Steve Teixeira
Xavier Pacheco

Borland® Delphi™ 6

DEVELOPER'S GUIDE

"Steve and Xavier have done it again! They've crafted their developer's guide so that you can take advantage of the depth and breadth of Delphi 6 programming."

–David Intersimone
VP, Developer Relations
Borland

SAMS

Borland® Delphi™ 6 Developer's Guide

Steve Teixeira and Xavier Pacheco

SAMS

201 West 103rd St., Indianapolis, Indiana, 46290 USA

Borland®
Delphi™ 6 Developer's Guide

International Standard Book Number: 0-672-32115-7

Library of Congress Catalog Card Number: 2001086071

Printed in the United States of America

First Printing: October 2001

04 03 02 01 4 3 2 1

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.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Executive Editor
Michael Stephens

Acquisitions Editor
Carol Ackerman

Development Editor
Tiffany Taylor

Managing Editor
Matt Purcell

Project Editor
Christina Smith

Production Editor
Rhonda Tinch-Mize

Indexer
Sharon Shock

Proofreader
Harvey Stanbrough

Technical Editor
John Ray Thomas
Tom Theobold

Team Coordinator
Pamalee Nelson

Media Developer
Dan Scherf

Interior Designer
Anne Jones

Cover Designer
Aren Howell

Page Layout
Octal Publishing, Inc.

Contents at a Glance

Introduction

Part I: Development Essentials

1 Programming in Delphi

2 The Object Pascal Language

3 Adventures in Messaging

Part II: Advanced Techniques

4 Writing Portable Code

5 Multithreaded Techniques

6 Dynamic Link Libraries

Part III: Database Development

7 Delphi Database Architecture

8 Database Development with dbExpress

9 Database Development with dbGo for ADO

Part IV: Component-Based Development

10 Component Architecture: VCL and CLX

11 VCL Component Building

12 Advanced VCL Component Building

13 CLX Component Development

14 Packages to the Max

15 COM Development

16 Windows Shell Programming

17 Using the Open Tools API

Part V: Enterprise Development

18 Transactional Development with COM+/MTS

19 CORBA Development

20 BizSnap Development: Writing SOAP-Based Web Services

21 DataSnap Development

Part VI: Internet Development

22 ASP Development

23 Building WebSnap Applications

24 Wireless Development

Index

Table of Contents

Introduction 1

Who Should Read This Book 2
Conventions Used in This Book 2
Delphi 6 Developer's Guide Web Site 2
Getting Started 3

PART I Development Essentials 5

1 Programming in Delphi 7

The Delphi Product Family 8
Delphi: What and Why 10
The Quality of the Visual Development Environment 11
The Speediness of the Compiler Versus the Efficiency of the Compiled Code 12
The Power of the Programming Language Versus Its Complexity 13
The Flexibility and Scalability of the Database Architecture 14
The Design and Usage Patterns Enforced by the Framework 15
A Little History 15
Delphi 1 16
Delphi 2 16
Delphi 3 17
Delphi 4 18
Delphi 5 18
Delphi 6 19
The Delphi IDE 19
The Main Window 20
The Form Designer 22
The Object Inspector 22
The Code Editor 22
The Code Explorer 23
The Object TreeView 23
A Tour of Your Project's Source 24
Tour of a Small Application 26
What's So Great About Events, Anyway? 28
Contract-Free Programming 28
Turbo Prototyping 29
Extensible Components and Environment 29

The Top 10 IDE Features You Must Know and Love 30
1. Class Completion 30
2. AppBrowser Navigation 30
3. Interface/Implementation Navigation 31
4. Dock It! 31
5. The Object Browser 31
6. GUID, Anyone? 31
7. C++ Syntax Highlighting 32
8. To Do. 32
9. Use the Project Manager 32
10. Use Code Insight to Complete Declarations and Parameters 33
Summary 33

