, for localized versions appropriate translations for the days of the week and the month names should be used. Whether or not a program supports the long date format in addition to the commonly expected short format is subject to the discretion of the program's creators.

TIME FORMATS

Time data is generally either the time of day, or an amount of elapsed time. The time of day can be displayed using either a 12-hour or a 24-hour clock. Twenty-eight minutes before midnight is expressed as *11:32 pm* using the 12-hour clock, and *23:32* using the 24-hour clock. Note that the A.M./P.M. suffix is expected when a 12-hour clock is used for time display. Whether or not the am/pm is capitalized may be left up to the user. The U.S. uses the 12-hour clock, while much of the rest of the world, including Europe, use and expect the 24-hour display.

Both the time of day and elapsed time use a time separator. The colon character (:) is the most common separator, but a period (.) is not uncommon. A few countries, such as Switzerland, use a comma (,) for the time separator. In addition, some countries also require you to display hours using two digits (for example 09:09 am, or 01:23). In the Windows Control Panel, this setting is labeled Leading Zero.

LIST SEPARATORS

The comma (,) and semicolon (;) characters are the most predominantly used separators for a list of numbers (10, 20, 30) or text items (blue; gold; green). In general, countries that use a comma for the decimal separator (or decimal point, as discussed earlier in this chapter), will use the semicolon as the list separator. It is important to support list separators other than commas, because otherwise it would be impossible to differentiate between commas used as list separators and commas used as decimal separators in a list of numbers.

INPUT AND OUTPUT CONSIDERATIONS

When a localizable program performs input and output to disk files, you must also be aware of the translation issues that can arise when exchanging data files across country borders. For instance, can your localized U.S. word processor exchange documents, without losing characters with your localized German version of the same word processor? Does your program permit the entry of, and correctly sort, international characters in filenames? Other problems can occur when your application must print on to international paper sizes—paper sheets that are probably not 8½ by 11 inches. These concerns of input and output handling are described in the next few sections.

FILE I/O

As noted previously, the internal file input and output routines of both DOS and Windows are the same in all localized versions. A well-behaved program's error-handling system generally doesn't require modification, as long as the error messages are translated into the target language.

Another file I/O related concept that developers should be aware of is directory and file names. DOS and Windows do support extended characters in directory and file names, and so programs should support the entry of these characters in directory and file names. However, both DOS and Windows present some technical hurdles to the use of extended characters in directory and file names. Some extended characters are not present in all code pages, or may have different ASCII values in different code pages. If users use any of these characters in their file or directory names, files that are named under one code page may not be accessible when another code page is loaded. In addition, Windows will map some extended characters from the Windows code page (code page 1004) to the OEM character set, which effectively prevents users from retrieving files with those characters. Software should handle these cases as cleanly as possible, without relying on advisories in the documentation. However, the documentation should strongly recommend that users refrain from using extended characters in file and directory names. For more information on code pages, please refer to the discussion earlier in this chapter.

FILE FORMATS

If your program uses its own proprietary file format for storing data, it is important to design that format with the data values stored separately from the data formats. When your program loads a data file, it should load the data values first, and then read the formatting information and apply it to the values. For example, numeric data can be stored in a number of different formats that do not include such format settings as the decimal and thousands separators and the leading zero. After your program has read and interpreted a numeric data item, it can then apply the appropriate numeric format for that item. The format may be specific for that particular data value, it may be the setting for the entire file, or it may be the program's defaults. The settings from the Control Panel can be used as the defaults for Windows programs, and in fact you are strongly encouraged to use the Control Panel settings.

In addition to numeric data, other types of data values can be stored independently of their formats. Date values are typically stored as Julian numbers, with January 1, 1900 given a value of 1, and each succeeding day is 1 greater than the preceding day. Time values are often stored as a fractional value between 0 and 1, with noon having the value .5. The advantage of using these methods of encoding date and time data is that data items may include both a date and a time. The ordinal portion of the encoded value gives the date, and the fractional part the time. Once the data item has been read and decoded, the appropriate format can be applied.

Monetary data should also be stored separately of its formats, but must be handled differently than other types of data. The currency symbol for a particular data item should not be changed without the user taking some specific action on that item. In particular, the currency symbol setting from the Windows Control Panel should only be used to set a program's defaults. Once the user has entered some monetary data, the Control Panel setting should not affect the currency symbol for that data.

In addition to data values, several other types of information should be given special treatment when saved in a file. These types include data labels and function keywords. Data labels such as yearly quarters (as in `First Quarter, 1992`) and keywords (such as `Total`) should be saved as tokenized values. As a data file is loaded, these values can be matched to the appropriate label or keyword string from the program's resource file. When program data is encoded using this method, users see their data displayed in their local language. This feature

is a real advantage if your program is used in multiple European markets. For example, a German user can send a data file to a colleague in France, and that French user will see the data displayed in French, not German.

KEYBOARD INPUT

When processing user input, a program should handle all of the characters that a user can possibly enter for each code page that might be used with it. In addition, the program should understand and properly handle all of the various keyboard drivers and the different methods they provide for entering characters. For further information on code pages and keyboard drivers, refer to the previous discussions in this chapter and your DOS documentation.

MOUSE INPUT

With one notable exception, no changes in your program's mouse support should be necessary for localized versions. Mouse input is straightforward, as mouse drivers provide information on the movement of the mouse and the state of the mouse buttons. The exception regards a program's on-screen text and other user interface elements. These various elements often change position when a program is translated. If your program is dependent on the position of text or other on-screen elements for any reason, including its mouse support, it is critical to update these dependent portions with the new positions.

OUTPUT

When localizing your software, it is important to recognize that U.S. standard letter-size paper, 8½ by 11 inches, is not the standard for much of the world. For countries that use the metric system, the predominant standard size for paper is A4, which is based on metric units and measures 210 by 297 millimeters (21 by 29.7 centimeters or approximately 8.25 by 11.7 inches). When writing your software, it is important to keep these different paper sizes in mind, and modify your output code accordingly. A graphics image that is always centered for U.S. standard letter size paper will not be centered on A4 paper. European computer users, who use A4 size paper, expect the software they use to support A4 paper correctly and compensate for different sizes of paper.

In addition to different paper sizes, it is also important to properly use the multiple paper trays that are a common feature in many currently available printers. Many laser printers sold in the U.S. feature multiple paper trays, including a tray for standard letter size paper, and one for legal size paper (8½ by 14 inches). These U.S. printers usually support A4 paper as well. When queried appropriately by an application, these printers may respond with information about the currently available paper trays and the paper they contain. Before a program sends output to a printer, it should first send the appropriate printer control codes for the desired paper size and paper tray.

DISPLAY

Two additional comments need to be made about your software's screen displays. If you have written your program as a WYSIWYG (what you see is what you get) application, you have no doubt paid close attention to matching the screen display to the printed output, taking into account that U.S. users generally use 8½-by-11-inch paper. In order to maintain your program's WYSIWYG properties when it is localized for markets where A4 paper is standard, it is necessary to take different paper sizes into account. Ideally, the WYSIWYG setting in your software should be tied to the printer's page size setting, whether it is A4, U.S. letter size—whatever.

If your program employs any on-screen measuring devices such as rulers, it is important to remember that most of the rest of the world uses the metric system. Needless to say, computer users who are accustomed to the metric system will not look favorably on software that displays rulers laid out using the English system.

CHARACTER SUPPORT, SORTING, AND SEARCHING

When creating a product for international use, even simple functions such as converting a lowercase letter to uppercase require special handling to correctly accommodate international characters. Simple string comparisons, such as the C library function `strcmp()` will fail when given equivalent international characters. For example, an accented character such as *á* should probably be treated as equal to the English letter *a*, but the standard C library routines will

treat these letters as unequal. How these routines are implemented affects all searching and sorting operations within the product. Suggestions for dealing with these problems are described next.

BORLAND LIBRARIES AND THE WINDOWS API

Both DOS and Windows applications will require various character functions for performing such operations as character identification and uppercasing or lowercasing a character. Windows developers can either make use of the Windows string-manipulation functions documented in the Windows API, Volume I, or they can implement their own routines. Developers of DOS programs do not have this choice, as the routines included in the Borland C++ Runtime Libraries are insufficient. As documented in the Borland C++ Library Reference, these routines do not handle the extended character set, as they recognize only characters with an ASCII value of 127 or less. Because of this severe limitation, these routines should not be used. DOS developers should plan to write their own routines for identifying characters and uppercasing and lowercasing characters.

CHARACTER IDENTIFICATION

A set of character identification routines should include functions to determine whether a character is an alphabetic, uppercase, lowercase, numeric, or alphanumeric (either alphabetic or numeric) character. Some developers may find that their programs need additional routines for identifying punctuation characters or handling currency symbols (which may be one or several characters, as discussed earlier).

When developing routines for uppercasing and lowercasing characters, developers need to be mindful of several requirements. First, these functions should act only on alphabetic characters. All other characters should be returned unchanged. Second, the uppercase routine should ignore any uppercase characters. Similarly, the lowercase routine should not change any lowercase characters. Both routines can quickly modify alphabetic characters that have ASCII values of 127 or less by subtracting 32 from the ASCII value to uppercase, and adding 32 to the ASCII value to lowercase.

The proper handling of the extended character set is straightforward for most countries. When lowercase accented characters are uppercased, they are mapped to the same uppercase base character with the same accent. For example, *é* is mapped to *É*. If no uppercase character with the same accent exists, the lowercase character is changed to the base uppercase character without an accent, and the accent is not preserved. Similarly, uppercase characters are lowercased by mapping them to the same lowercase base character with the same accent. If no lowercase character with the same accent exists, the uppercase character is changed to the base lowercase character without an accent, and the accent is lost. The most notable exception to these rules is France. For France, lowercasing is handled the same way, but uppercasing is different. Lowercase characters are mapped to the unaccented base character when uppercased. The accent is not preserved. For example, for most countries, *é* is mapped to *É*, but in France, *é* is mapped to *E*.

SEARCHING AND STRING COMPARISONS

For any product that includes sorting or searching features, it is important to research what is customary for these features in the target markets. Users in different countries have different expectations with regards to sort order and character matching and comparisons. If a product handles sorting and searching incorrectly for a target market, that product can be rendered impossible to use and doomed to failure in that market.

For searching, it is important for programs to distinguish between case-sensitive comparisons and case-insensitive comparisons of characters. For case-insensitive comparisons, programs should make use of the uppercase and lowercase routines described earlier. In most languages and countries, accented characters should be considered the same for searching as their unaccented counterparts. In the U.S., for example, when searching for all words that begin with the character *a*, the user would expect to retrieve all entries beginning with any type of *a*, accented or not. In general, this is the same in other countries when the base character is considered equivalent to the accented character. However, there are cases where the accented character and the base character are not considered to be equivalent. For example, in Sweden *å* and *ä* are not equivalent to *a*. Therefore, a Swedish user searching for all words that

begin with *a* would not expect to retrieve any words beginning with either *å* or *ä*. Programs should be able to search for all characters present in the current character set, regardless of whether or not the current language uses all of those characters.

COLLATION SEQUENCES

Collation sequences, or sort sequences, vary from country to country. A collation sequence delineates the order in which characters are sorted by assigning a value, or weight, to each character. A collation sequence is determined by a number of different considerations. These include the order of the alphabetic characters, whether or not accented characters are given the same weight as their unaccented base counterparts, and whether or not accented characters have their own position in the sequence. In many countries, the position of numeric characters is not specified, and numbers are usually sorted after all alphabetic characters. However, in some countries, there are specific requirements for sorting numbers. In Sweden, numbers are sorted before all alphabetic characters. In most other Scandinavian countries, numbers are specifically sorted after all alphabetic characters. Although collation sequences usually include only alphanumeric characters, it is important for sort routines to handle punctuation characters and symbols. The convention is to sort these characters after all alphanumeric characters.

Language differences also affect collation sequences. Accented characters are treated differently in different languages and countries. In some countries, all accented characters are given the same weight as the base unaccented character, while in other countries, some or all of the accented characters are weighted individually. For example, in Spain all *a*'s with accents are given the same weight in the collation sequence as the unaccented character *a*. Words that begin with the letters *a, á, à, â, ä, å,* and *æ* will appear together, unordered, in a sorted word list. The collating sequence for Sweden presents a different picture with regards to the family of *a* characters. In Sweden, *a* and *á* are given the same weight. The other *a* family characters, *å* and *ä,* are sorted separately (in that order) near the end of the alphabetic characters in the Swedish collating sequence. This collating sequence also includes an example of two characters, *y* and *ü,* that have equal sort weights even though they have different base characters.

Character expansion and compression can also affect collation sequences. For example, the *ß* character in German expands to *ss* and is treated as two characters for sorting purposes. In Spanish, the two character sequences *ch* and *ll* are treated as a single character.

For localization purposes, it is safe to assume that the collation sequence is different for each target market. Therefore, a collation sequence should be defined for every target market. Each collation sequence should include all possible characters used by the program, regardless of whether the various target languages use all of these characters. Ideally, all of the collation sequences defined should be included in the base product. The product can determine which sequence to use by either matching the sequence to the current language version or by checking the language setting in the Windows Control Panel settings. Finally, it is important to include all of the defined collation sequences in the program's documentation. Users will want to know in what order their data will be sorted. In addition, it is important for users to understand how the program will handle alphabetic characters symbols included in the code page that are not generally used in their country or language (for example, handling accented characters in the U.S.).

Defining a collation sequence for a target market is a straightforward process. First, it is important to research the collating conventions of that target market. Once this research is complete, the developer can create a table containing entries for all of the characters in the code page generally used in the target market. Each of these entries should list the character and a value specifying that character's position in the collation sequence. The positional values typically begin at 1 and increase as the sort order progresses. If two characters are assigned the same weight in the collation sequence, they should be given the same positional value. In the Spanish example discussed earlier, *a*, *á*, *à*, *â*, *ä*, *å*, and *æ* should all be assigned the same positional value in the collation sequence.

WINDOWS CONTROL PANEL

As described earlier, Windows programs can and should access and use the Control Panel's International settings. These settings can be accessed statically, that is, only once when the program is started, or dynamically, whenever

the user executes the Control Panel and changes the settings. There are advantages and disadvantages to both of these approaches. Dynamically reflecting any changes made while a program is running gives the user the maximum amount of flexibility. It allows him or her to globally change those Windows settings without having to exit and restart the individual program. However, continuously checking for changes can slow a program's performance. Dynamically reading changes in the Control Panel's settings can cause problems when data is shared with another program that is accessing the Control Panel settings statically. It is very important to decide at what point your program will read the Control Panel settings and how it will support them, and then document this feature thoroughly.

If your program will link to other programs, understanding at what point these programs access and use the Control Panel settings is very important. This understanding is crucial if your program will communicate with other programs using the Windows DDE (Dynamic Data Exchange) capability. Because DDE provides a data bridge (but not a data conversion) between different products, it is important that linked products follow the same data and country conventions. The most important of these conventions are decimal and list separators, but correctly passing all data formats is necessary to avoid data corruption and to ensure accurate data transfer. To this end, products that share data must be using the same Control Panel settings. One of the major disadvantages of accessing the Control Panel settings statically is that the user can change the settings after the program has read them during startup. Subsequent Control Panel changes will not be used by such a program, causing an incompatibility with other programs, including those started under the new Control Panel settings and those accessing the settings dynamically. This incompatibility can cause major data loss or corruption when the data formats have changed, causing incorrect interpretation of data. For example, a date with a period separator may be misinterpreted as an erroneous numeric value, or a changed decimal separator setting may cause currency values to vary unexpectedly. For the developer, the best course of action is to understand how other programs to which your program will be linking operate and match their capabilities. Then design your program accordingly, and carefully document these features for the user.

QUALITY CONSIDERATIONS

There are several things to consider when choosing a translator or translation agency for your software. The most obvious of these is that the translator needs to be fluent in both the language you used for your product's menus and prompts, as well as the targeted language. It would be difficult, if not impossible, for a translator who didn't speak English to translate a product's program strings from English to another language. In addition, it is important for the translator to be a computer user who is familiar with software similar to your product. Technical writing is also a useful skill for translators to have. Translators do not need to have programming skills, as long as they understand completely how your product works and how it interacts with the hardware on which it runs. It is important to consider the level of technical expertise possessed by the translators, and match the requirements of the tools used for translating the program text, help text, and documentation with their expertise. Due to the size and content of help resource files, documentation translators often translate the help text. Documentation translators may not be as well versed in technical issues as the translators accustomed to translating program files.

The translator does not necessarily have to be a native speaker of the target language, although that is certainly a plus. Many software companies have found it most practical to use translators and translation agencies located in the target country. They feel that this gives them an advantage and may help the localized product gain acceptance in that local market. However, they also have to overcome the logistical obstacles of customs, currency exchange, and distance. Whether these obstacles outweigh any advantage gained from using a local translator is something software developers have to decide for themselves. Regardless, the translator you choose should be familiar with any relevant cultural issues, and common practices, data formats, and terminology used in the target country.

To ensure consistency within a product, we recommend that a single translator or translation agency handle the translation of all translatable parts of a product, including program strings, help text, and manuals. A single translator or agency will potentially have a better overall view of the product, and is more likely to use the same terminology consistently throughout the program text and documentation. There is nothing quite as annoying as trying to use a product that has several different terms for the same item or feature.

Once the translation is complete, it is essential that the translator's work be reviewed by an editor. The editor will proofread the translated text for errors, as well as review consistency, clarity, and understandability.

The process of verifying, or testing, localized versions of software can be divided into two distinct phases. The first phase takes place during development of the base product. The overriding goal of this phase is to confirm as early as possible that no unexpected functional changes will be required for localized versions. One technique commonly used to achieve this goal is the preparation of a "mock translation." A mock translation consists of the base version of the program, with the program strings extended to their maximum lengths. Extended characters are used whenever possible in the strings. It is important for the mock translation to stretch the program to its limits. Mock translations provide several benefits, including early testing of any provided translation facilities and early detection of any unresourced (and therefore "untranslated") program strings. Often, a mock translation will uncover problems associated with extended characters, program strings overrunning their length, or movement of user interface elements (on-screen text, dialog options, prompts, and so on). Problems in this latter group are especially troublesome for DOS products, but can adversely affect localization efforts on any platform or environment.

In addition to problems discovered through preparation of a mock translation, all localization-related features should be thoroughly tested as a part of base product verification. These features include all data formatting, code page support, user input routines, display and output features, and all character-related functionality (sorting, searching, and so on). Since these features are a part of the base product, they are necessarily included in the base product testing.

Once all translation has been completed and a localized version of the product has been produced, the second phase of verification takes place. During this phase, the localized version itself is tested. Its functionality is compared with the functionality of the base product, and any required functional changes are carefully verified. All regression tests used for testing the base product are executed on the localized version, and any failures of these tests are thoroughly investigated. If possible, the regression tests are automated, as this testing must be repeated for each localized version of the product. Ideally, no functional changes have been required and no regression test failures are encountered. After this second verification phase is complete, the localized product is ready for distribution to the target market.

CHAPTER 15

HOW TO WRITE A TSR

BY KARL SCHULMEISTERS

This chapter discusses how to write a Terminate and Stay Resident (TSR) application or an interrupt handler using Borland C++ for the MS-DOS operating system, beginning with a general discussion of TSRs, interrupts, and interrupt handlers. You will then begin to build a simple TSR while learning about TimerTick and MS-DOS IdleLoop Interrupts. I will discuss the TesSeRact standard for TSRs, and you will see how to pop up a TSR using a hot key. The sample code will allow the TSR to pop up both graphics and alphanumeric display modes. After a brief discussion of Microsoft Windows, you'll learn how to unload your TSR.

Device drivers, TSRs, interrupts, IRQs, and InLine assembler

General constraints and TSR programming practices

Doing useful things with your TSR

In this chapter, I assume you have a working knowledge of 80x86 assembly language, the MS-DOS INT 21h API, PC video devices, and the Borland C++ 3.0 Library Reference. If you want to learn more about these topics, I suggest the following texts:

Dos Programmer's Reference, 2nd Edition, published by Que Corporation.

Intel iAPX 286 Programmer's Reference Manual, published by Intel Corporation Literature Department.

The MS-DOS Encyclopedia, published by Microsoft Press.

The MS-DOS Programmer's Reference, Version 5.0, published by Microsoft Corporation.

PC & PS/2 Video Systems, published by Microsoft Press.

The Peter Norton Programmer's Guide to IBM PC, published by Microsoft Press.

These are texts that I have found useful in my work, and are secondary only to the keyboard and monitor on my desk. Note that for work with MS-DOS applications, I use the programmer's reference for iAPX 286 CPU. I find this manual to be more comprehensive and easier to read than those covering earlier members of the Intel 80x86 family. Though I do own a copy of the 80386 manual, I prefer the 80286 version, although either of these will do. For those readers who are completely unfamiliar with assembly language programming, I recommend that you seek out a more introductory text (and see Chapter 14, "Using Borland C++ with Other Products"). The Intel manuals are very much reference works and do not attempt to teach you how to use assembly language.

Also useful, especially for those just getting started in writing low-level systems software:

Turbo Debugger 3.0 User's Guide, published by Borland International.

Turbo Assembler 3.0 User's Guide, published by Borland International.

Starting with MS-DOS 3.0, Microsoft made some significant changes to the way MS-DOS interacts with TSRs. Where I provide code examples, these examples will make the assumption that MS-DOS version 3.0 or later is being used. I discuss in the text how to accommodate TSRs running under earlier versions of MS-DOS.

DEVICE DRIVERS, TSRS, INTERRUPTS, IRQS, AND INLINE ASSEMBLER

MS-DOS provides an application with a set of standard methods for interacting with the PC hardware. These methods comprise the MS-DOS APIs (Application Programming Interfaces). They are fundamentally simple, not very powerful, and require a great deal of programming to accomplish even simple tasks. Borland C++ groups these methods into Library functions to provide much greater functionality and power. However, since Borland C++ must address all PC configurations, it is limited to using only those APIs that are meaningful for the majority of PCs.

If you wish to interact with the PC hardware in ways not envisioned by the designers of MS-DOS, or if you need to interact with a custom piece of hardware, you must work around MS-DOS and Borland C++. Fortunately both systems provide a mechanism for doing so. Borland allows for the declaration of interrupt functions through the use of the `interrupt` function type definition and inclusion of assembly language interleaved inline with your C++ code (see Chapter 14, "Using Borland C++ with Other Products"). MS-DOS provides a mechanism for installing device drivers for custom hardware, and for loading similar programs using the Terminate and Stay Resident (TSR) MS-DOS function request.

DEVICE DRIVERS

Device drivers are conceptually simple programs. They receive and process information from a piece of hardware, control the piece of hardware in response to commands from MS-DOS and applications, and convert the data gathered from the hardware into a standard format that can be understood by either MS-DOS or the application. The complexities of device drivers come from the need to be fast (timing in the hardware world is all-important), the lack of debugging tools, the lack of standard APIs, and the sometimes unexpected and undocumented behavior of the device being controlled.

LOAD-ON-DEMAND DRIVERS

TSRs are similar to device drivers. The biggest visible difference is that TSRs are loaded into memory by MS-DOS from the command prompt or a batch file, and device drivers are loaded using the `DEVICE=` entry in the config.sys file. In fact, some TSRs are device drivers. An example is the Microsoft Mouse program. `DEVICE=mouse.sys` loads the mouse driver from your config.sys file. Running mouse.com from the command prompt installs the same driver. Hence, everything I discuss about TSRs in this chapter applies to device drivers as well. However, device drivers must also conform to the MS-DOS device driver specification which imposes further requirements. I will cover some of these requirements later in the chapter. For a more detailed discussion of MS-DOS device drivers, I strongly recommend Chapter 12 of the *Dos Programmer's Reference,* Second Edition.

PORTS AND IRQS

So how does a program communicate with a piece of hardware? Through interrupts, IRQs, and ports. Ports are one-byte-wide gateways that respond to the `in` and `out` instructions of the 80x86 family of central processing units (CPUs). Each IRQ (interrupt request) is one of the 15 dedicated hardware lines that a piece of hardware can set (activate). Setting an IRQ line notifies the CPU that an interrupt request is pending. When the CPU acknowledges this request, a chip called the PIC (Programmable Interrupt Controller) inserts an `int` instruction into the instruction sequence being read by the CPU.

INTERRUPTS

When the CPU encounters an `int` instruction, it pushes the `flags`, `cs`, and `ip` registers onto the current stack. It then multiplies the parameter of the `int` instruction by 4, and uses that as an offset into segment 0. The double word (or `dword`) at that memory location is used as the new CS:IP. In a sense, the CPU "vectors" to the address stored at that particular entry in segment 0. This is why segment 0 is often referred to as the *interrupt vector table*.

The `int` instruction issued by the PIC corresponds to the IRQ line that was set. This is where things get sloppy, because the `int` instruction generated on the PC is not the same as the IRQ that caused it. It is offset by 8. For example, IRQ 0

generates an `int 08`, which vectors through the DWORD offset 0:20h. Thus a mouse card with its jumper set to INTERRUPT 5 is actually set to IRQ 5. The resultant interrupt that the CPU uses is 0Dh which uses the vector at 0:34h.

Why should you care? Because often, documentation that accompanies hardware interchanges the terms INTERRUPT and IRQ freely. In this chapter, *INTERRUPT* will be used only to refer to the actual INT instruction or vector used. *IRQ* will be used to refer to hardware interactions.

Table 15.1 describes the INTERRUPT vectors used by the IBM PC AT (and its successors).

TABLE 15.1. TABLE OF ALL DOS VECTORS.

INTERRUPT Vector	*IRQ*	*Description*
00h 0:0000h		*Divide by zero*. This interrupt is generated by the processor when a divide-by-zero is detected. Most compilers and debuggers hook this vector and gracefully terminate the current program.
01h 0:0004h		*Processor single step*. Not really a vector; when encountered by the processor, it causes the processor to execute the next instruction and stop.
02h 0:0008h		*Non-Maskable Interrupt*. Also called NMI. In the PC memory, parity error invokes this error.
03h 0:000Ch		*Break point*. This is the only single-byte instruction. This allows debuggers to easily insert this instruction into the instruction stream.

continues

TABLE 15.1. CONTINUED

INTERRUPT Vector	*IRQ*	*Description*
04h 0:0010h		*Overflow*. This interrupt is generated by the processor when an arithmetic overflow is detected. Most compilers and debuggers hook this vector and gracefully terminate the current program.
05h 0:0014h		*ROM BIOS print screen vector*. Pressing Shift-PrintScreen issues this interrupt. The ROM BIOS then converts this to as many INT 17h calls as required to print the screen to LPT1.
06h 0:0018h		Unused.
07h 0:001Ch		Unused.
08h 0:0020h	0	*Hardware timer tick*. This is generated by the timer hardware. Applications should not use this because INT 1Ch provides the same service with reduced complexity.
09h 0:0024h	1	*Keyboard interrupt vector*. Every keypress generates this vector. Care must be taken when writing to this vector since applications such as Microsoft Word assume close compliance with ROM BIOS behavior.
0Ah 0:0028h	2	*Reserved*. In AT-class machines, this is the vector that the second Programmable Interrupt Controller is acknowledged through.

INTERRUPT Vector	*IRQ*	*Description*
0Bh 0:002Ch	3	*COM2 port controller*. This is a hardware-generated vector that is normally disabled.
0Ch 0:0030h	4	*COM1 port controller*. This is a hardware-generated vector that is normally disabled.
0Dh 0:0034h	5	*CRT vertical retrace interval*. Used to detect the vertical retrace interval in CGA monitors to eliminate snow on-screen.
0Eh 0:0038h	6	*Floppy disk controller vector*. This vector is beyond the scope of this chapter and should be avoided.
0Fh 0:003Ch	7	*Printer port controller*. Some printer hardware can be configured to generate interrupts to improve performance. MS-DOS does not support this function by default. Often this vector is used by accessory cards as a spare hardware vector.
10h 0:0040h		*ROM BIOS video services*. This vector provides character-mode video services.
11h 0:0044h		*ROM BIOS equipment configuration check*. This is a service provided by the BIOS to allow verification of basic equipment installed at boot time. Not all external drive hardware correctly updates the values returned by this.

continues

TABLE 15.1. CONTINUED

INTERRUPT Vector	*IRQ*	*Description*
12h 0:0048h		*ROM BIOS memory size check*. Returns the size of the system memory detected at system boot.
13h 0:004Ch		*ROM BIOS disk services*. Provides sector-level programming of disks. Used by disk device drivers.
14h 0:0050h		*ROM BIOS COM port driver*. Provides rudimentary I/O to the asynchronous communication ports.
15h 0:0054h		*ROM BIOS cassette service*. Used to interface to the cassette interface on the PC XT, now used by network cards.
16h 0:0058h		*ROM BIOS keyboard driver*. This is a higher-level keyboard handler than INT 09h. Since this vector provides ASCII character codes, it is easier to program to than INT 09h.
17h 0:005Ch		*ROM BIOS printer driver*.
18h 0:0060h		*ROM BIOS BASIC*. Used to invoke or prevent invocation of ROM BIOS BASIC.
19h 0:0064h		*System Restart*. Invoking this vector will cause the system to restart with the ROM BIOS initialization sequence.
1Ah 0:0068h		*ROM BIOS clock services*. This vector allows the program to get

INTERRUPT Vector	*IRQ*	*Description*
		and set the system clock. Care must be used in calling this vector so as not to destroy the date "rollover" flag.
1Bh 0:006Ch		*Control Break vector*. This interrupt is caused when Ctrl-Break is pressed on the keyboard. This is converted internally and reissued as an INT 23h. Because the default handler makes changes to the MS-DOS keyboard buffer, most applications intercept this feature at INT 23h.
1Ch 0:0070h		*Software timer tick*. This vector is generated by the ROM BIOS 18 times per second. TSRs that "awaken" often should chain to this interrupt.
1Dh 0:0074h		This is not an interrupt vector; it is a vector that points to the current video parameter table.
1Eh 0:0078h		This is not an interrupt vector; it is a vector that points to the current disk parameter table.
1Fh 0:007Ch		This is not an interrupt vector; it is a vector that is used to map the PC extended character set (characters 80h–FFh) for CGA Graphics mode.
20h 0:0080h		*MS-DOS 1.0 program terminate*. This is the mechanism used by MS-DOS 1.0 to terminate programs.

continues

TABLE 15.1. CONTINUED

INTERRUPT Vector	*IRQ*	*Description*
		For compatibility reasons, this exists in all versions including MS-DOS 5.0.
21h 0:0084h		*MS-DOS API vector*. All MS-DOS function requests pass through this vector.
22h 0:0088h		*MS-DOS terminate vector*. Provides a mechanism for executing a post-termination clean-up routine.
23h 0:008Ch		*MS-DOS Control-C vector*. This interrupt is generated by MS-DOS whenever a Ctrl-C or Ctrl-Break is detected.
24h 0:0090h		*MS-DOS critical error interrupt*. Also known as the *hard error interrupt*. This interrupt is generated by MS-DOS whenever an error occurs that MS-DOS cannot recover from.
25h 0:0094h		*MS-DOS absolute disk read*. This is used by various MS-DOS utilities to access reserved parts of the MS-DOS file system. FORMAT and SYS are two examples.
26h 0:0098h		*MS-DOS absolute disk write*. This is used by various MS-DOS utilities to access reserved parts of the MS-DOS file system. FORMAT and SYS are two examples.
27h 0:009Ch		*MS-DOS Terminate and Stay Resident request*. This is the vector originally used in MS-DOS 1.0 to

INTERRUPT Vector	*IRQ*	*Description*
		transfer control back to the MS-DOS command prompt permanently without unloading the application from memory.
28h 0:00A0h		*MS-DOS idle loop vector*. This interrupt is generated by MS-DOS while scanning for keyboard input. Many applications do not support this feature and therefore cannot be relied upon to be issued every time it is safe to issue MS-DOS function requests.
2Fh 0:00BCh		*Multiplex and TSR communications vector*. This vector is used by MS-DOS to invoke internal services such as APPEND. It is also used by various TSRs to verify installation and to pass data between the TSR and any foreground application.
40h 0:0100h		*PC XT floppy disk interface*. Not used in ISA-, EISA-, or MCA-class machines.
41h 0:0104h		*Fixed disk parameter table*. Not an interrupt vector; this points to the fixed disk parameter table.
43h 0:010Ch		*Graphics character table*. Not an interrupt vector; this points to the table that defines the alternate character set that is used in EGA+ graphics modes.
70h 0:01C0h	8	IRQs 8–15 generate interrupts 70h–77h.

INFORMATION EXCHANGE

Once a TSR or a Device Driver is installed, no standard mechanism exists within the MS-DOS API for communicating directly to the TSR. Since you often wish to check the status of a device or command the TSR to do something, some mechanism outside the standard MS-DOS API needs to be used. The preferred mechanism is through an interrupt.

As part of the installation process, the TSR intercepts the predefined interrupt. The TSR does this by first saving the existing entry in the interrupt vector table into local memory. It then replaces this entry with a new segment:offset reference that points to the start of the TSR's custom interrupt handler. Since an INT instruction does not modify any of the CPU's registers, the calling program can pass information directly to the TSR in this fashion.

A well-behaved TSR will first check for an identifier unique to the TSR and the invoking application before modifying the contents of the registers. If the identifier is not found, the TSR should simply pass control to the old interrupt vector table entry. It should do this before modifying any of the CPU's registers, in case some other TSR is using this vector for communications. This is known as *chaining* an interrupt vector, since you trap only those interrupts that are specifically aimed at your TSR, and pass "along the chain" any interrupts that your TSR does not recognize.

Unfortunately not all TSRs are well behaved, and it is precisely the possibility of conflicting interrupt vectors that can cause problems when more than one TSR is loaded. Two standards attempt to address this potential trap.

As shown in Table 15.1, INT 2Fh provides multiplex and TSR communication services. The *MS-DOS Encyclopedia* suggests that INT 2Fh be used for determining the presence of a TSR and for communicating with it. Two "standard" methods exist for using this vector. The one suggested by the *MS-DOS Programmer's Reference* involves placing an identification value in the AH register and a function value in the AL register. MS-DOS reserves values AH = 00h–7Fh for use internally by MS-DOS, but imposes no further restrictions. Since this leaves only 128 other possible identifiers, the risk of conflict is very high.

In an attempt to address this without resorting to randomly using other interrupt vectors, a group of TSR developers agreed upon a standard for use of the INT 2Fh vector. Today this standard is known a the TesSeRact Standard. Copies of this standard may be obtained by writing to:

TesSeRact Development Team
1657 The Fairways, Suite 101
Jenkintown, PA 19046
CompuServe 70731,20
MCIMail 315-5415

I will discuss this standard in greater detail later in the chapter.

MULTITASKING

When OS/2 was first announced, the PC press clamored that almost no one needs multitasking. Horse hockey. All of us have at one time or another done two or more things at once. MS-DOS constrains us to doing only one at a time. So if you need to quickly do a calculation while writing a business letter, you must first shut down your word processor, start your calculator, write down the answer, and then restart your word processor. I would rather press a special key sequence and have a calculator pop up over my business letter, do my calculation, push another key, and return to my word processor. This is the single biggest use of TSRs. It is this idea that was at the core of Borland's hugely successful SideKick application.

While general business applications such as SideKick are available, numerous others specific to your environment are not. By the end of this chapter, you will know how to build a TSR application that fits your needs.

MS WINDOWS CAVEATS

Microsoft Windows is rapidly becoming the standard operating environment on the PC platform. TSRs generally do not work well with Microsoft Windows. This is to some extent because Microsoft Windows enhances MS-DOS functionality by working directly with some of the PC hardware. Therefore certain features that are standard to many TSRs will either function improperly or cause your system to crash. Fortunately Microsoft Windows provides a mechanism for detecting whether Windows is running.

I will discuss this in greater detail in the section "Microsoft Windows Gotchas." For now, you simply need to understand that what you are about to read primarily applies to the world of MS-DOS without Microsoft Windows.

USEFUL VECTORS

While there are some 45 Interrupt vectors described in Table 15.1 as predefined by either MS-DOS or the ROM BIOS, only a few are useful to most programmers. The most useful are:

INT 23h, also called the Control-C vector. By chaining to this vector, an application can customize response to CTRL-C or CTRL-BREAK keyboard input. Its usefulness is not limited to TSRs. Any application that needs to prevent automatic termination should hook this vector. Borland C++ and Turbo C++ provide the `ctrlbrk` library call to aid in implementing this handler.

INT 24h, also called the Critical Error vector. MS-DOS issues this interrupt whenever it encounters a failure that might indicate a hardware failure. This is the source of the infamous `Abort, Retry, Ignore` message. Usefulness is not limited to TSRs. All applications that wish to control response to such errors or prevent MS-DOS from displaying this message at the current cursor position should implement this handler. Borland C++ provides the `harderr`, `hardresume`, and `hardretn` library calls to aid in implementing this handler. INT 24h handlers must not issue any MS-DOS function requests above 0Ch.

INT 21h, also called the MS-DOS API vector. All function requests for MS-DOS are routed through this vector. By chaining to this vector, an application can enhance MS-DOS functionality, detect when it is safe for a TSR to do file I/O, monitor MS-DOS service requests or provide a variety of other services.

INT 09h, also called the Keyboard Interrupt. All keyboard input flows through this vector. By chaining to this vector, an application can filter and modify the keyboard input stream. A TSR can detect hot keys or simulate keyboard input.

INT 1CH, also called Timer Tick. This interrupt is issued by the BIOS 18 times per second. A TSR can chain to this vector so that it is guaranteed a mechanism for wakeup.

INT 28h, also called the Idle Loop Interrupt. MS-DOS issues this interrupt while waiting for keyboard input. This interrupt is issued only when it is safe for an application to issue MS-DOS function requests (INT 21h) above 0Ch.

INT 0Bh and INT 0Ch, also called the COM port vectors. When the serial communication ports are configured in Interrupt Mode, these interrupts are generated whenever a character is received. For an in-depth discussion of writing handlers for these two vectors, see Chapter 17, "High-Speed Serial Communications."

INT 05h, also called the Print Screen vector. Some networks disable this function when installed. By installing a TSR handler for this vector after the network is loaded, this function can be re-enabled.

INT 14h, also called the BIOS COM vector. This vector is called by applications that wish to use the BIOS to communicate to COM1: or COM2:. `MODE LPT1:=COM1:` hooks this vector and re-routes characters to the INT 17h vector for printing.

INT 17h, also called the BIOS Print vector. All printer output in MS-DOS passes through this vector. A print spooler would need to chain to this vector to prevent an application from corrupting background printing.

INT 5Ch, also called the NetBIOS Interrupt. This is the mechanism used to communicate with the low-level network APIs by IBM's PC-Net and Microsoft's LanManager.

FLOATING POINT EMULATION

The original 8087 chip in the PC was set up to issue its interrupts to the same vector as used by the NMI vector. PC floating point packages hook this vector. With the advent of the 80287 floating point coprocessor in the IBM PC AT, this changed. The 80287 interrupts on IRQ 13, which generates an INT 75h. The BIOS recognizes this, and to simplify the work of the floating point packages (and to ensure compatibility), reissues this as a software-generated INT 02h. If you use a floating point library in your TSR, these are two vectors you are likely to need to unhook when you unload your TSR.

GENERAL CONSTRAINTS AND TSR PROGRAMMING PRACTICES

There are some general principles and constraints that need to be observed when writing TSRs. If you follow these guidelines, you simplify your debugging task, and reduce the likelihood of the strange behaviors that are typically associated with TSRs.

CHAINING VERSUS HOOKING INTERRUPTS

In most cases you should chain interrupt handlers and pass control onto the next handler in the chain when you have completed your processing. When dealing with hardware interrupts from devices supported by ROM-BIOS, this allows you to ignore the need to control the Intel 8259 Programmable Interrupt Controller (PIC) chip. When dealing with software, this approach reduces the likelihood of unexpected side effects.

Of course there are exceptions. If you are writing an interrupt handler for a custom piece of hardware, you should be responsible for "dismissing" the interrupt through the 8259 PIC and issuing an IRET instruction when you are done. Chapter 17, "High-Speed Serial Communications," goes into greater detail on programming the 8259 PIC as part of the discussion of how to maximize the throughput of the asynchronous communication port (COM port) interrupt mode.

The other reason for not passing control to the next handler in the interrupt chain is to avoid whatever actions that handler will take. Examples of those are the `ctrlbrk` and `harderr` library calls. For the `ctrlbrk` library call, Borland's C++ 3.0 Library Reference specifically states, "The handler does not have to return; it can `longjmp` to return to an arbitrary point in the program." By coding a `longjmp` to an arbitrary point in the program, the programmer can ensure that execution continues at a known point. If no `ctrlbrk` handler were installed, or if control were simply passed along the INT 23h interrupt chain, MS-DOS would terminate the program whenever a CTRL-C were pressed.

ISSUING THE TSR FUNCTION REQUEST

The MS-DOS function request to Terminate and Stay Resident is INT 21h AH = 31h. It takes as a parameter in DX the number of paragraphs of memory to allocate to the RAM-resident portion of the TSR. A *paragraph* of memory is 16 bytes. A paragraph is the difference in the physical addresses of two adjacent segment values that have identical offsets. It is called a paragraph to differentiate it from the term *segment*, which refers to 64K in 8086 addressing mode (also called *real mode*). This is clearly more suited for code written in assembly language; however, two methods can be used to calculate this value when writing the TSR in Borland C++.

The simpler method has the disadvantage that it does not automatically adjust the size of memory allocated to the TSR as the size of the RAM-resident portion changes during development of the TSR. Initially, you set DX to 0xFFFF (all of memory). Then compile and link the complete TSR. Use the /vsm flags with TLINK to generate a complete map file of the executable file. From the map file, you get the complete size of the TSR. To this size, add 100h bytes for the PSP. Divide by 16, rounding up to get the number of paragraphs used by the TSR. To accommodate growth during future development, you can usually add another 200h–300h bytes. The biggest drawback of this approach is that unless you rigorously inspect the map file to ensure that you have not exceeded the allocated space with your latest changes, you will exceed the TSR's space allocation at some point in your development process. A common error message that occurs is `ERROR - Cannot Load COMMAND.COM`. This is because your TSR has written data over the memory header that was built by MS-DOS to describe the block of memory immediately after the TSR. Unfortunately, this error message is not always displayed immediately.

It would be nice to automate this process. This would require a variable that points to the last byte of memory used by the TSR. Since there is no documented variable that indicates the last byte used in a C or C++ program, I link in an assembly language file containing the following code fragment:

```
SEGMENT FOOBAR
public _EndMem
_EndMem EQU $
ENDS
```

It is crucial that the file containing this code be the last file on the link statement, and that the segment name be unique. This will ensure that `_EndMem`

indeed points to one past the last byte used by the program. The underscore is added to the beginning of `EndMem` to allow reference from within C++, since all C++ variables have an underscore added to the beginning of their actual symbol names. You then issue MS-DOS function request 51h (get PSP segment). Next you con-vert the PSP segment value and the address of `EndMem` into physical addresses by multiplying the segment value of both addresses by 16 and adding the offset. You then subtract the PSP physical address from the `EndMem` physical address. Divide the result by 16 and round up to get the size in paragraphs.

This second approach adds code complexity to the TSR (code complexity means added size). TSR size is almost always an issue, so the trade-off is between ease of development and final TSR size.

Another simpler but less flexible approach is to build your complete TSR using 0xFFFF as the value passed to the MS-DOS 31h function.

INDOS FLAG

MS-DOS is a single-tasking operating system. Hence it has internal assumptions that prevent multiple programs that are running simultaneously from requesting MS-DOS services at the same time. Such a simultaneous request would result in corruption of MS-DOS internal data structures and eventually hang the machine. The danger, of course, is that by using the corrupted data structures, MS-DOS might corrupt information stored on the disk before hanging.

The developers of MS-DOS anticipated this problem since they did provide the Terminate and Stay Resident function. They added the so-called *InDos* flag to the MS-DOS data structures. This flag is set to TRUE (nonzero) whenever MS-DOS is in the process of servicing a function request. The location of this flag can be determined by issuing MS-DOS function request 34h (INT 21h AH=34h). The pointer to the InDos flag is returned in ES:BX.

A TSR that needs to perform I/O through MS-DOS must first check the status of the InDos flag. If the flag is clear, it is safe to issue any MS-DOS function request. If it is set, the TSR has two options: postpone the desired I/O until such a time as InDos is clear or abort the I/O.

One exception to this behavior is made to accommodate old software. In some older programs, the program's INT 24h interrupt handler never returns control to MS-DOS. If this were to occur with InDos set, it would never be cleared. The result would be a variety of odd side effects. Hence, when processing a critical error, MS-DOS first clears the InDos flag before issuing the INT 24h critical error notification. It also sets the Critical Error flag (CritErr) to indicate this condition. In MS-DOS 3.0 and later, CritErr is located at InDos – 1. In earlier versions it is located at CritErr + 1.

Why should you as a TSR author care? After all, if InDos is clear, it's safe to issue any MS-DOS function request, isn't it? MS-DOS assumes that if a function request is received prior to any response to the INT 24h Critical Error notification, the application has chosen to abort the function request that caused the critical error in the first place. Thus, if your TSR relied on InDos alone and issued an MS-DOS function request while processing an INT 24h, you might cause some unwanted side effects. Therefore, a TSR should check both InDos and CritErr to determine if it is safe to proceed with I/O.

A second exception to the InDos rule occurs in versions of MS-DOS *prior to 3.0*. In these versions, function calls 00h through 0Ch switched MS-DOS internal stacks without setting the InDos flag. The result is that a simultaneous function call detects InDos as clear, but causes the system to crash when it proceeds with I/O.

Two solutions exist for this problem. You can trick MS-DOS into using the critical error stack for all of the TSRs function requests. You do this by setting the CritErr flag prior to issuing the TSR's function request, and clearing it upon completion. Alternatively, you can postpone the I/O request until the next INT 21h that is issued by the foreground application. If this request is above 0Ch, you can safely save the pending request, issue your request first, and upon completion of the TSR's I/O, reload the pending request and jump to the INT 21h vector as though nothing had happened.

Actually, a third option exists for those of us who are not constrained by a need to run on all popular versions of MS-DOS. Simply insert an MS-DOS version check in the start-up code and issue an error if a version prior to MS-DOS 3.0 is detected. Since the only machines that are still constrained to run MS-DOS versions prior to 3.0 are certain laptops with MS-DOS in ROM, this is usually an acceptable solution.

MS-DOS IDLE LOOP INTERRUPT—INT 28H

I already mentioned that the authors of MS-DOS anticipated the need to prevent multiple programs from simultaneously submitting MS-DOS function requests. The InDos flag served this function; however, it was not the mechanism originally published by Microsoft. Instead, the Int 28h vector, also known as the MS-DOS idle loop interrupt, was the mechanism defined for use by TSRs wishing to submit MS-DOS function requests.

Int 28h is issued by MS-DOS on a regular basis whenever MS-DOS is polling the console for input. The default handler for the Int 28h vector is initialized to an IRET instruction. By chaining an interrupt handler ahead of the IRET instruction, a TSR can ensure that it will receive control whenever MS-DOS is polling for keyboard input. The TSR is guaranteed that it is safe to submit INT 21h function requests to MS-DOS. In MS-DOS versions prior to 3.0, there are some limitations as to which functions may be called. I briefly discussed these limitations in the section on the InDos Flag. For greater detail, refer to the *DOS Programmer's Reference*, Second Edition, pages 580–581.

The Idle Loop interrupt is issued only by MS-DOS, and only when the console is being polled as a result of a MS-DOS function request. Since the MS-DOS console APIs are very limited in functionality, most applications that do significant keyboard input bypass these APIs completely and use one of the two ROM-BIOS keyboard handlers directly (Int 16h and Int 09h). If your TSR is going to be used only from the MS-DOS prompt, this is not a significant limitation. If instead you wish to access your TSR from within Borland's Quattro Pro, Microsoft Word, Borland's Paradox or other such applications, then Int 28h should not be the only mechanism you use to detect when it is safe to submit MS-DOS function requests.

WHEN AND HOW TO ACTIVATE YOUR TSR

Most TSRs are activated by some type of keyboard hot key. This is accomplished by hooking the INT 09h vector and waiting for the appropriate key sequence to be pressed. If the TSR is required to provide functionality other than keyboard input (as, for example, a background print spooler), two other mechanisms are most often used. The TSR can hook the Timer Tick interrupt (1Ch) or the MS-DOS Idle Loop interrupt (28h).

Which one you use depends on how much processing the TSR is going to do and what level of impact on the foreground application can be tolerated. Most print spoolers chain to both Timer Tick and MS-DOS Idle Loop. To avoid impacting system performance, most printing is done when an Int 28h is received. However, to guarantee some progress, since many applications inhibit MS-DOS generation of Int 28h, print spoolers will also do some limited printing upon receiving a Timer Tick (1Ch).

To observe the difference between printing using the Idle Loop interrupt and using the Timer Tick, try the following experiment. Create a large text file. Start printing this file using the PRINT utility that is part of MS-DOS. If you have a dot-matrix printer observe how quickly the file is being printed with only the command prompt displayed. Then start a keyboard-intensive application such as Microsoft Word or Borland 1-2-3. Notice how much slower printing becomes. If you are using a full-page laser printer, you will not be able to observe the difference in line-by-line printing speed. Time how long it takes to print the whole document once while displaying the command prompt, and once while idling in the application.

USE OF CLI

A TSR usually is active while interrupting another foreground application. However, another TSR might try to interrupt your TSR at inopportune moments. To prevent this, you use the *Clear Interrupt flag* (*CLI*) instruction. This prevents any interrupts (other than NMI) from being acknowledged by the CPU until a *Set Interrupt flag* (*STI*) is issued. You need to note a few things about these instructions:

- CLI does not prevent executing an INT instruction; it simply prevents the CPU from acknowledging any hardware interrupts.
- CLI does not prevent hardware interrupts from occurring. It just postpones processing them until after the STI instruction.
- The longer you keep interrupts disabled, the more interrupts will be processed immediately after the STI instruction (and usually before the instruction immediately following the STI).
- After issuing a CLI, you will be unable to receive keyboard input or any other input that is interrupt driven.

Hence you should use the CLI instruction sparingly. Namely, you should disable interrupts for only as long as necessary. You also need to be sure that your code can tolerate subsequent interrupts as soon as the STI instruction is issued.

STACK USAGE

The recommended minimum stack size for MS-DOS programs is 128 bytes. When an interrupt occurs, such as Int 28h Idle Loop, six bytes are immediately used to push FLAGS, CS, and IP. If you then push all of the rest of the CPU registers to preserve them prior to entering your interrupt handler, you use 18 bytes of stack for AX, BX, CX, DX, SI, DI, BP, ES, and DS, for a total of 24 bytes of stack. If your handler is written in C and uses the `interrupt` keyword, you always use these 24 bytes of stack. Furthermore, if you use a subroutine for some calculations, you push another four bytes for a near call (IP, BP) and six bytes for a far call (CS, IP, BP), plus two bytes for every `int` and `char` variable passed, and four bytes for every far pointer. Assume that you have pushed 36 bytes onto the stack and a Timer Tick interrupt occurs. Very quickly you push another 24 bytes onto the stack. You have now used one half of the stack space the program had allocated to itself, and have only called a subroutine passing in two far pointers.

It is clear that you cannot keep gobbling stack space in the manner I describe before you quickly run out of stack space. Unfortunately, no warning occurs when you do run out of stack space. Instead, all sorts of odd side effects can occur, not all of them immediately. You therefore must be very conservative in your use of stack space. In my work, I use the following rules to keep my code from overflowing stack space:

- Only save those registers your handler modifies.
- When calling C subroutines, try to use near pointers only.
- If the interrupt handler implements significant functionality, consider switching to a local, preallocated stack while processing the interrupt.

Switching to a local stack can also simplify some of the code in the interrupt handler. If the Stack, Data, and TSR code are all in the same segment, use of BP as an index pointer can avoid using CS overrides to access local data.

One way to decide whether to use a preallocated local stack is to build one during development. If I initialize the contents of the entire stack to a known

value (I use my initials: `'KS'`), and then examine the stack after running the program through its paces, I can determine how much of the allocated stack was used—the unused portion will be the only part of the stack that still contains `'KS'`.

INPUT AND OUTPUT

TSRs need to perform two types of input and output: console-oriented and file-system-oriented. Due to the complexities of the MS-DOS file system, it is not practical to write your own disk access routines. Also, any performance impact, due to the added code contortions required to determine whether it is safe to submit a function request to MS-DOS, is negligible compared to the time required to extract the data from a spinning disk.

This is not the case for reading the keyboard or for displaying characters on-screen. The existing MS-DOS console routines are slow and limited in functionality. Therefore, as I mentioned earlier, most TSRs chain to INT 09h to detect keyboard input, and display characters on the screen by writing directly to video memory.

Listing 15.1 demonstrates hooking and chaining interrupts, use of the InDos flag, and use of both the Timer Tick and Idle Loop interrupts.

LISTING 15.1. A DEMONSTRATION OF HOOKING AND CHAINING INTERRUPTS, USE OF THE INDOS FLAG, AND USE OF THE TIMER TICK AND IDLE LOOP INTERRUPTS.

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

#define ABORT 0

char far *    pfInDos;            // Pointer to the InDos Flag
char far *    pfCritErr;          // Pointer to the CritErr Flag
void far *    pOldCritErr;        // Pointer to the Old Critical Error handler
void far *    pOldIdleLp;         // Pointer to the Idle Loop chain
void far *    pOldTimerTick;      // Pointer to the Timer Tick Chain
int           fIsTSR = 0;         // Flag to prevent use of Timer Tick and
                                  //   IdleLoop handlers before going TSR
int           CursorX = 0,        // Current position to display cursor at
              CursorY = 0;
unsigned int TempDS;              // Used to store DS while setting up
                                  //    pseudo IRET to old Interrupt vector
```

continues

LISTING 15.1. CONTINUED

```
char chDot[] = {'.'};     // Character to display on screen

/* c_break - install a CTRL-C handler that ignores all CTRL-Cs
 *
 *     This routine is used to demonstrate the library function
 *  ctrlbrk
 *
 */
int c_break(void)
{
   return (~ABORT);
}

/* Int24_Hdlr - install a Critical Error Handler
 *
 *    This routine is used to demonstrate the mechanism for
 *    installing a critical error handler without using the
 *    library routine
 *    harderr
 *
 */
void interrupt
Int24_Hdlr()
{
}

/* Int28_Hdlr - Install a handler for the MS-DOS Idle Loop vector
 *
 *    This routine demonstrates the behavior of the MS-DOS Idle Loop
 *    Interrupt.  Whenever this routine is invoked, you advance the
 *    coordinates of where the Timer Tick routine prints '.' to
 *    the next line.
 *
 *  NOTE: You chain this vector instead of "hooking it."  This
 *     demonstrates the difficulties of doing this using
 *     a C Interrupt routine. The 'interrupt' keyword is designed for
 *     routines like 'Int24_Hdlr' which "hook" rather than "chain"
 *     to interrupt vectors.  To allow you to "chain," you first clean
 *     off the registers that the Borland C Compiler pushes onto the
 *     stack, then you push Flags and the far pointer to the old
 *     interrupt handler on the stack.  Then you push back the registers
 *     that the compiler originally pushed. Then when the flow of
 *     execution reaches the end of the routine, the IRET instruction
 *     actually transfers control to the old Interrupt Vector, instead
 *     of back to the routine that was interrupted.  Since the address of
 *     the interrupted routine is still on the stack, when the old
 *     interrupt handler issues an IRET, it will return to the interrupted
```

```
 *     program.
 *     The reason you have to do this is because of the order the compiler
 *     pushes these registers onto the stack, and the limits that
 *     are placed on the in-line assembler functionality.  Specifically
 *     the DS register is pushed somewhere in the middle of
 *     the group of registers. Once it is "pop'ed" with the original
 *     value, you have no way of accessing the variable into
 *     which you stored the pointer to the old interrupt vector. If you
 *     were using a full function assembler, you would force this data
 *     to be stored in the code segment, and use a CS: override to access
 *     pOldIdleLp.  Unfortunately BASM doesn't allow you to create labels
 *     within an ASM block that can be referenced as data pointers by
 *     the body of the C++ routine.
 */
void interrupt
Int28_Hdlr()
{
       union REGS       InRegs, OutRegs;

       // pop All of the registers, push Flags, CS:IP of the old
       // Handler routine. then push back all of the registers
       asm{
              pop     bp;        // Clean up stack prior to jumping down chain
              pop     di;
              pop     si;
              pop     ax;        // This is actually DS, but you can't afford to
              mov    TempDS, AX  // lose your ability to address the data
                                 // segment
              pop     es;
              pop     dx;
              pop     cx;
              pop     bx;
              pop     ax;

              pushf              // push flags, CS, IP of pOldIdleLp onto stack
              mov     AX, WORD PTR pOldIdleLp + 2          // CS
              push    AX
              mov     AX, WORD PTR pOldIdleLp              // IP
              push    AX

              push    ax
              push    bx
              push    cx
              push    dx
              push    es
              mov    ax, TempDS    // push original DS back onto stack
```

continues

LISTING 15.1. CONTINUED

```
            push    ax
            push    si
            push    di
            push    bp
    }

  // Short circuit this handler's functionality until you are installed
  // as a TSR
       if( fIsTSR )
       {
               asm {cli}
               if( CursorY++ >24 )
               {
                       CursorY = 0 ;
               }
               CursorX = 0;

               asm {sti}
       }
}


/* TimerTick - Install a handler for the Int 1Ch Timer Interrupt
 *
 *    This routine demonstrates the behavior of the Timer Tick
 *    Interrupt.  Whenever this routine is invoked, you print a '.' and
 *    advance the cursor position by 1.
 *
 *    NOTE: Since you chain to this interrupt as well, you have to go
 *    through the same contortions as you did for the Int28h handler
 */
void interrupt
TimerTick()
{
      union REGS       InRegs, OutRegs;

      // pop All of the registers, push Flags, CS:IP of the old
      // Handler routine. then push back all of the registers
      asm{
             pop     bp;         // Clean up stack prior to jumping down chain
             pop     di;
             pop     si;
             pop     ax;         // This is actually DS, but you can't afford to
             mov    TempDS, AX   // lose your ability to address the data segment
             pop     es;
```

```
            pop     dx;
            pop     cx;
            pop     bx;
            pop     ax;

            pushf               // push flags, CS, IP of pOldIdleLp onto stack
            mov    AX, WORD PTR pOldIdleLp + 2        // CS
            push    AX
            mov    AX, WORD PTR pOldIdleLp            // IP
            push    AX

            push    ax
            push    bx
            push    cx
            push    dx
            push    es
            mov     ax, TempDS      // push original DS back onto stack
            push    ax
            push    si
            push    di
            push    bp
}

if( fIsTSR )
{
    asm{
        pushf;            // Save state of interrupt flag
        cli;
    }

  // Print a '.' at CursorX,CursorY using the ROM Bios
  //   Display Character String function
    asm{
        mov     AH, 013h;         // Video BIOS Print String Function
        mov     AL, 1;            //    Subfunction 1
        mov     BX, 7;            //  Video Attribute to use
        mov     CX, CursorY;      //  Row to start on
        mov     DH, CL
        mov     CX, CursorX       //  Column to start at
        mov     DL, CL
        mov     CX, DS
        push    ES;               // place ptr to chDot in ES:BP
        push    BP;
        mov     ES, CX
        mov     BP, offset chDot
        mov     CX, 1
        int     010h;             // Print '.' at CursorX, CursorY
        pop     BP;
```

continues

LISTING 15.1. CONTINUED

```
            pop     ES;
        }

   // Update the cursor position by 1
        if( ++CursorX > 79 )
        {
            CursorX = 0;
        }
        asm{popf}        // restore interrupt flag state
    }

}


/* EXAMPLE1.main - Example1 shows the use of Int 28h and Timer Tick vector
 *
 *    This routine does very little except hook up the
 *    various interrupt vectors, and then issues the Terminate and Stay
 *    Resident request
 */
void
main(void)
{

    union REGS       InRegs, OutRegs;
    struct SREGS     SegRegs;
    char far *       pOurFunc;    // Pointer to your Interrupt function
                                  //     handlers. Used only to clarify code

  // Initialize pointers to the InDos and CritErr flags
       InRegs.x.ax = 0x3400;    // Get InDos Flag
       intdosx( &InRegs, &OutRegs, &SegRegs );
       pfInDos =(char far *)MK_FP(SegRegs.es, OutRegs.x.bx );
       if (_osmajor < 3 )
    {
              pfCritErr = pfInDos + 1;
    }
    else
    {
              pfCritErr = pfInDos - 1;
    }

  // Set the control break handler using the C++ library functions
       ctrlbrk(c_break);

  // Set the Critical Error handler using the standard MS-DOS Function
  // request mechanism, but first save the old vector
       InRegs.h.ah = 0x35;         // Get Interrupt Vector Function Request
```

```
        InRegs.h.al = 0x24;         // Specifically request the CritErr vector
        intdosx( &InRegs, &OutRegs, &SegRegs );
        pOldCritErr =  MK_FP(SegRegs.es, OutRegs.x.bx);

        pOurFunc = (char far *)Int24_Hdlr;
        InRegs.h.ah = 0x25;         // Set Interrupt Vector Function Request
        InRegs.h.al = 0x24;         // Specifically request the CritErr vector
        SegRegs.ds = FP_SEG( pOurFunc);
        InRegs.x.dx = FP_OFF( pOurFunc );
        intdosx( &InRegs, &OutRegs, &SegRegs );

        asm { cli }
    // Now chain the Int28 handler
        InRegs.h.ah = 0x35;         // Get Interrupt Vector Function Request
        InRegs.h.al = 0x28;         // Specifically request the Idle Loop Vector
        intdosx( &InRegs, &OutRegs, &SegRegs );
        pOldIdleLp =  MK_FP(SegRegs.es, OutRegs.x.bx);

        pOurFunc = (char far *)Int28_Hdlr;
        InRegs.h.ah = 0x25;         // Set Interrupt Vector Function Request
        InRegs.h.al = 0x28;         // Specifically request the Idle Loop Vector
        SegRegs.ds = FP_SEG( pOurFunc);
        InRegs.x.dx = FP_OFF( pOurFunc );
        intdosx( &InRegs, &OutRegs, &SegRegs );

    // Now chain the timer tick event
        InRegs.h.ah = 0x35;         // Get Interrupt Vector Function Request
        InRegs.h.al = 0x1C;         // Specifically request the Timer Tick
        intdosx( &InRegs, &OutRegs, &SegRegs );
        pOldTimerTick =  MK_FP(SegRegs.es, OutRegs.x.bx);

        pOurFunc = (char far *)TimerTick;
        InRegs.h.ah = 0x25;         // Set Interrupt Vector Function Request
        InRegs.h.al = 0x1C;         // Specifically request the CritErr vector
        SegRegs.ds = FP_SEG( pOurFunc);
        InRegs.x.dx = FP_OFF( pOurFunc );
        intdosx( &InRegs, &OutRegs, &SegRegs );

    // Prepare to Terminate and Stay Resident.  First you disable all
    // Interrupts, so that you can set the "fIsTSR" flag without risk
    // of a Timer Tick occurring before you have actually "gone tsr"
        fIsTSR = -1;
        InRegs.x.ax = 0x3100;
        InRegs.x.dx = 0x500;    // This value is determined by inspecting
                                //    the map file after compilation
        intdos( &InRegs, &OutRegs );
}
```

> The above TSR will require that you reboot your machine to disable it. Before running it, save any open data files, and close all applications. I strongly recommend that you run this example from within Turbo Debugger using the Resident option. For more details on how to use this feature, see "Debugging TSRs" in Chapter 12, "Debugging Techniques."

If you do run the above TSR, it will draw a single dot (.) in the first column of the display. This is because while MS-DOS loops waiting for console input at the command prompt, an Int 28h is being issued every Timer Tick event. Since the TimerTick event is used to draw the dot and Int 28h is used to move to the next row, you get only one dot per line. If you now press a single key followed by Enter, you should see a string of dots appear, as multiple Timer Tick events are generated while command.com attempts to parse whatever key you pressed.

Unfortunately, you have disabled CTRL-C and provided no way to interact with the TSR through the keyboard. Your only recourse is to reboot the machine. It is rather interesting to note how much code was required to implement a TSR that basically does very little. As a general rule, low-level programming requires more code to accomplish a given task. If you assume that you make mistakes in proportion to the number of lines of code you write, it begins to become clear why writing even a simple TSR takes much more effort and time than writing a more complex program that uses only the C++ run-time libraries.

DEALING WITH OTHER TSRS—THE TESSERACT STANDARD

As I mentioned earlier, a standard exists for communicating between the RAM-resident portion of your TSR, any transient utility portion your TSR may have, and other TSRs. This standard is called the TesSeRact standard. This standard defines 15 basic functions that are used by most TSRs. For a TSR to be considered TesSeRact-compliant, it must support the Check Install function and the Return ParameterPtr function. The TSR must also include a data header as part of the TSR's Int 2Fh interrupt handler. Since this requires mixing

of code and data, the simplest way to accomplish this is to write the entry of the INT 2Fh interrupt handler in Borland TASM, and to link the assembler module using TLINK.

Communication between the RAM-resident portion of the TSR and the transient portion is accomplished by issuing an INT 2F instruction after first setting up the parameters of the particular subfunction desired. The two subfunctions that are supported by *all* TesSeRact compliant TSRs are CHK_INSTALL and GET_PARMPTR.

During the install phase of the TSR, a check should be made for a previously installed copy. The syntax of the CHK_INSTALL call is

```
AX = 5453h               ; TesSeRact function id signature
BX = CHK_INSTALL == 0    ; Function 0
DS:SI = Ptr to Id string for this TSR
```

Upon return from the Int 2Fh, if a previous copy has been found, AX will be –1 and CX will contain the TesSeRact handle for the already-installed TSR. If no previous copy is found, CX will contain the value to use for this TSR as its TesSeRact handle, and AX will not be equal to –1.

```
AX = 5453h               ; TesSeRact function id signature
BX = CHK_INSTALL == 0    ; Function 0
DS:SI = Ptr to Id string for this TSR
```

The other required function is the GET_PARMPTR function. The syntax of this function is

```
AX = 5453h               ; TesSeRact function id signature
BX = GET_PARMPTR == 1    ; Function 1
CX = TSR Handle          ; Handle of target TSR
```

Upon return AX will be 0 and ES:BX will point to the data block labeled i2f_TSRData in the next code sample.

This next code fragment implements the two required TesSeRact functions as well as the header. This fragment needs to be placed in an .ASM file that is assembled using TASM and linked to the main TSR using TLINK.

LISTING 15.2. USING THE TesSeRact INTERFACE.

```
; This code fragment defines the Int 2Fh interface used by TesSeRact
;       compliant TSRs
```

continues

LISTING 15.2. CONTINUED

```
CHK_INSTALL     EQU     01b
GET_PARMPTR     EQU     010b
CHK_HOT_KEY     EQU     0100b
SET_I24         EQU     01000b
GET_DATAPTR     EQU     010000b
SET_HOTKEY      EQU     0100000b
ENABLE_TSR      EQU     0100000000b
DISABLE_TSR     EQU     01000000000b
UNLOAD_TSR      EQU     010000000000b
RESTART_TSR     EQU     0100000000000b
GET_STATUS      EQU     01000000000000b
SET_STATUS      EQU     010000000000000b
POPUP_TYPE      EQU     0100000000000000b
CALL_USER_PROC  EQU     01000000000000000b
PUT_KEYBD       EQU     010000000000000000b
TESSERACT_SIG   EQU     05453h

.model small
.code
       EXTRN pOldInt2f:DWORD
       public C i2f_Hdlr
i2f_Hdlr PROC C
       jmp             i2f_10_CodeStart
i2f_TSRData     LABEL   BYTE
       szProgId        db      'MY_TSRID'            ; 8 byte TSR ID string
       TSR_Handle      dw      ?
       fFuncSupported  dd      CHK_INSTALL + GET_PARMPTR
       HotKeyScanCd    db      ?                     ; Scan code for HotKey
                                                     ; activation
       KBDShiftState   db      ?                     ; Shift state for HotKey
       HotKeyId        db      ?                     ; Which HotKey to use
                                                     ; if more than one
                                                     ; supported
       cOtherHotKeys   db      ?                     ; Number of other hot keys
                                                     ; supported beyond primary
       pOtherHotKeys   dd      ?                     ; Pointer to other hot key
                                                     ; descriptors
       TSR_Status      dw      ?                     ; TSR Status flag
       TSR_PSP         dw      ?                     ; PSP of TSR
       TSR_DTA         dw      ?                     ; DTA of TSR
       TSR_DS          dw      ?                     ; Data Segment for TSR

i2f_10_CodeStart:
       cmp     AX, TESSERACT_SIG                     ; Check if you have a
       jne     i2f_30_Next2f                         ; Tesseract request
```

```
        push    DS                                      ; Save Caller's DS

        push    CS                                      ; Set up access to TSR data
        pop     DS
    ASSUME DS:CODE

        or      BX, BX                                  ; Check for Function 0
        jne     i2f_50_ChkGetParm

; Function 0 is the Check install function
;       DS:SI points to the caller's IdString
;       CX is the depth of the Tesseract chain to this point
i2f_20_ChkInstall:
        pop     DS                                      ; Recover Caller's DS, but
        push    DS                                      ;   Leave it on stack
    ASSUME DS:NOTHING
        push    CX                                      ; Save CX
        push    SI                                      ; Save caller's SI
        push    DI                                      ; Save caller's DI
        push    ES                                      ; Save caller's ES

        push    CS                                      ; set ES:DI to point to
        pop     ES                                      ;  szProgId
    ASSUME ES:CODE
        lea     DI, szProgId
        mov     CX, 8                                   ; CX == length of id string
        rep     CMPSB                                   ; Compare szProgId with
                                                        ;  passed in string
        pop     ES                                      ; Clean up stack
     ASSUME ES:NOTHING
        pop     DI
        pop     SI
        pop     CX
        jnz     i2f_25_NoMatch

; You got a match, so return your TSR's Tesseract handle and indicate
;       success
        mov     CX, CS:TSR_Handle
        mov     AX, 0FFFFh
        stc
        jmp     SHORT i2f_40_Leave

; Here you handle a missed match.  In this case, you increment the current
;   TESSERACT chain depth and pass control to the next 2F handler
i2f_25_NoMatch:
        inc     CX
        pop     DS                                      ; clean up stack

; Jump to the next handler in the chain
```

continues

LISTING 15.2. CONTINUED

```
i2f_30_Next2f:
       jmp     DWORD PTR [pOldInt2f]

; This is the common return point for this handler
i2f_40_Leave:
       pop     DS                                      ; clean up stack
       iret

; This is where you test for and handle the other required Tesseract function:
;  GetUserParameterPointer.  This is function 01 and CX contains the
;       Tesseract Handle for the target TSR.  Success means you zero AX
;       and return a pointer to the data area in ES:BX
i2f_50_ChkGetParm:
    ASSUME DS:CODE                              ; from above
       cmp     CX, TSR_Handler                  ; Check if this is for you
       je      i2f_60_ChkBX

       pop     DS                               ; TSR handles don't match so
       jmp     i2f_30_Next2f                    ; pass this request down the
                                                ; chain

i2f_60_ChkBX:
       cmp     BX, 1
       je      i2f_70_RetParm

       mov     AX, 0FFFFh                       ; Unsupported Function - indicate
       stc                                      ; error
       jmp     i2f_40_Leave

i2f_70_RetParm:                                 ; Return pointer to Parm Table
       push    CS
       pop     ES
       lea     BX, i2f_TSRData
       xor     AX, AX
       jmp     i2f_40_Leave

       endp
end
```

Again you see an example of how a lot of low-level code is required to accomplish relatively simple tasks. The above TesSeRact example is by no means complete, but it does provide a basis for understanding how to use the TesSeRact functions.

DOING USEFUL THINGS WITH YOUR TSR

Your focus to this point has been how to structure a TSR and some of the special techniques and coding practices required. It would be nice to add the ability to activate the TSR with a hot-key sequence. Most of the interactions programs have with the keyboard hardware is in a polling mode. This is fine for a foreground application that can loop while waiting for input from the keyboard. Since a TSR is only active in response to either TimerTick or some other event, you cannot poll the keyboard without a high likelihood of missing the key you are looking for.

Instead, you need to use some mechanism that will notify you when a key is pressed so that you may examine it immediately. While MS-DOS does not provide any such mechanism, the ROM BIOS does in the form of interrupt vector 09h. As Table 15.1 indicates, hardware IRQ 1 is mapped to INT 09h. A hardware IRQ 1 event is generated whenever a key is pressed. The 8259 PIC translates this into an INT 09h instruction which transfers control to the ROM-BIOS routines that extract the keypress information from the hardware and store it in a more usable form. This is a case where you don't wish to duplicate the complexity of the underlying service routines. The preferred mechanism is to call the existing interrupt handler as a subroutine to process the actual keypress and then to check for particular scan codes after they have been generated by the underlying interrupt handler. Since the underlying interrupt handler uses an IRET instruction to return, you must push the FLAGS register onto the stack before executing the far call to the old interrupt handler. The combination of the PUSHF instruction and the far call will cause the IRET to return to your handler. Since your handler too was invoked by an INT instruction, your handler will still need to return using the IRET instruction.

SAFE FILE I/O

A pop-up calculator or similar tool is all fine and well, but in most cases you wish to retrieve or store data to the multi-megabyte hard drive that you spent good money on. Replicating the file system functionality that exists within

MS-DOS is a waste of effort and precious memory. It also is almost guaranteed to be made obsolete by the latest MS-DOS revision. Clearly you would prefer to use the MS-DOS function calls. To be able to use the MS-DOS file system from within a TSR, you need to ensure the following:

- The MS-DOS InDos flag is not set.
- The MS-DOS CritError flag is not set.
- A file handle is available for use and you can access the desired file.
- A critical error will not cause the foreground application to behave unpredictably.
- A Ctrl-C will not cause unexpected behavior.

I have already discussed all but one of these functions separately. You will now put these together to build a routine that safely stores your data to a file. How do you ensure that you can safely write data to the file? This actually breaks down into two separate topics. The simpler of the two is how to ensure that you have access to the desired file. If you were to change to the drive and directory your target file is in, you would first have to save the current drive and directory. Then, after completion of your disk I/O, you would need to change back to the saved values. This is cumbersome, and extremely slow when using floppy drives. The correct solution is to use MS-DOS function request 3Dh (open file handle), followed by the appropriate sequence of function request 3Fh (read handle) and Function 40h (write handle) requests. Upon completing the current sequence of input or output, the file should be closed using MS-DOS function request 3Eh.

Why can't you simply open the file when you install the TSR and read or write from the handle whenever required? The answer is found in how MS-DOS treats handles. A handle is an index into an array of file identifiers that MS-DOS maintains on a per-process basis. This is done for a variety of reasons. One of them is to allow all processes to rely on the values of the pre-opened handles that the MS-DOS command interpreter, command.com, would create as part of the execution process. Another is to allow multiple Open requests for the same file, without losing track of which file is referenced by how many handles.

In any case, the effect is that Handle == 5 for your TSR is unlikely to refer to the same file as Handle == 5 for whatever process is in the foreground when you

wish to Read or Write data. You have two options for handling this. Since the table of handles is usually stored in the Program Segment Prefix (PSP), you could use MS-DOS function request 50h (set PSP segment) to change the current PSP to your TSR's PSP. This would allow you to access files using handles that were returned for Open Handle requests issued prior to the TSR issuing the MS-DOS function request 31h (Terminate and Stay Resident). If you wish to use any of the standard C file I/O library calls, you must follow this approach. However, this requires that you first save the current PSP value using MS-DOS function request 51h (get PSP segment), and restore it immediately upon completion of file I/O.

A major failing of this approach comes from assumptions made by other applications. If a foreground application itself has interrupt handlers that assume no PSP change will occur, you could have some very disastrous consequences. Although this assumption should not be made, in practice, the problem occurs after.

This approach also does not guarantee that the data you Write to the handle is actually written to the file. MS-DOS buffers file data internally, and only writes the data to the file when either the buffer is full, the buffer is reused, or the file is closed. This performance enhancement (buffer sizes are selected to be multiples of disk sector sizes) has the side effect that if the system crashes before the buffer is written to the file, the data is lost. Since you do not have control of the stability of the foreground application, if reliability in saving the data to a file is important, it is simpler to Open a new handle to the file, Write the data to the file, then Close the handle to the file.

This is also the preferred method when using versions of MS-DOS prior to 4.0. These versions of MS-DOS allowed a maximum of 20 files to be open simultaneously even if the FILES= command in config.sys were set to a value greater than 20.

In Listing 15.3, you put the five requirements that you started this section with into practice. Note that I do not use any inline assembler in setting up the MS-DOS I/O Requests. I use it only to issue the CLI instruction prior to checking the InDos and CritErr flags. This is because the MS-DOS function requests involved use of the DS register to pass data. Rather than concern myself with how I might affect access to my C variables by setting DS to some other value, I let the run-time libraries handle this.

LISTING 15.3. WRITING TO A FILE FROM WITHIN TSR MODE.

```
#include <dos.h>

extern char far * pfInDos;     // Pointer to the InDos flag
extern char far * pfCritErr;   // Pointer to the CritErr flag
/* TSR_Write_File - Write to a file from within TSR mode
 *
 *     TSR_Write_File takes as inputs
 *     pBuff - a pointer to the data buffer to be written
 *     cBytes - size of the data buffer as count of the bytes
 *     pFileName - full path name of the file including drive letter
 *
 *     TSR_Write_File returns
 *     0 = if successful
 *     -1 = if an error occurred and the write should be delayed
 *
 */
int
TSR_Write_File(
        char far * pBuff,     // Data Buffer to dump to file
        unsigned int cBytes,  // number of bytes in pBuff
        char far * pFileName  // pointer to the target file name
        )
{

        Set_IgnoreCtrlC();    // Set a handler that forces ignore of
                              // any Control C events that might occur
        Set_FailCritErr();    // Set a handler that forces all critical
                              // errors to FAIL

        InRegs.h.ah = 0x3d;   // MS-DOS Function Request Open
        InRegs.h.al = 0x02;   // Read/Write Deny All other access
        SegRegs.ds = FP_SEG(pFileName);
        InRegs.x.dx = FP_OFF(pFileName);
        asm { cli }
        if ( !*pfInDos && !*pfCritErr )
        {
              intdosx( &InRegs, &OutRegs, &SegRegs );
        }
        else
        {
                return( -1 );
        }
        asm{ sti}

      /* Check that carry flag is not set */
       if ( OutRegs.x.flags & 0x01 )
```

```
{
        return( -1 );  // Indicate that no I/O was done
}

hFile = OutRegs.x.ax

InRegs.h.ah = 0x40;    // Write File Function Request
InRegs.x.ax = hFile;   // File Handle
InRegs.x.cx = cBytes; // Number of Bytes to write
SegRegs.ds = FP_SEG(pBuff);
InRegs.x.dx = FP_OFF(pBuff);
asm { cli }
if ( !*pfInDos && !*pfCritErr )
{
       intdosx( &InRegs, &OutRegs, &SegRegs );
}
else
{
       return( -1 );
}
asm{ sti}

/* Check that carry flag not set */
if ( OutRegs.x.flags & 0x01 )
{
        return( -1 );    // Indicate that no I/O was done
}

InRegs.h.ah = 0x3e;      // Close File handle Function Request
InRegs.x.ax = hFile;
asm { cli }
if ( !*pfInDos && !*pfCritErr )
{
       intdos( &InRegs, &OutRegs );
}
else
{
       return( -1 );
}
asm{ sti}

/* Now release the Control C and Critical Error Vectors */
Clear_CtrlC();
Clear_CritErr();

return( 0 );
```

Note that the above example aborts the file I/O if it encounters fInDos or fCritErr as set. A more robust solution would be to use a `setjmp` call to save the state before returning –1, so that the I/O could later be restarted. Unfortunately `setjmp` and `longjmp` are not usable within TSR routines because of the assumptions they make about the stack segment. However, you could implement them specifically for use in your TSR. For `setjmp`, you simply need to save the task state in the TSR's data segment. A task state consists of:

- All segment registers (CS, DS, ES, SS)
- Register variables (SI, DI)
- Stack pointer (SP)
- Frame base pointer (BP)
- Flags

The only way you can implement this is if your TSR switches its stack upon entry, since otherwise the SP and BP values you restore by the `longjmp` would be meaningless. A simple way to think of `setjmp` is to use the analogy of a time-out in sports. When a player calls "Time out," the referees take note of all of relevant positions. When the referee calls "Time in," every attempt is made to re-create the positioning prior to the time-out call. If `setjmp` is "Time out," `longjmp` is "Time in." After the time-out and before the time-in, the players can move about. Similarly, a program can clean up some error condition and return to action by using `longjmp`.

If you were able to use `setjmp` and `longjmp`, you would insert a `setjmp` call immediately after the CLI instruction for each of the MS-DOS function requests. Upon detection of either fInDos or fCritError being set, you would also set a flag internal to the TSR indicating that a `longjmp` should be issued to complete the I/O at the next opportunity. Your Timer Tick (1Ch) or MS-DOS Idle Loop (28h) interrupt vector handlers would then be modified to check for this internal flag, and to issue a `longjmp` to complete the I/O request.

SAVING SCREEN INFORMATION

Listing 15.1 outputs characters using the ROM-BIOS video display routines. It makes no attempt to save any of the screen information that it overwrites. This is not acceptable for a real-world TSR. The goal of a TSR is to provide additional information, not to destroy existing display information. The simplest way to save display screen information is to snapshot the section of screen memory you are about to trash into a local buffer. To restore the screen information all you need to do is to copy the information back over the data your TSR has displayed. Simple in concept. Potentially complex in implementation. The complexity is due to the variety of standard, and extended video modes that exist in the PC environment. The following table lists the normal modes that are available.

TABLE 15.2. STANDARD VIDEO DISPLAY MODES AVAILABLE ON IBM PC-COMPATIBLE COMPUTERS.

Mode	*Type*	*MDA*	*CGA*	*EGA*	*MCGA*	*VGA*	*Pixel res.*	*Char. res.*	*No.of colors*
0	Alphanumeric	N	Y	Y	Y	Y	320X200	40X25	16
0	Alphanumeric	N	N	Y	N	Y	320X350	40X25	16
0	Alphanumeric	N	N	N	Y	N	320X400	40X25	16
0	Alphanumeric	N	N	N	N	Y	360X400	40X25	16
1	Alphanumeric	N	Y	Y	Y	Y	320X200	40X25	16
1	Alphanumeric	N	N	Y	N	Y	320X350	40X25	16
1	Alphanumeric	N	N	N	Y	N	320X400	40X25	16
1	Alphanumeric	N	N	N	N	Y	360X400	40X25	16
2	Alphanumeric	N	Y	Y	Y	Y	640X200	80X25	16
2	Alphanumeric	N	N	Y	N	Y	640X350	80X25	16
2	Alphanumeric	N	N	N	Y	N	640X400	80X25	16
2	Alphanumeric	N	N	N	N	Y	720X400	80X25	16
3	Alphanumeric	N	Y	Y	Y	Y	640X200	80X25	16
3	Alphanumeric	N	N	Y	N	Y	640X350	80X25	16
3	Alphanumeric	N	N	N	Y	N	640X400	80X25	16
3	Alphanumeric	N	N	N	N	Y	720X400	80X25	16

continues

TABLE 15.2. CONTINUED

Mode	*Type*	*MDA*	*CGA*	*EGA*	*MCGA*	*VGA*	*Pixel res.*	*Char. res.*	*No.of colors*
4	Graphics	N	Y	Y	Y	Y	320×200		4
5	Graphics	N	Y	Y	Y	Y	320×200		4
6	Graphics	N	Y	Y	Y	Y	640×200		2
7	Alphanumeric	Y	N	Y	N	Y	720×350	80×25	2
7	Alphanumeric	N	N	N	N	Y	720×400	80×25	2
0Dh	Graphics	N	N	Y	N	Y	320×200		16
0Eh	Graphics	N	N	Y	N	Y	640×200		16
0Fh	Graphics	N	N	Y	N	Y	640×350		2
10h	Graphics	N	N	Y	N	Y	640×350		4
10h	Graphics	N	N	Y	N	Y	640×350		16
11h	Graphics	N	N	N	Y	Y	640×480		2
12h	Graphics	N	N	N	N	Y	640×480		16
13h	Graphics	N	N	N	Y	Y	320×200		256

Table 15.2 describes the standard capabilities of the five standard adapters that exist for the PC family of machines. So what is missing? All entries for the PCjr, any Super-VGA resolutions (usually defined as 800×600 pixels), any resolutions above S-VGA (1024×760, 1280×1024, 1680×1260), any greater color densities (256 colors, 16bit color, 24bit "Truecolor"). How do you accommodate all of these other combinations, if you can't even list them all?

The answer is you don't. For those programmers who wish to program the CRT Controller hardware directly to take maximal advantage of these modes, I refer you to Richard Wilton's book *PC & PS/2 Video Systems*. Instead, you will take advantage of the capabilities provided by the INT 10h Video BIOS calls. Many video cards that exceed the capabilities of the above table do so only when using the custom drivers that are provided with the video card. What these drivers primarily do is enhance the capabilities of the basic INT 10h BIOS services so that applications written to this interface continue to function.

Why use the INT 10h BIOS services when they have a reputation for being slow and cumbersome? Because they reduce your need to adapt your code for a variety of video adapters and modes. As you have seen before in this chapter, the lower the level at which you interact with the system, the more code that needs to be written to accomplish the same task. While writing and reading data from video memory is relatively easy when the display adapter is functioning in alphanumeric mode, it is quite complex and mode-dependent when the display adapter is functioning in Graphics mode. By using the INT 10h BIOS services, you can reduce some of the complexity involved in dealing with an adapter that is functioning in graphics mode.

The major flaw in this methodology is that some applications choose to control the video hardware directly without updating the values that the INT 10h BIOS functions rely on. If you choose to pop up your application in this environment, any data you try to display is likely to look like garbage. I will ignore this issue in this discussion.

The INT 10h BIOS functions rely on status information that is stored in BIOS Data area (40:*xxxx*). An application or TSR can read the data in this area to determine various current video parameters. Table 15.3 describes the more useful entries in this table. A complete listing of this table is available on pages 436–437 of *PC & PS/2 Video Systems*.

Table 15.3. Video status information contained in the BIOS data area.

Description	*Address*	*Size*
Current BIOS Video Mode number	40:0049h	Byte
Number of character columns	40:004Ah	Word
Size of the Video Buffer in Bytes	40:004Ch	Word
Offset of the start of the Video Buffer	40:004Eh	Word
Cursor Position Array. There is one entry for each of the 8 possible Video Pages. For each entry, the high-order byte is the cursor row and the low-order byte is the cursor column.	40:0050h	8 Words

continues

TABLE 15.3. CONTINUED

Description	*Address*	*Size*
Starting and ending scan lines for the alphanumeric cursor. This is a WORD containing the top line in the high-order byte and the end line in the low-order byte.	40:0060h	2 Bytes
Currently displayed Video Page.	40:0062h	Byte
Current CRT Control Register value; this is saved here because it is a Read Only register on some video display adapters.	40:0065h	Byte
Highest displayable character row, assuming that the top row is row 0 (why they do columns differently is anyone's guess).	40:0084h	Byte
Character height in pixels—this is used for generating characters in graphics mode.	40:0085	Word

The following table lists the useful INT 10h BIOS functions, their parameters, and a brief description of their function. When the description refers to "vector 1Fh" and "vector 43h," I am referring to the corresponding entry in the interrupt vector table. The values for these entries are actually pointers to the graphics character data. For an example of a program that modifies these vectors, see the MS-DOS utility GRAFTBL. Pay particular attention to function 1120h since it is used to modify the most common of these vectors. Also pay attention to function 13h since it uses these vectors to display characters in graphics mode.

Table 15.4. INT 10h BIOS Functions

Function	*Parameters*	*Description*
Oh	AH = 0 AL = Video Mode number from above Table. For EGA, MCGA, and VGA adapters, setting Bit 7 inhibits the clearing of the new Video buffer Returns: nothing	This function selects the new video mode, programs the hardware, and clears the new video buffer (unless inhibited by Bit 7). No sanity checking is done to prevent selection of a mode that is not supported by the currently installed display adapter.
02h	AH = 2 BH = Video page to use DH = Character row DL = Character column Returns: nothing	This function updates the appropriate entry at 40:50h. If the target video page is the currently selected one, it also causes the displayed cursor to be repositioned.
03h	AH = 3 BH = Video page to use Returns: CH = Top Line of Cursor CL = End Line of Cursor DH = Character Row DL = Character Column	This returns the appropriate entry from 40:50h, as well as 40:60.

continues

TABLE 15.4. CONTINUED

Function	*Parameters*	*Description*
05h	AH = 5 AL = Video Page Returns: nothing	This sets the current video page to be displayed. It is ignored for CGA displays, and no check is made for the presence of the corresponding memory for the other adapters.
09h	AH = 9 AL = ASCII Code to display BH = Background Pixel Value BL = Foreground Pixel Value (Graphics) Character attribute (Alphanumeric) CX = Repeat count Returns: nothing	This function writes a character at the current cursor position. For graphics modes, the default character set is used. On CGA displays this is built for 00-7Fh and pointed to by vector 1Fh for 80H-FFh. All other adapters use vector 43h to point to this character set. NOTE, this does NOT move the cursor.
0Ah	AH = Ah AL = ASCII Code to display BH = Background Pixel Value BL = Foreground Pixel Value (Graphics) CX = Repeat count Returns: nothing	This is identical to 09h except that in Alphanumeric mode it only fills in the character and not the attribute.

Function	*Parameters*	*Description*
0Eh	AH = 0Eh AL = ASCII Code to display BH = Video Page for early BIOS versions BL = Foreground pixel value in graphics modes. Returns: nothing	This function writes a character at the current cursor position and updates the cursor position. However, it does not provide a mechanism for setting the attribute byte in alphanumeric modes.
0Fh	AH = 0Fh Returns: AH = Number of character columns AL = Video Mode Number BH = active Video page	This function pulls the return value from the data table at 40:xxxx.
1100h 1101h 1102h 1104h	AH = 11h AL = 0,1,2,4 BH = Character height in Pixels (must be a multiple of 2) BL = which character table to replace CX = Number of characters in table DX = ASCII code of first character ES:BP = Address of table	This function loads a user-defined Alpha-numeric character table into RAM. This is usable on EGA, MCGA, and VGA systems. Subfunction 0: allows custom height Subfunction 1: 8x14 characters Subfunction 2: 8x8 characters Subfunction 4: 8x14 characters, ignored by VGA.

continues

TABLE 15.4. CONTINUED

Function	*Parameters*	*Description*
	AH = 11h return CX = Character size CL = # rows displayed ES:BP = ptr to character definition table	
1103h	AH = 11h AL = 3h VGA BL[4,1,0] = Select table to use if Attrib Bit 3 is 0 BL [5,3,1] = Select table to use if Attrib Bit 3 is 1 MCGA, EGA BL[1,0] = Select table to use if Char Attrib Bit 3 is 0 BL[3,2] = Select table to use if Char Attrib Bit 3 is 1	For you, this is required primarily for the MCGA. If you load a character table using subfunction 0,1,2,4 then you must issue subfunction 3 to load those tables into the adapter's internal font pages.
1120h	AH = 11h Al = 20h ES:BP = address of user-specified 8×8 pixel graphics characters	This points the 1Fh vector to the table pointed at ES:BP. This vector is used by the CGA- and CGA-compatible modes on the EGA and VGA modes for support of characters that correspond to ASCII 80h-FFh.

Function	*Parameters*	*Description*
1121h 1122h 1123h 1124	AH = 11h AL = 21h-24h CX = size of character definition, used by subfunction 21. BL = Number of character rows per screen 0 == ?? Specified in DL 1 ==> 14 rows 2 ==> 25 rows 3 ==> 43 rows	This loads the graphics characters into the vector and is used in all graphics modes other than CGA-compatible modes, and contains the characters 00h-7Fh for the EGA, MCGA and VGA displays.
12h	AH = 12h BL = 10h Returns: BH = 0 = Color, 1 = Mono BL = Amount of video RAM 0 == 64kBytes 1 == 128KBytes 2 == 192KBytes 3 == 256KBytes CX = flags BL = 30h AL = 0 Select 200 scan line mode AL = 1 Select 350 scan line mode AL = 2 Select 400 scan line mode BL = 34h AL = 0 Enable cursor display AL = 1 Disable cursor display	This function sets or gets the status of various useful bits of data.

continues

TABLE 15.4. CONTINUED

Function	*Parameters*	*Description*
13h	AH = 13h AL = 0 ==> BL contains Attribute, do not update Cursor 1 ==> BL contains Attribute update Cursor position 2 ==> String contains Attribute bytes, do not update Cursor 3 ==> String contains Attribute bytes, update cursor BH = Video Page BL = Attribute - If Bit 7 set in graphics mode, then character is XORed into the screen display. CX = String Length DH = Character row to start at DL = Character Column to start at ES:BP = address of string	Display a character string. The most useful function for what you want to do.

So how do you use all this information? The basic logic is simple. When a hot key indicates that you are supposed to pop-up, you first check the video mode. If it is an alphanumeric mode, you calculate where on the screen you wish to place your window, save that data area to a local buffer by reading video memory directly, and then use INT 10h AH=13h, AL=0 to draw your window. When the TSR "goes away," you simply copy the save buffer back to video memory. Please note, on some CGA adapters, this process of copying characters directly

to video memory can cause flicker and snow during the transfer of data. It is possible to add code that specifically eliminates this. However, since most display adapters sold these days are of EGA quality or better, I will ignore this problem. To address this problem I refer you to pages 66–75 of *PC & PS/2 Video Systems*.

If you are in graphics mode, you could follow the same procedure. There are two catches: calculating the size and location of the area to save is somewhat more difficult since it depends on the display mode, you are not guaranteed that the contents of the 1Fh and 43h character vectors have not been changed to something illegible. To avoid displaying garbage, you should first load these vectors with "good" data.

Before you rush out and issue an INT 10h AX = 1122h, you first need to save the value of the two video vectors and any of the parameters used. You need to do this so that you can restore these values prior to "going away." The following code pops up in response to Ctrl-Shift-K and displays "Hello Reader" in the middle of the display. To make the message "go away" press Ctrl-Shift-S. For clarity, the fragment in Listing 15.4 assumes that the TSR is already installed and that TimerTick and INT 09h have already been "hooked." The code also assumes that data for the 1Fh and 43h vectors resides in another file. The best way to gather this data is to use TurboDebugger to snapshot the values of a "standard" system.

LISTING 15.4. SAVING SCREEN INFORMATION IN TSRS.