2 The Object Pascal Language 35

Comments 36
Extended Procedure and Function Features 37
Parentheses in Calls 37
Overloading 37
Default Value Parameters 38
Variables 39
Constants 41
Operators 43
Assignment Operators 43
Comparison Operators 43
Logical Operators 44
Arithmetic Operators 45
Bitwise Operators 46
Increment and Decrement Procedures 46
Do-and-Assign Operators 47
Object Pascal Types 48
A Comparison of Types 48
Characters 50
A Multitude of Strings 51
Variant Types 63
Currency 75
User-Defined Types 75
Arrays 76
Dynamic Arrays 77
Records 78
Sets 80
Objects 82
Pointers 83

Type Aliases 86
Typecasting and Type Conversion 87
String Resources 88
Testing Conditions 88
The if Statement 88
Using case Statements 89
Loops 90
The for Loop 90
The while Loop 91
repeat..until 92
The Break() Procedure 92
The Continue() Procedure 92
Procedures and Functions 93
Passing Parameters 94
Scope 98
Units 99
The uses Clause 100
Circular Unit References 101
Packages 101
Using Delphi Packages 102
Package Syntax 102
Object-Oriented Programming 103
Object-Based Versus Object-Oriented Programming 105
Using Delphi Objects 105
Declaration and Instantiation 105
Destruction 106
Methods 107
Method Types 108
Properties 110
Visibility Specifiers 111
Inside Objects 112
TObject: The Mother of All Objects 113
Interfaces 114
Structured Exception Handling 118
Exception Classes 121
Flow of Execution 123
Reraising an Exception 125
Runtime Type Information 126
Summary 127

3 Adventures in Messaging 129

What Is a Message? 130
Types of Messages 131

How the Windows Message System Works 132
Delphi's Message System 133
Message-Specific Records 134
Handling Messages 135
Message Handling: Not Contract Free 138
Assigning Message Result Values 139
The TApplication Type's OnMessage Event 139
Sending Your Own Messages 140
The Perform() Method 140
The SendMessage() and PostMessage() API Functions 141
Nonstandard Messages 142
Notification Messages 142
Internal VCL Messages 143
User-Defined Messages 144
Anatomy of a Message System: VCL 146
The Relationship Between Messages and Events 154
Summary 154

PART II Advanced Techniques 155

4 Writing Portable Code 157

General Compatibility 158
Which Version? 158
Units, Components, and Packages 160
IDE Issues 160
Delphi-Kylix Compatibility 161
Not in Linux 162
Compiler/Language Features 162
Platform-isms 163
New Delphi 6 Features 163
Variants 163
Enum Values 163
$IF Directive 164
Potential Binary DFM Incompatibility 164
Migrating from Delphi 5 164
Writable Typed Constants 164
Cardinal Unary Negation 164
Migrating from Delphi 4 165
RTL Issues 165
VCL Issues 165
Internet Development Issues 165
Database Issues 166

Migrating from Delphi 3 166
Unsigned 32-bit Integers 166
64-Bit Integers 168
The Real Type 168
Migrating from Delphi 2 168
Changes to Boolean Types 168
ResourceString 169
RTL Changes 169
TCustomForm 169
GetChildren() 170
Automation Servers 170
Migrating from Delphi 1 171
Summary 171

5 Multithreaded Techniques 173

Threads Explained 174
Types of Multitasking 174
Using Multiple Threads in Delphi Applications 175
Misuse of Threads 175
The TThread Object 176
TThread Basics 176
Thread Instances 180
Thread Termination 180
Synchronizing with VCL 182
A Demo Application 185
Priorities and Scheduling 187
Suspending and Resuming Threads 190
Timing a Thread 190
Managing Multiple Threads 192
Thread-Local Storage 192
Thread Synchronization 196
A Sample Multithreaded Application 210
The User Interface 211
The Search Thread 219
Adjusting the Priority 224
Multithreading BDE Access 227
Multithreaded Graphics 233
Fibers 238
Summary 244