```
.model small
.code

        EXTRN   pOld09:DWORD, pOld1Ch:DWORD
        EXTRN   pNew43h:DWORD, HotKeyScanCd:BYTE
        EXTRN   KBDShiftState:BYTE
        PUBLIC C i09_Hdlr, TimerTick

TSRToggle       DB      0       ;True if TSR is to toggle State at Timer Tick
TSRState        DB      0       ; True if TSR is "pop-ed up"
OurMSG          DB      "Hello Reader"
pOld43h         DD      ?       ; Save vectors for the Old character table
pOld1Fh         DD      ?       ; pointers
VidSeg          DW      0B800h  ; Segment of Start of video memory
cbScreenData    DW      ?       ; Amount of data saved
```

continues

LISTING 15.4. CONTINUED

```
pScreenData     DD      ?       ; Where you lopped the data from
SaveBuf         DW      4096 DUP (?)    ; Use a 4k buffer to handle Graphics
                                        ;   modedata

        lES     DI, pScreenData         ; Set up area to restore
        mov     CX, cbScreenData
        lea     SI, SaveBuf             ; Set up area to put back

; i09_Hdlr - Interrupt 09 Handler
;
;       This handler is called whenever any key is pressed, including
;       the shift keys.  It is extremely low-level, dealing with scan codes.
;       To maintain compatibility with BIOS functionality, this handler
;       does as little as possible.  It reads the scan code in from
;       port 060h and then calls the previous Int 09h handler.
;
;       After the old handler returns, the Scan code is compared to the
;       value stored in HotKeyScanCd (see TesSeRact example).  If a
;       match exists, check to see if the desired Ctrl and Alt key
;       is depressed as well by issuing an INT 16h AH=02h and comparing
;       the returned value against KBDShiftState.  If a match is found,
;       set a flag that requests the TSR change its state at the
;       next timer tick interval.
;
i09_Hdlr PROC C
    ASSUME DS:NOTHING, ES:NOTHING
        in      AL, 060h                ; Get the scan code into AL from
        push    AX                      ;   KBD port and save it

        pushf                           ; Simulate an Interrupt to the
        call    DWORD PTR [pOld09]      ;  old handler, so it will IRET here

        pop     AX                      ; Recover the Scan code
        cmp     AL, HotKeyScanCd        ; and check it
        je      i09_10_ChkShift

        iret                            ; No match

i09_10_ChkShift:
        mov     AH, 02h                 ; Get KBD status flags from BIOS
        int     16h                     ;   into AL

        and     AL, 0Fh                 ; Mask off ones you don't care about
        cmp     AL, KBDShiftState       ;   and check for a match
        je      i09_20_GotIt
```

```
        and     al, 03h                 ; Check for shift state since it
        and     al, KBDShiftState       ;  is reported separately for both
        jnz     i09_20_GotIt            ;  sides

        iret                            ; No Match

i09_20_GotIt:
        mov     TSRToggle, 0FFFFh       ; Set mark for Timer tick
        iret

        ENDP


; TimerTick - Int 1Ch handler, the workhorse of the example
;
;       This routine monitors the value in TSR_Toggle.  If a change of
;       state is requested, check TSRState to determine which of the
;       two subroutines to call TSRPopUp or TSRGoAway and dispatch
;       appropriately.
;
TimerTick PROC C
        ASSUME DS:NOTHING, ES:NOTHING   ; You got here via interrupt
                                        ; So you only know CS
        pushf                           ; Save state of interrupt flag
        cli
        cmp     TSRToggle, 0FFh         ; Check for toggle request
        je      tt_20_DoToggle

tt_10_GoChain:                          ; Go chain to the next handler
        popf
        jmp     DWORD PTR [pOld1Ch]

tt_20_DoToggle:
        mov     TSRToggle, 0            ; Reset the toggle request
        cmp     TSRState, 0FFh          ; Check if you are "pop-ed" up
        je      tt_30_GoAway

        call    DoPopUp                 ; You were requested to pop-up
        jmp     tt_10_GoChain

tt_30_GoAway:
        call    DoCleanUp               ; You were requested to "go away"
        jmp     tt_10_GoChain

        ENDP

BIOS_DATA_SEG   EQU     040h    ; Segment of BIOS data area
VID_MODE        EQU     049h    ; Offset of video mode in 40: data area
ROWS            EQU     084h    ; Offset of Row entry in BIOS data
```

continues

LISTING 15.4. CONTINUED

```
COLS            EQU     04Ah    ; Offset of Column count entry
VID_PAGE        EQU     062h    ; Offset of current Video page value
VID_SIZE        EQU     04Ch    ; Size of the current video buffer
VID_START       EQU     04Eh    ; Start of the current video buffer

; DoPopUp - Pop you up
;
;       This routine checks the display mode, and calls the
;       appropriate setup routines. It then displays "HELLO Reader" in
;       the center of the screen and returns.
;
DoPopUp PROC NEAR
    ASSUME DS:NOTHING
        pushf
        sti                             ; Allow interrupts
        push    CX                      ; Save registers
        push    DX
        push    DI
        push    SI
        push    ES
        push    DS

        push    CS                      ; Get DS to point to local vars
        pop     DS
    ASSUME DS:CODE
        mov     AX, BIOS_DATA_SEG
        mov     ES, AX                  ; Set up pointer to BIOS data area
        mov     AL, ES:[VID_MODE]       ; AL := Video Mode

        cmp     AL, 4                   ; Check for mode 4-6 graphics
        jb      dpu_20_NotGraph         ; Low modes are all alphanumeric

        cmp     AL, 6
        ja      dpu_50_ChkHigh          ; Check for modes above 0D




dpu_10_IsGraph:
        cli
        call    LoadGraphChar           ; Load the known array of graphics
                                        ; Characters
        call    SetRowCol               ; Returns with DX set to screen
                                        ; center
        call    SaveGraph               ; Save the graphics display

        jmp     SHORT dpu_30_ShowMsg
```

```
dpu_20_NotGraph:
        call    SetRowCol               ; Returns with DX set to screen
                                        ; center
        call    SaveAlpha               ; Save graphics display

dpu_30_ShowMsg:         ; OK use BIOS 10h AH=13 to paint string on screen
        mov     AX, BIOS_DATA_SEG
        mov     ES, AX
        mov     BH, ES:[VID_PAGE]

        push    CS                      ; ES:BP := "Hello Reader"
        pop     ES
        lea     BP, OurMsg
        mov     CX, LEN "Hello Reader"

        mov     BL, 7                   ; Attribute to use
        mov     AH, 013h
        mov     AL, 0                   ; Issue string but leave cursor alone
        int     13h

dpu_40_Leave:
        pop     DS
        pop     ES
        pop     SI
        pop     DI
        pop     DX
        pop     CX
        popf
        ret

dpu_50_ChkHigh:
        cmp     AL, 0Dh                 ; Check for modes 0Dh and above
        jb      dpu_20_NotGraph

        jmp     dpu_10_IsGraph

        ENDP

; DoCleanUp -  Clean up the Display screen
;
;       All this routine does is copy the video data buffer back
;       over the "Hello reader".  It then checks if it needs to reset
;       the 1Fh and 43h vectors and does so if you are in graphics mode.
;
DoCleanUp PROC NEAR
    ASSUME DS:NOTHING
        sti                             ; Allow interrupts
        push    CX                      ; Save registers
        push    DI
        push    SI
```

continues

LISTING 15.4. CONTINUED

```
        push    ES
        push    DS

        push    CS                      ; get DS to point to local vars
        pop     DS
    ASSUME DS:CODE

        lES     DI, pScreenData         ; Set up area to restore
        mov     CX, cbScreenData
        lea     SI, SaveBuf             ; Set up area to put back
        rep     MOVSB


        mov     AX, BIOS_DATA_SEG
        mov     ES, AX                  ; Set up pointer to BIOS data area
        mov     AL, ES:[VID_MODE]       ; AL := Video Mode

        cmp     AL, 4                   ; Check for mode 4-6 graphics
        jb      dcu_20_NotGraph         ; Low modes are all alphanumeric

        cmp     AL, 6
        ja      dcu_30_ChkHigh          ; Check for modes above 0D

dcu_10_IsGraph:         ; You have a Graphics mode, so reset the two graphics
        cli             ; mode character vectors.  NOTE, your CLI so nothing
                        ; corrupts while you do this.

        lES     DI, pOld1Fh             ; First reset 1Fh
        mov     CX, ES                  ; CX:DI == pOld1Fh
        xor     AX, AX
        mov     ES, AX
        mov     ES:[01Fh * 4], DI
        mov     ES:[(01Fh * 4) + 2], CX

        lES     DI, pOld43h             ; Now reset 43h
        mov     CX, ES                  ; CX:DI == pOld1Fh
        xor     AX, AX
        mov     ES, AX
        mov     ES:[043h * 4], DI
        mov     ES:[(043h * 4) + 2], CX

dcu_20_NotGraph:
        pop     DS
        pop     ES
        pop     SI
        pop     DI
```

```
        pop     CX
        ret

dcu_30_ChkHigh:
        cmp     AL, 0Dh                 ; Check for modes 0Dh and above
        jb      dcu_20_NotGraph

        jmp     dcu_10_IsGraph

        ENDP


; LoadGraphChar - Load Graphics Character array
;
;      This subroutine loads an 8x8 character set for use in
;       graphics mode into the character generator.  Because this causes a
;       reset of the 43h vector (and possibly the 1Fh vector) you save
;       both of these vectors first, so that they can be reset later.
;       You assume interrupts are disabled.
;
LoadGraphChar PROC NEAR
        push    BP
        xor     AX, AX          ; address Interrupt Vector Table
        mov     ES, AX
        lES     DI, DWORD PTR ES:[1Fh * 4]
        mov     WORD PTR [pOld1Fh], DI
        mov     WORD PTR [pOld1Fh+2], ES

        xor     AX, AX          ; address Interrupt Vector Table
        mov     ES, AX
        lES     DI, DWORD PTR ES:[43h * 4]
        mov     WORD PTR [pOld43h], DI
        mov     WORD PTR [pOld43h+2], ES

        mov     AX, BIOS_DATA_SEG
        mov     ES, AX                  ; Set up pointer to BIOS data area
        mov     DL, ES:[ROWS]           ; DL := rows on screen

        lES     DI, pNew43h     ; Get pointer to table of known char values
        mov     BP, DI

        mov     AX, 01123h
        mov     BL, 0                   ; Let DL set row count
        int     10h

; NOTE: The correct thing to do here would be to check for the presence
;       of an MCGA, and lock the fonts into it if it's present. For
;       readability you won't do that in this sample.
```

continues

LISTING 15.4. CONTINUED

```
        pop     BP
        ret
        endp


; SaveAlpha - Save the Alphanumeric values currently at the display
;
;       Puts the data into the local SaveBuf. NOTE, the amount you
;       store is twice the length of "Hello Reader" because you
;       save the attributes as well.
;
SaveAlpha  PROC NEAR
        push    DX                      ; Trashed by MUL
        mov     AX, BIOS_DATA_SEG
        mov     ES, AX

        mov     SI, ES:[VID_START]      ; SI:= Start of video buffer
        xor     AX, AX                  ; Zero AH
        mov     AL, BYTE PTR ES:[ROWS]  ; AX := rows * columns * 2 / 2 for
        mov     CX, WORD PTR ES:[COLS]
        mul     CX
        add     SI, AX                  ; SI := Start of paint area

        cmp     BYTE PTR ES:[VID_MODE], 7       ; Get Segment of video mem
        ja      sa_20_IsA000
        jb      sa_10_IsB000

        mov     AX, 0b800h
        mov     VidSeg, AX
        jmp     SHORT sa_30_GotSeg

sa_10_IsB000:
        mov     AX, 0B000h
        mov     VidSeg, AX
        jmp     SHORT sa_30_GotSeg

sa_20_IsA000:
        mov     AX, 0A000h
        mov     VidSeg, AX

sa_30_GotSeg:
        push    DS                      ; Set up pointers for move into
        pop     ES                      ; SaveBuf
    ASSUME DS:NOTHING, ES:CODE
        mov     DS, AX
        lea     DI, SaveBuf
```

```
        mov     CX, LEN "Hello Reader"
        mov     cbScreenData, CX
        mov     WORD PTR [pScreenData], SI
        mov     WORD PTR [pScreenData+2], DS
        rep     MOVSW                   ; Move attributes as well as chars
        pop     DX
        ret
        ENDP

; SetRowCol - Return the ROW:Column in DX DH == Row, DL = Column
;
;       Calculate the RowColumn position of the center of the screen
;       for use by the 13h function and return the position in DX
;
SetRowCol  PROC NEAR
        mov     AX, BIOS_DATA_SEG
        mov     ES, AX                  ; Set up pointer to BIOS data area
        xor     DX
        mov     AX, DX
        mov     DX, WORD PTR ES:[COLS]
        shr     DX, 1                   ; Divide by two for screen middle
        mov     AL, BYTE PTR ES:[ROWS]  ; AX := rows * colums * 2 / 2 for
        shr     AX, 1
        mov     DH, AL
        ret
        ENDP

; SaveAlpha - Save the Alphanumeric values currently at the display
;
;       Puts the data into the local SaveBuf. NOTE, the amount you
;       store is twice the length of "Hello Reader" because you
;       save the attributes as well.
;
SaveAlpha  PROC NEAR
        push    DX                      ; Trashed by MUL
        mov     AX, BIOS_DATA_SEG
        mov     ES, AX

        mov     SI, ES:[VID_START]      ; SI:= Start of video buffer
        xor     AX, AX                  ; Zero AH
        mov     AL, BYTE PTR ES:[ROWS]  ; AX := rows * columns * 2 / 2 for
        mov     CX, WORD PTR ES:[COLS]
        mul     CX
        add     SI, AX                  ; SI := Start of paint area

        cmp     BYTE PTR ES:[VID_MODE], 7       ; Get Segment of video mem
        ja      sa_20_IsA000
        jb      sa_10_IsB000
```

continues

LISTING 15.4. CONTINUED

```
        mov     AX, 0b800h
        mov     VidSeg, AX
        jmp     SHORT sa_30_GotSeg

sa_10_IsB000:
        mov     AX, 0B000h
        mov     VidSeg, AX
        jmp     SHORT sa_30_GotSeg

sa_20_IsA000:
        mov     AX, 0A000h
        mov     VidSeg, AX

sa_30_GotSeg:
        push    DS                      ; Set up pointers for move into
        pop     ES                      ; SaveBuf
    ASSUME DS:NOTHING, ES:CODE
        mov     DS, AX
        lea     DI, SaveBuf
        mov     CX, LEN "Hello Reader"
        mov
        rep     MOVSW                   ; Move attributes as well as chars
        pop     DX
        ret
        ENDP

; SaveGraph - Save the graphics data currently in the display
;
;       Puts the data into the local SaveBuf. WARNING! This routine
;       is not the best way to save off the data.  For simplicity,
;       you save a chunk of memory that should be big enough in standard
;       video modes to cover the area that you write to with subfunction
;       12h.  The correct solution would be to save ONLY that area that
;       you overwrite.  This requires 2 pages of assembler code for
;       each combination of pixel dimension x color count.  As shown in
;       table 02 this is 7 subroutines.
SaveGraph  PROC NEAR
        mov     AX, BIOS_DATA_SEG
        mov     ES, AX

        mov     SI, ES:[VID_START]      ; SI:= Start of video buffer
        mov     AX, ES:[VID_SIZE]       ; point to the middle of the buffer
        shr     AX, 1


        cmp     BYTE PTR ES:[VID_MODE], 7       ; Get Segment of video mem
```

```
        ja      sg_20_IsA000
        jb      sg_10_IsB000

        mov     AX, 0b800h
        mov     VidSeg, AX
        jmp     SHORT sg_30_GotSeg

sg_10_IsB000:
        mov     AX, 0B000h
        mov     VidSeg, AX
        jmp     SHORT sg_30_GotSeg

sg_20_IsA000:
        mov     AX, 0A000h
        mov     VidSeg, AX

sg_30_GotSeg:
        push    DS                      ; Set up pointers for move into
        pop     ES                      ; SaveBuf
    ASSUME DS:NOTHING, ES:CODE
        mov     DS, AX
        lea     DI, SaveBuf
        mov     CX, 4096
        mov     cbScreenData, CX
        mov     WORD PTR [pScreenData], SI
        mov     WORD PTR [pScreenData+2], DS
        rep     MOVSB
        ret
        ENDP
```

As the code mentions, the routine SaveGraph is at best a kludge. This is because in graphics mode, adjacent rows are not adjacent in memory. This is an artifact of early displays which were interlaced. An interlaced display (like your TV) paints all of the even rows first, followed by all of the odd rows. This is to reduce hardware costs. In any case, to understand more about pixels, and saving the data they use, see Chapter 5 of *PC & PS/2 Video Systems*.

WINDOWS AND OTHER GOTCHAS

If you install your TSR as part of your standard start-up sequence and at some later point load Microsoft Windows, you will be in the dangerous realm of running a TSR under Microsoft Windows. The simple solution to this is to

unload your TSR before you load Microsoft Windows. While workable, it is inconvenient and not very elegant. Instead it would be preferable to understand what negative interactions might occur with Microsoft Windows and prevent them from occurring.

GOTCHAS OF WINDOWS IN GENERAL

Three sets of issues exist, corresponding loosely to the three modes Microsoft Windows can use. In all modes, Microsoft Windows drives the display in graphics mode, and preempts the INT 09h keyboard interrupt vector. Microsoft has spent much time and effort in researching how best to use the graphics hardware available. The result is that they do not make any pretense of supporting the INT 10h functionality. The only way to pop up in MS-Windows is to do it in real mode and program the low-level graphics hardware directly. Not a good idea—especially since some of that hardware uses read-only registers, which makes restoring the appropriate mode and data nigh unto impossible.

GOTCHAS OF WINDOWS STANDARD MODE

In standard mode, Microsoft Windows switches the CPU mode dynamically between 8086 real-mode emulation mode and 80286 *protected mode*. In protected mode, *segments* are replaced with *selectors*. Physical address calculations cannot be made on selectors. Hence a TSR should avoid making such calculations while Microsoft Windows is running.

GOTCHAS OF WINDOWS 386 ENHANCED MODE

In enhanced mode, Microsoft Windows switches the CPU mode dynamically between 8086 real mode and 80386 enhanced mode. In this mode, Microsoft Windows saves a snapshot of the interrupt vector table during start-up. It then prevents any further direct access to this region of memory by invoking the virtual memory features of the 80386. When Windows launches an MS-DOS shell, or an MS-DOS program using a PIF file, it creates a *virtual machine* (VM) for each incarnation launched. Each VM is given access to a virtual block of memory that is almost identical to that which the application would have were it running only under MS-DOS. The catch is in the *almost*.

Any changes the application makes to the protected region of the interrupt vector table apply only within that particular virtual machine. Thus if an application were to set an interrupt vector to point to some other handler, this change would only be reflected in the local copy of that chunk of memory. This does not apply to the majority of the free MS-DOS memory.

Hence, if you were to invoke the transient portion of your TSR and request that the TSR unload itself, the TSR would be unloaded; however, the interrupt vectors would not be unhooked. This could lead to the very results discussed in the section on interrupt vectors.

Another feature of the MS-DOS virtual machine is that all standard MS-DOS devices (LPT1–LPT3, COM1, and COM2) that are present during the start-up of Microsoft Windows are virtualized as well. Specifically, Windows installs default handlers for these devices. It then prevents MS-DOS VMs from directly modifying the state of these devices. Instead, Windows allows only one VM at a time to control a port and only in the limited mode that the default handlers provide. There are two ways to circumvent this. You can write a replacement for the default handlers that Windows installs. This topic requires a complete book in itself. Alternatively, you can fool Windows into not recognizing any device your TSR wishes to manage. Windows uses the presence of port values in the 40:0 BIOS data region for detecting these devices. By zeroing the entry that corresponds to the device you wish to manage, you fool Windows into ignoring the presence of that device. However, this doesn't avoid the problem of not being able to change the interrupt vector.

The last gotcha in 386 Enhanced mode is that the Timer Tick interrupt is virtualized as well. As a result, a time-critical TSR that is running in one of the MS-DOS VMs is not guaranteed to receive every timer event that occurs.

DETECTING THE PRESENCE OF WINDOWS

Fortunately Microsoft provides two mechanisms for detecting the presence of Windows. The safest solution to all of the above problems is to disable all reconfiguration of the TSR while Windows is running. Microsoft Windows notifies other applications of start-up by issuing INT 2Fh AX = 1605h. Upon termination, Windows issues INT 2Fh AX = 1606h. By monitoring these two events, you can lock out reconfiguration requests while Microsoft Windows is running.

To prevent a TSR from being installed within an MS-DOS VM, you can issue INT 2Fh AX = 1600h during the start-up of the TSR. If this returns with AX != 0080h you know Microsoft Windows is running, and you abort the installation of the TSR.

EOIs

At various points in this chapter, I have made reference to the Intel 8259 Programmable Interrupt Controller. I mentioned earlier that chaining to hardware interrupt vectors was preferable to hooking them since it allowed you to ignore dealing with the 8259 PIC. This is true as a general case. If, however, you are writing a handler for a custom piece of hardware, you do not have this luxury since the default interrupt handler will not properly reset the 8259 chip.

Specifically, when the CPU acknowledges an interrupt to the 8259 PIC, the PIC masks all lower-priority interrupts until it receives an End-Of-Interrupt instruction. This is commonly referred to as an EOI. A properly written hardware interrupt handler will issue the EOI as soon as it is safe to do so. The primary concern is how to handle nested interrupts, because as soon as the EOI is issued, the same interrupt could be generated again. This could be the Timer Tick, or a network card that is receiving data. If the interrupt handler is complex, the usual approach is to use a software flag to prevent reentrance, and to issue the EOI as soon as this flag is set.

To issue an EOI on a PC or PS/2, you need to know which IRQ you are handling. If you are responding to IRQ 0–7, then you need only dismiss the EOI at the primary PIC. AT-class machines have a second PIC that supports IRQs 8–15. This second PIC is cascaded into the primary PIC on IRQ 2. Therefore, if you are responding to IRQ 8–15, you need to issue an EOI to both PICs. The following code fragment issues an EOI to both PICs.

```
EOI     EQU     020h
        cli                     ; Disable Interrupt processing by
                                ; the CPU
        mov     AL, EOI
        out     020h, AL        ; Issue the EOI to the primary PIC
        mov     AL,   EOI
        out     0a0h, AL        ; Issue the EOI to the secondary PIC
```

For an example of the use of this sequence of instructions, please refer to Chapter 16, “High-Speed Serial Communications.”

UNLOADING AND CLEANING UP

One of the disadvantages of using TSRs is that they consume memory in the valuable region below the 1 Megabyte boundary, even when they are not being used. While MS-DOS provides a published mechanism for loading TSRs, no mechanism is defined for unloading TSRs when they are no longer needed. If you wish to unload your TSR, you therefore must make some assumptions and depend on some undocumented calls in MS-DOS.

RELEASING INTERRUPT VECTORS

You primarily want to unload a TSR to free the memory that it consumes. Since this memory can then be reused by other applications, you must first ensure that any interrupt vectors that point to code in your TSR are reset to the values they contained prior to installation of your TSR. Since you saved away the value of the original vector, this should be easy. Just issue MS-DOS function request 25h (set interrupt vector) with the original value, and you're done, right? Wrong!

You can only unhook your interrupt handler if no other TSR or application has chained to the vector after your TSR's installation routine was run. You check this by comparing the value of the pointer in the interrupt vector table with the address of the entry point to your interrupt handler. You can extract the value from the interrupt vector table by either reading it directly from the interrupt vector table, or by issuing MS-DOS function request 35h (get interrupt vector). I prefer reading it directly since that can be done faster and with less code (this is one of the reasons that Table 15.1 lists the memory addresses of the interrupt vectors). You then compare the result with the address of your interrupt handler. If they differ, then you know some other program has chained itself to that vector, and you must abort unloading your TSR.

If you were to continue to unload the TSR, and high-handedly replace the interrupt vector with its original value, the program that hooked the vector after you would cease to function properly.

The second toughie in unhooking your interrupt handlers is that Borland C++ hooks some interrupt vectors of its own during program start-up. This is to provide various enhanced behaviors such as clean termination in the case

of divide-by-zero. Any third-party library routines that were linked in may have done the same. Since you use an explicit call to MS-DOS using `intdosx` to install your program as a TSR, you bypass all of the standard Borland C++ process termination and cleanup routines.

There is no simple way to solve this problem. I prefer to use Turbo Debugger to inspect the interrupt vector table. When the program initially loads, I write down a copy of the interrupt vector table, and place a breakpoint at `_main`. When that breakpoint is hit, I re-inspect the interrupt vector table to identify which vectors the library start-up routines have changed.

I then repeat the process, but this time I set breakpoints to trigger whenever anything changes one of the vectors I had identified in the first pass. This is done by selecting Breakpoints | Change global memory and entering the address of the interrupt vector that was changed. When a breakpoint is encountered, I inspect the code that is executing. This allows me to identify where the start-up routine is storing the original value. This is the only completely reliable method for identifying what vectors need to be unhooked, and where to find the original vector values.

Note that, as I mentioned earlier, floating point packages will use INT 75H and INT 02H to emulate the floating point processor when it is not installed. If you are compiling your code with floating point processing enabled, the above process should be repeated on hardware that has a floating point processor, as well as hardware that does not. Otherwise, you will not be guaranteed to have discovered all of the potentially offending vectors.

The alternative is to write the TSR in assembly language. Note that Borland is not alone in not documenting what vectors the run-time libraries use. My first encounter with this problem came while using Microsoft C 5.00. It appeared as a defect that destabilized MS-DOS, but only on some machines, and only when they ran certain applications after the TSR had been unloaded. As you may well imagine, this was not an easy defect to track down.

RELEASING MEMORY

Now that you have unhooked the interrupt vectors that were used by your TSR, you can begin to free memory. But before you free memory, you should understand a little about how MS-DOS allocates and manages memory. Every time a program issues MS-DOS function request 48h (allocate memory),

MS-DOS finds the first block of free memory that is at least as large as the allocation request plus 16 bytes. It then builds a 16-byte header (also called the memory arena header, or arena header for short) that contains a status flag, the Process ID (PId) of the process that is requesting the allocation, and the size of the memory block being allocated. If the block of free memory being used is larger than the size requested, a second 16-byte header is built immediately after the newly allocated memory. This header contains a 0 for the PId to indicate "unowned" or free memory. The block size contains the balance of memory that was available in the original chunk, minus 32 bytes for the two memory headers.

When MS-DOS loads, it first creates a single header to describe all of available memory below the 1M boundary. As each successive device driver and program is loaded, MS-DOS allocates memory to it from the initial single memory block. The result is that once the command prompt is displayed, memory has been organized into a series of allocated blocks that are "owned" by various device drivers and the command interpreter (command.com usually). Since device drivers do not have the MS-DOS memory re-allocation function requests available for use, and since the first program loaded is the command interpreter, you initially have some blocks of allocated memory followed by a single block of free memory.

Because MS-DOS does not implement any "memory garbage collection" and only performs "memory compaction" on adjacent free memory blocks, many applications assume the above memory model. To save disk space, and to improve the speed at which programs load, some programs are written so that the program that loads initially is a small configuration program, that then reallocates the initial size to include all of the rest of memory and loads the main program as an overlay. This works fine unless a "memory hole" develops. A memory hole is a block of free memory that is bounded above and below by allocated memory blocks.

MEMORY HOLE ISSUES

If a memory hole exists, it is possible that the initial program is small enough to fit into the memory hole. When this program goes to reallocate memory, it is only able to grow the initial allocation up to the size of the memory hole. The program then exits with the error message `insufficient memory to load`. CHKDSK still reports the availability of hundreds of thousands of bytes of memory, leading to much consternation and hair pulling. Clearly memory holes should be avoided.

What then creates a memory hole? The most common cause is the unloading of a TSR. Because MS-DOS allocates memory on a first-come/first-served basis, the memory allocated to a TSR is in that series of allocated blocks in the lower part of memory. If any TSR has been loaded after your TSR, then it will have memory allocated immediately following your memory block. If you unload your TSR in this configuration, and free your block of memory, you will have created a memory hole. Therefore, just like with interrupt vectors, you need to verify that all TSRs that were loaded subsequent to yours have been unloaded before you unload your TSR.

As a side note, if a TSR were to allocate memory while running, one of the side effects might be to create a memory hole the size of whatever program was loaded at the time of the allocation.

How do you detect if any TSRs or programs occupy memory above you? A couple of mechanisms exist. The most reliable method walks the chain of memory blocks, identifies which ones belong to the TSR, and looks for allocated memory directly above the TSR. Note that you must think through this logic carefully. If you are going to use a run-time program to communicate with the TSR, the TSR must discount any memory that has been allocated to this tool.

Another thorny issue is that you need to free *all* of the memory allocated to your TSR. This includes the environment segment as well as any segments that the C++ run-time libraries may have allocated to themselves during start-up. Ideally this latter case is avoided by compiling the TSR using the small or tiny memory models. By walking the complete list of memory blocks, and freeing all of those that are "owned" by your TSR, you can accomplish your goal. The following code fragment walks the list of memory blocks, identifying the blocks "owned" by the TSR. It also tracks the highest in-use block of memory that is not "owned" by the TSR. Finally it checks to see if the highest in-use block of memory is located above the TSR.

```
#include <dos.h>

typedef struct
{
       unsigned char BlockType;        // 'Z' == Last Block, 'M' otherwise
       unsigned int  PIdOwner;         // PId of owning process
       unsigned int  BlockSize;        // Block size in "paragraphs"
} MARENA;
```

```
/* Walk_mem - Walk the memory chain and free any blocks owned by this TSR
 *
 *     This routine takes no inputs, it returns 0 if successful, and -1
 *     if it was unable to unload the TSR.
 *
 *     WARNING! this subroutine relies on calls that are not publicly
 *             documented by Microsoft
 */
int
Walk_mem()
{
       union REGS   InRegs, OutRegs;
       struct SREGS SegRegs;
       MARENA far * pMemHdr;           // Ptr to memory header being inspected
       unsigned int PId;               // Process Id
       unsigned int oPId;              // PId of last allocated block seen
       unsigned int i = 0,
                    OurBlock[5];       // Array to hold segments owned by
                                       //   your TSR

       InRegs.x.ax = 0x5100;           // Get PSP segment, PId == PSP segment
       intdos( &InRegs, &OutRegs  );
       PId = OutRegs.x.bx;

       InRegs.x.ax = 0x5200;           // Get Dos Parameter Block
       intdosx( &InRegs, &OutRegs, &SegRegs );

     /* See the DOS Parameter Block description on p633 and 634 of the DOS
      * Programmers Reference*/
       pMemHdr = (MARENA far *) MK_FP(SegRegs.es, OutRegs.x.bx - 2);

      /* Walk the list of memory blocks until the last one is encountered */
       while (pMemHdr->BlockType != 'Z' )
       {
             /* An Owner PId == 0 indicates the block is free */
              if (pMemHdr->PIdOwner != 0 )
              {
                   /* If this block doesn't belong to your TSR, remember
                    * its offset so that later you can check it against your
                    * PSP offset. NOTE, here is where you would add any
                    * code to ignore a utility that is used to unload the TSR*/
                     if ( pMemHdr->PIdOwner != PId )
                     {
                            oPId = pMemHdr->PIdOwner;
                     }
                     else
                     {
                            /* Since this is one of the memory segments "owned"
                             * by your TSR, save its segment address so that
```

```
                                * later you can free it if it is safe to do so.
                                * NOTE, since you are walking the memory HEADERS,
                                * the actual memory segment is 16 bytes offset
                                * from the Header.
                                  OurBlock[i++] = FP_SEG(pMemHdr) + 16;
                        }
       }
      /* Scroll to the next memory header. NOTE: The BlockSize is given
       * in paragraphs (16-byte chunks), and the size of the header is
       * not included, so add an extra paragraph to the BlockSize and
       * multiply by 16.*/
        pMemHdr += (pMemHdr->BlockSize + 1) * 16;
  }

/* Check to see if any program is installed above you. */
  if ( oPId > PId )
  {
          // Abort the TSR unload if this is true,
          return(-1);
  }
  else
  {
        /* Walk the list of blocks allocated to your TSR freeing them */
         asm {cli}
         for( ; i >= 0; i--)
         {
                InRegs.h.ah = 0x49;   // Free Mem Request
                SegRegs.es = OurBlock[i];
                intdosx( &InRegs, &OutRegs, &SegRegs );
         }
  }
  return( 0 );
}
```

ENVIRONMENT SEGMENT

The above code fragment will free the segment values of all of the segments owned by the TSR, including the environment segment that was allocated to the TSR when it was initially loaded. Why then did you not get rid of the environment segment as part of the original TSR process? After all, it would have freed up some memory. The MS-DOS Exec function request (4Bh) first allocates memory for the applications parameter block, and then for the program itself. The MS-DOS command interpreter (command.com) passes a

complete copy of the MS-DOS environment segment as the program's parameter block. Since users can set environment segments to be quite large, freeing this segment just prior to issuing the MS-DOS TSR function request would cause a memory hole of unknown size to occur.

DETECTING OTHER TSRS

The last two sections, "Releasing Interrupt Vectors" and "Releasing Memory," have a common theme. A TSR cannot safely be unloaded unless all of its interrupt vectors can be released, and unless no TSRs exist in memory that have been loaded after your TSR was. Since you have no control over which vectors are likely to be chained onto, or which programs are loaded after your TSR, your code must first ensure that all conditions are met that allow safe unloading of TSRs before you proceed with release of any vectors or memory.

This means that if a sequence of TSRs is to be loaded, they should be loaded in reverse order of how they are likely to be unloaded. Some TSRs will cause problems no matter what. Some early TSRs attempted to ensure their ability to have first crack at keyboard data, or the timer tick event, by continually rechaining to these vectors whenever they detected that another interrupt handler had been installed. The solution to these types of TSRs is to avoid them, or to always load them last. Symptoms of these types of TSRs are the sudden inability for your TSR to unload, even though you are sure you have not loaded any TSRs after your own.

ADVANCED TOPICS

WALKING THE DEVICE DRIVER CHAIN

As I mentioned earlier, a TSR can be used as a load-on-demand device driver. Note, this is limited to *character devices* only (a character device is one that performs I/O as a stream of bytes rather than as a *block* of bytes. *Block devices* are used for disk drivers). To accomplish this, the TSR must first be written to conform to the MS-DOS device driver structure. The details of writing an

MS-DOS device driver are beyond the scope of this chapter. Refer to the example TEMPLATE.SYS on page 471 of *The MS-DOS Encyclopedia*. Once you have written your device driver, the following modification will allow you to load it dynamically.

MS-DOS maintains a linked list of device driver headers known as the *device driver chain*. This list can be accessed by issuing MS-DOS function request 52h. The data block that is returned contains a pointer to the beginning of the NULL device. The NULL device is the first device in the device driver chain. By inserting itself in the chain, immediately after the NULL device, the TSR has become a device driver. The following structure definition describes the data pointed to by ES:BX upon return:

```
typedef struct _DPB
{
     char far * pDriveParmBlock;        // Pointer to First Drive Parameter
                                        // Block
     char far * pDeviceControlBlock;    // Pointer to First Device Control Block
     char far * pClock$;                // Pointer to the current Clock$ device
                                        // driver
     char far * pCON;                   // Pointer to the current CON device
                                        // driver
     union
     {
          struct
          {
               unsigned char cDrives;   // Number of Logical Drives supported
               unsigned int  cbSector;  // Number of Bytes/Sector for installed
                                        // Block Devices
               char far *    pBuff;     // Pointer to the start of the Disk
                                        // buffer chain
               char far *    pNULL;     // Pointer to the Null device driver,
                                        // also the start of the Device
                                        // Driver Chain
          } _20;
          struct
          {
               unsigned int  cbSector;  // Number of Bytes/Sector for installed
                                        // Block Devices
               char far *    pBuff;     // Pointer to the start of the Disk
                                        // buffer chain
               char far *    pDrvTbl;   // Pointer to the Logical Drive Table
               char far *    pFCB;      // Pointer to the start of the MS-DOS
                                        // FCB chain
               unsigned int  cFCBSwap;  // Num. of "Keep" FCBs from FCBS
                                        // config.sys command
```

```
                unsigned char cBlockDev; // Number of Block Devices in system
                unsigned char cDrives;   // Number of Logical Drives, from
                                         // LASTDRIVE command
                char far *    pNULL;     // Pointer to the Null device driver,
                                         // also the start of the Device Driver
                                         // Chain

            } _30;
        } v;
} DosParmBlock_t;
```

If you write the TSR to support Device Open/Close and Read/Write, you can now dispense with the TesSeRact mechanism for communicating with your TSR. Instead you can issue MS-DOS function request 3Dh (Open) to get a handle to the TSR. You can exchange data with the TSR Device Driver using MS-DOS function requests 3Fh and 40h (Read and Write handle).

Even though the INT 21h MS-DOS function request interface is much easier to use than the INT 2Fh Multiplex API, this mechanism is not often used for two main reasons. Function request 52h is not an officially documented MS-DOS function request (you will not find it in *The MS-DOS Encyclopedia*, but you will find it in *DOS Programmer's Reference*, 2nd Edition) and may be changed or eliminated by Microsoft in future versions of MS-DOS. Secondly, and almost more importantly, MS-DOS device drivers are only invoked when InDos is set. Hence, no MS-DOS function request can be called from within a device driver!

The second limitation implies that all I/O must be delayed until some later Timer Tick event, or must be done at an extremely low level. The added complexity in implementing this must be taken into consideration when designing a TSR device driver.

CHAPTER 16

HIGH-SPEED SERIAL COMMUNICATIONS

BY GORDON FREE

A BRIEF HISTORY OF IBM PCS AND UARTS

Prior to the advent of personal computers, most interaction between computers and users took place over *serial data links* (so named because data traveled one bit after another). These links did not demand too much speed—a person can type and read only so fast. Besides, mechanical considerations limited the speed of these devices. (For example, Teletype terminals had a top speed of about 15 characters per second.)

A brief history of IBM PCs and UARTs

Processing incoming data

Implementing a high-speed serial interface class library

The interface was fairly simple: individual characters shifted out a single bit at a time over a wire and then reassembled into a whole byte at the other end. This interface was thoroughly defined by the Electronic Industries Association (EIA) in the mid-1960s as the *RS-232 specification*. With industry acceptance of this standard, you could easily mix and match devices and computers (so long as you didn't do such things as hook a device to a device or a computer to a computer).

If you've read much about serial or modem communications, you've no doubt come across the term *baud rate*. A simple correlation exists between baud rate and the number of characters you can transmit each second. The RS-232 standard specifies that each character or byte of data should start and end with at least one bit in a known state. This feature enables the receiver to stay in sync with the beginning and end of each byte. As a result, each byte of data requires two extra bits. To convert from baud rate to characters per second (cps), divide the baud rate by 10. For example, 2400 baud yields a maximum throughput of 240 cps. Convert to bits per second to compare. While technically a 2400 baud line can send 2,400 bits per second, you need to factor out the two-bit overhead when comparing to non-RS-232 links that don't carry such overhead. Multiply the cps by 8 bits per character to do that. This means that a 2400 baud line has the same throughput as a 1,920-bit-per-second synchronous link ($2400/10 \times 8 = 1920$).

While the first personal computer manufacturers knew 15 cps might be fine for the keyboard interface, they understood that applications such as graphics and form entry must display information in a more timely fashion. Enter *memory-mapped displays*. With these, information is presented to the user almost as fast as the CPU can write to memory. Still, many RS-232 devices are still out there (some highly specialized for particular industries).

When the IBM PC first appeared in 1981, one of the few interfaces available was a serial interface port to enable users to connect RS-232 devices to the PC. In fact, support for serial communications is built into the ROM BIOS. Serial printers and modems were the most common serial devices at that time (mice weren't yet in vogue). Printer technology had improved beyond the Teletype and could print at speeds of 1200 baud or more. (Centronic parallel printers could, of course, go even faster.) Modems, enabling a computer to talk to a similarly equipped remote computer over a telephone line, operated at speeds

of 300 or even 1200 baud. The BIOS was written to support these speeds and, in fact, could be pushed to 9600 baud, although the CPU speed set a more realistic maximum of 2400 baud.

The design of the hardware itself was a different story, however. IBM used the relatively new Universal Asynchronous Receiver/Transmitter from Intel (a.k.a UART, or the 8250 chip). Intel claimed the UART, assuming it had an adequate clock, could operate at speeds up to or exceeding 57,600 baud. Then an amazing thing happened: Normally conservative IBM engineers fed the UART a clock that enabled setting the UART to speeds of 115,200 baud. Remember, the BIOS could handle only 2400 baud reliably, and that probably was considered overkill. When you look at the overall hardware design of the original IBM PC, nothing stands out as so over-designed for that time as the serial ports: not the CPU (the 8088 wasn't that much more powerful than the Z80 or 6800, after all), not the memory (the original PC came with only 16K of RAM), not the display (remember CGA?), and certainly not the mass storage (160K and 180K floppies anyone?). Even today, 486 machines cannot send a single byte over the serial link faster than the original IBM PC. Needless to say, people took some time figuring out how to exploit all the power of the serial port.

Let's come down from the clouds momentarily to put things into perspective. Running the serial link at 115,200 baud gives a maximum throughput of 92,160 bits per second. Modern local area networks (LANs) can exceed 10,000,000 bits per second—two orders of magnitude faster! Clearly, the serial port is no match for a state-of-the-art LAN, but it still has advantages. First, every PC has one. (Probably less than one in every 20 PCs has a network adapter.) Also, the serial port has a well-defined standard interface. Every PC can connect to every other PC through the serial port. (Try connecting an EtherNet adapter to an ARCnet adapter!)

While not blazing, the PC's serial port surely provides reasonable data transfer rates. One road, however, blocks your path to high-speed serial communications. Remember, the PC's BIOS supports speeds only up to 2400 baud (maybe 4800, or even 9600 if you're lucky). Today, though, you're beginning to see modems operating at 57.6 Kbaud and even 115.2 Kbaud over standard phone lines, and when transferring data between machines, nothing is too fast. What, then, can you do about a too-slow interface? This situation isn't unique to the

PC. The video display has a similar problem: going through the BIOS routines to display information is too slow. The solution is to bypass the BIOS routines and go straight to the hardware. For the video display, write directly to the video RAM. To obtain higher speeds, do the same for the serial port—go straight to the hardware, that is, the UART.

LIFE, THE UART, AND EVERYTHING

Programming directly to the UART requires a little knowledge about this device, although to program effectively, the more you know, the better. Fortunately, numerous books and articles about the different varieties of UARTs are at your disposal (see "For Further Reference" at the end of this chapter), so little space is used here discussing how the UART works. The real goal of this chapter is to show you how to push the UART to its limits within the friendly confines of C++, although some assembly language for the timing of critical matters is given.

Unlike the video display adapters, which are mapped directly within the CPU's memory space, the UART is mapped in the CPU's I/O space. Instead of using a memory MOV instruction to get data in or out, you must use an IN or OUT instruction. The Borland C++ runtime library provides routines to perform just these operations (`inportb` and `outportb`). You can manipulate every function of the UART through these two routines.

DETERMINING THE BASE I/O ADDRESS

First, decide where within the 64K I/O space the UART is mapped. You have several ways to determine the base I/O address of a UART. The simplest is to recall that the first serial port in every system since the earliest PC is at 3F8h in the I/O address space. If you have a second serial port, locate it at 2F8h. You can hard-code these values into your programs because they work in most cases. As you might guess, this can also get you into trouble.

While the BIOS is almost worthless for serial data I/O, it does serve useful purposes. For example, every time you reboot the computer, the BIOS searches for working serial UARTs and keeps track of the base I/O addresses. This

information is stored in segment 0040h, starting at offset 0. When you boot your computer, the BIOS bootstrap code searches the well-known UART addresses looking for an active UART (presumably by testing for some particular behavior characteristic of UARTs). Most BIOSs start with 3F8h, followed by 2F8h, 3E8h, and 2E8h (PS/2s search for 3F8h, 2F8h, 3220h, and 3228h). Upon finding a working UART, it enters the base address in the BIOS data area and advances its internal pointer to the next slot. This process continues until the BIOS finds four ports (the maximum supported by the BIOS) or checks every address it knows.

Unfortunately, some BIOSs (particularly older XT BIOSs) know only about COM1 and COM2—if you install more than two serial ports, the addresses may not load correctly into the BIOS data area. Also, the exact interpretation of COM1 can differ in context. Usually, COM1 is synonymous with a 3F8h I/O address. In fact, when setting jumpers on add-in serial cards, configuring a port as COM1 means just that. But if you think of COM1 as being the first port listed in the BIOS data area, it may not be 3F8h.

Recall how the BIOS fills in port addresses. If you have a single serial port in your system and set the jumper on that card to be COM2, the first serial port the BIOS finds is at I/O address 2F8h, which then is put in the first slot of the BIOS data area. Does your program call that COM1? You might be confused, knowing you set the card to be COM2.

It's easy to see why the I/O port address of a serial port confuses both users and programmers. In addition, many existing serial cards were separated from their documentation long ago, making determining the address for which a card is set nearly impossible.

Moreover, some programs indicate they are using a particular serial port by zeroing out the address in the BIOS data area. (OS/2 1.x is one example.) For COM3 and COM4, it's impossible to tell whether a 0 address in the BIOS data area means that the port doesn't exist, that the BIOS doesn't know about ports above COM2, or that the port is in use.

HOW THE UART WORKS

The UART performs two basic tasks: taking data given by the CPU while shifting that data out the Transmit Data Line one bit at a time, and collecting

bits from the Receive Data Line and presenting them to the CPU as a whole byte. The transmitter and receiver sections of the UART each contain two data registers. This feature allows the UART to begin receiving the next character while waiting for the CPU to read the current one, and for a byte to be queued up for sending while one is being shifted out. If the software doesn't provide bytes to transmit in a timely fashion, nothing bad happens; you simply have idle periods on the line between bytes. However, if the software doesn't read data out of the UART before the next byte is complete, that data is lost forever.

Some UARTs don't store the newest byte if an overrun occurs, but most overwrite the previous one.

THE REGISTERS

As mentioned before, all programming of the UART is done through registers, which are memory locations on the UART chip itself. Most UARTs have a total of 10 registers available to software applications. Take a brief look at each one.

0 Data Register—The CPU tells the UART which byte to transmit next by writing to this register. Reading from this register, you get the last byte received by the UART.

1 Interrupt Enable—This register tells the UART which events should generate an interrupt.

2 Interrupt ID—The CPU can poll this register to tell which event, if any, caused an interrupt.

3 Line Control—This register controls how data bytes transmit in terms of stop bits and parity.

4 Modem Control—This register allows the CPU to set the RTS and DTR lines along with two general-purpose lines (OUT1 and OUT2).

5 Line Status—This register indicates if receive data is available, the transmitter is empty, or an error has occurred.

6 Modem Status—This register allows the CPU to poll the state of the CTS (Clear-To-Send) and DSR (Data-Set-Ready) lines as well as determine if any change in these lines occurred since this register was last read. These lines are typically used as handshake lines to indicate when a remote device is ready to send or receive data.

7 Scratch—This register is not found on all models of UARTs. It is available for temporary storage of values by the CPU. Be aware that some internal modems use this register for configuration purposes, and writing to this register may lock up the UART.

Rather than use up more I/O addresses, the designers of the UART decided some registers would have dual meanings. In particular, you can use the first two registers (base + 0 and base + 1) to hold the baud rate divisor. The UART takes the basic input clock (1.8432MHz on PCs), divides it by 16, and then divides it again by the divisor value to get the final baud rate. Setting the baud rate divisor to 1 gets the UART at its fastest speed. But how does the UART know which meaning to apply to data written to and read from the first two registers? The value of the high order bit in the Line Control register actually determines this. When this bit is 0, the first two registers are mapped to the Data I/O and Interrupt Enable registers, respectively; when it is 1, these registers correspond to the low and high order bytes of the baud rate divisor. Think of this as a mini EMS-style paging scheme. Future UARTs may remap some of the other registers as well. (In fact, the Intel 82510 on some AboveBoards already does that.)

PROCESSING INCOMING DATA

Now you know how to get data in and out of the UART, but how does the UART inform the CPU when data is available? It does this in two different ways; one is easy, the other is more complicated. Guess which is more efficient.

I'll start with the simplest approach: polling. Recall the Line Status register of the UART. Bit 0 indicates whether data is in the receive buffer. Reading the data from the Data register clears this bit until the next byte comes in.

Periodically checking whether this bit is set and then reading the data in the data register is a fairly simple matter. Of course, this procedure is fairly inefficient, and if you are busy doing something else, you can lose data. (At 115,200 baud a new character comes in every 87 microseconds).

This point brings us to interrupts. Writing software to handle hardware interrupts can be a reasonably straightforward procedure. Processing UART interrupts is more involved than usual, though. Let's start with a general discussion about how a hardware interrupt handler works.

Any board plugged into the PC's motherboard can generate an interrupt. The original PC architecture provides a choice of seven different interrupt lines. The Intel CPU, however, has only a single interrupt line (no doubt to reduce the number of pins required). The solution is to multiplex the seven hardware interrupt lines into this single line by an interrupt arbitrator, which takes the form of another Intel chip, the Peripheral Interrupt Controller (the PIC, or 8259). This device continuously polls the seven interrupt lines and issues an interrupt to the CPU on behalf of any device asserting an interrupt line. If more than one device requests an interrupt, the PIC uses a simple priority scheme to decide which goes first. Namely, each line receives a priority (from 0 to 7, with 0 the highest), and the one with the highest priority goes first. If an interrupt is already in progress and another one with a higher priority comes in, the new interrupt takes over. If a lower priority interrupt occurs, it waits until all higher priority interrupts finish.

How does the PIC know when an interrupt is done? This depends on cooperation from the software. All hardware interrupt service routines must issue an End-of-Interrupt (EOI) by outputting the value 20h to the base PIC register (conveniently, also 20h) prior to performing an interrupt return (IRET). How does this software interrupt handler gain control in the first place? When the PIC interrupts the CPU, it also supplies the vector number of the appropriate service routine. For IRQs 0 through 7, these vectors are 08h through 0Fh. If the software pointed to by these vectors fails to inform the PIC of an EOI, no further interrupts on this or any lower priority interrupts can occur. (If you've ever seen a mouse cursor hang while the keyboard still responds, you may have seen this in action.)

What happens if more than one device is set up to use the same interrupt? In general, on the ISA bus (the one found in most non-PS/2s), the process doesn't work well. While one device is trying to assert an interrupt, the other device

is trying not to. One device or the other may prevail, or neither may be able to assert an interrupt. Even if both devices' interrupts could get through, the PIC is set to recognize only the leading edge of an interrupt request. This feature means that the PIC sees two overlapping interrupts from different devices as a single interrupt. Because of these problems, many of the IRQs are reserved for use by one specific device (see Table 16.1). The newer buses (EISA and MCA) correct these problems, and devices can safely share IRQs.

TABLE 16.1. RESERVED IRQ ASSIGNMENTS.

IRQ	*Assignment*
IRQ0	Clock timer (18.2 times a second)
IRQ1	Keyboard
IRQ2	Available
IRQ3	COM2
IRQ4	COM1
IRQ5	Hard Disk on XT or LPT2 if installed
IRQ6	Floppy Disk
IRQ7	LPT1

As you see, the original XT didn't leave many IRQs available for add-in boards (at most one or two). To rectify this, the PC/AT adds a second 8259 PIC, extending the number of available hardware interrupt lines by seven. (The cascaded second PIC uses the IRQ2 on the first PIC.) All 286 or better PC systems have this second PIC. Unfortunately, only 16-bit bus cards can support these extra lines, and of those, few actually do. This condition is unfortunate because conflicting IRQs present one of the biggest headaches in configuring systems.

Returning to the interrupt handler software, the basic functions for any hardware interrupt handler are:

- To place the address of the interrupt service routine (ISR) into the appropriate slot of the interrupt vector table, depending on the IRQ level the supported device uses.

- To enable interrupts on the PIC for the appropriate IRQ level.
- To enable interrupt generation on the supported device.
- To issue an EOI at the completion of each interrupt.

As mentioned before, the UARTs slightly complicate this procedure. Determining which IRQ the UART is configured for is the first step when setting up UARTs for interrupts. Unfortunately, no foolproof way exists to determine this, because the IRQ setting for any particular COM port is not stored anywhere that software can read. By convention, however, COM1 and COM2 almost always are set for IRQ4 and IRQ3, respectively, which means if a COM port has a base address of 3F8h, its IRQ level is most likely four. In addition, if the base address is 2F8h, an IRQ level of three is your best guess. Although exceptions to this mapping occur, those are rare. Note that the mapping is between port addresses and IRQs, not between port numbers (which as you saw earlier, can be arbitrarily based on the ports the ROM BIOS finds).

For ports beyond COM2, things become a little more difficult. COM3 and COM4 violate the rule requiring ISA devices not to share the same IRQ. The designers of the PC tried to work around this problem by putting in a switch that essentially can disconnect each UART from the interrupt line going to the PIC. This switch takes the form of the OUT2 line of the UART (a general-purpose line for which the UART manufacturer intended no specific use). When the OUT2 bit is set in the UART's Modem Control register, any interrupts generated by the UART pass to the PIC. When the OUT2 bit is low, the PIC never sees the interrupt. So long as no two UARTs have their OUT2 lines set at the same time, no contention for the interrupt line results. This action works fine as long as you only use one port at a time, all communication software remembers to turn off the OUT2 line when they're done with it, the BIOS initially clears the bits, and only UARTs share the interrupt. In other words, this scheme doesn't work too well.

All that aside, how do you know which IRQ to use for a given UART? Your best method is to look at the base I/O address, and if it is in the 300s, assume IRQ4; otherwise, use IRQ3. It sounds crazy, but this procedure works more often than not. As a precaution, allow your program's users to specify the I/O address and the IRQ level.

Once you know which IRQ to use for a particular UART, put the address of your handler routine into the proper spot in the interrupt vector table. Then

tell the UART the type interrupts you want. The UART generates an interrupt when receiving a character and when the transmit buffer is empty. It can also generate an interrupt when any of the modem status lines (CTS and DSR) change or when a receive error occurs (such as an overrun). You do this by setting any combination of bits in the Interrupt Control register. Finally, don't forget to turn on the OUT2 bit so the interrupt makes it to the PIC.

Table 16.2 summarizes the meanings assigned to the various bits in each of the UART's registers. These bit definitions apply to all 8250 compatibles. Some of the reserved bits (indicated by a constant 0) are used in some of the more advanced UARTs. For example, the 16650 uses some of these bits to control the internal 16-byte FIFO queue.

TABLE 16.2. UART REGISTERS.

	bit 7	*bit 6*	*bit 5*	*bit 4*
0 Data I/O	data bit 7	data bit 6	data bit 5	data bit 4
1 Interrupt Enable	0	0	0	0
2 Interrupt ID	0	0	0	0
3 Line Control	Aux Regs	Set Break	Stick Parity	Even Parity
4 Modem Control	0	0	0	Loop back
5 Line Status	0	Transmit Hold Empty	Transmit Shift Empty	Received Break
6 Modem Status	Carrier Detect	Ring Detect	DSR	CTS
	bit 3	*bit 2*	*bit 1*	*bit 0*
0 Data I/O	data bit 3	data bit 2	data bit 1	data bit 0
1 Interrupt Enable	Modem Status Change	Error on Received Data	Transmit Idle	Data Received

continues

TABLE 16.2. CONTINUED

	bit 3	*bit 2*	*bit 1*	*bit 0*
2 Interrupt ID	0	Interrupt	ID	No Interrupt Pending
3 Line Control	Parity Enable	Stop Bits	Word Length	
4 Modem Control	Out2	Out1	RTS	DTR
5 Line Status	Frame Error	Parity Error	Overrun	Data Ready
6 Modem Status	Carrier change	Ring change	DSR change	CTS change

The transmit interrupt doesn't appear to work reliably on all UARTs. It is, therefore, a good idea to have a plan B in place so communications don't come to a halt if a transmit interrupt is missed.

SHARING IRQS WITH OTHER DEVICES

As you've seen, sharing IRQs among devices doesn't work very well in the ISA environment. This problem is, hopefully, temporary as the new bus architectures become commonplace. To be safe, you may want to write your interrupt handlers to share the interrupt with other handlers. This simple process involves storing away the address of the current interrupt handler and then verifying that the UART you're monitoring actually generated any interrupts

you received (by examining the interrupt ID register). If you get an interrupt but your UART doesn't show any interrupts pending, pass it along to the previous interrupt handler.

> Some BIOSs set up a default interrupt handler that masks interrupts at the PIC if an interrupt makes it all the way down the chain to the BIOS. For this reason, you may want to reenable interrupts on the PIC after chaining.

If you have written or are writing TSRs (see Chapter 15, "How to Write a TSR"), you are aware of one problem with chaining interrupts: you can unhook yourself from the chain only if you are at the front of the chain. IBM proposes a solution to this in its BIOS technical reference manual for the PS/2 (see "For Further Reference" at the end of this chapter).

If every handler in the chain follows this suggestion, applications may insert and remove themselves from the chain at will. To be good software citizens, we all should strive to cooperate by following these ground rules. (Because few commercial applications do this, you can help spread the word.)

REDUCING INTERRUPT LATENCY PROBLEMS

That's all there is to high-speed serial communications—bypass the BIOS and set up an interrupt handler so you don't continuously poll, right? Not quite. Remember, at 115,200 baud, bytes come in one every 87 microseconds. Processing an interrupt is one of the slowest operations on any CPU. You need to save the current state of the machine and do a long jump to the interrupt handler, which flushes any prefetch queue used by the CPU to instruction execution speed. On a slow XT-class machine (8088/4.77 MHz), this operation leaves little of the 87 microsecond window for any real work. Also, serial interrupts are in the middle of the priority list (closer to the end if you count

the second PIC) and must wait for higher priority interrupts to complete. (You probably can count on losing at least one character 18.2 times every second.) You can see, therefore, that interrupts don't provide the total solution.

You need a hybrid approach. Use the interrupt to tell you when to start polling the UART. This approach works especially well if the data is sent in bursts (called block-oriented communications). To be safe, observe the standard handshake usage of the modem control lines to determine when a remote machine is ready for data. If you keep your blocks relatively small (about 256 to 512 bytes), you can turn off interrupts inside the polling loop to guarantee no loss of data.

A HIGH-SPEED SERIAL INTERFACE CLASS LIBRARY

Now you have enough background to implement a high-speed serial communications class library. For this example, choose a simple byte-count-oriented protocol called Digital's Data Communication Message Protocol (DDCMP). While this protocol is not too common, it is fairly well designed. Don't worry too much about the specifics of the protocol.

Listing 16.1 is the header file with all UART-related constants. Lines 14–26 map IRQs to the appropriate vector in the interrupt table. Lines 35–111 define mnemonics for the various UART registers and bit values.

LISTING 16.1. GENERAL UART INFORMATION.

```
 1  // ************************************************************************
 2  //                    Secrets of the Borland C++ Masters
 3  //
 4  //   Module: uarts.h
 5  //
 6  //   Purpose: General UART information
 7  //
 8  //   Author: Gordon G. Free
 9  //
10  // ************************************************************************
11  #ifndef  UARTS_H
12  #define  UARTS_H
13
```

```
14  #define  IRQ0           0x08
15  #define  IRQ2           0x0A
16  #define  IRQ3           0x0B
17  #define  IRQ4           0x0C
18  #define  IRQ5           0x0D
19  #define  IRQ7           0x0F
20  #define  IRQ9           0x71
21  #define  IRQ10          0x72
22  #define  IRQ11          0x73
23  #define  IRQ12          0x74
24  #define  IRQ13          0x75
25  #define  IRQ14          0x76
26  #define  IRQ15          0x77
27
28  #define  ENAB_IRQ4      0xEF
29  #define  ENAB_IRQ3      0xF7
30
31
32  // -------------------------------
33  //          UART Registers
34  // -------------------------------
35  #define  DATA_IN        0               // read only  (DLAB=0)
36  #define  DATA_OUT       0               // write only (DLAB=0)
37
38  #define  BAUD_RATE_LO   0               // read/write (DLAB=1)
39  #define  BAUD_RATE_HI   1               // read/write (DLAB=1)
40
41  #define  INTR_ENABLE    1               // read/write (DLAB=0)
42  #define     INTR_ON_RCV       0x01
43  #define     INTR_ON_XMIT      0x02
44  #define     INTR_ON_ERR       0x04
45  #define     INTR_ON_CHANGE    0x08
46
47  #define  INTR_ID        2               // read only  (DLAB=0)
48  #define     INTR_NOTPENDING   0x01
49  #define     INTR_ID_MASK      0x0E
50  #define     INTR_ERR          0x06
51  #define     INTR_RCV          0x04
52  #define     INTR_XMIT         0x02
53  #define     INTR_CHANGE       0x00
54  #define     FIFO_ENAB1        0x40        // 16550+ only
55  #define     FIFO_ENAB2        0x80        // 16550+ only
56  #define     FIFO_16550        0x80        // 16550+ only
57  #define     FIFO_16550AF      0xC0        // 16550+ only
58
59  #define  FIFO_CNTRL     2               // write only (DLAB=0)
60  #define     FIFO_ENABLE       0x01
61  #define     RCV_FIFO_ENAB     0x02
```

continues

LISTING 16.1. CONTINUED

```
 62  #define     XMIT_FIFO_ENAB    0x04
 63  #define     DMA_ENABLE        0x08        // 16550+ only
 64  #define     RCV_BYTE1         0x00        // 16550+ only
 65  #define     RCV_BYTE4         0x40        // 16550+ only
 66  #define     RCV_BYTE8         0x80        // 16550+ only
 67  #define     RCV_BYTE14        0xC0        // 16550+ only
 68
 69  #define  LINE_CNTRL     3               // read/write
 70  #define     DATA5             0x00
 71  #define     DATA6             0x01
 72  #define     DATA7             0x02
 73  #define     DATA8             0x03
 74  #define     ONE_STOP_BIT      0x00
 75  #define     TWO_STOP_BIT      0x04
 76  #define     NO_PARITY         0x00
 77  #define     ODD_PARITY        0x08
 78  #define     EVEN_PARITY       0x18
 79  #define     STICK_PARITY      0x20
 80  #define     SEND_BREAK        0x40
 81  #define     OTHER_REGS        0x80        // DLAB
 82
 83  #define  MODEM_CNTRL    4               // read/write
 84  #define     DTR               0x01
 85  #define     RTS               0x02
 86  #define     OUT1              0x04
 87  #define     OUT2              0x08
 88  #define     LOOPBACK          0x10
 89
 90  #define  LINE_STATUS    5               // read/write
 91  #define     DATA_RCV          0x01
 92  #define     OVER_RUN          0x02
 93  #define     PARITY_ERR        0x04
 94  #define     FRAME_ERR         0x08
 95  #define     BREAK             0x10
 96  #define     XMIT_HOLD_EMPTY   0x20
 97  #define     XMIT_SHIFT_EMPTY  0x40
 98
 99  #define  MODEM_STATUS   6               // read/write
100  #define     CTS_CHANGED       0x01
101  #define     DSR_CHANGED       0x02
102  #define     RI_CHANGED        0x04
103  #define     DCD_CHANGED       0x08
104  #define     RLSD_CHANGED      0x08
105  #define     CTS               0x10
106  #define     DSR               0x20
107  #define     RI                0x40
108  #define     DCD               0x80
```

```
109  #define     RLSD              0x80
110
111  #define  SCRATCH_REG    7                // read/write (8250A+)
112
113  #define  INTRCNTRL1     0x20
114  #define  INTRCNTRL2     0xA0
115  #define     EOI         0x20
116  #define     SPECIFIC    0x40
117
118  #define MAXBAUDRATE  115200L
119  #define NULL_PORT_ADDR  0
120  #endif
121
```

Listing 16.2, SERCLASS.H, describes the `comm_channel` class you use as a base for your DDCMP class. Lines 14–23 define the shared interrupt header you use to allow chaining of interrupt handlers. You can examine it in more detail when you get to its actual code. Lines 25–30 declare all the instance data needed for serial communications over a given port, including an instance of the shared interrupt header which fills with executable code prior to hooking any interrupts (a bit unconventional, but allows for a nearly unlimited number of interrupt handlers).

LISTING 16.2. IMPLEMENTING A SERIAL I/O INTERFACE CLASS.

```
 1  // ************************************************************************
 2  //                    Secrets of the Borland C++ Masters
 3  //
 4  //   Module: serclass.h
 5  //
 6  //   Purpose: Implement serial I/O interface class.
 7  //
 8  //  Author: Gordon G. Free
 9  //
10  // ************************************************************************
11  #ifndef SERCLASS_H
12  #define SERCLASS_H
13
14  // Shared interrupt handler stub code
15  typedef struct SHARED_INT_STUB_S {
16     unsigned short entry_jmp;
17     struct SHARED_INT_STUB_S far *prevHandler;
18     unsigned short signature;
```

continues

LISTING 16.2. CONTINUED

```
19     unsigned char  chain_flags;
20     unsigned short reset_jmp;
21     unsigned char  reserved[7];
22     unsigned char  entry_stub[20];
23  } SHARED_INT_STUB_T, far *PSHARED_INT_STUB_T;
24
25  class comm_channel {
26     protected:
27        int port_id;                  // port number 0=COM1, 1=COM2, etc.
28        unsigned short port_addr;     // I/O address of UART
29        int port_irq;                 // irq of UART
30        SHARED_INT_STUB_T intr_stub;  // ISR stub code
31
32     public:
33        comm_channel(int comm_id);
34        unsigned short GetPortAddr() {return port_addr;};
35        unsigned long GetBaudRate();
36        void SetBaudRate(unsigned long NewBaudRate);
37        void TransmitSerialByte(unsigned char);
38        void WaitForTransmitEmpty();
39        int  WaitForRTS(unsigned short);
40        void TransmitDataBurst(unsigned char*, unsigned short);
41        ~comm_channel();
42  };
43
44  #define  COM1           0
45  #define  COM2           1
46  #define  COM3           2
47  #define  COM4           3
48
49  #endif
```

Lines 33–41 define the methods available, including functions to query and change the baud rate and to send single and blocks of data. Notably absent is a function for receiving data. For performance reasons, `DDCMP_channel`, your derived class, fills in this gap.

Listing 16.3, DDCMP.H, describes the DDCMP class. Lines 26–37 define the standard DDCMP header which precedes all data going over the link. Don't concern yourself with many of the fields, because your simple implementation won't use them. Perhaps the most important field is the `Count` field. It tells the receiving machine how many bytes to expect in the rest of the block, allowing blocks of arbitrary size and classifying the protocol as a *Byte Count* protocol (as opposed to a fixed block or framed protocol).

LISTING 16.3. DIGITAL'S DATA COMMUNICATIONS MESSAGE PROTOCOL CLASS.

```
// ********************************************************************
//                    Secrets of the Borland C++ Masters
//
//   Module: ddcmp.h
//
//   Purpose: Digital's Data Communications Message Protocol class.
//      This module implements a subset of DDCMP.  It can be used to
//      send and receive information frames.
//
//  Author: Gordon G. Free
//
// ********************************************************************
#ifndef DDCMP_H
#define DDCMP_H
#include "serclass.h"

#define SYNC_BYTE             22
#define MAX_WAIT_TIME         1000
#define INFORMATION_FRAME     1
#define SENT_OK               0
#define SEND_TIMEDOUT        -1
#define FALSE                 0
#define TRUE                  !FALSE

// The DDCMP Frame Header Structure.
typedef struct {
   unsigned char  Sync;          // second Sync character
   unsigned char  FrameClass;    // type of frame
   struct {
      unsigned Count:14;         // size of info field
      unsigned Flags:2;          // misc frame flags
   } ByteFrame;
   unsigned char  ReceiveCount;  // ack of bytes sent
   unsigned char  SendCount;     // number of bytes in info field
   unsigned char  Address;       // destination address
   unsigned short CRC;
} DDCMP_HDR_T;

#define DDCMP_HDR_SIZE  sizeof(DDCMP_HDR_T)

// Derive DDCMP class from basic serial I/O class

class DDCMP_channel : comm_channel {
protected:
      void *pReceiveFrameHdr;       // ptr to input frame buffer
      void *pReceiveFrameData;      // ptr to input data buffer
```

continues

LISTING 16.3. CONTINUED

```
47        int  DataBufferFilled;        // flag to indicate data in buffer
48        unsigned char LastSentFrame;  // sequence id
49        unsigned char IntrMaskSave;   // PIC interrupt mask for restore
50        int far InterruptHandler();   // interrupt service routine
51
52  public:
53        DDCMP_channel(int port_id);
54        int  SendInformationFrame(void*, unsigned short);
55        void ReceiveInformationFrame(void *);
56        int IsDataAvailable() { return DataBufferFilled; };
57        ~DDCMP_channel();
58  };
59
60  #endif
```

Lines 45–49 declare the instance data you need for each channel of communications, which primarily determines where to store incoming data. Notice that `DDCMP_channel` derives from the `comm_channel` class (line 43).

Lines 53–57 define the methods available to a `DDCMP_channel` object: `SendInformationFrame` and `ReceiveInformationFrame`. Because receiving data is asynchronous in nature (that is, you don't know when the data will come in, so you don't want to wait around for it), `ReceiveInformationFrame` just makes a buffer available for receiving data. Use the `IsDataAvailable` method to determine when reception of data is complete. In addition, all the methods of the base `comm_channel` class are available.

Listing 16.4, SERCLASS.CPP, contains the actual code for the base serial communication class, `comm_channel`. Lines 22–39 comprise the class contructor which determines the proper I/O address and IRQ level for the specified COM port. If all goes well, the port is marked as in use by zeroing out its address in the BIOS data area, thereby preventing another instance of `comm_channel` over the same COM port.

LISTING 16.4. IMPLEMENTING A SERIAL I/O INTERFACE CLASS.

```
1  // ************************************************************************
2  //                    Secrets of the Borland C++ Masters
3  //
4  //   Module: serclass.cpp
```

```
 5  //
 6  //   Purpose: Implement serial I/O interface class.
 7  //
 8  //  Author: Gordon G. Free
 9  //
10  // ********************************************************************
11  #include <dos.h>
12  #include "serclass.h"
13  #include "shareint.h"
14  #include "uarts.h"
15
16  unsigned short DetermineBIOSComAddr (int comm_id);
17  unsigned short SetBIOSComAddr (int comm_id, unsigned short new_addr);
18
19  // ------------------------------------------------------------------
20  // comm_channel constructor
21  // ------------------------------------------------------------------
22  comm_channel::comm_channel(int comm_id)
23  {
24     // verify that requested port is within range
25     if ((comm_id >= COM1) && (comm_id <= COM4)) {
26
27        // Initialize instance data
28        port_id = comm_id;
29        port_addr = DetermineBIOSComAddr(comm_id);
30
31        // If channel was assigned, zero out BIOS data area
32        if (port_addr != NULL_PORT_ADDR) {
33           // Assume that all UARTs with I/O addresses in the 300's are IRQ4
34           port_irq = ((port_addr&0xFF00) == 0x300) ? 4 : 3;
35           SetBaudRate(MAXBAUDRATE);
36           SetBIOSComAddr(comm_id, 0);
37        }
38     }
39  }
40
41  unsigned short DetermineBIOSComAddr (int comm_id)
42  {
43     unsigned short far *bios_comm_addr;
44
45     // Get I/O address of UART from BIOS data area starting at 40:0
46     FP_SEG(bios_comm_addr) = 0x40;
47     FP_OFF(bios_comm_addr) = comm_id*2;
48
49     return(*bios_comm_addr);
50
51  }
52
```

continues

LISTING 16.4. CONTINUED

```
// ---------------------------------------------------------------
// SetBIOSComAddr - set BIOS Ram for a particular port's I/O address.
// ---------------------------------------------------------------
unsigned short SetBIOSComAddr (
  int comm_id,                          // 0=COM1,1=COM2,2=COM3,3=COM4
  unsigned short new_addr               // I/O address to set
)
{
   unsigned short far *bios_comm_addr;  // ptr to BIOS Ram
   unsigned short original_addr;        // current address in BIOS RAM

   // Set I/O address of UART in BIOS data area starting at 40:0
   FP_SEG(bios_comm_addr) = 0x40;
   FP_OFF(bios_comm_addr) = comm_id*2;

   original_addr = *bios_comm_addr;
   *bios_comm_addr = new_addr;

   return(original_addr);
}

// ---------------------------------------------------------------
// GetBaudRate - determine current baud rate setting for comm channel
// ---------------------------------------------------------------
unsigned long comm_channel::GetBaudRate()
{
   unsigned short BaudDivisor;          // divisor value read from UART
   unsigned long  BaudRate;             // converted baud rate value
   unsigned short InterruptMask;        // temp storage for intr enable

   // Don't want any interrupts while using alternative registers
   InterruptMask = inportb(port_addr+INTR_ENABLE);
   outportb(port_addr+INTR_ENABLE, 0);

   // Read baud rate out of alternate register set
   outportb(port_addr+LINE_CNTRL
   , inportb(port_addr+LINE_CNTRL) ¦ OTHER_REGS);
   BaudDivisor = inport(port_addr+BAUD_RATE_LO);
   outportb(port_addr+LINE_CNTRL
   , inportb(port_addr+LINE_CNTRL) & ~OTHER_REGS);

   outportb(port_addr+INTR_ENABLE, InterruptMask);

   if (BaudDivisor > 0)
      BaudRate = MAXBAUDRATE / BaudDivisor;
   else
      BaudRate = 0L;

```

```
101     return(BaudRate);
102  }
103
104  // ------------------------------------------------------------------
105  // SetBaudRate - program UART for comm channel to desired baud rate
106  // ------------------------------------------------------------------
107  void comm_channel::SetBaudRate(unsigned long NewBaudRate)
108  {
109     unsigned long  BaudRate;             // desired baud rate
110     unsigned short BaudDivisor;          // converted divisor value
111     unsigned short InterruptMask;        // temp storage for intr enable
112
113     // verify that requested baud rate is within range
114     if ((NewBaudRate > 0) && (NewBaudRate <= MAXBAUDRATE)) {
115        BaudDivisor = MAXBAUDRATE / NewBaudRate;
116
117        // Don't want any interrupts while using alternative registers
118        InterruptMask = inportb(port_addr+INTR_ENABLE);
119        outportb(port_addr+INTR_ENABLE, 0);
120
121        WaitForTransmitEmpty();
122
123        // set baud rate in UART
124        outportb(port_addr+LINE_CNTRL,
                    DATA8+ONE_STOP_BIT+NO_PARITY+OTHER_REGS);
125        outport(port_addr+BAUD_RATE_LO, BaudDivisor);
126        outportb(port_addr+LINE_CNTRL, DATA8+ONE_STOP_BIT+NO_PARITY);
127
128        outportb(port_addr+INTR_ENABLE, InterruptMask);
129     }
130  }
131
132  // ------------------------------------------------------------------
133  // WaitForTransmitEmpty - poll UART until all data has been sent
134  // ------------------------------------------------------------------
135  void comm_channel::WaitForTransmitEmpty()
136  {
137     unsigned char xmit_status;
138
139     do {
140       xmit_status = inportb(port_addr+LINE_STATUS)
141          & (XMIT_HOLD_EMPTY+XMIT_SHIFT_EMPTY);
142     } while (xmit_status != (XMIT_HOLD_EMPTY+XMIT_SHIFT_EMPTY));
143  }
144
145  // ------------------------------------------------------------------
146  // WaitForTransmitEmpty - poll UART until all data has been sent
147  // ------------------------------------------------------------------
```

continues

LISTING 16.4. CONTINUED

```
148  int  comm_channel::WaitForRTS(unsigned short count)
149  {
150
151     unsigned char cts_status;
152
153     do {
154       cts_status = inportb(port_addr+MODEM_STATUS) & CTS;
155     } while (count-- && cts_status != CTS);
156
157     return (count);
158  }
159
160  // ------------------------------------------------------------------
161  // TransmitSerialByte - send data out serial channel
162  // ------------------------------------------------------------------
163  void comm_channel::TransmitSerialByte(unsigned char ByteOut)
164  {
165     // Wait to be sure UART is ready [NOTE: a broken UART could
166     // cause an infinite loop!]
167     while (!(inportb(port_addr+LINE_STATUS) & XMIT_HOLD_EMPTY));
168     outportb(port_addr+DATA_OUT, ByteOut);
169  }
170
171  // ------------------------------------------------------------------
172  // TransmitDataBurst - send block of data out serial channel
173  // ------------------------------------------------------------------
174  void comm_channel::TransmitDataBurst(
175     unsigned char *DataBuffer,
176     unsigned short Length
177  )
178  {
179     register int i;
180
181     for (i=0; i<Length; i++) {
182        while (!(inportb(port_addr+LINE_STATUS) & XMIT_HOLD_EMPTY));
183        outportb(port_addr+DATA_IN, *(DataBuffer++));
184     }
185
186  }
187
188  // ------------------------------------------------------------------
189  // comm_channel destructor
190  // ------------------------------------------------------------------
191  comm_channel::~comm_channel()
192  {
193     // Restore original I/O address in BIOS data area
```

```
194     SetBIOSComAddr(port_id, port_addr);
195  }
196
```

Lines 77–102 read the baud rate divisor from the UART and convert it to an actual baud rate. Lines 107–130 reverse the process by taking a baud rate, converting it to a divisor, and setting the UART to that new rate. Notice you send all data before changing baud rates (line 121).

Lines 135–143 comprise the `WaitForTransmitEmpty` method which polls the UART until all data is transmitted.

Lines 148–158 contain the `WaitForRTS` method which loops for a specified number of passes waiting for the remote machine to assert its RTS line (tied to your CTS line), indicating that the remote software is ready for a block transfer.

Lines 163–169 control transmitting a single byte out the UART, while lines 174–186 repeat the process for an entire block of data.

Lines 191–195 include the class destructor which restores the UART address to the BIOS data area.

Listing 16.5, DDCMP.CPP, contains the code for the derived byte count block oriented serial communication class, `DDCMP_channel`. Its contructor (lines 30–50) is considerably more involved than the one for `comm_channel`. After the instance data initializes and you verify that the parent class is successfully created (line 41), prepare to hook a handler into the appropriate interrupt chain. First create an instance of the interrupt header which can link into the list of interrupt handlers. This header allows you to execute a C++ method as an interrupt handler (complete with the proper `this` instance data). Once you have the interrupt header, hook into the interrupt chain (line 43) and then enable interrupts on the UART and the PIC (lines 44–47). You now are ready to receive UART interrupts.

LISTING 16.5. DIGITAL'S DATA COMMUNICATIONS MESSAGE PROTOCOL CLASS.