6 Dynamic Link Libraries 247

What Exactly Is a DLL? 248
Static Linking Versus Dynamic Linking 250

Why Use DLLs? 252
Sharing Code, Resources, and Data with Multiple Applications ..252
Hiding Implementation 252
Creating and Using DLLs 253
Counting Your Pennies (A Simple DLL) 253
Displaying Modal Forms from DLLs 256
Displaying Modeless Forms from DLLs 259
Using DLLs in Your Delphi Applications 261
Loading DLLs Explicitly 263
The Dynamically Linked Library Entry/Exit Function 266
Process/Thread Initialization and Termination Routines 266
DLL Entry/Exit Example 267
Exceptions in DLLs 271
Capturing Exceptions in 16-Bit Delphi 271
Exceptions and the Safecall Directive 272
Callback Functions 273
Using the Callback Function 276
Drawing an Owner-Draw List Box 276
Calling Callback Functions from Your DLLs 277
Sharing DLL Data Across Different Processes 279
Creating a DLL with Shared Memory 280
Using a DLL with Shared Memory 284
Exporting Objects from DLLs 287
Summary 293

PART III Database Development 295

7 Delphi Database Architecture 297

Types of Databases 298
Database Architecture 299
Connecting to Database Servers 299
Overview of Database Connectivity 299
Establishing a Database Connection 300
Working with Datasets 300
Opening and Closing Datasets 301
Navigating Datasets 305
Manipulating Datasets 310
Working with Fields 315
Field Values 315
Field Data Types 316
Field Names and Numbers 317

Manipulating Field Data 317
The Fields Editor 318
Working with BLOB Fields 324
Filtering Data 330
Searching Datasets 332
Using Data Modules 336
The Search, Range, Filter Demo 337
Bookmarks 347
Summary 348

8 Database Development with dbExpress 349

Using dbExpress 350
Unidirectional, Read-Only Datasets 350
dbExpress Versus the Borland Database Engine (BDE) 350
dbExpress for Cross-Platform Development 351
dbExpress Components 351
TSQLConnection 351
TSQLDataset 354
Backward Compatibility Components 358
TSQLMonitor 358
Designing Editable dbExpress Applications 359
TSQLClientDataset 359
Deploying dbExpress Applications 360
Summary 361

9 Database Development with dbGo for ADO 363

Introduction to dbGo 364
Overview of Microsoft's Universal Data Access Strategy 364
Overview of OLE DB, ADO, and ODBC 364
Using dbGo for ADO 365
Establishing an OLE DB Provider for ODBC 365
The Access Database 367
dbGo for ADO Components 367
TADOConnection 368
Bypassing/Replacing the Login Prompt 370
TADOCommand 372
TADODataset 373
BDE-Like Dataset Components 373
TADOQuery 375
TADOStoredProc 375
Transaction Processing 375
Summary 377

PART IV Component-Based Development 379

10 Component Architecture: VCL and CLX 381

More on the New CLX 383
What Is a Component? 383
Component Hierarchy 384
Nonvisual Components 385
Visual Components 385
The Component Structure 387
Properties 388
Types of Properties 389
Methods 390
Events 390
Streamability 392
Ownership 393
Parenthood 394
The Visual Component Hierarchy 394
The TPersistent Class 395
TPersistent Methods 395
The TComponent Class 395
The TControl Class 397
The TWinControl and TWidgetControl 398
The TGraphicControl Class 399
The TCustomControl Class 400
Other Classes 400
Runtime Type Information 403
The TypInfo.pas Unit: Definer of Runtime Type Information 405
Obtaining Type Information 407
Obtaining Type Information on Method Pointers 416
Obtaining Type Information for Ordinal Types 420
Summary 428

11 VCL Component Building 429

Component Building Basics 430
Deciding Whether to Write a Component 430
Component Writing Steps 431
Deciding on an Ancestor Class 432
Creating a Component Unit 433
Creating Properties 435
Creating Events 445
Creating Methods 451
Constructors and Destructors 452

Registering Your Component 454
Testing the Component 456
Providing a Component Icon 458
Sample Components 459
Extending Win32 Component Wrapper Capabilities 459
TddgRunButton—Creating Properties 470
TddgButtonEdit—Container Components 477
Design Decisions 477
Surfacing Properties 478
Surfacing Events 478
TddgDigitalClock—Creating Component Events 481
Adding Forms to the Component Palette 485
Summary 488