```
1  // ************************************************************************
2  //                    Secrets of the Borland C++ Masters
3  //
4  //   Module: ddcmp.cpp
5  //
```

continues

LISTING 16.5. CONTINUED

```
 6  //   Purpose: Digital's Data Communications Message Protocol class.
 7  //      This module implements a subset of DDCMP.  It can be used to
 8  //      send and receive information frames.
 9  //
10  //  Author: Gordon G. Free
11  //
12  // ********************************************************************
13  #include <dos.h>
14  #include "ddcmp.h"
15  #include "shareint.h"
16  #include "uarts.h"
17
18  // Declare external function to set up a shared interrupt stub.
19  extern "C" void cdecl far CopyTemplate(void far *, void *
20          , int (far DDCMP_channel::*)());
21
22  unsigned short CalculateCRC(
23     unsigned char   *DataBuffer,
24     unsigned short  Length
25  );
26
27  // --------------------------------------------------------------------
28  // DDCMP_channel constructor
29  // --------------------------------------------------------------------
30  DDCMP_channel::DDCMP_channel (
31     int port_id                  // 0=COM1, 1=COM2, 2=COM3, 3=COM4
32  ) : comm_channel(port_id)
33  {
34
35     // Initialize instance data
36     LastSentFrame = 0;
37     pReceiveFrameHdr = pReceiveFrameData = 0;
38     DataBufferFilled = FALSE;
39
40     // check if base class constructor succeeded
41     if (port_addr != NULL_PORT_ADDR) {
42        CopyTemplate(&intr_stub, this, &(DDCMP_channel::InterruptHandler));
43        HookSharedInterrupt(IRQ0+port_irq, &intr_stub, SPECIFIC+prt_irq);
44        outportb(port_addr+INTR_ENABLE, INTR_ON_RCV);
45        outportb(port_addr+LINE_STATUS, DSR+OUT2);
46        IntrMaskSave = inportb(INTRCNTRL1+1);
47        outportb(INTRCNTRL1+1, IntrMaskSave & ~(0x01<<port_irq));
48     }
49
50  }
51
```

```
52  // ------------------------------------------------------------------
53  // SendInformationFrame - Transmit a DDCMP information frame along with
54  //    CRCs.
55  // ------------------------------------------------------------------
56  int DDCMP_channel::SendInformationFrame (
57     void *DataBuffer,                    // Data to be sent as information
58     unsigned short numDataBytes          // number of bytes of information
59  )
60  {
61     DDCMP_HDR_T      frameHdr;           // Allocate a frame header
62     unsigned short   CRC1, CRC2;         // two different CRCs are sent
63     int rc=SENT_OK;                      // return code
64
65     // Initialize frame header
66     frameHdr.Sync = SYNC_BYTE;
67     frameHdr.ByteFrame.Count = numDataBytes;
68     frameHdr.ByteFrame.Flags = 0;
69     frameHdr.FrameClass = INFORMATION_FRAME;
70     frameHdr.ReceiveCount = 0;
71     frameHdr.SendCount = ++LastSentFrame;
72
73     // Calculate two separate CRC values.
74     frameHdr.CRC = CalculateCRC((unsigned char *)&frameHdr,
                                   DDCMP_HDR_SIZE-2);
75     CRC2 = CalculateCRC((unsigned char *)DataBuffer, numDataBytes);
76
77     // Send first Sync byte
78     TransmitSerialByte(SYNC_BYTE);
79
80     // Wait for UART to complete the send before expecting any
81     // response from remote.
82     WaitForTransmitEmpty();
83
84     // Turn off interrupts and start sending data ...
85     disable();
86
87     // as soon as remote is ready!
88     if (WaitForRTS(MAX_WAIT_TIME)) {
89        TransmitDataBurst((unsigned char *)&frameHdr, DDCMP_HDR_SIZE);
90        TransmitDataBurst((unsigned char *)DataBuffer, numDataBytes);
91        TransmitSerialByte(CRC2 & 0xFF);
92        TransmitSerialByte((CRC2 & 0xFF00) >> 8);
93
94     // no response from remote, tell caller & reset frame count
95     } else {
96         rc = SEND_TIMEDOUT;
97         —LastSentFrame;
98     }
99
```

continues

LISTING 16.5. CONTINUED

```
100     enable();
101     return(rc);
102  }
103
104  // ------------------------------------------------------------------
105  // ReceiveInformationFrame - Setup to receive an information frame.
106  //   Caller should poll IsDataAvailable to determine completion.
107  //
108  //   NOTE: we assume that caller provides a buffer big enough to hold
109  //      the largest buffer to be sent!
110  // ------------------------------------------------------------------
111  void DDCMP_channel::ReceiveInformationFrame(
112     void *DataBuffer
113  )
114  {
115     pReceiveFrameData = DataBuffer;
116     DataBufferFilled = FALSE;
117  }
118
119  // ------------------------------------------------------------------
120  // InterruptHandler - Process incoming receive interrupt.
121  // ------------------------------------------------------------------
122  int far DDCMP_channel::InterruptHandler()
123  {
124     void far* FrameBuffer;       // local pointer to frame buffer
125     void far* DataBuffer;        // local pointer to data buffer
126     unsigned short FrameSize;    // number of bytes in frame buffer
127     unsigned short CRC;          // storage for CRC value
128
129
130     // Check to be sure that UART really has an interrupt pending
131     if ((inportb(port_addr+INTR_ID)&INTR_ID_MASK) == INTR_NOTPENDING)
132       return FALSE;  // tell interrupt stub to chain
133
134     // Check to be sure that data is available from UART
135     if (!(inportb(port_addr+LINE_STATUS) & DATA_RCV))
136       goto Finish_up_and_leave;
137
138     // If first byte is not a Sync character then we are out of sync!
139     if (inportb(port_addr+DATA_IN) != SYNC_BYTE)
140       goto Finish_up_and_leave;
141
142     // Be sure that we don't overwrite data in our buffer and that
143     // the pointers are to valid memory
144     if ((DataBufferFilled) ¦¦ (pReceiveFrameHdr == 0)
145     ¦¦ (pReceiveFrameData == 0))
146       goto Finish_up_and_leave;
```

```
147
148     // initialize local storage used by inline asm.
149     FrameBuffer = pReceiveFrameHdr;
150     DataBuffer = pReceiveFrameData;
151     FrameSize = DDCMP_HDR_SIZE;
152
153     // Turn off UART interrupt generation
154     outportb(port_addr+INTR_ENABLE, 0);
155
156     // We held off as long as we could... but we gotta go to
157     // assembly language for true performance.  Sorry.
158     asm {
159        mov  dx, [this.port_addr]         // get base I/O address
160
161        les  di, [FrameBuffer]            // es:di -> frame buffer
162        mov  si, [FrameSize]              // si = size of frame
163
164        add  dx, MODEM_CNTRL              // signal remote to send burst
165        mov  al, CTS+DSR                  // by setting CTS
166        out  dx, al
167        add  dx, DATA_IN-MODEM_CNTRL
168     }
169
170  NextByte1:
171     asm {
172        add  dx, LINE_STATUS-DATA_IN      // dx = base register
173        mov  cx, MAX_WAIT_TIME
174     }
175
176  Wait_For_Data1:
177     asm {
178        in   al, dx                       // wait for UART to say there is
179        and  al, DATA_RCV                 // a new byte
180        jnz  Got_Data1
181        loop Wait_For_Data1               // can't wait forever!
182        jmp  Timed_Out
183     }
184  Got_Data1:
185     asm {
186        add  dx, DATA_IN-LINE_STATUS      // read the new byte and store it
187        in   al, dx
188        stosb
189        dec  si                           // continue for remaining bytes
190        jnz  NextByte1
191        les  di, [DataBuffer]             // move on to information data
192        mov  si, [FrameBuffer.ByteFrame]  // get frame length out of header
193        and  si, 3FFFh
194     }
```

continues

LISTING 16.5. CONTINUED

```
195
196  // NOTE: we duplicate above code for sake of speed.
197
198  NextByte2:
199     asm {
200        add  dx, LINE_STATUS-DATA_IN      // dx = base register
201        mov  cx, MAX_WAIT_TIME
202     }
203
204  Wait_For_Data2:
205     asm {
206        in   al, dx                       // wait for UART to say there is
207        and  al, DATA_RCV                 // a new byte
208        jnz  Got_Data2
209        loop Wait_For_Data2               // can't wait forever!
210        jmp  Timed_Out
211     }
212  Got_Data2:
213     asm {
214        add  dx, DATA_IN-LINE_STATUS      // read the new byte and store it
215        in   al, dx
216        stosb
217        dec  si                           // continue for remaining bytes
218        jnz  NextByte2
219     }
220
221  Wait_For_CRC1:
222     asm {
223        in   al, dx                       // wait for UART to say there is
224        and  al, DATA_RCV                 // a new byte
225        jnz  Got_CRC1
226        loop Wait_For_CRC1               // can't wait forever!
227        jmp  Timed_Out
228     }
229  Got_CRC1:
230     asm {
231        add  dx, DATA_IN-LINE_STATUS      // read the new byte and store it
232        in   al, dx
233        mov  ah, al
234     }
235
236  Wait_For_CRC2:
237     asm {
238        in   al, dx                       // wait for UART to say there is
239        and  al, DATA_RCV                 // a new byte
240        jnz  Got_CRC2
241        loop Wait_For_CRC2               // can't wait forever!
```

```
242         jmp  Timed_Out
243      }
244   Got_CRC2:
245      asm {
246         add  dx, DATA_IN-LINE_STATUS      // read the new byte and store it
247         in   al, dx
248         xchg al, ah
249         mov  [CRC], ax
250      }
251
252     // You'd probably want to verify the CRC at this point!
253
254     DataBufferFilled = TRUE;
255
256   Timed_Out:
257   Finish_up_and_leave:
258
259     // Re-enable UART interrupts
260     outportb(port_addr+MODEM_CNTRL, DSR+OUT2);
261     outportb(port_addr+INTR_ENABLE, INTR_ON_RCV);
262
263     // Tell PIC we are done
264     disable();
265     outportb(INTRCNTRL1, EOI);
266     return TRUE;
267
268   }
269
270   // ------------------------------------------------------------------
271   // DDCMP_channel destructor
272   // ------------------------------------------------------------------
273   DDCMP_channel::~DDCMP_channel()
274   {
275      UnhookSharedInterrupt(IRQ0+port_irq, &intr_stub);
276      outportb(port_addr+INTR_ENABLE, 0);
277      outportb(port_addr+LINE_STATUS, 0);
278      outportb(INTRCNTRL1+1, IntrMaskSave);
279   }
280
281   // ------------------------------------------------------------------
282   // CalculateCRC - calculate the CCITT-16 CRC for given buffer.
283   //   It's all magic to me.
284   // ------------------------------------------------------------------
285   unsigned short CalculateCRC(
286      unsigned char   *DataBuffer,
287      unsigned short  Length
288   )
289   {
```

continues

LISTING 16.5. CONTINUED

```
290     int i;
291     unsigned short CRC = 0xFFFF;
292     unsigned char curByte;
293
294     // Loop through each byte in buffer
295     for (i=0; i<Length; i++) {
296          curByte = *(DataBuffer++);
297
298          // Close eyes, wave hands...
299          asm {
300             push    bx
301             mov     bx,[CRC]
302             mov     al,[curByte]
303             xor     al,bl
304             mov     bl,al
305             shl     al,1
306             shl     al,1
307             shl     al,1
308             shl     al,1
309             xor     bl,al
310             xchg    bh,bl
311             mov     al,bh
312             mov     ah,bh
313             shr     ah,1
314             shr     ah,1
315             shr     ah,1
316             shr     ah,1
317             xor     bl,ah
318             sub     ah,ah
319             shl     ax,1
320             shl     ax,1
321             shl     ax,1
322             xor     bx,ax
323             mov     [CRC],bx
324             pop     bx
325          }
326     }
327     return(CRC);
328 }
```

Lines 56–102 implement the frame transmission method. You create a temporary DDCMP header structure (line 61) and initialize it for the particular frame you plan to send. You then calculate the two CRC values (lines 74–75) so the remote can verify correct reception.

DDCMP transmissions always start with a SYNC character, which you activate by invoking the base classes `TransmitSerialByte` method (line 78). Now you must wait for the go-ahead from the remote (signaled by your CTS going high, which you don't expect to occur until the SYNC byte is fully transmitted) before safely blasting the data across. While that is being done, you can be calling the `WaitForTransmitEmpty` method (line 82). Once you have the go-ahead from the remote (line 88), blast the header, data, and final CRC across (lines 89–92).

The `ReceiveInformationFrame` method (lines 111–117) stores a pointer to the data buffer and marks the buffer as empty. The application must poll the `IsDataAvailable` method to determine when data arrives.

Lines 122–268 represent the real meat of the `DDCMP_channel` class, namely, the actual interrupt handler code. The shared interrupt header (which you set up in the constructor section) calls this method when a hardware interrupt occurs. This interrupt header also passes `DDCMP_channel::InterruptHandler` the appropriate pointer to its instance data, which allows multiple instances of `DDCMP_channel` to share the same IRQ. Before you start receiving the data, make certain that your UART is the cause for the interrupt (lines 131–136), that the data is the expected SYNC byte (lines 139–140), and finally, that you have empty buffers to put the data into (lines 144–146).

At line 154, turn off the UART interrupts to avoid generating spurious interrupts as the data is blasting in. Now switch to assembly language to read the UART data efficiently as it becomes available. Then store it into memory. You sacrifice size for speed and simplicity by duplicating the UART polling loops (lines 176–183) and UART data reads (lines 184–190).

Once you receive all the frame header data, use the count field to determine how many bytes to expect (lines 191–192). Continue reading in the data bytes and finish with the final CRC. When all the data is in, flag it as received (line 254) and set up the UART for the next interrupt (lines 260–265).

The class destructor (lines 273–279) unhooks you from the interrupt chain and turns off UART interrupts.

Lines 285–328 perform the CCITT CRC-16 error detection calculation used by DDCMP. Don't worry about the details of this calculation (see "For Further Reference" for resources).

Listings 16.6 and 16.7 fall into the "practice what you preach" category. These routines provide the basis for implementing shared IRQs. The `HookSharedInterrupt` routine (lines 20–43) of Listing 16.6 inserts your interrupt header at the front of the interrupt chain. You then can unhook from the interrupt chain by calling `UnhookSharedInterrupt` (lines 49–94 of Listing 16.6), which scans the linked list of interrupt handlers (in case you're no longer at the front) and removes you.

LISTING 16.6. ROUTINES TO HOOK AND UNHOOK SHARED INTERRUPT VECTORS IN ACCORDANCE WITH IBM PS/2 TECHNICAL DOCUMENTATION.

```
 1  // ************************************************************************
 2  //                    Secrets of the Borland C++ Masters
 3  //
 4  //   Module: shareint.cpp
 5  //
 6  //   Purpose: Routines to hook and unhook shared interrupt vectors
 7  //     in accordance with IBM PS/2 technical documentation.
 8  //
 9  //   Author: Gordon G. Free
10  //
11  // ************************************************************************
12  #include <dos.h>
13  #include "shareint.h"
14
15
16  // ---------------------------------------------------------------------
17  // HookSharedInterrupt - hook Interrupt Service Routine (ISR) into
18  //    chain.
19  // ---------------------------------------------------------------------
20  void HookSharedInterrupt(
21     int                  intr_num,       // interrupt vector to hook
22     PSHARED_INT_STUB_T   pSharedHdr,     // ISR stub
23     unsigned char        specificEOI     // specific EOI to issue to PIC
24  )
25  {
26
27     // It's always a good idea to disable interrupts when mucking with
28     // a linked list shared between tasks or threads.
29     disable();
30
31     // Get current vector address
32     pSharedHdr->prevHandler = (PSHARED_INT_STUB_T) getvect(intr_num);
33
34     // See if we are first in-line
35     if (*(unsigned char *)pSharedHdr->prevHandler != IRET_OPCODE)
36     pSharedHdr->chain_flags = FIRST_INTR_HANDLER;
```

```
37
38     // Hook vector and issue specific EOI to clear any pending interrupts
39     setvect(intr_num, (PISR_T)pSharedHdr);
40     if (specificEOI != 0)
41        outportb(0x20, specificEOI);
42     enable();
43  }
44
45  // ------------------------------------------------------------------
46  // UnhookSharedInterrupt - remove Interrupt Service Routine (ISR) from
47  //    chain.
48  // ------------------------------------------------------------------
49  void UnhookSharedInterrupt(
50     int                  intr_num,       // interrupt vector to unhook
51     PSHARED_INT_STUB_T   pSharedHdr      // ISR stub
52  )
53  {
54     PSHARED_INT_STUB_T pNextHandler;     // ptr to next ISR
55
56      // Can't afford to have the chain change on us while we're
57      // modifying it!
58      disable();
59
60      // Get first ISR from vector table
61      pNextHandler = (PSHARED_INT_STUB_T)getvect(intr_num);
62
63      // See if we are at the head of the interrupt chain
64      if (pNextHandler == pSharedHdr) {
65         // Remove us and let next ISR know that it is first
66         pNextHandler = pSharedHdr->prevHandler;
67
68         // Be sure he is using the same scheme we are!
69         if (pNextHandler->signature == SHARED_SIGNATURE)
70       pNextHandler->chain_flags |= (pSharedHdr->chain_flags & FIRST_INTR_HANDLER);
71         setvect(intr_num, (PISR_T)pNextHandler);
72
73      // We're not first, so scan the chain for our entry
74      } else {
75
76         // Be sure that each entry is playing by the rules
77         while ((pNextHandler->signature == SHARED_SIGNATURE)
78         && (pNextHandler->prevHandler != 0L)) {
79
80       // If we find ourselves, unhook us
81       if (pNextHandler->prevHandler == pSharedHdr) {
82           pNextHandler->prevHandler = pSharedHdr->prevHandler;
83           goto AllDone;     // What's this?  A goto?
84       } else {
```

continues

LISTING 16.6. CONTINUED

```
85           pNextHandler = pNextHandler->prevHandler;
86       }
87         // Oh Oh!  Either we're not in the chain or someone in front
88         // of us isn't playing fair!
89         }
90      }
91
92  AllDone:
93     enable();
94  }
```

LISTING 16.7. ISRSTUB.ASM.

```
 1  ; ********************************************************************
 2  ;                    Secrets of the Borland C++ Masters
 3  ;
 4  ;   Module: isrstub.asm
 5  ;
 6  ;   Purpose: Set up an interrupt handling stub that allows daisy-chaining
 7  ;      interrupt service routines (ISR).  This stub will call a C++
 8  ;      function at interrupt time (passing the proper 'this' ptr) and
 9  ;      will chain to the next ISR if the called function returns a zero.
10  ;
11  ;  Author: Gordon G. Free
12  ;
13  ; ********************************************************************
14  .MODEL LARGE, C
15  .CODE
16
17  FARCALL_OPCODE   EQU 9Ah
18
19  PUBLIC CopyTemplate
20
21  ;  This is the ISR stub that gets copied into every chained interrupt
22  ;  handler.  This particular instance of the code never gets executed.
23  Template_Start:
24  TemplateProc PROC  FAR
25
26     ; Standard chained ISR header (see PS/2 BIOS Technical Reference)
27     Template_Entry:
28                  jmp    short StartStub
29     PrevHndlr:   DD   0
30     Signature:   DW   424Bh
31     Flags:       DB   0
```

```
32                jmp   short ResetLoc
33                DB   0,0,0,0,0,0,0
34
35    ; Start of unique ISR code
36    StartStub:
37          push   ax
38    MovDSValue:
39          mov    ax, 1234h        ; real DS value goes here
40          mov    ds, ax
41    MovThisValue:
42          mov    ax, 5678h        ; real this address goes here
43          push   ax
44    CallHandler:
45          ; NOTE: may need to swap stack at this point!
46          call   TemplateProc     ; real ISR address goes here
47          ; restore stack if swapped here.
48          and    ax, ax           ; should we chain?
49          pop    ax
50          jz      Chain_Interrupt
51          iret
52
53     Chain_Interrupt:
54          jmp     cs:[PrevHndlr]
55
56     ResetLoc:
57          ret
58  TemplateProc   ENDP
59  Template_End:
60
61  ; ---------------------------------------------------------------
62  ; CopyTemplate - copies the ISR stub template into the provided
63  ;  buffer and patches the appropriate values so that the ISR will
64  ;  be all set when called at interrupt time.
65  ; ---------------------------------------------------------------
66  CopyTemplate   PROC USES cx si di ds es, Destination:FAR PTR,
                   ThisValue:NEAR PTR, Handler:FAR PTR
67
68          ; Patch current DS value for loading at interrupt time
69          mov   ax, ds
70          mov   WORD PTR cs:[MovDSValue+1], ax
71
72          ; Patch specified this ptr value
73          mov   ax, [ThisValue]
74          mov   WORD PTR cs:[MovThisValue+1], ax
75
76          ; Patch call to specified ISR
77          les   di, [Handler]
78          mov   WORD PTR cs:[CallHandler+1], di
```

continues

LISTING 16.7. CONTINUED

```
79          mov   WORD PTR cs:[CallHandler+3], es
80
81          ; NOTE: the linker may try to optimize the far call to a
82          ;   PUSH CS and a near call.  We'll patch in the proper
83          ;   opcode, since our patched in ISR will be far.
84          mov   BYTE PTR cs:[CallHandler], FARCALL_OPCODE
85
86          ; Now, copy the template code into the provided buffer
87          les   di, [Destination]
88          mov   ax, cs
89          mov   ds, ax
90          mov   si, OFFSET Template_Start
91          mov   cx, Template_End - Template_Start
92      rep  movsb
93          ret
94  CopyTemplate   ENDP
95  END
```

isrstub.asm is an assembly module containing a template of the interrupt header code (lines 23–59), which links shared interrupts. The `CopyTemplate` routine patches in the appropriate DS and `this` value into the template along with the address of the method to be called (lines 69–84). Once this operation is done, copy the entire template into the current instance of the interrupt header (lines 87–92).

> The interface between the interrupt service routine stub and the C++ method to handle the interrupt event assumes that the object data pointer, `this`, is passed on the stack. If you enable Borland C++ 3.1's object data optimization, lines 42 and 43 of Listing 16.7 should be changed to load the pointer value into SI.

Finally, Listing 16.8, SERTEST.CPP, is a sample application that uses the serial communications class to send and receive data between computers over a serial cable. You send a packet of data over COM1 by pressing the return key. Data you receive from the remote is displayed and the buffer is reposted.

LISTING 16.8. SERTEST.CPP.

```
 1  #include <fstream.h>
 2  #include <conio.h>
 3  #include "ddcmp.h"
 4
 5  #define CR      13
 6  #define ESC     27
 7
 8  char RcvBuffer[1024];
 9  char XmitBuffer[]
10      = "Hello, this is the United States calling.  Are we reaching?";
11
12  int main()
13  {
14      DDCMP_channel cch1(COM1);
15      int kbdRqst;
16
17      cch1.ReceiveInformationFrame(&RcvBuffer);
18
19      while(1) {
20         if (cch1.IsDataAvailable()) {
21       cout << RcvBuffer;
22       cch1.ReceiveInformationFrame(&RcvBuffer);
23         }
24         if (kbhit()) {
25       kbdRqst = getch();
26       if (kbdRqst == CR)
27          cch1.SendInformationFrame(XmitBuffer, sizeof(XmitBuffer));
28       else if (kbdRqst == ESC)
29               break;
30         }
31      }
32
33      return 0;
34  }
```

FOR FURTHER REFERENCE

If you would like more information about serial communications programming for the IBM PC or data communcations in general, you may want to check out some of these books and articles. The hardware data books are available directly from the manufacturers or your local distributor.

Campbell, Joe. *C Programmer's Guide to Serial Communications*, Indianapolis, Indiana: Howard W. Sams, 1987. Still one of the few books dedicated entirely to serial communications. Has a fairly good reference on programming Hayes-compatible modems. UART section is a bit dry, however.

Duntemann, Jeff. "Structured Programming," *Dr. Dobb's Journal*, M&T Publishing, March 1991 to Janurary 1992. An excellent series introducing programming the UART. Done in the usual Duntemann style (that is, makes for good, interesting reading).

Intel Corporation. *Peripherals*, Santa Clara, California: Intel Corporation, 1990. Contains useful information on the 8259 PIC and other important chips inside the PC.

International Business Machines Corporation. *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference*, Second Edition, International Business Machines Corporation, 1988. A good definitive source on serial BIOS services as well as how to implement shared interrupts.

Jourdain, Robert. *Programmer's Problem Solver for the IBM PC, XT and AT*, New York: Brady Books, 1986. Has many good examples of different levels of UART programming along with other common programming challenges.

National Semiconductor Corporation. *Data Communications, Local Area Networks, UARTS*, Santa Clara, California: National Semiconductor Corporation, 1990. The definitive source for how to program UARTs. Written for hardware jockeys, definitely not for the faint of heart!

Phoenix Technologies Ltd. *CBIOS for IBM PS/2 Computers and Compatibles*, New York: Addison-Wesley Publishing Company. Details all available BIOS services, but it is mostly a reference manual and not very interesting reading.

Sargent III, Murray and Richard L. Shoemaker, *The IBM Personal Computer from the Inside Out*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1984. Still the best book I've seen describing the IBM PC hardware in an understandable fashion. Dated but still excellent.

Spragins, John D., Joseph L. Hammond, and Krzysztof Pawlikowski. *Telecommunications, Protocols and Design*, New York: Addison-Wesley Publishing Company, 1991. A good book on OSI communications modules and protocols. Contains a brief section on DDCMP.

Stone, Harold S. *Microcomputer Interfacing*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1982. One of my college textbooks. Much of it deals with transmission theory, but has good sections on RS-232 communications.

Tanenbaum, Andrew S. *Computer Networks*, Second Edition, Englewood Cliffs, New Jersey: Prentice Hall, 1988. The classic textbook for data communications.

TEMPLATES, PARSING, AND MATH

Using class templates

Using function templates

Parsing character strings

Three totally unrelated topics are covered in this chapter: the use of class and function templates, parsing techniques, and various options available to programs that rely on floating-point, binary-coded-decimal, or complex calculations.

If you program in C++, you already know how you can use classes to help reuse and recycle your code and increase your efficiency as a programmer. *Templates* raise the level of abstraction one degree and enable you to automatically create individual classes that work with different types of data. In other words, using a template, you can create a class definition that works with integers, a class that works with characters, and a class that works with floating-point numbers—but do it all in one class definition and without writing any new code. The idea of writing one class whose code can simultaneously support, say, a stack of integers and a stack of character strings might seem rather farfetched,

but believe me, it really works. A related technique enables you to create *function templates*, which are basically shorthand methods of generating a bunch of overloaded function definitions. The use of templates, fortunately, is both powerful and easy. Templates are described in the sections "Class Templates" and "Function Templates."

Parsing is the art of processing an input expression or command and translating the request into action. Many programmers concoct ad hoc scanning routines to break apart a user's input. These approaches are quite limited in what they can accomplish. The section "Parsing" describes a general-purpose method you can use to parse complex statements, arithmetic expressions, user input, command-line options, and more.

The chapter concludes with a look at the Borland C++ floating-point, binary-coded-decimal, and complex calculation options.

CLASS TEMPLATES

A C++ *class* is like a manufacturing mold used to create many copies of a particular part. In C++, the mold is the class definition, and each part is an instantiated object. Like the parts created by the mold, each object is essentially identical. If your class maintains a doubly linked list of integers, each object in that class works only with integers. If you then decide to create a doubly linked list of floating-point values, you need to retool your class definition. Suppose that later you need a doubly linked list of character strings, too. It's back to the factory floor to do some more retooling.

There is, however, a better way to address this type of problem. Borland C++ supports templates. A template creates classes, just as a class creates objects. The trick is that a template can perform certain substitutions in the class definition so that a single class definition can describe a doubly linked list of integers, floating-point numbers, character strings, or any other object you might think of. As such, a template is a *generic* class definition. Best of all, templates are remarkably easy to use. Keep templates in mind when you start writing similar class definitions. You might be able to write a single class template that describes all your similar classes.

To understand where a template might be used, consider the implementation of a simple stack data structure. Listing 17.1 shows a conventional C++ class inline definition for an integer stack. Listing 17.2 illustrates how the stack class is used in a sample application. The class's major member functions are `push()` to add a value to the stack and `pop()` to remove a value. Values are added, as shown in line 13, by assigning `st[sp++]` the value to be pushed on the stack. `sp` points to the stack location to be used for the next item. The conditional test `if (sp>=0)` provides a safety check that comes into play in an application described later in this chapter. An item is popped off the stack by decrementing `sp`, fetching the value at `st[sp]`, and returning that integer.

LISTING 17.1. A SAMPLE IMPLEMENTATION OF A CLASS THAT MAINTAINS A STACK OF INTEGER VALUES.

```
 1  // TISTACK.H
 2  // Implements a stack of int's.
 3  #ifndef TISTACK_H
 4  #define TISTACK_H
 5
 6  #define MAXSTACKSIZE 30
 7
 8  class TIStack {
 9  public:
10    TStack() { sp = 0; };
11    virtual void reset() { sp = 0; }
12    virtual void push( int value )
13    { if (sp>=0) st[sp++] = value; }
14    virtual int pop( void ) { return st[--sp]; }
15    virtual int overflow() {
16      return (sp == (MAXSTACKSIZE-1));
17    }
18    virtual int underflow() {
19      return (sp <= 0);
20    }
21  private:
22  int sp;
23  int st[MAXSTACKSIZE];
24  };
25  #endif
```

LISTING 17.2. SAMPLE PROGRAM TO USE THE INTEGER STACK CLASS.

```
 1  // INTSTACK.CPP
 2  // Demonstrates use of the TIStack class.
 3  #include <iostream.h>
 4  #include "tistack.h"
 5
 6  void main(void)
 7  {
 8  TIStack s1;
 9
10  s1.push(1);
11  s1.push(2);
12  s1.push(3);
13
14  cout << s1.pop() << ", " << s1.pop() << ", " << s1.pop() << "\n";
15
16  }
```

Now, consider what you need to do if you change this stack to support `float` values in place of integers. In Listing 17.1, you need to copy the `TIStack` class definition and then manually change the return result for `pop()` and the data type of the array in line 23. You now have two class definitions. If you decide to make more changes later on, you need to carefully insert the changes in both class definitions. What a bother. Add a third or fourth stack type and future modifications will clearly get out of hand.

The solution, obviously, is to use a template to describe the stack. In the template definition, a user-defined symbol acts as a placeholder for the data type and can be filled in when a particular type of stack is required.

Listing 17.3 shows the template definition for the `TStack` class. Notice the use of the keyword `template` in line 10 and the new syntax that defines `<class TYPE>`. `TYPE` becomes the placeholder for the data type, whatever it might be. In line 15, `TYPE` defines the type of the `value` parameter to push, the return type for `pop()` in line 17, and the type of the stack in line 26. In this basic form, the template is doing little more than substituting a macro for the data type; however, as you will soon see, the template is far more powerful than a macro.

LISTING 17.3. THE TEMPLATE-BASED DEFINITION OF A STACK.

```
1  // TSTACK.H
2  // Implements the TStack template.  The name TStack
3  // is used to eliminate conflicts with the standard
4  // stack.h file.
5  #ifndef TSTACK_H
6  #define TSTACK_H
7
8  #define MAXSTACKSIZE 30
9
10 template <class TYPE>
11 class TStack {
12 public:
13   TStack() { sp = 0; };
14   virtual void reset() { sp = 0; }
15   virtual void push( TYPE value )
16   { if (sp>=0) st[sp++] = value; }
17   virtual TYPE pop( void ) { return st[--sp]; }
18   virtual int overflow() {
19     return (sp == (MAXSTACKSIZE-1));
20   }
21   virtual int underflow() {
22     return (sp <= 0);
23   }
24 private:
25 int sp;
26 TYPE st[MAXSTACKSIZE];
27 };
28 #endif
```

Listing 17.4 illustrates use of the `TStack` template to create three separate stacks: one of integers, one of floating-point numbers, and one of character strings. Each of the stacks is defined in lines 9–11. Notice the use of the angle brackets to enclose the stack type. The stack type parameter matches the `<class TYPE>` parameter in the `TStack` template definition in Listing 17.3. Remarkably, a single class definition enables you to create stacks for three entirely different types of data. This is the essence of a template definition. When you create similar classes that perform similar manipulations of different data types, consider using a template instead of multiple class definitions.

LISTING 17.4. THE TSTACK TEMPLATE IN USE.

```
// TEMPLATE.CPP
// Demonstrates use of a template.
#include <iostream.h>
#include "tstack.h"

void main(void)
{

TStack<int> s1;
TStack<float> s2;
TStack<char *> s3;

s1.push(1);
s1.push(2);
s1.push(3);

s2.push(1.001);
s2.push(71.002);
s2.push(3.14159);

s3.push("String 1");
s3.push("String 2");
s3.push("String 3");

cout << s1.pop() << ", " << s1.pop() << ", " << s1.pop() << "\n";
cout << s2.pop() << ", " << s2.pop() << ", " << s2.pop() << "\n";
cout << s3.pop() << ", " << s3.pop() << ", " << s3.pop() << "\n";

}
```

So far, the template has been used only to store the built-in data types. But why stop there? A template is not just a macro substitution. Indeed, a template is far more powerful than a macro. Listing 17.5 illustrates this power by using the existing `TStack` template to store a stack of arbitrary objects. The `TLogEntry` class was used in Chapter 13, "Using Borland C++ with Other Products," to illustrate the container class libraries. `TLogEntry` is derived from the `Object` type that is used as the root of all the object-based container libraries.

LISTING 17.5. THE TSTACK TEMPLATE USED TO CREATE A STACK OF OBJECTS.

```
// TEMPLAT2.CPP
// Demonstrates use of a template and adds
```

```
// a class derived from the container class
// Object class.
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <object.h>
#include "tstack.h"

class TLogEntry : public Object {
public:
  TLogEntry() {};
  TLogEntry(char * NewCallSign,
    char * NewContactNum,
    char * NewExchange,
    int NewMode,
    int NewBand,
    long NewTime,
    long NewDate ) : Object () {
      strcpy( CallSign, NewCallSign );
      strcpy( ContactNum, NewContactNum );
      strcpy( Exchange, NewExchange );
      Mode = NewMode;
      Band = NewBand;
      ContactTime = NewTime;
      ContactDate = NewDate;
       }
  virtual hashValueType hashValue() const;
  virtual classType isA() const {return __firstUserClass;}
  virtual int isEqual( const Object& testObject) const
    { if (stricmp( ((TLogEntry&)testObject).CallSign, CallSign ))
       return 0;
       else return 1;
     }
  virtual char *nameOf() const {return "TLogEntry";}
  virtual void printOn( ostream& outputStream) const
      { outputStream << CallSign;}
  friend void Display(Object& o, void *);
private:
  char CallSign[11];
  char ContactNum[11];
  char Exchange[35];
  int Mode, Band;
  long ContactTime;
  long ContactDate;
};

hashValueType TLogEntry::hashValue() const
{
  // This hash computation algorithm is adapted
```

continues

LISTING 17.5. CONTINUED

```
52    // from Borland's STRNG.CPP implementation
53  /*
54    hashValueType   hash = hashValueType(0);
55    int len = strlen( CallSign );
56    for( int i = 0; i < len; i++ ) {
57      hash ^= CallSign[i];
58      hash = _rotl( hash, 1 );
59    };
60    return hash;
61  */
62    return atoi( ContactNum );
63  }
64
65
66
67  void main(void)
68  {
69
70  TStack<int> s1;
71  TStack<float> s2;
72  TStack<char *> s3;
73  TStack<TLogEntry> s4;
74
75  s1.push(1);
76  s1.push(2);
77  s1.push(3);
78
79  s2.push(1.001);
80  s2.push(71.002);
81  s2.push(3.14159);
82
83  s3.push("String 1");
84  s3.push("String 2");
85  s3.push("String 3");
86
87  s4.push( TLogEntry("KF7VY", "0001", "59", 0, 0, 0, 0) );
88  s4.push( TLogEntry("N7VPL", "0002", "59", 0, 0, 0, 0) );
89
90  cout << s1.pop() << ", " << s1.pop() << ", " << s1.pop() << "\n";
91  cout << s2.pop() << ", " << s2.pop() << ", " << s2.pop() << "\n";
92  cout << s3.pop() << ", " << s3.pop() << ", " << s3.pop() << "\n";
93  cout << s4.pop() << ", " << s4.pop() << "\n";
94
95  }
```

SPECIAL RULES REGARDING CLASS TEMPLATES

A class template can have more than one argument. For example, you might write:

```
template <class TYPE1, class TYPE2>
```

Both arguments can be used within the generic class definition.

Each object must provide member functions or overloaded operators, as needed, to implement the operations used within the template class. For example, if your template class performs arithmetic on objects of type `TYPE`, the arithmetic operators must be defined for the object. You must ensure that each object also has a copy constructor.

Class templates can also include variable parameters, similar to the parameters you use in a function. Consider the `TStack` template and its use of the `MAXSTACKSIZE` manifest constant to determine the size of the stack. To be completely flexible, `TStack` should enable you to set the stack size when you select the data type. To do this, you might make `MAXSTACKSIZE` into a variable argument having a default value, like this:

```
template <class TYPE, int MAXSTACKSIZE = 30>
```

Now, when you create a class of objects from this template, you can optionally specify a new stack size and, as in this example, create a stack of 100 integers:

```
TStack<int, 100>;
```

The value 100 overrides the default of 30. You can specify any constant expression in the definition. That means the use of `const int` or macro symbols is acceptable, but the use of a variable is not. If you omit the stack size, as shown here, the size defaults to 30:

```
TStack<int>;
```

The `TStack` example in this chapter uses inline member functions. If you want to define the member functions out-of-line (and you probably want to define them as out-of-line functions), you need to incorporate special syntax before each out-of-line member function. An example is presented below. Notice how you must duplicate the `template<>` definition itself: the member function type, the template name with each of its parameters in brackets, and finally, the member function. Here's an example that implements `TStack`'s `push()` member function out of line:

```
template <class TYPE, int MAXSTACKSIZE=64>
void TStack<TYPE,MAXSTACKSIZE>::push(TYPE value)
{ if (sp>=0) st[sp++] = value; }
```

Within the class definition itself, `push()` is defined (but not implemented) by writing only:

```
virtual void push(TYPE value);
```

FUNCTION TEMPLATES

C++ also provides a function template or *generic function* capability. You should consider using a function template when you find yourself writing a sequence of identical, overloaded functions. For example, consider the `abs()` function, which returns the absolute value of its parameter. To implement `abs()` so that it can accept `int`, `float`, `long`, and `double` parameters, you could write the function four separate times and let C++'s overloaded function feature sort out which function to call depending upon the data type. Your set of `abs()` definitions might look something like this:

```
int abs(int x)
{
  return (x<0) ? -x : x;
}

long abs(long x)
{
  return (x<0) ? -x : x;
}

float abs(float x)
{
  return (x<0) ? -x : x;
}

double abs(double x)
{
  return (x<0) ? -x : x;
}
```

Of course, since you read the section on class templates, I'll bet you see a better solution! That's right, use a function template. Instead of writing all of those `abs()` functions, write a single function template:

```
template <class TYPE>
TYPE abs( TYPE x )
{
  return (x<0) ? -x : x;
};
```

That's all you need do to write a generic `abs()` function. This function works with any data type, provided that a unary `operator-()` function is defined for the data object.

As with other functions, you can declare an overloaded function independent of the function template. For instance, if you separately add a definition for:

```
char * abs( char * ) {...}
```

the compiler treats this as an overloaded function.

Conceivably, you could define a macro to implement the `abs()` function:

```
#define abs(x) ((x)<0) ? -(x) : (x) )
```

Any type you substitute for *x* will be expanded in the macro, even if *x* is a `struct`, `union`, or array, and it doesn't make sense to perform the absolute value. Worse though, a macro causes a blanket substitution to occur anywhere in the file. If you later decide to implement an overloaded `abs()` function, such as:

```
long abs( struct TagRecord x) {;};
```

you quickly run into trouble. Your function never becomes a function but instead translates into this bizarre macro expansion:

```
long ( (struct TagRecord x)<0 ?
      -(struct TagRecord x) : (struct TagRecord x) ) {;};
```

As you might suspect, macro definitions, while very useful in C, are not so necessary in C++. Instead, many macros can be replaced with function templates, which provide improved type checking through overloaded functions.

PARSING

Parsing is the act of scanning through a string of characters and carving the string into meaningful chunks. Consider the first sentence in this paragraph. Our eyes parse the words from the sentence by recognizing the white space between each word. Our brains process the words to determine that the order

is correct and has some valid meaning. Many programs must parse input lines to determine a course of action. These input lines might be a set of command-line options, a command type in response to a prompt, or a relatively complex instruction such as an equation that must be evaluated and solved. This section describes methods of parsing user input. You will gain a limited appreciation and understanding of what goes on inside the compiler as it compiles your programs.

Breaking an input stream into its constituent parts is called *lexical analysis*. Lexical analysis carves the stream of input characters into words or symbols. The syntax of a language describes valid ways of putting the symbols and punctuation together. Syntax analysis checks that the symbols and punctuation appear in the proper order. Finally, semantic analysis interprets the meaning of the statement and ensures that syntactically valid statements actually make some sense. If you have ever seen a linguistics textbook, you probably saw Noam Chomsky's famous sentence: "Colorless green ideas sleep furiously." This is a syntactically correct but meaningless sentence. Ideas are not usually green, and especially not colorless green. Have you ever seen an idea sleep furiously? Syntax analysis only indicates that this is a potentially useful sentence. It is up to semantic analysis to figure out what the statement actually means. In the context of a compiler, for example, semantic analysis produces a sequence of machine instructions.

Before you dive into parsing, you might be able to solve some command interpretation problems using simpler techniques than full-scale lexical, syntactic, and semantic analysis. You might be able to use `sscanf()`, for scanning formatted input out of a string buffer, or the `strtok()` token scanning function.

USING *SSCANF()*

`sscanf()`, defined in stdio.h, is the string-based version of `scanf()`. As you already know, `scanf()` reads input from stdin and, based upon a formatting string that you supply, parcels the input data into a set of target variables. `sscanf()` and `scanf()` operate identically except that `sscanf()` scans a character buffer that you provide. By using `sscanf()` you can process text input entered in a dialog box or on the command line.

```
     sscanf() is defined as:
int sscanf( const char *buffer,
            const char *format, [, address, ...]);
```

`sscanf()` scans the characters in *`buffer`* according to the format specifier in *`format`*, storing the result in the variable-size list of destination variables. *`format`* should contain format specifiers like those used in `scanf()` or `printf()`.

You can use `sscanf()` to parse through relatively simple lists of data, as shown in this example:

```
char * buffer = {"Seattle 100 Spokane 175"};
char city1[20], city2[20];
int dist1, dist2;

sscanf(buffer, "%s %d %s %d", &city1, &dist1, &city2, &dist2);
printf("%s=%d, %s=%d\n", city1, dist1, city2, dist2);
```

This example extracts "Seattle" from `buffer` and places it in the `city1` character array, extracts "100" and stores the result in the integer variable `dist1`, extracts "Spokane" and places the result in `city2`, and extracts "175" and stores this value in `dist2`.

A significant problem in using `sscanf()` is that `sscanf()` can neither read multi-word character strings nor handle punctuation symbols such as comma or semicolon. Punctuation symbols, also known as *delimiters*, mark the boundaries of the data fields. In place of `sscanf()`, you can use the `strtok()` library function, which does recognize delimiter characters, to extract the portions of the buffer that fall between the delimiters.

USING *STRTOK()*

`strtok()` scans through a character array, stopping whenever it encounters special token characters that you specify. `strtok()` comes in two flavors, a near version and a far version:

```
char *strtok(char *s1, const char *s2);
char far *_fstrtok(char far *s1, const char *s2);
```

On the first call to `strtok()`, *`s1`* is the address of a character string to parse, and *`s2`* is the address of a list of potential delimiter characters. `strtok()` scans through *`s1`* until it finds one of the delimiter characters in *`s2`*. It returns a pointer to the first substring, or token, that it isolates. Subsequent calls scan deeper into the string, returning a pointer to the next token found, or to `NULL` if no more tokens are found.

When you use `strtok()` to scan a string, you pass the *s1* parameter only on the first call. On subsequent calls, set *s1* to `NULL`. This way, `strtok()` knows that it should continue where it left off during the previous call. When `strtok()` finds a token, it returns a pointer to the first character in the token, and changes the byte where it finds the delimiter character to a `'\0'` null byte. This produces a null terminated string. On the next call to `strtok()`, it begins searching at the first character past the null byte.

Listing 17.6 uses `_fstrtok()` to extract the tokens from the input_buffer string (this program was compiled under the large memory model). Notice the use of two delimiter characters. The `strtok()` (or `_fstrtok()`) function stops when it reaches any one of the delimiters specified. The output from this program displays:

```
Seattle
100
Spokane
175
Yakima
38
```

LISTING 17.6. DEMONSTRATION OF THE `strtok()`/`_fstrtok()` FUNCTION.

```
 1  // strtok.c
 2  #include <stdio.h>
 3  #include <string.h>
 4  void main(void)
 5  {
 6    char input_buffer[]="Seattle,100;Spokane,175;Yakima,38";
 7    char * token = NULL;
 8    char delimiters[] = ",;";
 9
10    token = _fstrtok( input_buffer, delimiters );
11
12    while (token) {
13      if (token) printf("%s\n", token);
14      token = _fstrtok( NULL, delimiters );
15    };
16  }
```

CONSTRUCTING A FORMAL PARSER

In this section, you will develop a relatively simple set of classes to implement basic arithmetic expression evaluation and simple command parsing. In the sample program, you can type a BASIC-like statement, PRINT, followed by a typical algebraic expression that includes arithmetic operators, conditional tests such as equality or less-than-or-equal-to, and logical operators such as AND and OR. This sample code abides by the standard algebraic rules of hierarchy that cause unary +, -, and NOT to be performed before multiplication and division, which in turn is performed before addition and subtraction.

The parsing technique used is called *recursive descent parsing* and is applicable to many types of input that must be handled by applications. You can, for instance, use the technique illustrated here to parse arbitrarily complex command sequences such as those that might appear in a command-driven program or in a computer adventure game. I chose recursive descent parsing for this example because it is flexible, relatively easy to understand, and easy to implement.

Parsing, as noted earlier, is normally divided into three steps: lexical analysis, syntax analysis, and semantic analysis. In some parsers (such as some compilers), these steps are conducted independently, each on a separate pass through the source to be interpreted or compiled. In the sample program presented in this section, all three steps occur incrementally and simultaneously. In other words, the syntax analysis stage calls the lexical analyzer to fetch one or more tokens. Once sufficient tokens are read, syntax analysis verifies the statement and takes immediate action. In the sample program used in this section, this means that the program evaluates the expression as it steps character-by-character through the input string.

Lexical analysis carves the *source statement* (the statement to be evaluated) into logical entities called *tokens*. To the computer, a statement is just a long sequence of characters. The lexical analyzer breaks the characters into recognizable groups such as IF, +, AND, and 456.

Syntax analysis, or *parsing*, ensures that the tokens returned by the lexical analyzer appear in the correct order. When the syntax analyzer recognizes a valid sequence of tokens, the parser can evaluate a portion of the input expression or statement. During the parsing process, lexical analysis and syntax analysis are intermingled. When the syntax analyzer needs a new token, it calls the lexical analyzer.

THE SAMPLE LANGUAGE

The parser in this example processes a simple language consisting of a small number of keywords and expressions. It is used to evaluate statements such as:

```
PRINT 35*(6+35/2)
```

The syntax of the language is shown as a series of syntax diagrams in Figure 17.1. These diagrams show how properly formed sentences are constructed. The way the language syntax is defined enforces the algebraic rules of hierarchy so that multiplication and division will be performed prior to addition and subtraction. You can see this by looking at the syntax definitions of `simpleexpression` and `term`. A `simpleexpression` consists of one or more `terms` followed by the addition or subtraction operator. In `term`, the multiplication and division operators are recognized as having higher priority than the operators within `simpleexpression`. The section "Syntax Analysis" discusses this in greater detail.

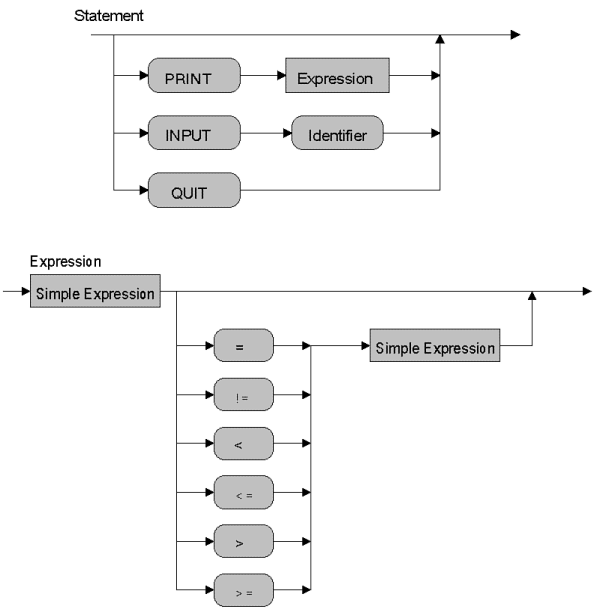


Figure 17.1. The syntax diagrams of the sample language used as a parsing example.

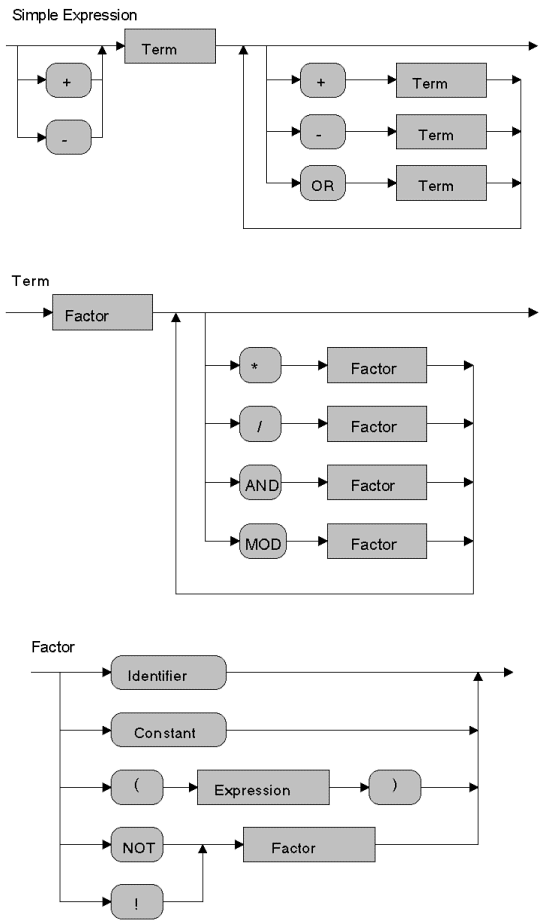


Figure 17.1. continued

LEXICAL ANALYSIS

The lexical analyzer reads the source program character by character. Once it recognizes a valid sequence of characters, it returns the entire group as a token. Most lexical analyzers must recognize a large number of tokens—many more than just letters and numbers. For example, the lexical analyzer described in

this chapter must recognize +, -, (,), and many other standard arithmetic symbols. Some symbols are more difficult than others to recognize. For example, < by itself is the *less-than* relational operator. But when followed immediately by =, the sequence <= becomes the *less-than-or-equal-to* relational operator. Parsers that support character strings (the sample analyzer in this chapter does not) must recognize first a leading double quotation mark, then an arbitrary number of characters, and last a trailing double quotation mark.

A practiced programmer can put together a lexical analyzer quite quickly. The first step is to carefully study the syntax of the language to be parsed and identify all the symbols and characters permitted in the language. In the case of the expression evaluator, the allowable characters include the alphabetic letters, integer numbers, arithmetic operators, and a few punctuation symbols such as the left and right parentheses. Once the character set is identified, you can set to work on the construction of the lexical analyzer.

The lexical analyzer not only carves up the input stream, it usually returns a coded token value that indicates the type of the token found. It may also convert the character representation of a number ("457") into its integer format, or convert identifiers from lowercase to uppercase.

A lexical analyzer class is defined in Listing 17.7. Most of the `#define` macros are used to specify the tokens returned by the lexical analyzer. The values in lines 9–16 correspond to keywords such as `PRINT`, `AND`, or `QUIT`, and their values correspond to values stored in the internal `keywords[]` symbol table (see line 75 and Listing 17.8, lines 31–40).

LISTING 17.7. THE LEXICAL.H CLASS HEADER.

```
 1  // LEXICAL.H
 2  #ifndef  LEXICAL_H
 3  #define  LEXICAL_H
 4
 5  #define MAXSTMTLENGTH 78  // Max length of a input
 6  #define MAXSYMLENGTH 16   // Max length of symbol
 7
 8  // Token values returned by getsymbol()
 9  #define ENDOFLINE       0
10  #define KAND            1
11  #define KINPUT          2
12  #define KMODULUS        3
13  #define KMOD            4
14  #define KOR             5
```

```
15  #define KPRINT          6
16  #define KQUIT           7
17
18  #define NOT             60
19  #define OTHER           61
20  #define IDENTIFIER      62
21  #define INTCONSTANT     63
22  #define LP              64
23  #define RP              65
24  #define ADDOP           66
25  #define SUBOP           67
26  #define MULOP           68
27  #define DIVOP           69
28  #define EQOP            70
29  #define COMMA           71 // not used in example
30  #define SEMICOLON       72 // not used
31  #define COLON           73 // not used
32  #define BACKSLASH       74 // not used
33  #define LESOP           75
34  #define LEQOP           76
35  #define NEQOP           77
36  #define GTOP            78
37  #define GEQOP           79
38
39  // Possible error codes
40  #define E_INVALIDCHAR     0
41  #define E_DIVIDEBYZERO    1
42  #define E_OUTOFSTACK      2
43  #define E_INVALIDEXPR     3
44  #define E_RPEXPECTED      4
45
46  class LexicalAnalyzer {
47  public:
48  char symbol[MAXSYMLENGTH];  // Current input symbol
49  int token;                  // Current tokenized symbol
50  int tokenvalue;             // Current value of #s
51
52  LexicalAnalyzer(void) {};
53  // Call init_lexicalyzer before calling getsymbol()
54  void init_lexicalyzer(char * line_to_parse);
55  unsigned getsymbol(void);
56
57  // error and InSet() could be placed in a separate
58  // non-class module but are here for convenience.
59  void error( int errcode );
60  int InSet(
61     short * set, int toFind, int num_entries );
62
63  private:
```

continues

LISTING 17.7. CONTINUED

```
64  void lookup_keyword( char * sym, int& keywrd);
65  char get_next_char(void);
66
67  char statement[MAXSTMTLENGTH];     // Statement to be parsed
68  char *BufPtr;                      // Ptr into statement
69  char *pSymbol;                     // Ptr into symbol
70  char ch;                           // Current input char
71
72  #define NUMVALIDCHARS 48  // # of valid input chars
73  static const char validchars[NUMVALIDCHARS];
74  static unsigned map[26];  // Symbol table to locate
75  static char *keywords[];  // keywords quickly
76  static char *error_messages[];  // Err msg text
77  };
78  #endif
```

The primary interface to the lexical analyzer is its `getsymbol()` function. Each time `getsymbol()` is called, the lexical analyzer returns the next token from the input statement. It sets its member variables `token`, `tokenvalue`, and `symbol` depending on the token recognized. `token` contains an integer value corresponding to the `#define` constants. If `getsymbol()` sees a left parenthesis `(`, it sets `token` to `LP`; if `getsymbol()` sees an integer constant, it sets `token` to `INTCONSTANT` and `tokenvalue` to the value of the integer constant.

Before the lexical analyzer is used, its `init_lexicalyzer()` function is called to initialize and prepare the lexical analyzer for the next statement. A couple of member functions, `error()` and `InSet()`, are helper functions that aren't really needed as part of the `LexicalAnalyzer` class. They should be placed in a separate utility module and not implemented as member functions. They are placed here as a convenient way to keep the number of modules small for the purposes of this example.

The implementation of the lexical analyzer is shown in Listing 17.8. This source code module will be linked into a complete program later in this chapter. The first section of the listing initializes the `static` tables used during parsing. The `validchars[]` character array is a table of all characters defined in the language. The lexical analyzer reads through its input character by character. If it encounters a character not defined in this table, the lexical analyzer issues an error message.

LISTING 17.8. THE IMPLEMENTATION OF THE LEXICALANALYZER CLASS.

```
// LEXICAL.CPP
// Implements class LexicalAnalyzer, which performs
// lexical analysis on a statement.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "lexical.h"

// This table is the list of all characters recognized
// by the parser.  If the statement to be parsed contains
// a character NOT in this table, an error is issued.
const char LexicalAnalyzer::validchars[NUMVALIDCHARS] = {
  ' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
  'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
  'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '0', '1', '2',
  '3', '4', '5', '6', '7', '8', '9', '(', ')', '+',
  '-', '*', '/', '=', '<', '>', '!', '\0' };

// The next two tables implement a simple hash table
// to quickly look up an identifier to see if it is
// a keyword.
unsigned LexicalAnalyzer::map[26] = {
  1, 0, 0, 0, 0, // A, B, C, D, E
  0, 0, 0, 2, 0, // F, G, H, I, J
  0, 0, 4, 0, 5, // K, L, M, N, O
  6, 7, 0, 0, 0, // P, Q, R, S, T
  0, 0, 0, 0, 0, // U, V, W, X, Y
  0 // Z
};

char * LexicalAnalyzer::keywords[ ] = {
  "",            // 0 unused
  "AND",         // 1
  "INPUT",      // 2
  "MODULUS",        // 3
  "MOD",         // 4
  "OR",         // 5
  "PRINT",      // 6
  "QUIT"        // 7
};

char * LexicalAnalyzer::error_messages[] = {
"Invalid character in input.",
"Attempt to divide by zero.",
"Expression too complicated.",
"Invalid expression.",
```

continues

LISTING 17.8. CONTINUED

```
48  "Right parenthesis ')' expected."
49  };
50
51
52  //=====================================
53  // error displays an error message corresponding
54  // to errcode.
55  void LexicalAnalyzer::error( int errcode )
56  {
57    printf("!!!! %s\n", error_messages[errcode] );
58    // Force scanning to halt
59    ch = ENDOFLINE;
60  };
61
62  //=====================================
63  // Searches for toFind in the array of short integers
64  // pointed to by set, returning 1 if toFind was found,
65  // or 0 if not found.
66  int LexicalAnalyzer::InSet(
67      short * set, int toFind, int num_entries )
68  {
69    for( int i=0; i<num_entries; i++)
70      if ( *(set+i) == toFind ) return 1;
71    return 0;
72  }
73
74
75  //=====================================
76  // Read the next character from the input line
77  // Returns null char '\0' if at end of line
78  char LexicalAnalyzer::get_next_char(void)
79  {
80    if (ch)
81    // If non-null, then append to symbol string
82    {
83        *pSymbol++ = ch;
84        *pSymbol = '\0';
85        ch = *BufPtr++;
86    };
87    return ch;
88  }
89
90  //=====================================
91  // Determine if sym exists in the keyword table.
92  // If it does, return the table address.  If it
93  // doesn't exist, then say it's an IDentifier
```

```
 94  void LexicalAnalyzer::lookup_keyword(
 95       char * sym, int& keywrd)
 96  {
 97    unsigned index;
 98    int foundit;
 99
100    // Compute a 'hash code' using the symbol's first
101    // letter.  Map that through map[] to point to a
102    // potential match in the keywords[] table.
103    index = map[sym[0] - 'A' ];
104    if (index == 0)
105      // This symbol is definitely not a keyword
106      keywrd = IDENTIFIER;
107    else
108    do {
109      foundit = strcmp( sym, keywords[index] );
110      if (foundit == 0)
111        // if sym matches keyword table entry
112        // then return the index value
113        { keywrd = index; return; };
114      if (foundit < 0)
115        // if sym < table entry, then
116        // we are all done; return as IDentifier
117        { keywrd = IDENTIFIER; return; }
118      // otherwise, decrement to next position in
119      // the table.
120      index--;
121    } while (1);
122  };
123
124  //=====================================
125  // Initializes the lexical analyzer to
126  // parse the string given as its parameter.
127  void LexicalAnalyzer::init_lexicalyzer(
128       char * line_to_parse)
129  {
130    strcpy( statement, line_to_parse);
131    BufPtr = (char *)&statement;
132    // Before calling get_next_char() for the first
133    // time, need to ensure that ch != ENDOFLINE.
134    // This is done so that get_next_char() can
135    // efficiently detect ENDOFLINE and return ch
136    // as the result.
137    ch = ' ';
138    ch = get_next_char();
139  }
140
141  //=====================================
142  // This is the "guts" of the lexical analyzer.
```

continues

LISTING 17.8. CONTINUED

```
143  // getsymbol() scans the input to identify the
144  // next valid token in the language.  On return
145  // 'token' is set to a coded representation of
146  // the item it found, 'symbol' contains the string
147  // (such as identifier name), and for constant
148  // values, 'tokenvalue' contains the value.
149  unsigned LexicalAnalyzer::getsymbol(void)
150  {
151    // Initialize pointer to beginning of symbol string
152    pSymbol = (char *)&symbol;
153    do {
154      ch = toupper(ch);
155      if ( memchr( validchars, ch, NUMVALIDCHARS ) == NULL )
156        error(E_INVALIDCHAR);
157      // Note that getsymbol() reads the statement character
158      // by character until it hits endofline or identifies
159      // a valid character sequence.
160      switch (ch) {
161        case '\0' : { return token = ENDOFLINE; }
162        case ' ' :  {
163          // Ignore blanks by skipping over them.
164          while (ch == ' ') ch = get_next_char();
165          // Reset symbol to null string, no need
166          // to save the blanks.
167          pSymbol = (char *)&symbol;
168          break;
169        };
170        // if ch is a letter, then obtain a keyword
171        // or identifier.
172        case 'A': case 'B': case 'C': case 'D': case 'E':
173        case 'F': case 'G': case 'H': case 'I': case 'J':
174        case 'K': case 'L': case 'M': case 'N': case 'O':
175        case 'P': case 'Q': case 'R': case 'S': case 'T':
176        case 'U': case 'V': case 'W': case 'X': case 'Y':
177        case 'Z' : {
178          while (isalpha(ch)) ch = toupper(get_next_char());
179          // Truncate long symbols to a shorter length
180          if (strlen(symbol) > MAXSYMLENGTH)
181              symbol[MAXSYMLENGTH] = '\0';
182          // Call lookup_keyword() to see if the symbol is
183          // a keyword.  The function sets token either to a
184          // coded value of the keyword, or IDENTIFIER.
185          lookup_keyword( symbol, token );
186          return token;
187        };
188        // Read a numeric value from the input.
```

```
189       // If you wanted, you could modify this section
190       // to parse out floating point value.
191       case '0': case '1': case '2': case '3': case '4':
192       case '5': case '6': case '7': case '8': case '9' : {
193         tokenvalue = 0;
194         while (isdigit(ch)) {
195           tokenvalue = tokenvalue*10 + (ch - '0');
196           ch = get_next_char();
197         };
198         return token = INTCONSTANT;
199       };
200       // Recognize special punctuation symbols
201       case '(' : {
202         ch = get_next_char();
203         return token = LP; }
204       case ')' : {
205         ch = get_next_char();
206         return token = RP; }
207       case '+' : {
208         ch = get_next_char();
209         return token = ADDOP; }
210       case '-' : {
211         ch = get_next_char();
212         return token = SUBOP; }
213       case '*' : {
214         ch = get_next_char();
215         return token = MULOP; }
216       case '/' : {
217         ch = get_next_char();
218         return token = DIVOP; }
219       case '!' : { ch = get_next_char();
220                    if (ch == '=') {
221                      ch = get_next_char();
222                      return token = NEQOP;
223                  }
224                    else return token = NOT;
225           }
226       case '=' : {
227         ch = get_next_char();
228         return token = EQOP; }
229       case '<' : { ch = get_next_char();
230                    if (ch == '=') {
231                      ch = get_next_char();
232                      return token = LEQOP;
233                  }
234                    else
235                    if (ch == '>') {
236                      ch = get_next_char();
237                      return token = NEQOP;
```

continues

LISTING 17.8. CONTINUED

```
238                   }
239                      else return token = LESOP;
240             }
241         case '>' : { ch = get_next_char();
242                      if (ch == '=') {
243                        ch = get_next_char();
244                        return token = GEQOP;
245                      }
246                      else return token = GTOP;
247             }
248       }; // switch
249    } while (ch);
250    return token;
251  };
```

The `map` and `keywords` arrays implement a simple hashing mechanism for quickly looking up an identifier to determine whether the identifier is a keyword. In this example, a fancy table lookup is hardly needed, but this technique can easily be expanded if you wish to create a parser for a more complicated language. When the lexical analyzer spots an identifier, it calls `lookup_keyword()` (see line 94). `lookup_keyword()` uses the first letter of the identifier as a hash code. The keyword `AND`, for example, uses the letter *A*, minus the ASCII code for the letter *A*, to produce a hash code of 0. The letter *B* translates to 1, the letter *C* to 2, and so on. These values are used to compute an `index` into the `map[]` array (see line 103). If `map[`*`first letter`*`]` is zero, the identifier is not a keyword. If the `map[]` entry is nonzero, the entry value is used as the `index` into `keywords[]`. If `map[index]` is nonzero, `map[index]` is the index in the `keywords[]` array where keywords beginning with the letter *A* are stored.

A quick loop is used to compare the identifier with the keywords stored in the table (see lines 108–121). If there is no match and the symbol is greater than the table entry, `index` is decremented by one, and the comparison is made on the next lower entry in the `keywords` table. To understand how this works, run through the `lookup_keyword()` function by hand. Use `MODULUS` as the identifier to look up. Notice that when more than one keyword starts with the same letter, the keywords are stored together in the table in ascending order. The `map[]` index, though, returns the index of the last keyword having the common first letter, and the hash algorithm searches through the similar words from top to bottom.

The core of the lexical analyzer centers on the `get_next_char()` function and `get_symbol()`. `get_next_char()` (lines 78–88) extracts one character from the input line, appends it to the symbol string, and returns the character value in the `ch` variable. `get_symbol()` (lines 149–251) uses `get_next_char()` to scan through the input line one character at a time. In line 154, `get_symbol()` converts `ch` to uppercase. In this sample implementation, the conversion could be performed inside `get_next_char()`; however, if the parser is modified to recognize strings or individual character constants, you don't want these automatically converted to uppercase. For this reason, the uppercase operation is handled external to `get_next_char()` so that you can choose whether to do the conversion.

`get_symbol()` uses a `switch` statement to determine how to handle the input character (see lines 160–248). The terminating null byte (`case '\0'` in line 161) is translated into the `ENDOFLINE` token. This value is sensed by the syntax analysis stage and used to recognize that the entire line has been read. In line 162, the blank character is processed. Blanks serve only as separators, so they can safely be thrown away. The code uses a `while()` loop (line 164) to throw out groups of blanks in one fell swoop.

The lexical analyzer becomes more interesting when it recognizes identifiers, integer constants, and special symbols of the language. All alphabetic characters are trapped by the cases in lines 172–177 (remember that `ch` was converted to uppercase, so only uppercase letters need appear in the `case` statement). Line 178 continues to read characters as long as it sees alphabetic characters. When it encounters a nonalphabetic character, scanning stops. As written, identifiers must consist only of alphabetic characters. If you want, you can modify this code so that identifiers must start with an alphabetic character but can include digits or the underscore. Add a test for `isdigit()` and the underscore (_) in the condition for the `while` loop in line 178. Line 185 calls `lookup_keyword()` to determine whether the identifier happens to be a keyword. If it is a keyword, token is either set to the appropriate keyword token value such as `KAND` or `KMOD` or set to `IDENTIFIER` for nonkeyword symbols.

Numeric values are scanned similarly to alphabetic identifiers, except that the value of the integer constant is computed as the characters are scanned (see lines 191–199). `token` is set to `INTCONSTANT`, and `tokenvalue` has the value of the constant.

Next come the special punctuation symbols. Most of these symbols are quite easy. For example, lines 201–203 process the left parenthesis and set `token` to `LP`. Some punctuation symbols consist of two characters. For these, `get_symbol()`

must look ahead to the next character. For example, this simple language recognizes `!` as equivalent to the `NOT` keyword. It also recognizes `!=` as equivalent to the not-equal relational operator. Lines 219–225 show how this sequence is processed.

You can add more punctuation symbols by adding them to the `validchars[]` array and adding the appropriate code within `get_symbol()` to process the symbols. Be sure to add manifest constants to symbolically represent the token value.

SYNTAX ANALYSIS

It is easier to understand syntax analysis and the relationship between the syntax analysis code and the syntax diagrams of the language if you look at the syntax analyzer's source code. Listing 17.9 is the header file for the `SyntaxAnalyzer` class. Its only interesting public member function is `statement()`. To process a line of input, you pass the input string to `statement()`. `statement()` then copies the line to a safe place (although in this example code it does nothing destructive, so it could use the original line), and interprets the statement.

LISTING 17.9. THE HEADER FILE FOR THE SYNTAX ANALYZER CLASS.

```
 1  // SYNTAX.H
 2  #ifndef SYNTAX_H
 3  #define SYNTAX_H
 4
 5  #include "tstack.h"
 6
 7  #define NUMADDOPS 3
 8  #define NUMMULOPS 5
 9  #define NUMRELOPS 6
10  #define NUMFACTORTYPES 4
11
12  class SyntaxAnalyzer {
13  public:
14  SyntaxAnalyzer() {};
15  void statement( char * line_to_parse );
16  private:
17  virtual void factor(void);
18  virtual void term(void);
19  virtual void simpleexpression(void);
```

```
20  virtual void expression(void);
21  virtual void DoPrintStmt(void);
22  virtual void DoInputStmt(void);
23  LexicalAnalyzer Lex;
24  TStack<int> stack;
25  static short AddOps[NUMADDOPS];
26  static short MulOps[NUMMULOPS];
27  static short RelOps[NUMRELOPS];
28  static short FactorTypes[NUMFACTORTYPES];
29  };
30  #endif
```

The lexical analyzer is used only within the syntax analyzer; therefore, it is defined as the `Lex` object in line 23. Remember the generic `TStack` created in the "Class Templates" section of this chapter? It is used here to implement a stack of integers (see line 24). This stack is used to help evaluate each expression. The remaining functions in the private declaration of the class correspond to components in the syntax diagrams. You will learn more about these functions in the next section.

The syntax analyzer's code is best understood if you start at the bottom of Listing 17.10, which contains the implementation of the syntax analyzer class (Listing 17.10 is a module that will be linked with Listings 17.8 and 17.11 to create an executable program). Imagine you are about to parse this statement:

```
PRINT 3+5
```

To understand the syntax analyzer, follow this statement through the processing provided by the `SyntaxAnalyzer` class. Start at the implementation of `statement()` (lines 223–239). The lexical analyzer is initialized in line 225; this sets up the lexical analyzer to begin scanning characters starting at the letter `P` in `PRINT`. The internal stack, used to evaluate the expression, is reset in line 226; the first symbol is read from the input (line 227); and, depending on the `token` found, a function is called to continue the parsing. If the statement begins with `PRINT` (see line 230), `token` is set to the `KPRINT` constant, and `DoPrintStmt()` takes over. If the statement begins with `INPUT`, control is handed over to `DoInputStmt()`. Notice how the keyword `QUIT` causes the program to exit (see lines 236–237). If you want to recognize additional statements, you should add code to detect the appropriate keywords in this function.

LISTING 17.10. THE IMPLEMENTATION OF THE SYNTAX ANALYZER AND EXPRESSION EVALUATOR.

```
// SYNTAX.CPP
// Performs syntax analysis of the tokens
// returned by the LexicalAnalyzer.
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lexical.h"
#include "syntax.h"

short SyntaxAnalyzer::AddOps[] =
  { ADDOP, SUBOP, KOR };
short SyntaxAnalyzer::MulOps[] =
  { MULOP, DIVOP, KAND, KMOD, KMODULUS };
short SyntaxAnalyzer::RelOps[] =
  { EQOP, NEQOP, LESOP, LEQOP, GEQOP, GTOP };
short SyntaxAnalyzer::FactorTypes[] =
  { IDENTIFIER, INTCONSTANT, LP, NOT };

//====================================
// Implements the syntax of an expression's factor.
// This is the most finite component, such as a constant,
// an identifier, or another parenthetical expression.
void SyntaxAnalyzer::factor(void)
{
  if (stack.overflow()) Lex.error( E_OUTOFSTACK );
  if
  (Lex.InSet( FactorTypes, Lex.token, NUMFACTORTYPES ) == NULL)
    Lex.error(E_INVALIDEXPR );
  else {
    switch (Lex.token) {
      case IDENTIFIER: {
        // This example code does not now support
        // identifiers.  If it did, here is where you
        // should look up the identifier contained
        // in 'symbol', retrieve its value, and push
        // it onto the stack.  You could also recognize
        // keywords here, such as sin(), cos(), etc.
        // For functions, you should use the symbol
        // to determine which function to execute,
        // syntax check for "(", call expression to
        // parse the parameter, check for ")", etc.
        stack.push(0);  // Default value of 0 for now
        Lex.getsymbol();
        break;
      }
```

```
47        case INTCONSTANT: {
48          stack.push( Lex.tokenvalue);
49          Lex.getsymbol();
50          break; }
51        case LP: {
52          Lex.getsymbol();
53          expression();
54          if (Lex.token != RP)
55            Lex.error( E_RPEXPECTED );
56          else Lex.getsymbol();
57          break;
58        }
59        case NOT: {
60          Lex.getsymbol();
61          factor();         // Note the direct recursion
62          if (stack.pop()) stack.push(0);
63          else stack.push(1);
64          Lex.getsymbol();
65          break;
66        }
67      }; //switch
68    };
69  };
70
71  //====================================
72  // Handles multiplication operators
73  void SyntaxAnalyzer::term(void)
74  {
75    int saved_token;
76    int divisor;
77
78    factor();
79    // while token is a multiplication-type operator
80    while ( Lex.InSet( MulOps, Lex.token, NUMMULOPS ) != NULL )
81    {
82      saved_token = Lex.token;
83      Lex.getsymbol();
84      factor();
85      switch (saved_token) {
86        case MULOP: {
87          stack.push( stack.pop() * stack.pop() );
88          break; }
89        case DIVOP: {
90          divisor = stack.pop();
91          if (divisor == 0) Lex.error( E_DIVIDEBYZERO );
92          else
93          stack.push( stack.pop() / divisor );
94          break;
95        }
96        case KAND: {
```

continues

LISTING 17.10. CONTINUED

```
 97          stack.push( stack.pop() & stack.pop() );
 98          break; }
 99        case KMOD: case KMODULUS: {
100          divisor = stack.pop();
101          if (divisor == 0) Lex.error( E_DIVIDEBYZERO );
102          else
103          stack.push( stack.pop() % divisor );
104          break;
105        }
106      };// switch
107    };// while
108  };
109
110  //=====================================
111  // Handles the addition operators
112  void SyntaxAnalyzer::simpleexpression(void)
113  {
114    int saved_token;
115    int UnaryOp = OTHER;
116    if (Lex.token == ADDOP) {
117      UnaryOp = ADDOP;
118      Lex.getsymbol();
119    }
120    else if (Lex.token == SUBOP) {
121      UnaryOp = SUBOP;
122      Lex.getsymbol();
123    };
124    term();
125    if (UnaryOp == SUBOP)  stack.push( - stack.pop() );
126    // while token is an addition-type operator ...
127    while ( Lex.InSet( AddOps, Lex.token, NUMADDOPS ) != NULL )
128    {
129      saved_token = Lex.token;
130      Lex.getsymbol();
131      term();
132      switch (saved_token) {
133        case ADDOP: {
134          stack.push( stack.pop() + stack.pop() );
135          break; }
136        case SUBOP: {
137          stack.push( - stack.pop() + stack.pop() );
138          break; }
139        case KOR: {
140          stack.push( stack.pop() || stack.pop() );
141          break; }
142      };//switch
143    };//while
```

```
144  };
145
146  //====================================
147  // Handles the relational or conditional
148  // operators.
149  void SyntaxAnalyzer::expression(void)
150  {
151    int saved_token;
152
153    simpleexpression();
154    // While token is a relational operator ...
155    while (Lex.InSet(RelOps, Lex.token, NUMRELOPS) != NULL) {
156      saved_token = Lex.token;
157      Lex.getsymbol();
158      simpleexpression();
159      switch( saved_token ) {
160        case EQOP:  {
161          stack.push( stack.pop() == stack.pop() );
162          break;
163        };
164        case LESOP:  {
165          stack.push( stack.pop() > stack.pop() );
166          break;
167        };
168        case GTOP: {
169          stack.push( stack.pop() < stack.pop() );
170          break;
171        };
172        case LEQOP: {
173          stack.push( stack.pop() >= stack.pop() );
174          break;
175        };
176        case NEQOP: {
177          stack.push( stack.pop() != stack.pop() );
178          break;
179        };
180        case GEQOP: {
181          stack.push( stack.pop() <= stack.pop() );
182          break;
183        };
184      };//switch
185    };//while
186  };
187
188  //====================================
189  // Parses the PRINT <expression> statement
190  void SyntaxAnalyzer::DoPrintStmt(void)
191  {
192     Lex.getsymbol(); // Eat the PRINT keyword
193     expression();
```

continues

LISTING 17.10. CONTINUED

```
194     if (stack.underflow())
195        cout << "Error in expression.\n";
196     else
197       cout << stack.pop() << "\n";
198  };
199
200  //=====================================
201  // Parses the INPUT <identifier> statement
202  void SyntaxAnalyzer::DoInputStmt(void)
203  {
204    int value;
205    Lex.getsymbol(); // Eat the INPUT keyword
206    if (Lex.token != IDENTIFIER)
207      Lex.error(E_INVALIDEXPR);
208    else {
209      // This implementation does not now support
210      // identifiers.  If you wish to add them, you
211      // should add 'symbol' to a symbol table, and
212      // set its value to 0.
213      cout << "? ";
214      cin >> value;
215      // Insert 'symbol', 'value' into symbol table
216      Lex.getsymbol();
217    };
218  };
219
220  //=====================================
221  // Initializes the lexical analyzer and begins
222  // parsing 'line_to_parse'.
223  void SyntaxAnalyzer::statement( char * line_to_parse )
224  {
225    Lex.init_lexicalyzer( line_to_parse );
226    stack.reset();
227    Lex.getsymbol();
228    if (Lex.token != ENDOFLINE)
229    {
230      if (Lex.token == KPRINT)
231        DoPrintStmt();
232      else
233      if (Lex.token == KINPUT)
234        DoInputStmt();
235      else
236      if (Lex.token == KQUIT)
237        exit(0);
238    };
239  };
```

Moving up to DoPrintStmt(), getsymbol() is called to bring in the next symbol. The PRINT keyword is disregarded because it is now unnecessary. It is used only to indicate that this is a PRINT statement. This sets the current token to INTCONSTANT and tokenvalue to 3. Next, because PRINT should be followed by an arithmetic expression, the expression() function is called. Upon return from parsing a valid expression, the topmost value on the internal stack is displayed as the result.

The syntax analysis gets much more interesting inside expression(). expression() immediately calls simpleexpression(), which in turn effectively calls term(), which calls factor(). The parser is now deep inside a series of nested function calls. At factor(), after checking for a potential out-of-stack-space error condition (see line 26), processing of the expression begins. Look at lines 46–50. Here the INTCONSTANT token is recognized. The result of this is to push the constant 3 onto the internal stack and read the next symbol, the + symbol. factor() returns to term() (at line 80). Because the + symbol is not a multiplication operator, term() exits back to simpleexpression(). simpleexpression() recognizes the addition operator (at line 128, the ADDOP token that represents + is found in AddOps). The ADDOP is saved into saved_token, a new symbol is read (the constant 5), and term() is called again. As before, term() calls factor(), which sees the constant 5, and the result is a push of 5 to the stack. The stack now contains:

```
5
3
```

with 3 on the bottom and 5 on top. Again, factor() exits to term() and term() exits to simpleexpression(). Inside simpleexpression(), the ADDOP operator is processed (see lines 133–135). The top two values on the stack are popped and added together, producing 8, which is pushed back on to the stack. The stack now contains:

```
8
```

simpleexpression() returns to expression(), which does not see any relational operators and therefore exits. This takes control back to DoPrintStmt() where the value on the top of the stack is printed, displaying the answer:

```
8
```

For practice, you might try following the code by hand using other types of expressions, such as 3*5, or 3+4*5. Parenthetical expressions, such as 3*(4+5) are handled internally to factor(). The code in lines 51–58 processes the left

parenthesis by disregarding the (symbol and reading the next token, and then, interestingly, calling `expression()` to process the subexpression in the parentheses. Notice that because `expression()` is already at the top of the calling chain, the result is a recursive call. Upon return from `expression()`, the right) parenthesis should be the next symbol.

If you own a reverse-Polish-notation calculator, you might recognize the steps taking place here. As the expression is broken apart into smaller pieces, the values and intermediate results are pushed onto a stack. Arithmetic operators are applied to the top two elements of the stack, leaving a single resultant value in their places. In this fashion, the syntax analyzer interprets the expression and produces a result. A compiler would behave differently here. Rather than evaluating the expression during parsing, the compiler emits instructions that, when executed on the computer, evaluate the expression.

If the parser supported variable identifiers (it recognizes them but doesn't actually do anything with them), it would place the identifiers into a symbol table. In the case of the interpreter, the symbol table would hold the symbol and its current value. In `factor()`, the symbol would be looked up in the symbol table and its present value pushed onto the stack (see `case IDENTIFIER`, lines 32–46). You would also need to add an assignment statement capability, to process statements like:

```
TOTAL = 100
```

To handle this statement you would need to detect `IDENTIFIER` inside `statement()`. For practice, you might want to try adding the code needed to support this feature. After spotting the `IDENTIFIER` token, look for the `EQOP` token, and call `expression()` to process the result. Like the `PRINT` statement, upon return from `expression()`, the top of the stack holds the value of the expression (in the preceding example, 100). This value is then associated with `TABLE` and stored into the symbol table. To simplify creation of the symbol table, you probably want to look up each symbol within `getsymbol()`. If the symbol is an identifier, add it to the symbol table when you first see it.

The parse.cpp program, in Listing 17.11 shows how the syntax and lexical analyzers are used. You can use this short program as a test bed if you choose to make modifications to the parsing code. To compile and link, create a project file containing syntax.cpp, parse.cpp, and lexical.cpp.

LISTING 17.11. A SAMPLE PROGRAM SHOWING HOW TO CALL THE SYNTAX ANALYZER TO PARSE AN EXPRESSION.

```
// PARSE.CPP
// Demonstrates how to parse complex expressions
// and statements using lexical, syntactic, and
// semantic analysis.
#include <stdio.h>
#include "lexical.h"
#include "syntax.h"

void main (void)
{
  char input_statement[MAXSTMTLENGTH];
  SyntaxAnalyzer Interpret;

  printf("Simple Expression Calculator\n\n"
         "Demonstrates a simple parsing technique.\n\n");

  printf("Enter statement (type QUIT to exit):\n");

  do {
    printf("> ");
    gets(input_statement);
    Interpret.statement(input_statement);
  } while (1); // Syntax analyzer handles QUIT to exit
}
```

BORLAND C++ MATH OPTIONS

Borland C++ provides several data types and libraries designed to support every type of mathematical operation needed by your applications. You are already familiar with basic floating-point operations using the `float` or `double` data types. If you have a math coprocessor (8087, 80287, 80387, or the combined CPU plus math processor in the 80486), you can use Borland C++ features that greatly speed up the computation of floating-point numbers. In addition, C++ programmers have access to two predefined classes that implement arithmetic in *binary-coded-decimal* (BCD) format and complex number arithmetic. BCD is especially valuable for computations involving money, such as dollars and cents, because BCD numbers do not have the problems asociated with rounding typical of the floating-point data types.

By default, the floating-point code generated by the Borland C++ compiler works on all types of PC systems, either with or without a math coprocessor. The code supports both systems through a special emulation library. At runtime, if the program detects the presence of a math coprocessor, the coprocessor is used and your application obtains all the speed benefits of the coprocessor. If no math coprocessor is available on the system, the program automatically calls special emulation library routines. The emulation library simulates the hardware coprocessor, providing the same functions and identical numeric formats. Of course, because the emulation library is in software, it runs more slowly than the coprocessor. You choose the floating-point operation mode using the IDE menu selection Option | Compiler | Advanced Code Generation dialog box. Beneath the heading Floating Point, you can select None, Emulation, 8087, or 80287/387. Select Emulation mode (the default setting) if your program uses floating-point operation and must run on both types of systems. If using bcc, use the `-f` command line switch to select the emulation library.

If you use the emulation library, you might occasionally want to disable the detection of a math coprocessor. For instance, on an old IBM PC I once had, the CPU board had a switch selection to indicate the presence or absence of a math coprocessor. The switch was set incorrectly, and I never bothered to fix it until I sold the computer to someone else. (Obviously, I didn't use it for number crunching, or I would have reset the switch!) Anyway, any software that detected and tried to use an 8087 coprocessor got very upset when run on that old PC. You might want to disable the use of the coprocessor in order to test performance or debug a problem. You can easily control all use of the math coprocessor by setting an environment variable named `87` to either `Y` or `N`. Setting:

```
C:> SET 87=N
```

tells the Borland emulation routines that you don't want to use the math coprocessor. Setting `87` to `Y` indicates that you do want to use the coprocessor.

If your application uses no floating-point operations whatsoever, you can choose the None, or `-f-`, compiler option. This just tells the linker that it doesn't need to look in the math libraries, producing a tiny saving in the linker's execution time. If you leave the Floating Point option in Emulation mode, the linker will read through the libraries, but it won't actually bring in extra code. You cause no harm nor increased code size by leaving the setting in Emulation mode.

The default floating-point emulation library provides your application with the greatest flexibility. Depending on your program's requirements, however, the floating-point emulation library might not be the best solution for your application. Borland provides three additional floating-point calculation options that enable you to fine-tune the floating-point code generation.

If you are certain that your application runs only on systems with a math coprocessor, you can tell the compiler to produce code that works only when the coprocessor is present. Your application benefits by eliminating the emulation library, and you save nearly 10K of code. The disadvantage is that if no coprocessor is found, the application won't run. To select math coprocessor support in the IDE, select either the 8087 or 80287/387 floating-point option on the Advanced Code Generation dialog. When using BCC, use `-f87` (to select 8087 support) or `-f287` (to select 80287 or 80387 support).

Borland C++'s fast floating-point option lets the compiler ignore some of the standard rules regarding conversion of different floating-point formats (`float`, `double`, `long double`) during certain calculation sequences. The name fast floating point is a bit of a misnomer because it merely speeds up calculations that involve conversion from a higher precision format to a lower precision format. For many calculations, this option has little or no impact on the speed of your programs. Because the relaxation of the conversion rules has little impact (though it can cause some loss of accuracy problems), the compiler's default selection is to generate fast floating-point code. You can set (or override) the fast floating-point option using the Fast Floating Point option beneath the Options heading in the Advanced Code Generation dialog box. For the command line compiler use `-ff` to turn off the fast floating-point option.

Table 17.1 summarizes the floating-point options.

TABLE 17.1. FLOATING-POINT OPTIONS SUMMARY.

IDE option	*Bcc option*	*Usage*
None	`-f-`	Program does not use floating-point arithmetic.
Emulation	`-f`	Default emulation library
8087	`-f87`	Requires 8087 coprocessor

continues

TABLE 17.1. CONTINUED

IDE option	*Bcc option*	*Usage*
80827/387	`-f287`	Requires 80287 or 387
Fast Floating	`-ff`	Relax conversion rules.

BCD DATA TYPE

C++ programmers (but not C programmers) can use the Borland-provided `bcd` class. BCD arithmetic can help overcome the rounding-off errors inherent in using the standard floating-point representations of numbers. The problem with floating-point numbers is that the binary number system can't exactly represent many values. Most of the time, the inaccuracy is hidden many digits to the right of the decimal point and is never seen. However, if you perform repeated calculations on such numbers, the rounding-off error can progressively become so large that it begins to affect the accuracy of your results. If you run the following short program, you can see the effect of this rounding-off problem:

```
#include <stdio.h>
void main(void)
{
  float Total=0.0;
  float Factor = .05;

  for (int i=1; i<=100; i++) Total = Total + Factor;

  printf("%g\n", Total);
  printf("%g\n", Total -  5.0 );
};
```

The output from this program produces:

```
5
9.53674E-07
```

The reason for the very small second number is rounding-off error. The value of `Total` displays as 5 but is actually a tiny fraction greater than 5.0. When a true 5.00000000 is subtracted from `Total`, the discrepancy is made apparent.

The rounding-off errors of binary-based floating-point numbers are particularly noticeable in financial software applications. For this reason, spreadsheets and accounting software almost always use—or at least have available—a BCD format for their internal numeric representation. The advantage is cleaner representation of decimal numbers, especially those involving money. In the BCD format, numbers up to about 17 digits are stored in base 10, using 4 bits for each decimal digit. The maximum range extends from about 1×10^{-125} to $1 \times 10^{+125}$. Arithmetic operations are performed in much the same way you manually add, subtract, multiply, and divide numbers using pencil and paper. The result is much more reliable representation and computation of decimal numbers.

To use `bcd` numbers, you must `#include<bcd.h>`, which contains the `bcd` class definition. To define a variable of type `bcd`, use the standard syntax, as shown in this example:

```
#include <stdio.h>
#include <bcd.h>
void main(void)
{
  bcd Total=0.0;
  bcd Factor = 0.05;

  for (int i=1; i<=100; i++) Total = Total + Factor;

  printf("%g\n", (double)real(Total));
  printf("%Lg\n", real(Total -  5.0) );
};
```

Notice that you can't use a `bcd` variable directly in a `printf()` statement (however, you can use `bcd` numbers with `cout`). To use a `bcd` variable in `printf()`, use the `bcd` friend function `real()` to convert the `bcd` number into a `long double`. Notice the use of `(double)` to recast the `real()` return result in the first `printf()` statement shown in the preceding example.

If you run this program, you might be surprised at the result. The second `printf()` displays:

1.1969e–16

This problem is in the conversion arithmetic from `bcd` to `long double`. Unfortunately, Borland did not do a complete implementation of the `bcd` class. Internally, they do frequent conversion to `double`, perform the calculation, and then convert back to BCD format, thereby losing the benefit of the `bcd` numeric type.

SPECIAL SITUATIONS USING *BCD* ARITHMETIC

You can convert `bcd` format numbers to other nemeric formats and set the number of decimal places of accuracy. These and other topics are noted here:

- Use the `real()` friend function to convert `bcd` format into `long double`. If you want to convert `long double` to either `double` or `float`, use a recast operator, such as `(float) real(...)` or `(double) real(...)`.
- The number of decimal places of accuracy can be set when defining a `bcd` object. To set the accuracy, directly call the `bcd` constructor using this format:

  ```
  bcd Total = bcd(initial_value, decimal_places);
  ```

  For example:

  ```
  bcd Total=bcd(0.0, 2);
  ```

  initializes `Total` to zero with two decimal places of accuracy.
- When values have greater accuracy than can be represented in the number of decimal places provided, they are rounded.
- You can use any of the standard library functions such as `sin()` and `cos()`. The `bcd` class provides overloaded function equivalents for `bcd` type expressions.
- `bcd` values have rounding-off errors too. Obviously, there is no way to represent irrational numbers such as 1/3 or pi.

COMPLEX DATA TYPE

A special `complex` class, with implementation similar to the `bcd` class, is also available. To use the `complex` class, `#include<complex.h>`. You can initialize a complex value by writing:

```
complex a(10,5);
```

In this example, `5` is the imaginary part of the complex number. You can use `complex()` also as a recast function:

```
complex a;
...
a = complex(1,5);
```

The first argument corresponds to the real part and the second to the imaginary component of the complex number. If you omit the imaginary part (`complex(1)`), the imaginary part defaults to zero.

All the basic arithmetic functions (`+`, `-`, `*`, and `/`) and many of the math functions are overloaded to support the complex data type. You can test for equality and inequality, but you can't make less-than or greater-than comparisons.

You can extract the real value portion of a complex number using the overloaded friend function `real(complex number)`. To extract the imaginary part, use the overloaded function `imag()`. Other friend functions are available, including `polar(distance, angle)`, to convert a polar coordinate, specified by `distance` and `angle`, to a complex number. See complex.h for a complete list of all member functions and overloaded functions.

APPENDIX

SOURCES FOR SOFTWARE TOOLS, UTILITIES, AND LIBRARIES

Magazine sources

Mail-order sources

Shareware and freeware

Many development tools, such as utility programs and libraries, can help you get your job done more quickly and with higher quality. This book's companion disks contain a number of freeware and shareware utility programs that can help you get your jobs done a bit easier. Numerous commercial products are also available to aid the professional programmer. Unfortunately, you may not have easy access to all the latest software gadgets. Unless you live in Silicon Valley, your local computer dealers may not stock the latest software development tools and utility software. If you cannot find

local sources for the software tools that you need, you may want to turn to mail-order and online sources to obtain specialized software. This section provides a modest list of software distributors, both shareware and commercial, and online information services.

Please note that any reference to a publisher, distributor, or online service in this book does not represent an endorsement or recommendation to do business with the company. The lists of companies are not intended as an exhaustive list of all possible businesses, but are presented only for your convenience. Where we highlight specific products (some of the chapters provide introductions to specific commercial and shareware development tools that we believe can improve your productivity), you can be assured that we have used these products and we believe they add value to your software development efforts.

MAGAZINE SOURCES

The technical computer magazines shown in Table A.1 are good sources of information about the latest development technologies.

TABLE A.1. SUGGESTED PC TECHNICAL COMPUTER PUBLICATIONS.

Publication	*Telephone Number*
C++ Report	1-212-274-0640
Computer Language	1-303-447-9330*
Dr. Dobb's Journal	1-303-447-9330*
Inside Turbo C++	1-800-223-8720

Publication	*Telephone Number*
Journal of Object-Oriented Programming	1-212-274-0640
PC Techniques	1-602-483-0192

** Subscriptions to Computer Language and Dr. Dobb's Journal are handled by a central clearinghouse; however, the two publications are independent of one another.*

In these magazines, you will find not only technical articles describing the latest developments, but advertisements for many of the technical tools that you will not find anywhere else.

MAIL-ORDER SOURCES

Several mail-order companies specialize in software development tools (see Table A.2). For a catalog, call the number shown.

TABLE A.2. MAIL-ORDER COMPANIES THAT SELL SOFTWARE DEVELOPMENT TOOLS.

Mail-Order Company	*Telephone Number*
Programmer's Paradise	1-800-445-7899
Programmer's Warehouse	1-800-323-1809
The Austin Code Works	1-512-258-0785
The Programmer's Shop	1-800-421-8006

SHAREWARE AND FREEWARE

Freeware is software that is literally free and is being given away. Many programmers produce collections of routines or small tools that they believe are of interest to other programmers, but that do not justify the complexities of trying to sell them. So they give the tools away. In some cases, freeware is quite

sophisticated. For instance, some applications have been written as part of publicly sponsored research projects. Because these applications were funded by taxpayers, the results are available to all of us at no charge.

Shareware is *not* freeware. Shareware is software that has been professionally produced and is downloadable from an online library for your examination. Through shareware, you are given the opportunity to try it before you buy it. If you like the shareware software and find that you will be using it, you should pay for the software by sending the requested purchase price to the distributor or author. Purchase and registration of your shareware software usually entitles you to additional software features, newer versions, and printed documentation.

Shareware provides consumers with a wide variety of affordable software, tools, utilities, and libraries. Of special importance to programmers is that shareware provides a distribution mechanism for software tools that would not ordinarily be distributed through retail channels because of their limited sales volume (for obvious reasons, retail stores prefer to stock only top-selling items). Shareware, then, provides a wide assortment of tools and libraries that would not otherwise be available to us. Best of all, through the shareware system, each of us can give the software a detailed examination before purchasing it. Shareware won't last forever, however, unless we as consumers reward the authors who contribute their talents and time to provide us with this unique try-before-you-buy distribution mechanism. Several sharware programs are distributed on the diskette include with this book so that you can give these programs a quick check before you buy.

You can obtain shareware and freeware software through a variety of outlets. Disks are available from mail-order firms that sell software by the disk, typically at about $5 per disk. Large collections of software are also available on CD-ROM. The mail-order firms often run their own bulletin board systems (BBSs), providing access to their libraries electronically. A few libraries are even available through the online services. If you purchase disks of shareware, the price of the disk does not excuse you from paying a shareware fee to the author. When you buy shareware by the disk, your $5 goes to a business that copies and distributes the disks; the author of the software receives nothing.

The online services and many local bulletin board systems also provide access to enormous libraries of contributed software. The online services are convenient because you can search their libraries electronically, and they are more

likely to have the latest software. On many occasions, when I needed a particular utility or routine *immediately*, access to an online database provided me with a solution quickly and inexpensively. However, downloading via modem can be both time-consuming and expensive, particularly for large applications or libraries. Regardless of how you choose to obtain shareware and freeware—mail-order disks, CD-ROM, or online—you will benefit greatly by accessing the programming libraries that are available to you.

The following sections provide contact information for distributors of shareware and freeware. Many online services, such as CompuServe and BIX, are accessible from international locations. Not shown, but not to be overlooked, is your locally run bulletin board system. BBSs are run by computer enthusiasts, and sometimes by local stores or computer clubs, to provide online community forums. Access is usually available free, although donations are always welcome. Many systems are linked by private telephone lines to form networks, such as the respected FidoNet.

Most BBSs include libraries of online information relevant to the interest areas of their users. In other words, BBSs might include programming tools, collections of online artwork, or technical support information, depending on the needs of the BBS's users. Some BBSs include access to CD-ROM libraries, because a single online CD-ROM can provide access to 500 megabytes of data. If you do not know of any BBS systems in your area, ask other computer owners or check with local computer stores or computer clubs. Many high schools and colleges have groups of computer enthusiasts that usually know everything there is to know about the local BBS scene. Some areas, particularly where there is a high concentration of technology-related jobs, have local special-interest computer publications. In Seattle, for instance, see the *Puget Sound Computer User*, or in Silicon Valley see *Computer Currents*. These publications include a listing of all local BBSs and the telephone number at which you can contact the BBS.

MAIL-ORDER SHAREWARE AND FREEWARE DISK DISTRIBUTORS

The Software Labs, Inc.
1-800-359-9998

Public Brand Software
1-800-426-3475

CD-ROM DISC DISTRIBUTORS

Bureau of Electronic Publishing, Inc.
1-800-828-4766

CD-ROM, Inc.
1-303-231-9373

SELECTED ONLINE SERVICES AND LIBRARIES

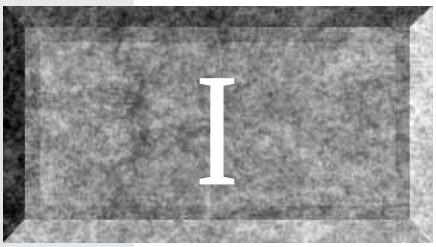
America Online
8619 Westwood Center Dr.
Vienna, VA 22182
1-800-827-6364

BIX
General Videotex Corporation
1030 Massachusetts Avenue, 4th Floor
Cambridge, MA 02138
1-617-354-4137
1-800-695-4775

CompuServe
P.O. Box 20212
Columbus, OH 43220
1-800-848-8990 (U.S. only)
0800 289 458 (U.K. only)

GEnie
401 N Washington St.
Rockville, MD 20850
1-800-638-9636

Prodigy Services (see Ziffnet offering)
445 Hamilton Ave.
White Plains, NY 10601
1-800-PRODIGY



INDEX

SYMBOLS

#pragma hdrstop statement, 48
* (pointer component), 184
-? MAKE utility command-line switch option, 83
> (DOS redirection operator), 183
@ symbol, 71
\ (backslash) character, 67, 686-688
\ \ (backslash, double), 181
_argc system variables, 193
_argv system variable, 193
_chdrive() function, 181
_chmod() function, 178
_dos_allocmem() routine, 149
_fullpath() function, 172
_searchenv() function, 186
_splitpath() function, 171
~ (tilde) character, 30
4PRINT utility, 97

A

-a MAKE utility command-line switch option, 81
abs() function, 648
abstract classes, 204
accelerators
    international compatibility, 502-503
    resources, 502-503
    translating, 502
access rights, 179
access() function, 179
accessing
    command-line parameters, 193-194
    files
        controlling, 110
        directories, 183-185
    revisions, 116
activate() command, 257
activating TSRs, 542
active functions, tracking list of, 388
Add command, 107

Add Watch variable dialog box, 389
add-in boards, IRQs, 605
adding
  files
    to archives, 113-115
    to libraries, 107
  items to Transfer menu, 30
  variables to Watch windows, 398
addresses
  bits, 124
  converting, 131
  distinguishing from values in assembly language program, 474-475
  I/O, UART, 600-601
  initializing pointers, 133
  segment registers, 125
  segment:offset, 129
addressing
  far, 126
  memory, 124-126
  near, 126
Advanced Code Generation dialog box, 51
affine transform, 237
algorithms, changing to improve program speed, 441-447
alloca() function, 149
allocating
  arrays
    integers, 147
    multidimensional, 160
    new operator, 160
  buffers, 145
  memory, 143, 159-160
    blocks, 149-151
    discarding, 143, 144
    DOS, 149-150
    dynamically, 141-142
allocation errors, trapping, 160-161
allocations, preinitializing, 159
allocmem() routine, 149
Alt-key sequence, 497
amplitude, 339
analog-to-digital converter (a-to-d converter), 339
Animate... function, 403
ANSI violations warnings, 42
APIs (Application Programming Interfaces), 525
applications
  developing file control problems, 103-105
  inserting dialog boxes, 201
  SideKick, 535
arbitrary data files, managing, 105
arc() call, 243
archives, 110-111
  adding files, 113-115
ARG assembler directive, 486
arithmetic operators, 133
arrays
  allocating
    integers, 147
    multidimensional, 160
    new operator, 160
  indicing, 380-381
  scanning characters, 651-652
  specifying number of items, 147
ASCII files
  managing ASCII source files, 105
  tracking version histories, 106-109
ASliceOfAmerica() function, 357
asm keyword, 471-472
assembly language, 467-469
  arithmetic operations, 480-482
  constant identifiers (macros) versus variables, 476
  CPU's instruction set, 469-470
  functions, calling, 472-473
  global variables, accessing, 473-474
  jump instructions, 479
  local variables in functions, 476
  pass-by-reference parameters, accessing, 477-478

pass-by-value parameters, accessing, 477-478
procedures, calling, 472-473
statement labels, 479
structure components, accessing, 478-479
sum function, 483-484
Turbo Assembler, 482-488
running TASM, 487-488
statements, 483
sum function, 483-484
values and address, distinguishing between, 474-475
writing code with built-in assembler, 470-472
assert() function, 163
assert() macro, 227-229
assert.h. assert() header file, 399
ATTIC shareware program, 105-109
attrib parameter, constant values, 177
autoexec.bat file, international compatibility, 498
automatic duration variables, 139

B

-b MAKE utility command-line switch option, 81
Back trace function, 403
backslash
character (\ ), 67
double (\ \ ), 181
bag container library, 204-206
bar charts, drawing, 306-318
bar() function, 306-314
batching commands, 77
baud rate, 598
divisor, 621
bcc command-line compiler, 44, 138
bcd
arithmetic, 680
data type, 679-680
BCX protected-mode (Borland C++ 2.0), 57
.bgi files
converting into .obj files, 330-331
linking with .chr files, 331-335
binary files, 105
binary-coded-decimal (bcd) format, 675
BIOS
COM vector, 537
data area, video status information in, 565-566
Print vector, 537
bit mask
constants, file attributes, 178, 186
values, chmod() function, 179
bitmaps, 244
international compatibility, 502
resources, 502
translating text, 505
bits
addresses, 124
registers, 124
UARTs, 607-608
BIX General Videotex Corporation, 688
block
copy bytes, 164
devices, 593
blocks, memory, *see* memory blocks
Borland C++
2.0
BCX protected-mode, 57
compatiblity with DOS 5.0, 57-58
compatibility with Microsoft C++, 466-467
loading, 15-16
Borland Graphics program, sample, 267-270
Borland Integrated Development Environment, *see* IDE

Borland International Data Structures (BIDS) library, 214
Break point byte instruction, 527
Breakpoint Modify/New dialog box, 392
breakpoints, 385, 391-394
  Turbo Debugger, 403-404
    inserting executable expressions directly before, 405-406
    setting conditional breakpoints, 405
    viewing, 406
Breakpoints | At... command, 403-404
Breakpoints | Changed Memory Global... command, 405-406
Breakpoints | Expression True Global... command, 406
Breakpoints dialog box, 392
BRIEF (Epsilon Programmer) editor, 36
buffers
  allocating, 145
  extracting portions, 651
  scanning characters, 650
built-in assembler, 470-472
  arithmetic operations, 480-482
  jump instructions, 479
  operators, 481-482
  statement labels, 479
builtins.mak file, 76
bypassing version control systems, 104
bytes
  converting to paragraphs, 150
  copying block, 164

C

C++ Report magazine, 684
C++ warnings, 43
CalcMinAndMax() function, 317
call stack, 141
Call Stack window (IDE debugger), 388
calling functions as near procedures, 138
calloc() routine, 147-148
calls
  arc, 243
  draw(), 243
  ellipse(), 243
  filled(), 243
  pieslice(), 243
  read(), 257
Cannot Load COMMAND.COM error, 539
$CAP EDIT instruction macro, 34
$CAP MSG() instruction macro, 34
case-sensitive comparisons, 516
casting operators,
  effect on expressions, 390
CD-ROM Disc Distributors, Bureau of Electronic Publishing, Inc., 688
CD-ROM, Inc., 688
central processing unit, *see* CPU
CGA, 245
chaining interrupts, 534, 538, 545-551
character
  arrays, scanning, 651-652
  buffers, scanning, 650
  devices, 593
  sets
    code pages, 495-496
    extended, 516
    foreign languages, 492-495
  sizes, selecting, 274-276
characters
  Alt-key sequence, 497
  identification routines, 515-516
  foreign languages (keyboard), 496-497
chart class, 249-251
chart drawing class, 248-249
charting, 296-306
chdir() function, 180-181

Check Install function, 552
check-in and check-out mechanism, 110-111
  maintaining ownership of files, 103-105
checking out files, 108-109, 115
CHK_INSTALL subfunction, 553
chmod() function, bit mask values, 179
Chord class, 356
chords, creating, 355-356
  in true polyphony, 360
    generating PCM data, 360-362
    playing PCM data, 362-365
.chr files
  converting into .obj files, 330-331
  linking with .bgi files, 331-335
chunks (songs), 362
circle() function, 272
class
  definitions
    generic, 640
    TStack, 642
  libraries
    borrowing from, 203
    container, 202-211
    interface, high-speed, 610-635
  templates, 640-650
    implementing operations, 647
    variable parameters, 647
classes, 214
  abstract, 204
  avoiding debugging code, 227-229
  chart, 249-251
    drawing, 248-249
  Chord, 356
  creating templates, 640
  DDCMP, 614-616
  deciding future use, 217-220
  defining, 215-217
  designing class interface, 214-215
  documenting, 223-224
  ezfile, 257
  fixedpoint, implementing fixed-point operations, 449-452
  gfxstyle, 240-241
  graphics, 248-252
  intelligent mouse classes, 245
  keeping concise, 220
  LexicalAnalyzer, 658-666
  line() device class, 241
  mouse_cursor, 252
  mouse_event, 253
  naming, 220-221
  object, creating from templates, 647
  operator overloading, 222-223
  PcmFile, 361-362
  PcmNote, 361-362
  pixarray, 244
  restricting access, 225-227
  scaler, 238-239
  Song, 357-360
  stacks, 641
  standard idioms, 222
  testing, 229-232
  TLogEntry, 644
  Turbo Vision library, 197
  viewport, 235, 242
  writing, 203
Clear Interrupt flag (CLI) instruction, 543
close() function, 384
closedir() function, 183
closegraph() function, 328
codes
  C
    choosing library functions, 464-466
    data types, acceptable range of values, 462-464
    general guidelines, 461
    header files, 467-469
    importing into Turbo Pascal programs, 456-460
    increasing portability, 460-461

Make files, obstacles with, 466-467
C++
choosing library functions, 464-466
data types, acceptable range of values, 462-464
general guidelines, 461
header files, 467-469
increasing portability, 460-461
Make files, obstacles with, 466-467
derived byte count block, 621
floating-point, 676
pages, 492-495
changing, 497-499
character sets, 495-496
fonts, 495-496
multilingual, 493
Windows, 493
returned error return codes, 384
stabilizing, 371-372
walking through, 373-374
$COL state macro, 32
collation sequences, 517-518
color
choosing active color in graphics, 284-290
in graphics, 245-247
gamma of monitors, 247
palette, 281-283
precisely specifying color amount, 284
.com files, 138-143
COM port vectors, 537
COM1 port controller, 529
COM2 port controller, 529
combining segment registers, 125
command-line
compilers
bcc, 138
compiling Turbo Debugger-compatible programs, 396
setting memory models, 130
switches, 49-56
interface, 105
MAKE utility, 71
options
GREP utility, 86-87
MAKE utility, 81-83
TDW (Turbo Debugger for Windows), 416-417
parameters, accessing, 193-194
switches, memory model selection, 130
commands
activate(), 257
Add, 107
batching together, 77
Breakpoints | At..., 403-404
Breakpoints | Changed memory global..., 406
Breakpoints | Expression True Global..., 406
Data | Add watch..., 399
Debug | Evaluate/Modify..., 390
Debug | Toggle Breakpoint, 392
Debug | Watches | Delete Watch, 390
Debug | Watches | Edit watch...., 390
Debug | Watches | Remove, 390
Debug | Watches... | Add watch..., 389
DOS ATTRIB, 104
fcbs, 12
File | Open, 423
File | Properties..., 417
GET, 112
Go to cursor, 402
install, 10
Options | Compiler | Optimizations..., 424

Options | Environment... | Preferences, 387
Options | Macros, 410
Options | Macros | Delete all, 411
Options | Macros | Remove..., 411
Options | Macros | Stop recording, 411
Program reset, 402
PUT, 113-116
relative drawing, 280
Run, 402
Run | Go to cursor, 392
Run | Program, 411
Run | Step over, 388
Run | Trace into, 388
Save | Modify TDW.EXE, 417
Select, 108-109
Step over, 402
Toggle Breakpoints, 392
TOUCH, 84
Trace into, 402
Until Return, 402
View | Hierarchy, 402
View | Windows messages, 416
Watches, 399
Window | Next, 389
Window | Previous, 389
Window | Tile, 388
Window | User screen, 388
Xtract, 108-109
compact memory models, 127-128
comparing
identifiers with keywords, 664
objects, 208
strings, 516-517
compatibility
international, 490-510, 520-521
character support, 514-518
output, 511-514
translator quality, 520
Windows Control Panel, 518-519
Compile menu, 42
compilers
command line, *see* command-line compilers
compatibility, 194
compiling, 209
IDE debugger, 385-386
improving speed of, 41
/s- switch (BC command line), 46
/x option in extended memory, 43-44
BCC command-line compiler, 44
disabling compiler optimizations, 45
disabling display of warning messages, 42-43
internal optimizer, 49-56
optionally including #include header files, 48
precompiled headers, 46-48
Turbo Debugger-compatible programs, 397
Turbo Profiler for compatibility, 424
complex data type, 681
compound waveforms, 339-341
CompuServe, 688
ComputeInvestmentYield() function, 382
Computer Language magazine, 684
concatenating strings, 171
conditional
breakpoints, 403
evaluating true expressions, 405
conditional directives, 72-73
MAKE utility, 78-80
structuring conditional expressions, 438
Conditions and actions dialog box, 403
$CONFIG state macro, 32

config.sys file, 9
  international compatibility, 498
configuration files, converting, 93-94
configuring
  DR DOS, 10-11
  mouse for left-hand uers, 41
  MS-DOS, 8-10
  UARTs, IRQs, 606
constant values, attribute parameter, 177
container libraries, 202-211
  bag, 204-206
  SortedArray, 209-211
  templates, 203
Control Break vector, 531
Control Panel (Windows), International settings, 518-519
Control-C vector, 536
controlling files, ownership, 103-105, 110
converting
  addresses, 131
  bytes to paragraphs, 150
copying
  bytes, block, 164
  transfer menu items into new project files, 95
CopyTemplate routine, 636
coreleft() function, 146
coreleft() function, 377
cosine function, 360
country-specific languages, 491
coverage analysis, 435
.cpp extension (source files), 143
.cpp file extension, 159
CPP utility, 60-65
  sample preprocessed program, 62-64
CPU (central processing unit)
  displaying register values, 388
  instruction set, 469-470
  int instruction, 526, 534
  ports, 526
  registers
    80x86, 122-124
  selecting to ensure peak productivity, 4
creating
  .com files, 138
  chords, 355-356
    in true polyphony, 360-365
  class templates, 640
  filenames, 173
  files, temporary, 173-174
  musical notes, 352-353
  note strings, 354
  object classes from templates, 647
  pointers
    to arrays, 147
    to specific locations, 133-134
  songs, 357-360
    chunks, 362
  sound effects, 351-352
  stacks (template), 643
  subdirectories, 180
    for projects, 111-113
creattemp() function, 174, 177-178
Critical Error vector, 536
CRT vertical retrace interval, 529
Ctrl-Break, 196-197
ctrlbrk() routine, 196
currency formats, 507-509
  symbols, 507-508
current
  pointer, 280-281
  points, 318
  selection, 109
custom-design editor, 285-290
customizing
  IDE editor, 36
    with Turbo Editor Macro Language (TEML), 36-38
cycles, 339

D

-d command-line compiler option, 394
d.xmax() function, 235
d.ymax() function, 235
DASS.EXE program, 365-366
DASSMAIN module, 366
Data | Add watch... command, 399
data
  compression, 24-26
  files, arbitrary, 105
  register, 602
  segments, sharing, 129
  transfer, 599
  types
    complex, 681
    simple, 159-160
    using long instead of float, 429
data() member, 249
dates, formats, 491, 509-510
DDCMP class, 614-616
-d*directory* MAKE utility command-line switch option, 81
ddm_allocate() function, 151
deallocating DOS memory, 150
Debug | Evaluate/Modify... command, 390
Debug | Toggle Breakpoint command, 392
Debug | Toggle breakpoint menu option, 387
Debug | Watches | Delete Watch command, 390
Debug | Watches | Edit watch.... command, 390
Debug | Watches | Remove command, 390
Debug | Watches... | Add watch... command, 389
Debugger options dialog box, 385
debugging
  adding extra program statements, 370
debugging in View | CPU mode, 418
IDE debugger, 387-388
  breakpoints, 385, 391-393
  Call Stack window, 388
  compiling, 385-386
  Registers windows, 388
  Watch windows, 388-389
  windows, 388
IDE's built-in debugger, 370
maintaining modification histories, 372
printf() statements, 394-395
Turbo Debugger, 370, 395
  at assembly language source level, 406
  breakpoints, 403-406
  compiling for compatibility, 396
  debugging TSRs, 411-412
  debugging Turbo Vision programs, 413
  Evaluate/Modify window, 401
  inserting executable expressions, 405-406
  inspector windows, 399-400
  macros, 409-410
  monitoring specific variables, 405
  protected-mode debugging, 407
  starting, 397-398
  tracing program execution, 402-403
  Variables window, 401-402
  viewing breakpoints, 406
  virtual debugging, 407-408
  Watch window, 399
unit testing, 371-372
walking through
  codes, 373-374
  designs, 373

declaring pointers
far, 133
types, 144
decrementing pointers, 127
$DEF state macro, 32
default directories, 181
defining
classes, 215-217
functions, 382-383
segment pointer keywords, 133
bcd type variables, 680
delay function, 349
deleting
items in Transfer menu, 30
subdirectories, 180
temporary files, 174
variables, 195
delimiters, 651
$DEP() instruction macro, 34
dependency statements, 66
design walkthroughs, 373
device drivers, 8-10, 525
chain, 594
linking with font files, 330
TSRs as load-on-demand device drivers, 594-595
versus TSRs, 526
devices, sharing IRQs, 608-609
diacritical marks, 496
dialog boxes
Add Watch variable, 389
Advanced Code Generation, 51
Breakpoint Modify/New, 392
Breakpoints, 392
Conditions and actions, 403
Debugger options, 385
Display Options, 432
displaying files, 201
Edit Breakpoint, 392
Evaluate/Modify, 390
inserting applications, 201
Modify/New Transfer Item, 30-31
Optimization Options, 49
Options | Compiler | Advanced Code Generation..., 51
Options | Compiler | Code Generation, 51
Options | Compiler | Code Generation..., 395
Options | Compiler | Optimizations..., 49, 386, 396
Options | Compiler... | C++ Options..., 386, 397
Options | Compiler... | Code Generation..., 130
Range, 400
Source debugging, 417
Statistics | Profiling options, 436
dialogs, *see* dialog boxes
Digital Audio and Sound Support (DASS), 365
digital sound, recording, 339-341
digital-to-analog converter (DAC), 246, 341
$DIR filename macro, 33
directives
conditional, 72-73
MAKE utility, 78-80
dot, 72-75
directories
accessing files, 183-185
closing streams, 183
displaying, 182
hard coding, 182
listings, 182-183
reading, 182-194
entries, 183
reference, 111-120
removing, 181
scanning, 183-185
searching, 182-194
files, 188
selecting
as current, 181
as default, 181

subdirectories, temporary files, 176
disabling inline functions, 386
discardable memory
schemes, 151
segments, 151
discarding
allocated memory, 143
memory blocks, 144, 150
disk
caching, 5, 16
write caching, 16-17
files, 106-109
disks, RAM, 18-20
display
international compatibility, 514
modes, video
available on IBM-PC compatible computers, 563-565
status information in BIOS data area, 565-566
Display Options dialog box, 432
displaying
directories, 182
files in hexadecimal format, 99
file dialog boxes, 201
register values of CPU, 388
text, 273-274
displays, memory-mapped, 598
disposable memory blocks, marking, 152
Divide by zero interrupt, 527
dmalloc.h header file, 152
documentation files, 105
documenting classes, 223-224
DoPrintStmt() function, 673
DOS
> (redirection) operators, 183
5.0 compatiblity with Borland C++ 2.0, 57-58
allocating memory, 149-150
application localization, 494
code pages, 492-495
country-specific conventions, 498
environment variables, 194-196
INT 21H subfunction, 149
international compatibility, 503-505
localized versions, 499-500
Protect Mode Interface (DPMI), 5-6
resources, 503-505
share.exe program, 112
vectors, table of, 527-533
DOS ATTRIB command, 104
dos.h file attributes, bit mask constants, 186
DOSKEY program, 20-21
macros, 21-23
dot directives, 72-75
double backslash (\ \ ), 181
Dr. Dobb's Journal magazine, 684
DR DOS, 7
configuring, 10-11
draw() call, 243
draw() function, 248
DrawBarChart procedure, 299, 306-314, 325
DrawBars() procedure, 325
drawing
bar charts, 307-314
line charts, 318-328
polygons, 295-296
DrawLineChart function, 299
DrawLineChart procedure, 318-328
DrawPieChart function, 299
DrawPoint() function, 318
drawpoly() function, 295-296
$DRIVE() filename macro, 33
drivers, keyboard, 496-499
drives, selecting, 181-182
-D*symbol* MAKE utility command-line switch option, 81
-D*symbol*=string MAKE utility command-line switch option, 82
DUMP utility, 99

dynamic
allocations, assigning to local pointers, 146
duration variables, 140
memory blocks, size, 148
variables, 453
dynamically
allocated memory, 141-142
freeing up, 377-378
discardable memory, 151

E

-e MAKE utility command-line switch option, 82
/e option (IDE), 43
Edit Breakpoint dialog, 392
editing
all variables, 401
breakpoints, 391-393
existing items in Transfer menu, 30
editions of software, tracking, 110
editors, 36
BRIEF (Epsilon Programmer), 36
IDE, customizing, 36
with Turbo Editor Macro language (TEML), 36-38
installing in Transfer menu, 31
MR_ED shareware programming editor, 36
$EDNAME filename macro, 33
EFFECTS module, 366
EGA, 245
EIA (Electronic Industries Association), 598
ellipse() call, 243
ellipses, drawing, 242
emulation library, 676
End-Of-Interrupt instruction (EOIs), 586, 604
EndAngle function, 305
environment
segment, 592
variables
DOS, 194-196
listing, 195
TMP, 173
$ERRCOL state macro, 32
$ERRLINE state macro, 32
$ERRNAME state macro, 32
errno global variables, 180
erroneous pointer values, 376
errors
allocation, trapping, 160-161
Cannot Load COMMAND.COM, 539
expression, 383
off-by-1, 379-380
out-of-disk-space, 384
out-of-range, 380
rounding-off, 679
Evaluate/Modify dialog box, 390-391
Evaluate/Modify window (IDE debugger), 389-391
Evaluate/Modify window (Turbo Debugger), 400
execDialog() function, 200
Execute expression, 404
Execute to... function, 402
execution
profile, 422-423
trace (programs), 394-395
Execution Profile window, 428-430
$EXENAME filename macro, 33
exp function, 360
expanded memory (EMS), 5
IDE usage, 44
utilizing to ensure system productivity, 5-6
explicit rules (MAKE utility), 69-70
exploded pies, 306
expression
operators (MAKE utility), 75-76
symbols (GREP utility), 87-88

expression() function, 673
expressions
  conditional structuring, 438
  effect of casting operators on, 390
  errors, 383
  Execute, 405
  short-circuited, 438
$EXT() filename macro, 33
extended
  memory (XMS), 5
    /x option, 43
    utilizing to ensure system productivity, 5-6
  registers, 124
extensions, .cpp source files, 143
external variables, setting, 381-382
extracting buffer portions, 651
ezfile class, 257

F

factor() function, 673
factorial() function, 429
far
  addressing, 126
  memory
    allocation routines, 150
    referencing, 126-127
  pointers
    declaring, 133
    normalizing, 132
farcoreleft() function, 377
farfree() routine, 150
farmalloc() routine, 144, 150-158
FASTOPEN utility program, 13
fcbs command, 12
fclose() function, 174
ff_fdate field, 189
ff_ftime field, 189
-f*filename* MAKE utility command-line switch option, 82
fields
  ff_fdate, 189
  ff_ftime, 189
File | Open command, 423
File | Properties... command, 417
file-compression programs, 98-99
filename macros
  $DIR, 33
  $DRIVE(), 33
  $EDNAME, 33
  $EXENAME, 33
  $EXT(), 33
  $NAME(), 33
  $OUTNAME, 33
filenames, 168-180
  creating, 173
  parsing, 168-171
  verifying uniqueness, 174
files
  access rights, 180
  adding
    to archives, 113-115
    to libraries, 107
  ASCII tracking, 106-109
  attributes, 178-180
    bit mask constants, 178, 186
    setting, 179
  .bgi
    converting into .obj files, 330-331
    linking with .chr files, 331-335
  binary, 105
  builtins.mak, 76
  checking out, 115
    multiple, 108-109
    single, 108
  .chr
    converting into .obj files, 330-331
    linking with .bgi files, 331-335
  .com, 138-143
  config.sys, 9

configuration
converting into new format project files, 94
converting into project files, 93
data, 105
dialog boxes, displaying, 201
directories, 183-185
displaying in hexadecimal format, 99
documentation, 105
ensuring access to desired files (TSRs), 558, 561-562
extension, .cpp, 159
font, linking with device drivers, 330
formats
international compatibility, 512-513
stream w+b (writeable binary), 174
graphics
driver files, 328-329
reading, 256-257
header
assert.h. assert(), 395
optionally including, 48
portability between programs, 467
see also header files
I/O international compatibility, 511
library, listing symbols referenced/defined within, 92-93
Make, 65-66
aborting, 71
incompatiblity between Borland C++ and MS C++, 466
inserting macros, 77-80
sample, 66-69
makefile.mak, 66
marking with keywords, 109
mode, 177
.obj, converting .bgi and .chr files into, 330-331
object, listing symbols referenced/defined within, 92-93
ownership, controlling, 103-105
project, 83
converting into configuration files, 93
converting into new format project files, 94
copying transfer menu items from one into another, 95
read-only, 103
read-write, 104
revisions, 110-111
scope, variables, 129
searching, 84
directories, 188
GREP utility, 85-88
Turbo Search and Replace, 88-97
whereis, 89
source
ASCII, managing, 105
.cpp extension, 143
swap, 14-15
target, 69
temporary
creating, 173-174
deleting, 174
functions, 173-177
filled() call, 243
filled_polygon() function, 242
fillpoly() function, 295-296
FIND utility, 84
find_file() function, 192
findfirst() function, 185-189
findnext() function, 185-189
Fixed disk parameter table, 533
fixedpoint class, implementing fixed-point operations, 449-452
floating points, 237, 537
code, 676

operations, Emulation mode, 676
options, 677-681
rounding, 679
flood fills, 243
floodfill() procedure, 291-292
Floppy disk controller vector, 529
fnmerge() function, 171
fnsplit() function, 168, 192
fonts
code pages, 495-496
files, linking with device drivers, 330
selecting, 274-276
stroked, 275
ViewPoint library, 244
ForEach function, 208-209
foreign languages
autoexec.bat file modifications, 498
changing, 490
character sets, 492-495
code pages, 492-495
config.sys file modifications, 498
country-specific, 491
diacritical marks, 496
keyboards, 496
entering characters, 496-497
see also international compatibility
formats
arithmetic, binary-coded-decimal (BCD), 675
currency, 507-509
date, 509-510
dates, 491
hard coded, 505
hexadecimal, displaying files in, 99
international compatibility, 505-510
files, 512-513
numeric, 507
time, 510
frame pointer, 473
free() function, 376
free() routine, 143, 376
freemem() routine, 149
freeware, 685-688
frequency, 339
Frequent errors warnings, 43
functions
_chdrive(), 181
_chmod(), 178
_fullpath(), 172
_searchenv(), 186
_splitpath(), 171
abs(), 648
access(), 179
alloca(), 149
analyzing programs by, 436
Animate..., 402
ASliceOfAmerica(), 357
assert(), 163
Back trace, 402
bar(), 306-314
CalcMinAndMax(), 324
calling
as near procedures, 138
C library functions, 459
in assembly language programming, 472-473
calls, replacing with lookup tables, 439-440
chdir(), 180-181
Check Install, 552
circle, 272
close(), 384
closedir(), 183
closegraph(), 328
ComputeInvestmentYield(), 382
coreleft(), 146, 377
cosine, 360
creattemp(), 174
d.xmax(), 235
d.ymax(), 235
ddm_allocate(), 151
debugging with Evaluate/Modify dialog, 390-391
defining, 382-383

delay, 349
DoPrintStmt(), 673
draw(), 248
DrawPoint(), 318
drawpoly(), 295-296
EndAngle, 305
execDialog(), 200
Execute to..., 401
exp, 360
expression(), 673
factor(), 673
factorial(), 429
farcoreleft(), 377
fclose(), 174
filled_polygon(), 242
fillpoly(), 295-296
find_file(), 192
findfirst(), 185-189
findnext(), 185-189
fnmerge(), 171
fnsplit(), 168, 192
ForEach, 208-209
free(), 376
fwrite(), 384
generic, 648
get, 244
get_next_char(), 665
get_symbol(), 665
getenv(), 195
getmaxx(), 272
getmaxy(), 272
getsymbol(), 658
getx(), 281
gety(), 281
grapherrormsg(), 271
graphresult(), 271, 275
hashValue(), 207-208
initgraph(), 328-329
initgraphics(), 270
inline, disabling, 386
INT 10h BIOS, 566-572
interator, 209
isA(), 207-208
isEqual(), 207-208
library, 167
  choosing when importing programs, 464-466
line(), 241
local variables in (assembly language programming), 476
malloc(), 384
memcpy(), 164, 380
mkdir(), 180-181
mktemp(), 174
moveto(), 280-281
nameOf(), 207, 208
nosound, 350
NoteStringToFreqency, 354
opendir(), 183-185
ortho_transform(), 244
piechart::draw(), 249-251
pieslice(), 274
PlayAmerica(), 358
PlayCMajorChord(), 355-356
PlayNote, 354
PlaySiren(), 351-352
PlaySound(), 345-346
PlayTone, 351
  distinguishing notes, 352
  playing major scale, 353-354
pop(), 641
printOn(), 207
push(), 641
put, 244
put_complex(), 135
putenv(), 195
readdir(), 183
real(), 681
realloc(), 148
restorecrtmode(), 329
Return ParameterPtr, 552
rmddir(), 180
rmdir(), 180-181
rmtmp(), 174

scale(), 238
searchdirectory(), 187
searchenv(), 190-193
searchpath(), 186, 190-193
set_new_handler(), 161
setaspectratio(), 292-295
setcolor(), 284-290
setfillpattern(), 284-292
setfillstyle(), 274, 284-292
constants used in calls to, 291
setgraphmode(), 329
setrgbpalette(), 284
setstrin.c, calling C library functions, 459
simpleexpression(), 673
sizeof(), 462-463
sound, 349-350
sscanf(), 650-651
StartAngle, 305
strcpy(), 380
strtok(), 651-652
system(), 182
templates, 640, 648-649
temporary files, 173-174
creattemp(), 177-178
mktemp(), 176-177
rmtmp(), 174
tempnam(), 175-176
tmpfile(), 174
tmpnam(), 174-175
textheight(), 276
textwidth(), 276
tmpfile(), 173-174
tmpnam(), 173
tracking list of currently active functions, 388
TSR function requests, 539-540
unscale(), 238
write(), 384
fwrite() function, 384

G

gamma (monitors), 247
generic
class definitions, 640
functions, 648
pointers, 144
GEnie, 688
GET command, 112
get function, 244
get_next_char() function, 665
GET_PARMPTR subfunction, 553
get_symbol() function, 665
getenv() function, 195
getmaxx() function, 272
getmaxy() function, 272
getsymbol() function, 658
getx() function, 281
gety() function, 281
gfxstyle class, 240-241
global variables
accessing in assembly language programming, 473-474
changing, 377
errno, 180
Go to cursor command, 401
goto statements, 440
grapherrormsg() function, 271
graphics
bitmaps, 244
character sizes, selecting, 274-276
charting, 296-306
class, 248-252
color, 245-247
applying to interior of objects, 284-290
choosing active color, 284-290
gamma of monitors, 247
palette, 281-284
displaying text, 273-274

drawing
bar charts, 307-314
ellipses, 242
establishing coordinates, 271
line charts, 318-328
lines, 242
parameters, 241
pie charts, 254-255
polygons, 242-296
driver, 270
driver files, 328-329
exploded pies, 306
filling objects, 274
with patterns, 296
flood fills, 243
fonts, 244
selecting, 274-276
hardware, international compatibility, 499
justifying text, 277-278
library, 270
logical rectangles, 238
monitors, 283
moving current pointer, 280-281
open intervals, 241
physical rectangles, 238
routines, returning errors, 271
scaling, 238
ViewPoint library, 234-236
benefits, 239-240
features, 240-247
viewports, 278-280
world coordinates, 236-237
Graphics character table, 533
Graphics Device Interface, 266
graphresult() function, 271, 275
GREP utility, 60, 84-88

H

-h MAKE utility command-line switch option, 83
halting execution, 196
hard
coded formats, 505
coding directories, 182
drives, installing PVCS Version Manager, 111-113
hardware
communicating with programs, 526
ensuring peak productivity, 2-3
international graphics compatibility, 499
Hardware timer tick, 528
hash code, 208
hashValue() function, 207-208
header files
#include, optionally including, 48
assert.h. assert(), 395
dmalloc.h, 152
portability between programs, 467
UARTs, 610-613
heap, 142-143
Help Compiler (Windows), 503
help
online, context-sensitive, 94-95
text
international compatibility, 503
translating, 503
hertz, 339
hexadecimal format, displaying files in, 99
HEXEDIT utility, 99
high-speed interface class library, 610-635
himem.sys, 8-10
alternatives to, 14
hooking interrupts, 538, 545-551
huge memory models, 128
huge pointers, 131-132
incrementing, 132
Hungarian notation, 221

I

-i MAKE utility command-line switch option, 82
I/O
  addresses, UART, 600-601
  files, international compatibility, 511
  ports, directly manipulating PC speaker, 345-349
  TSRs, 545-552
IBM PCs, 597-600
icons
  international compatibility, 502
  resources, 502
  translating text, 505
IDE
  /e option, 43
  /r option, 43
  built-in debugger, 370, 387-388
    breakpoints, 385, 391-394
    compiling, 385-386
    windows, 388-389
  editor, customizing with Turbo Editor Macro language (TEML), 36-38
  expanded memory, 44
  list of programs within, 28-29
  Transfer menu, 110
identifiers
  comparing with keywords, 664
  variables, 674
-I*directoryname* MAKE utility command-line switch option, 82
Idle Loop interrupt, 537, 545-551
$IMPLIB instruction macro, 34
implicit rules (MAKE utility), 69-71
importing C code into Turbo Pascal, 456-460
$INC state macro, 32
#include header files, optionally including, 48
increment operators, 132
incrementing pointers, 127, 132
InDos flag, 540-541, 545-551
initgraph() function, 270, 328-329
initializing
  pointers, 144
    to specific address, 133
  search, 186
inline functions, disabling, 386
input (keyboards/mouse), international compatibility, 513
inserting dialog boxes, 201
Inside Turbo C++ magazine, 684
inspector windows (Turbo Debugger), 399-401
install command, 10
installing
  editor in Transfer menu, 31
  install phase of TSRs, 553
  PVCS Version Manager, 111-113
instantaneous amplitude, 339
instructions
  Break point byte, 527
  Clear Interrupt flag (CLI), 543
  End-Of-Interrupt (EOIs), 586, 604
  int, 526, 534
  macros, 34-35
  Set Interrupt flag (STI), 543
INT 10h BIOS functions, 566-572
int instruction, 526, 534
integer stacks, 641
integrated debugger, *see* IDE debugger
Intel 8259 Programmable Interrupt Controller, 586
intelligent mouse classes, 245
interfaces, command-line, 105
interleave factor, ensuring peak productivity, 3-4
internal optimizer, 49-56
international compatibility, 498-505, 520-521
  accelerators, 502-503
  bitmaps, 502
  character support, 514-518

collation sequences, 517-518
DOS, 503-505
file I/O, 511
formats
currency, 507-509
dates, 509-510
files, 512-513
numeric, 507
time, 510
formatting, 505-510
help text, 503
icons, 502
input
keyboards, 513
mouse, 513
list separators, 510
output, 511-514
paper sizes, 513-514
screen display, 514
speed keys, 502-503
translator quality, 520-521
Windows Control Panel, 518-519
see also foreign languages
interrupt enable register, 602
INTERRUPT vectors, 527-533
interrupts, 347, 604-605
chaining, 545-551
versus hooking, 538
Divide by zero, 527
handler, 347
setting up, 362-365
ID register, 602
headers
code template, 634-636
shared, 613
hooking, 545-551
Idle Loop, 545-551
intercepting with TSRs, 534-535
latency, reducing, 609-610
MS-DOS idle loop, 542
Overflow, 528
software, 605
Timer Tick, 545-551
UART, 604
vectors
chaining, 534
most useful, 536-537
releasing, 587, 588
vector table, 526
INTERSOLV, Inc., 110
IRQs (interrupt request), 526, 604
add-in boards, 605
assignments, reserved, 605
sharing, 606-609
UARTs
configuring, 606
selecting, 606-607
isA() function, 207-208
isEqual() function, 207, 208
iterator functions, 209

J

Journal of Object-Oriented Programming magazine, 685
jump instructions (assembly language programming), 479
justifying text, 277-278

K

-k MAKE utility command-line switch option, 82
Keyboard Interrupt, 528, 536
keyboards, 496
Alt-key sequence, 497
drivers, 496-499
foreign languages, entering characters, 496-497
international compatibility, 496, 513
keystroke repeat rate, 23-24
keywords, 109
asm, 471-472

defining segment pointers, 133
near, 137
pascal, 457
PRINT, 673
kinetic energy, 338

L

labels, versions, 116-117
landscape mode, 97
languages, foreign,
see foreign languages
large memory models, 128
left mouse button, 40
Less frequent errors warnings, 43
Lex object, 667
lexical analysis, 650
parsing, 653-666
LexicalAnalyzer class, 658-666
$LIB state macro, 32
libraries, 105-109, 167-168, 515
class, *see* class libraries
container, *see* container libraries
emulation, 676
files
adding, 107
listing symbols referenced/defined within, 92-93
floating point, 537
functions, 167
choosing when importing programs, 464-466
graphics, 270
high-speed interface, 610-635
ViewPoint, 233-236
affine transform, 237
benefits, 239-240
features, 240-247
light slash pattern, 290
LINCHART sample program, 318-328
$LINE state macro, 32
line
charts, drawing, 318-328
control register, 602
status register, 603
line() device class, 241
line() functions, 241
lines, drawing, 241
linking
device drivers and font files, 330
modules, 134
links
serial data, 597
see also serial data links
list separators, 510
listings
4.1 Placement of $Header$ and Log keywords, 118
5.1. Demonstration use of the MK_FP() macro, 134
5.2. Source file for mixed-model programming, 135-136
5.3. Header file for the large model module, 136
5.4. Large model module with put_complex() function, 136
5.5. Sample program compilable into a .com file, 139
5.6 Program using malloc() and free(), 145
5.7 Example using the calloc() function, 147-148
5.8 Header file for discardable memory schemes, 152
5.9 Discardable memory manager listing, 153-157
5.10 Testing discardable memory manager, 158
5.11 Allocating arrays with new operator, 160
5.12 Example of set_new_handler() function, 161
5.13 Setting pointer to NULL, 162-163
6.1 Example of fnsplit() and fnmerge(), 170

6.2 Sample usage of the _fullpath() function, 172
listings, continued
6.3 Example of using tempnam(), 176
6.4 DIR type defined in dirent.h, 183
6.5 dirent structure, 184
6.6 Directory-reading functions, 185
6.7 ffblk structure, 187
6.8 Display directory listing with findfirst() and findnext(), 188-189
6.9 Example of _searchenv() function, 190
6.10 Code to locate application's own subdirectory, 192-193
6.11 Example of _argc and _argv system variables, 194
6.12 (Intercepting Ctrl-Break keystroke, 196-197
6.13 Sample TVISION application with TFileDialog, 198-200
6.14 TFileDialog in an ObjectWindows application, 202
6.15 Bag container library, 205-206
6.16 SortedArray container class, 210-211
8.1 Simple graphics program using ViewPoint library, 234
8.2 Chart drawing class, 248-249
8.3 Chart class, 249-251
8.4 Testing pie chart drawing capability, 252
8.5 Stretching boxes using mouse queue, 253-254
8.6 Selecting, with mouse, area to draw pie chart, 254-255
8.7 Targa File Reader, 256-257
8.8 PCX file viewer, 258-262
9.1 Sample Borland Graphics program, 267-270
9.2 Custom pattern editor, 285-290
9.3 Calibrating circle drawing algorithm for monitors, 292-295
9.4 PIECHART program, 299-305
9.5 BARCHART program, 307-314
9.6 LINCHART sample program, 318-328
9.7 Changing source code to link .bgi and .chr file, 331-335
11.1 Clobbering pointers/variables with array indicing, 380-381
11.2 Debugging functions with Evaluate/Modify dialog, 390-391
11.3 ObjectWindows initialization routine, 415
12.1 Sample program testing Turbo Profiler, 424
12.2 Using long data types instead of float data table, 429
12.3 Implementing factorial() nonrecursively, 431-432
12.4 Replacing calculations with lookup table, 439-440
12.5 Calculating prime numbers, 441
12.6 Prime number program avoiding division algorithms, 442-443
12.7 Locating duplicate numbers in consecutive numbers using bitmap, 443-445
12.8 Finding duplicate numbers arithmetically, 446-447
12.9 Fixed point operations, 449
12.10 Implementing fixed-point operations with fixedpoint class, 449-452
13.1 Turbo Pascal program calling written function, 456-457
13.2 sum.c sample function, 457

13.3 setstrin.c C function call C library function, 459
13.4 Pascal program, calling C routine, 459-460
13.5 Implementing increment function with asm stat, 471
13.6 Referencing variables by built-in assembly code, 474
13.7 Accessing pointer parameters, 477
13.8 Assembly code using struct type, 478-479
13.9 Calling external assembly language function, 483-484
13.10 Sample Turbo Assembler program, 485
14.1 Code segment storage of program text, 504-505
15.1 Hooking/chaining interrupts.InDos flag.Timer Tick interrupts.Idle Loop interrupts, 545-551
15.2 TesSeRact interface, 553-556
15.3 Writing to files from within TSR mode, 561
15.4 Saving screen information in TSRs, 573-583
16.1 UART header file, 610-613
16.2 Implementing a serial I/O interface class, 613-614
16.3 DDCMP class, 615-616
16.4 Implementing a serial I/O interface class, 616-621
16.5 DDCMP class, 621-628
16.6 Hooking and unhooking shared interrupt vectors, 630-632
16.7 ISRSTUB.ASM, 632-634
16.8 SERTEST.CPP, 636-637
17.1 Class maintaining stack of integer values, 641
17.2 Sample use of integer stack class, 642
17.3 Template-based definition of a stack, 643
17.4 The TStack template in use, 644
17.5 TStack template used to create a stack of objects, 644-646
17.6 Demonstratio of the strtok() _fstrtok() function, 652
17.7. The lexical.h class header, 656-658
17.8 Implementing the LexicalAnalyzer class, 659-664
17.9 Header file for syntax analyzer class, 666-667
17.10 Implementation of the syntax analyzer and expression evaluator, 668-672
17.11 Calling syntax analyzer to parse expressions, 675
loading Borland C++, 15-16
local variables, 139-141, 453
  allocating space, 140
localization, 490
  DOS, 499-500
  foreign languages
    changing, 490
    country-specific, 491
  resource files, 501-502
  Windows, 499-500
locked
  memory
    blocks, 151
    segments, 151
  revision, 111, 114-115
    overriding, 119-120
logic errors, 374
logical rectangles, 238
lookup tables, replacing function calls, 439-440
loops, improving, 437-438
low memory area, 7
lowercase, 515

LZEXE utility, 98-99

M

-m MAKE utility command-line switch option, 82
macro commands in Transfer programs, 31-35
macros
  assert(), 227-229
  DOSKEY program, 21-23
  in Turbo Debugger, 409-410
  inserting into make files, 77-80
  MK_FP(), 133
  MK_FP(), 134
  versus variables in assembly language programming, 476
mail-order companies, 685
maintaining revision histories of source code, 117-119
major scale, playing, 353-354
Make files, 65-66
  aborting, 71
  incompatiblity between Borland C++ and Microsoft C++, 466
  inserting macros, 77-80
  sample, 66-69
MAKE utility, 60, 65-66
  @ symbol, 71
  batching commands, 77
  builtins.mak file, 76
  command lines, 71
    options, 81-83
  conditional directives, 72-73, 78-80
  dot directives, 73-75
  explicit rules, 69-70
  expression operators, 75-76
  implicit rules, 69-71
  macros, inserting into make files, 77-80
  sample shell.mak file, 66-69
  TOUCH command, 84
makefile.mak file, 66
MAKER utility, 65
malloc() function, 384
malloc() routine, 143-165
managing
  arbitrary data files, 105
  ASCII source files, 105
manually halting programs, 399
mapping characters, 516
marking disposable memory blocks, 152
math options, 675-678
maximizing memory with Turbo Profiler, 428-432
  active profiling, 434-435
  analyzing programs by functions, 436
  changing algorithms, 441-447
  dynamic variables, 453
  goto statements, 440
  improving loops, 437-438
  increasing file I/O buffers, 452
  local variables, 453
  passing by address versus copying to stack, 447
  passive profiling, 434-435
  recycling memory, 454
  reducing memory, 453
  replacing
    float data types with fixed point comput, 448-452
    function calls with lookup tables, 439-440
  selecting program areas to profile, 425-428
  set compiler options, 439
  statistics provided by Profiler, 432-434
  structuring conditional expressions, 438
  writing in assembly language, 447-448

MCGA, 245
medium memory models, 128
$MEM() instruction macro, 34
members
  data(), 249
  read_row(), 263
  set_pen_color(), 246-247
  set_scale(), 237
memcpy() function, 164, 380
memory
  addressing, 124-126
  allocating, 143, 159-160
    discarding, 143-144
    DOS, 149-150
    dynamically, 141-142
  configuring to ensure peak productivity, 4-5
  discardable schemes, 151
  dynamically allocated, freeing up, 377
  expanded (EMS), 5
    IDE usage, 44
    utilizing to ensure system productivity, 5-6
  extended (XMS), 5
    /x option, 43
    utilizing to ensure system productivity, 5-6
  freeing (unloading TSRs), 589-592
  heap, 142-143
  managers, 14
  managing routines, 149
  maximizing, *see* maximizing memory
  operating near program's memory requirements, 384
  out-of-memory condition, 152
  paragraphs of memory (TSRs), 539
  RAM disks, 18-20
  referencing
    far, 126-127
    near, 126-127
  system, utilizing to ensure system productivity, 7
  trashing, 380-381
memory
  blocks
    allocating, 149-151
    discarding, 144, 150
    disposable, marking, 152
    dynamic, 148
    locked, 151
  models, 121-130
    compact, 127-128
    huge, 128
    large, 128
    medium, 128
    restrictions, 129
    selecting, 130
    setting command-line compiler, 130
    small, 128
    tiny, 127
    see also mixed models
  segments
    discardable, 151
    locked, 151
    relocation, 138
  trashers, 163
  watchers, 406
memory-mapped displays, 598
menu option, Debug | Toggle breakpoint, 387
menu-driven program, ATTIC, 106
menus
  Compile, 42
  Project, 83
  Transfer, 28-29
    adding items, 30
    deleting items, 30
    editing existing Transfer items, 30
    installing editor in, 31
    modifying, 29
Microsoft Windows, 535

C++ compatibility with Borland C++, 466-467
running TSRs under, 584-586
MIDI ports, 366
mixed models, 134-138
MK_FP() macro, 133
mkdir() function, 180-181
mktemp() function, 174-177
modem
control register, 602
status register, 603
modes
landscape, 97
portrait, 97
protected, Microsoft Windows in, 584
TSR, writing to files from within, 561
video display
available on IBM-PC compatible computers, 563-565
status information in BIOS data area, 565-566
View | CPU, debugging in, 418-419
modification histories, maintaining, 372
modifiers, pointers, 134-138
Modify/New Transfer Item dialog box, 30-31
modifying Transfer menu, 29
Module window, 428
modules
DASSMAIN, 366
EFFECTS, 366
linking, 134
MUSIC, 366
object, 105
PCM, 366
monitors
gamma, 247
graphics, 283
monochrome display, 245
mouse
buttons, customizing, 40
configuring for left-hand users, 41
event queues, 253-254
input, international compatibility, 513
mouse_cursor class, 252
mouse_event class, 253
moveto() function, 280-281
moving current pointer in graphics system, 280-281
MR_ED shareware programming editor, 36
MS-DOS
1.0 program terminate, 531
absolute disk
read, 532
write, 532
API vector, 532, 536
configuring, 8-10
Control-C vector, 532
critical error interrupt, 532
Exec function request (4Bh), 592
file system, accessing from within TSRs, 558
idle loop
interrupt, 542
vector, 533
Terminate and Stay, 532
terminate vector, 532
multidimensional arrays, allocating, 160
multilingual code pages, 493
multiple
files, checking out, 108-109
languages, supporting in programs, 95-97
Multiplex and TSR communications, 533
multitasking, 535

MUSIC module, 366
music synthesizer, 366
musical notes, creating, 352-353

N

-n MAKE utility command-line switch option, 82
$NAME() filename macro, 33
nameOf() function, 207-208
near
  addressing, 126
  keyword, 137
  memory referencing, 126-127
NetBIOS Interrupt, 537
networks, PVCS Version Manager
  file servers, storing information, 110
  installing, 111-113
new operator, 159-160
Non-Maskable Interrupt, 527
normalized pointers, 131
nosound function, 350
$NOSWAP instruction macro, 34
note strings, creating, 354
NoteStringToFrequency function, 354
null pointers, 144, 162-163
numeric formats, international compatibility, 507

O

.obj files, converting .bgi and .chr files into, 330-331
object
  classes, creating from templates, 647
  files, listing symbols referenced/defined within, 92-93
  modules, 105
  wildargs.obj, 193
objects
  coloring interior, 284-290
  comparing, 208
  defining container libraries, 207
  filling with
    colors, 274
    patterns, 274, 296
  Lex, 667
  storing stacks, 644
  TLogEntry, 207
ObjectWindows
  initialization routine, 415
  TFileDialog class, 201-202
OBJXREF utility, 92-93
octaves, 352
off-by-1 errors, 379-380
offset value, far pointers, 127
online, context-sensitive help, 94-95
open intervals, 241
opendir() function, 183-185
operator
  overloading, 222-223
  precedence, 383
operators
  arithmetic, 133
  built-in assembler operators, 481-482
  casting, checking effect of, on expressions, 390-391
  DOS, > (redirection), 183
  expression (MAKE utility), 75-76
  increment, 132
  new, 159-160
Optimization Options dialog box, 49
optimizations, compiler, disabling, 45
optimizing programs, 49-56
Options | Compiler | Advanced Code Generation... dialog box, 51
Options | Compiler | Code Generation... dialog box, 51, 395
Options | Compiler |

Optimizations... command, 424
Options | Compiler | Optimizations... dialog box, 49, 386, 396-397
Options | Compiler... | C++ Options... dialog box, 386, 397
Options | Environment... | Preferences.... command, 387
Options | Macros | Delete all command, 411
Options | Macros | Remove... command, 411
Options | Macros | Stop recording command, 411
Options | Macros command, 410
Options | Compiler... | Code Generation... dialog box, 130
ortho_transform() function, 244
out-of-disk-space errors, 384
out-of-memory condition, 152
out-of-range errors, 380
$OUTNAME filename macro, 33
output
  paper sizes, 513
  translation, 511-514
outtext() routine, 273-274
outtextxy() routine, 273-274
Overflow interrupt, 528
overriding locked revisions, 119-120
overwriting stack, 164
ownership files, controlling, 103-105, 110

P

paragraphs, 150
  converting from bytes, 150
paragraphs of memory (TSRs), 539
parameters
  accessing
    command-line, 193-194
    pass-by-reference (assembly language program), 477-478
    pass-by-values (assembly language program), 477-478
  attrib constant values, 177
  pointer, accessing, 477
  setviewport(), 278-280
  variable class templates, 647
  xasp, 292-295
  yasp, 292-295
parsing, 640, 649-681
  filenames, 168-171
  formal parser, 653
  lexical analysis, 653-666
  numeric values, 665
  puncuation symbols, 665
  recursive descent parsing, 653
  semantic analysis, 653
  source statement, 653
  subdirectories, 192
  syntax analysis, 653, 666-675
  tokens, 653
pascal keyword, 457
passive analysis, 434
patterns, customizing, 285-290
PC & PS/2 Video Systems, 564
PC speakers, 344-345
  direct access to, 345-349
PC Techniques magazine, 685
PC Tools (Central Point Software), 4
PC XT floppy disk interface, 533
PC-CACHE utility, 17
PCM module, 366
PcmFile class, 361-362
PcmNote class, 361-362
PCX file viewer, 258-263
physical rectangles, 238
PIC (Peripheral Interrupt Controller), 526, 604-605
pie charts
  drawing, 254-255
  routine, 299-306
PIECHART program, 299-305
piechart::draw() function, 249-251
pieslice() call, 243

pieslice() function, 274
pixarray class, 244
PlayAmerica() function, 358
PlayCMajorChord() function, 355-356
PlayNote function, 354
PlaySiren() function, 351-352
PlaySound() function, 345-346
PlayTone function, 351
  distinguishing notes, 352
  playing major scale, 353-354
pointer parameters, accessing, 477
pointers, 131-134, 162
  * components, 184
  creating
    to arrays, 147
    to specific locations, 133-134
  decrementing, 127
  far
    declaring, 133
    normalizing, 132
  generic, 144
  huge, 131-132
  incrementing, 127, 132
  initializing, 144
    to specific address, 133
  modifiers, 134-138
  normalized, 131
  NULL, 144, 162-163
  segment, 133
  segment:offset values, 131
  testing, 146
  types, declaring, 144
polling, 603-608
polygons, drawing, 242, 295-296
polyphonic music, creating, 360
  generating PCM data, 360-362
  playing PCM data, 362-365
pop() function, 641
Portability warnings, 42
portrait mode, 97
ports, 526
power supply, 492
precompiled headers, 46-48
preinitializing allocations, 159
preprocessing programs, 60
  sample, 62-64
previous points, 318
prime numbers, calculating, 441
PRINT keyword, 673
Print Screen vector, 537
PRINT statement, 673
Printer port controller, 529
printers, international compatibility, 499
printf() statements, 329, 394-395
printing, saving paper, 98
printOn() function, 207
private virtual functions, 224
privileges, 110-111
PRJ2MAK utility, 83
PRJCFG utility, 93
PRJCNVT utility, 94
$PRJNAME state macro, 33
procedures
  calling in assembly language programming, 472-473
  DrawBarChart, 299, 306-314, 325
  DrawBars(), 325
  DrawLineChart, 299, 318-328
  DrawPieChart, 299
  floodfill(), 291-292
  setallpalette(), 282-283
  setpalette(), 282-283
  settextjustify(), 277-278
Prodigy Services, 688
Program reset command, 402
program
  resources, 500
  statements, adding to source code (debugging technique), 370
Programmable Interrupt Controller (PIC) chip, 538
Programmer's Paradise mail-order company, 685
Programmer's Warehouse mail-order company, 685

programming
assembly language, 467-469
arithmetic operations, 480-482
constant identifiers (macros) versus variables, 476
CPU's instruction set, 469-470
functions, calling, 472-473
global variables, accessing, 473-474
jump instructions, 479
local variables in functions, 476
pass-by-reference parameters, accessing, 477-478
pass-by-value parameters, accessing, 477-478
procedures, calling, 472-473
statement labels, 479
structure components, accessing, 478-479
Turbo Assembler, 482-488
values and addresses, distinguishing between, 474-475
writing code with built-in assembler, 470-472
by individuals, ATTIC, 106-109
programs
analyzing by functions, 436
communicating with hardware, 526
DASS.EXE, 365-366
developing, 102-103
DOS share.exe, 112
DOSKEY, 20-21
macros, 21-23
execution
profile, 422-423
trace, 394-395
FASTOPEN utility program, 13
file-compression, 98-99
improving speed of, 428-432
keywords for retrieving source, 109
LINCHART sample program, 318-328
manually halting, 399
optimizing, 49-56
PIECHART, 299-305
preprocessing, 60
sample, 62-64
PVCS Version Manager, 110
shareware, ATTIC, 105-109
subdirectories, 191
supporting multiple languages, 95-97
THELP pop-up TSR program, 94-95
Transfer macro commands, 31-35
TSRs, *see* TSRs
Turbo Assembler, 28
Turbo Vision, debugging, 412-413
viewing value of variables during execution, 388-389
Winsight message intercept, 416, 418
project
files, 83
converting, 93-94
copying transfer menu items from one into another, 95
make facilities, 65
management duties, 110
Project menu, 83
projects, creating subdirectories for, 111-113
$PROMPT instruction macro, 35
protected
members (classes), 224
mode
debugging (Turbo Debugger), 406-407
Microsoft Windows in, 584
Public Brand Software, 687
public interface (classes), 224
publications, suggested PC technical computer publications, 684-685
Pulse Code Modulation (PCM), 342
generating PCM data, 360-362

playing PCM data, 362-365
pure tone, 339
push() function, 641
PUT command, 113-115
put command, 116
put function, 244
put_complex() function, 135
put_thestring() routine, 90
putenv() function, 195
PVCS Configuration Builder, 110
PVCS Version Manager, 105, 110-111
5.0, 105
accessing older revisions, 116
files
adding to archives, 113-115
checking out, 115
maintaining revision histories of source code, 117-119
setting up, 111-113
version labels, 116-117

Q

QCACHE utility, 17
qsort() routine, 390-391

R

-r MAKE utility command-line switch option, 82
/r option (IDE), 43
RAM disks, 18-20
Range dialog box, 400-401
$RC instruction macro, 35
read() call, 257
read-only files, 103
read-write files, 104
read_row() member, 263
readdir() function, 183
reading
directories, 182-194
graphics files, 256-257
real() function, 681
realloc() function, 148
recursive descent parsing, 653
redirection (>) operator (DOS), 183
reference directory, 111-120
referencing
heap area data, 125
memory
far, 126-127
near, 126-127
registers, 602-603
bits, 124
UARTs, 607-608
CPU, 80x86, 122-124
data, 602
dual meanings, 603
extended, 124
interrupt
ID, 602
enable, 602
line
control, 602
status, 603
modem
control, 602
status, 603
scratch, 603
segment, 122-124
shifting, 132
Registers window (IDE debugger), 388
relational operators, lexical analysis, 656
relative drawing commands, 280
relocation
information, 138
memory segments, 138
removing directories, 181
Reserved vector, 528
Resource Compiler (Windows), 500-501
resource
files, localization, 501-502

scripts, 500
resources
accelerators, 502-503
bitmaps, 502
DOS, 503-505
icons, 502
program, 500
speed keys, 502-503
string tables, 501-502
restorecrtmode() function, 329
restrictions, memory models, 129
Return ParameterPtr function, 552
returned error return codes, 384
reusable classes
avoiding debugging code, 227-229
deciding future use, 217-220
defining, 215-217
designing class interface, 214-215
documenting, 223-224
keeping concise, 220
libraries
Borland International Data Structures (BIDS), 214
Turbo Vision, 214
naming, 220-221
operator overloading, 222-223
restricting access, 225-227
standard idioms, 222
testing, 229-232
revision histories, maintaining source code, 117-119
revisions
accessing, 116
files, 110-111
locked, 111, 114-115
overriding, 119-120
tracking software, 105
tip, 110
right mouse button, 40
rmddir() function, 180
rmdir() function, 180-181
rmtmp() function, 174
ROM BIOS
BASIC, 530
cassette service, 530
clock services, 530
COM port driver, 530
disk services, 530
equipment configuration check, 529
keyboard driver, 530
memory size check, 530
print screen vector, 528
printer driver, 530
video services, 529
routines
_dos_allocmem(), 149
allocmem(), 149
calloc(), 147-148
character identification, 515-516
CopyTemplate, 636
ctrlbrk(), 196
extracting from libraries, importing into programs, 458
far memory allocation, 150
farfree(), 150
farmalloc(), 144, 150-158
free(), 143, 376
freemem(), 149
graphics, returning errors, 271
malloc(), 143-165
memory management, 149
ObjectWindows initialization, 415
outtext(), 273-274
outtextxy(), 273-274
pie chart, 299-306
put_thestring(), 90
qsort(), 390
setblock(), 149
settextjustify(), 273-274
settextstyle(), 274-276
RS-232 specification, 598
Run | Go to cursor command, 392
Run | Program command, 411

Run | Step over command, 387
Run | Trace into command, 387
Run command, 402
running ATTIC, 106

S

-s MAKE utility command-line switch option, 82
/s- switch (BC command line), 46
sampling frequency, 340
Save | Modify TDW.EXE command, 417
$SAVE ALL instruction macro, 35
$SAVE CUR instruction macro, 35
$SAVE PROMPT instruction macro, 35
saving
  paper, 98
  screen information (TSRs), 573-583
    video modes for IBM-PC compatible computers, 563-565
scale value, 249
scale() function, 238
scaler class, 238-239
scaling
  graphics, 238
  transforms, 254-255
scanning
  character
    arrays, 651-652
    buffers, 650
  delimiters, 651
scopes, file scope variables, 129
scratch register, 603
screen coordinates, 271
screen_device shell, 235
screen displays, international compatibility, 499, 514
scripts, resource, 500
search pattern, 187
searchdirectory() function, 187
searchenv() function, 190-193
searching, 516-517
  directories, 182-194
    files, 188
  for files, 84
    GREP utility, 85-88
    Turbo Search and Replace, 88-97
    whereis, 89
  initializing, 186
searchpath() function, 186, 190-193
segment
  pointers, 133
    defining keywords, 133
  registers, 122-124
    combining, 125
    shifting, 132
segment:offset
  addressing, 129-131
  pairs, 127
  values. pointers, 131
segments, 122, 584
  memory relocation, 138
  TSRs, 539
Select command, 108-109
Selected Online Services and Libraries, America Online, 688
selecting
  character sizes, 274-276
  directories
    as current, 181
    as default, 181
  drives, 181-182
  fonts, 274-276
  IRQs for UARTs, 606-607
  memory models, 130
selectors, 584
semantic analysis (parsing), 653
serial data links, 597
  baud rate, 598
Set Interrupt flag (STI) instruction,

543
set_new_handler() function, 161
set_pen_color() member, 246-247
set_scale() member, 237
setallpalette() procedure, 282-283
setaspectratio() function, 292-295
setblock() routine, 149
setcolor() function, 284-290
setfillpattern() function, 284-292
setfillstyle() function, 274, 284-292
  constants used in calls to, 291
setgraphmode() function, 329
setpalette() procedure, 282-283
setrgbpalette() function, 284
setstrin.c function, calling C library functions, 459
settextjustify() procedure, 277-278
settextjustify() routine, 273-274
settextstyle() routine, 274-276
setting
  file attributes, 179
  memory models, command-line compiler, 130
  PVCS Version Manager, 111-113
setviewport() parameter, 278-280
shareware program, ATTIC, 105-109
sharing
  data segments, 129
  IRQs, 606-609
short-circuited expressions, 438
side effects, 377
SideKick application, 535
simple data types, 159-160
simpleexpression() function, 673
single files, 108
size, memory blocks, 150
sizeof() function, 462-463
small memory model, 128
snr command-line options, 91-92
software
  common problems during development
    checking all returned error codes, 384
    erroneous pointer values, 376
    expression errors, 383
    failing to free up dynamically allocated memory, 377-378
    global variables, changing, 377
    ignoring scoping rules, 381-382
    logic errors, 374
    memory trashers, 380-381
    off-by-1 errors, 379-380
    out-of-disk-space errors, 384
    out-of-range errors, 380-381
    typographical errors, 378-379
    undefined functions, 382-383
    uninitialized pointer values, 376
    uninitialized variables, 375
  interrupt handlers, 605
  testing, 369
  tracking
    editions, 110
    revisions, 105
  VCS, 105
    PVCS Version Manager 5.0, 105
Software timer tick, 531
Song class, 357-360
songs, creating, 357-360
  chunks, 362
SortedArray container library, 209-211
sound, 338
  cards, 366-368
  converting into stream of digital data, 339-341
  effects, creating, 351-352
  function, 349-350
source
  codes
    maintaining revision histories, 117-119
    ownership, controlling, 101
  files

.cpp extensions, 143
ASCII, managing, 105
see also files
Source debugging dialog box, 417-418
source statement (parsing), 653
special trigraph character sequences, 96
specifying, file mode, 177
speed keys
international compatibility, 502-503
resources, 502-503
translating, 502
SS (stack segment), 125
sscanf() function, 650-651
STACKER, 98
STACKER 2.0 (Stac Electronics), 24-26
stacks
call, 141
classes, 641
creating templates, 643
frame, 472
integer, 641
objects, storing, 644
overwriting, 164
segment, *see* SS
reserving stack space (TSRs), 544-545
stand-alone configuration, storing information PVCS Version Manager, 110
StartAngle function, 305
starting Turbo Debugger, 397-399
state macros
$COL, 32
$CONFIG, 32
$DEF, 32
$ERRCOL, 32
$ERRLINE, 32
$ERRNAME, 32
$INC, 32
$LIB, 32
$LINE, 32
$PRJNAME, 33
statements
#pragma hdrstop, 48
dependency, 66
goto, 440
labels (assembly language programming), 479
PRINT, 673
printf(), 329, 394-395
Turbo Assembler, 483
static
data, 129
variables, 129, 140
duration variables, 140
preinitialized, 140
Statistics | Profiling options dialog box, 436
Step over command, 402
storing data, 139-141
strcpy() function, 380
streams
directories, closing, 183
format, w+b (writeable binary), 174
strings
comparing, 516-517
concatenating, 171
length, constants, 169
tables, resources, 501-502
stroked fonts, 275
strtok() function, 651-652
subdirectories
creating, 180
for each project, 111-113
deleting, 180
parsing, 192
temporary files, 176

subfunctions, DOS INT 21H, 149
sum function, 483-484
Super PC-Kwik, 17
SuperStor, 98
swap files, 14-15
switches
  command-line
    compiler switches, 49-56
    memory model selection, 130
  option, MAKE utility
    command-line, 81-83
symbols, currency, 507-508
syntax analysis (parsing), 653, 666-675
system() function, 182
System Restart, 530
systems
  clock
    counter, 347
    timer, 347, 362-365
  configuration, enhancing
    configuring memory, 4-7
    creating swap files, 14-15
    disk caching, 16-17
    DR DOS, configuring, 10-11
    faster loading of Borland C++, 15-16
    FASTOPEN utility program, 13
    hardware modifications, 2-3
    keyboard repeat rate, 23-24
    memory managers, 14
    MS-DOS, configuring, 8-10
    RAM disks, 18-20
    selecting CPU, 4
    setting interleave factor, 3-4
    utilizing extended or expanded memory, 5-6
  memory, utilizing to ensure system productivity, 7
  PVCS Version Manager, 105
  U.S. vs. other countries, 492-500
  variables
    _argc, 193
    _argv, 193
  version management
    ATTIC, 106-109
    PVCS Version Manager, 110-120

T

Targa File Reader, 256-257
target files, 69
$TASM instruction macro, 35
TDW, 413-414
  command-line options, 417-418
  examining messages, 414-416
templates, 639
  classes, 640-650
    creating, 640
    TStack, definition, 642
  code, interrupt headers, 636
  container libraries, 203
  function, 640, 648-649
  object classes, 647
  stacks, creating, 643
tempnam() function, 175-176
temporary files
    creating, 173-174
    creattemp() function, 177-178
    deleting, 174
    mktemp() function, 176-177
    rmtmp() function, 174
    tempnam() function, 175-176
    tmpfile() function, 174
    tmpnam() function, 174-175
  subdirectories, 176
  variables, 195
terminate-and-stay-resident programs, *see* TSRs
TesSeRact standard, 552-556
text, 106-109
  bitmaps, translating, 505
  displaying, 273-274
  icons, translating, 505
  justifying, 277-278

versions, 106-109
textheight() function, 276
textwidth() function, 276
TFileDialog class
ObjectWindows, 201-202
Turbo Vision, 197-201
The Austin Code Works mail-order company, 685
The Programmer's Shop mail-order company, 685
The Software Labs, Inc., 687
THELP pop-up TSR program, 94-95
thrash (system), 45
tilde character (~), 30
time formats, 510
Timer Tick, 536
interrupt, 545-551
tiny memory models, 127
tip revision, 110
TLogEntry class, 644
TLogEntry object, 207
TMP environment variable, 173
tmpfile() function, 173-174
tmpnam() function, 173-175
Toggle Breakpoints command, 392
tokens, 653
TOUCH command, 84
touch testing, 231
Trace into command, 402-403
tracking
files
binary, 110-111
source, 110-111
ASCII, version histories of, 106-109
software
editions, 110
revisions, 105
TRANCOPY utility, 95
transducer, 338
Transfer menu, 28-29
adding items, 30
copying menu items into new project files, 95
deleting items, 30
editing existing Transfer items, 30
installing editor in, 31
modifying, 29
Transfer menu (IDE), 110
Transfer programs, macro commands, 31-35
translating
accelerators, 502
help text, 503
speed keys, 502
text
bitmaps, 505
icons, 505
translator quality, 520-521
translation, 490, 500-505
bitmaps, 502
icons, 502
output, 511-514
program text, 500-501
trapping allocation errors, 160-161
TRIGRAPH utility, 95-97
TSRs (terminate-and-stay-resident programs)
accessing MS-DOS file system from within, 558
activating, 542
as load-on-demand device driver, 594-595
chaining interrupts, 534, 545-551
versus hooking interrupts, 538
Clear Interrupt flag (CLI) instruction, 543
debugging, 411-412
disadvantages, 587
ensuring access to desired files, 558-559, 562
environment segment, 592
floating point libraries, 537
function requests, 539-540

hooking interrupts, 545-551
Idle Loop interrupt, 545-551
InDos flag, 540-541, 545-551
input/output, 545-552
install phase, 553
intercepting interrupts, 534-535
keyboard usage, notifying, 557
MS-DOS idle loop interrupt, 542
multitasking, 535
paragraphs of memory, 539
placing into high memory, 10
releasing
interrupt vectors, 587-588
memory, 589-592
reserving stack space, 544-545
running under Microsoft Windows, 584-586
saving screen information
INT 10h BIOS functions, 566-572
video modes for IBM-PC compatible computers, 563-565
video status information in BIOS data area, 565-566
segments, 539
Set Interrupt flag (STI) instruction, 543
TesSeRact standard, 552-556
Timer Tick interrupt, 545-551
unloading, 593
freeing memory, 589-593
releasing interrupt vectors, 587-588
versus device drivers, 526
writing to files from within TSR mode, 561
TStack class
class definition, 642
template definition, 642
Turbo Assembler, 28, 482-488
running TASM, 487-488
statements, 483
sum function, 483-484
Turbo Debugger, 370, 396
at assembly language source level, 406-407
breakpoints, 403-405
conditional, 405
viewing, 406
compiling
for compatibility, 396-397
programs with command-line compilers, 396
debugging
Turbo Vision programs, 412-413
TSRs, 411-412
inserting executable expressions, 405-406
macros, 409-411
monitoring specific variables, 406
protected-mode debugging, 407
starting, 397-399
tracing program execution, 402-403
virtual debugging, 407-409
windows
Evaluate/Modify, 401
inspector, 399-401
Variables, 401-402
Watch, 399
Turbo Debugger for Windows, *see* TDW
Turbo Editor Macro Compiler, 36
Turbo Editor Macro Language (TEML), 36-40
customizing IDE editor, 36-38
Turbo Profiler, 422-423
active profiling, 434-435
analyzing programs by functions, 436
changing algorithms, 441-447
compiling for compatibility, 424
dynamic variables, 453
goto statements, 440
improving

loops, 437-438
speed of program, 428-432
increasing file I/O buffers, 452
local variables, 453
passing by parameter vs copying to stack, 447
passive profiling, 434-435
recycling memory, 454
reducing memory, 453
replacing float data types with fixed point computations, 448-452
replacing function calls with lookup tables, 439-440
running, 423
sample program testing, 424
selecting areas to profile, 425-428
setting compiler options, 439
statistics provided by, 432-434
structuring conditional expressions, 438
writing in assembly language, 447-448
Turbo Search and Replace, 88-92
Turbo Vision
debugging, 412-413
library, 214
TFileDialog class, 197-201
tutorial option, PVCS Version Manager, 111
typographical errors, 378-379

U

UARTs, 597-602
configuring IRQs, 606
header file, 610-613
I/O address, 600-601
interrupts, 604
polling, 603-608
registers, 602-603
bits, 607-608
see also registers
uninitialized
pointer values, 376
variables, 375
unit testing, 371-372
unloading TSRs, 593
freeing memory, 589-593
releasing interrupt vectors, 587-588
unscale() function, 238
Until Return command, 402
upper memory
area, 7
block, 9
uppercase, 515
-u*symbol* MAKE utility command-line switch option, 83
utilities
4PRINT, 97
CPP, 60-65
sample preprocessed program, 62-64
DUMP, 99
FIND, 84
GREP, 60, 84-88
HEXEDIT, 99
LZEXE, 98-99
MAKE, 60, 65-66
@ symbol, 71
batching commands, 77
builtins.mak file, 76
command lines, 71, 81-83
conditional directives, 72-73, 78-80
dot directives, 73-75
explicit rules, 69-70
expression operators, 75-76
implicit rules, 69-71
macros, inserting into make files, 77-80
sample shell.mak file, 66-69
TOUCH command, 84
MAKER, 65
OBJXREF, 92-93
PC-CACHE, 17

PRJ2MAK, 83
PRJCFG, 93
PRJCNVT, 94
QCACHE, 17
TRANCOPY, 95
TRIGRAPH, 95-97

V

values, distinguishing from address in assembly language program, 474-475
variable parameter declaration, 457
variables
  adding to Watch windows, 399
  automatic duration, 139
  changing, 399-401
  defining bcd type, 680
  deleting, 195
  dynamic, 453
  dynamic duration, 140
  environment
    DOS, 194-196
    listing, 195
    TMP, 173
  examining, 399-401
    value of individual variables, 389-391
  external, setting, 381-382
  file scope, 129
  global
    accessing in assembly language programming, 473-474
    changing, 377
    errno, 180
  identifiers, 674
  local, 139-141, 453
    in functions (assembly language programming), 476
  monitoring specific variables, 405-406
  static, 129, 140
  system
    _argc, 193
    _argv, 193
  temporary, 195
  unititialized, 375
  versus constant identifiers (macros), 476
  viewing
    and/or seeing all, 401-402
    value of during program execution, 389-390
Variables window (Turbo Debugger), 401-402
VCS (version control system), 101-105, 110
  PVCS Version Manager 5.0, 105
vectors
  BIOS
    COM, 537
    Print, 537
  COM port, 537
  COM1 port controller, 529
  COM2 port controller, 529
  Control Break, 531
  Control-C, 536
  Critical Error, 536
  CRT vertical retrace interval, 529
  DOS vectors, table of, 527-533
  Fixed disk parameter table, 533
  Floppy disk controller, 529
  Graphics character table, 533
  Hardware timer tick, 528
  Idle Loop Interrupt, 537
  INTERRUPT, 527-533
  interrupt
    chaining, 534
    most useful, 536-537
    releasing, 587-588
  Keyboard Interrupt, 528, 536
  MS-DOS
    1.0 program terminate, 531

absolute disk read, 532
absolute disk write, 532
API, 532, 536
Control-C, 532
critical error interrupt, 532
idle loop, 533
terminate, 532
Terminate and Stay, 532
Multiplex and TSR communications, 533
NetBIOS Interrupt, 537
Non-Maskable Interrupt, 527
PC XT floppy disk interface, 533
Print Screen, 537
Printer port controller, 529
Reserved, 528
ROM BIOS
BASIC, 530
cassette service, 530
clock services, 530
COM port driver, 530
disk services, 530
equipment configuration check, 529
keyboard driver, 530
memory size check, 530
print screen, 528
printer driver, 530
video services, 529
Software timer tick, 531
System Restart, 530
Timer Tick, 536
version
ASCII files, tracking, 106-109
control system (VCS), 101-105, 110
labels, 116-117
management systems
ATTIC, 106-109
PVCS Version Manager, 110-120
VGA, 245
View | Breakpoints window (Turbo Debugger), 406
View | CPU mode, debugging in, 418-419
View | Hierarchy command, 402
View | Windows messages command, 414-416
viewing
all variables, 401-402
breakpoints, 406
portals, 278-280
value of variables during program execution, 389-390
ViewPoint library, 233-236
affine transform, 237
benefits, 239-240
features, 240-247
viewports, 278-280
class, 235, 242
virtual
8086 support, 4
debugging, 407-409
machine (VM), 584
voltage, 339

W

-w MAKE utility command-line switch option, 83
w+b (writable binary) stream format, 174
warnings
ANSI violations, 42
C++, 43
disabling display when compiling, 42-43
Frequent errors, 43
Less frequent errors, 43
Portability, 42
Watch window
IDE debugger, 388
Turbo Debugger, 399
Watches command, 399
whereis, 89
wildargs.obj object module, 193

wildcard filename specification, 107
Window | Next command, 389
Window | Previous command, 389
Window | Tile command, 388
Window | User screen command, 388
windows
  Call Stack (IDE debugger), 388
  code page, 493
  Control Panel
    international portion, 506
    International settings, 518-519
  Evaluate/Modify
    IDE debugger, 388-390
    Turbo Debugger, 401
  Execution Profile, 428-430
  Help Compiler, 503
  inspector (Turbo Debugger), 399-401
  localization, resource files, 501-502
  localized versions, 499-500
  Module, 428
  Registers (IDE debugger), 388
  Resource Compiler, 500-501
  Variables (Turbo Debugger), 401-402
  View | Breakpoints (Turbo Debugger), 406
  Watch
    IDE debugger, 388-390
    Turbo Debugger, 399
Winsight message intercept program, 416, 418
workfiles, 110-111
world coordinates, 235-237
write() function, 384
$WRITEMSG instruction macro, 35

X-Z

/x option (extended memory), 43
xasp parameter, 292-295
Xtract command, 108-109
yasp parameter, 292-295

C Programming Starter Kit Deluxe Edition

- Advanced C
- Borland C++ Tips, Tricks, and Traps
- Borland C++ Power Programming
- C Programming Just the FAQ's
- C++ Interactive Course
- Crash Course in Borland C++ 4
- Killer Borland C++ 4
- Programming Windows Games with Borland C++
- Secrets of the Borland C++ Masters
- Teach Yourself ANSI C++ in 21 Days
- Teach Yourself Advanced C in 21 Days
- Special Edition Using Borland C++