12 Advanced VCL Component Building 489

Pseudo-Visual Components 490
Extending Hints 490
Creating a THintWindow Descendant 490
An Elliptical Window 493
Enabling the THintWindow Descendant 494
Deploying TDDGHintWindow 494
Animated Components 494
The Marquee Component 494
Writing the Component 495
Drawing on an Offscreen Bitmap 495
Painting the Component 497
Animating the Marquee 498
Testing TddgMarquee 508
Writing Property Editors 510
Creating a Descendant Property Editor Object 511
Editing the Property As Text 513
Registering the New Property Editor 517
Component Editors 522
TComponentEditor 523
TDefaultEditor 524
A Simple Component 524
A Simple Component Editor 525
Registering a Component Editor 526
Streaming Nonpublished Component Data 527
Defining Properties 528
An Example of DefineProperty() 529
TddgWaveFile: An Example of DefineBinaryProperty() 530

Property Categories 538
Category Classes 539
Custom Categories 540
Lists of Components: TCollection and TCollectionItem 543
Defining the TCollectionItem Class: TRunBtnItem 546
Defining the TCollection Class: TRunButtons 546
Implementing the TddgLaunchPad, TRunBtnItem, and TRunButtons Objects 547
Editing the List of TCollectionItem Components with a Dialog Property Editor 555
Summary 561

13 CLX Component Development 563

What Is CLX? 564
The CLX Architecture 565
Porting Issues 568
No More Messages 569
Sample Components 570
The TddgSpinner Component 570
Design-Time Enhancements 584
Component References and Image Lists 591
Data-Aware CLX Components 598
CLX Design Editors 608
Packages 613
Naming Conventions 613
Runtime Packages 615
Design-Time Packages 618
Registration Units 621
Component Bitmaps 622
Summary 623

14 Packages to the Max 625

Why Use Packages? 626
Code Reduction 626
A Smaller Distribution of Applications—Application Partitioning 626
Component Containment 627
Why Not Use Packages? 627
Types of Packages 628
Package Files 628
Using Runtime Packages 629
Installing Packages into the Delphi IDE 629

Creating Packages 630
The Package Editor 630
Package Design Scenarios 631
Package Versioning 635
Package Compiler Directives 635
More on {$WEAKPACKAGEUNIT} 636
Package Naming Conventions 637
Extensible Applications Using Runtime (Add-In) Packages 637
Generating Add-In Forms 637
Exporting Functions from Packages 644
Launching a Form from a Package Function 644
Obtaining Information About a Package 648
Summary 651

15 COM Development 653

COM Basics 654
COM: The Component Object Model 654
COM Versus ActiveX Versus OLE 655
Terminology 655
What's So Great About ActiveX? 656
OLE 1 Versus OLE 2 657
Structured Storage 657
Uniform Data Transfer 657
Threading Models 657
COM+ 658
COM Meets Object Pascal 658
Interfaces 658
Using Interfaces 661
The HResult Return Type 666
COM Objects and Class Factories 667
TComObject and TComObjectFactory 667
In-Process COM Servers 669
Out-of-Process COM Servers 672
Aggregation 672
Distributed COM 673
Automation 673
IDispatch 674
Type Information 675
Late Versus Early Binding 676
Registration 676
Creating Automation Servers 676
Creating Automation Controllers 692

Advanced Automation Techniques 700
Automation Events 700
Automation Collections 713
New Interface Types in the Type Library 723
Exchanging Binary Data 724
Behind the Scenes: Language Support for COM 727
TOleContainer 733
A Small Sample Application 733
A Bigger Sample Application 735
Summary 746

16 Windows Shell Programming 747

A Tray-Notification Icon Component 748
The API 748
Handling Messages 751
Icons and Hints 752
Mouse Clicks 752
Hiding the Application 755
Sample Tray Application 762
Application Desktop Toolbars 764
The API 764
TAppBar: The AppBar Form 766
Using TAppBar 775
Shell Links 779
Obtaining an IShellLink Instance 781
Using IShellLink 781
A Sample Application 790
Shell Extensions 799
The COM Object Wizard 801
Copy Hook Handlers 801
Context Menu Handlers 808
Icon Handlers 818
InfoTip Handlers 827
Summary 833

17 Using the Open Tools API 835

Open Tools Interfaces 836
Using the Open Tools API 839
A Dumb Wizard 839
The Wizard Wizard 843
DDG Search 855
Form Wizards 868
Summary 876

PART V Enterprise Development 877

18 Transactional Development with COM+/MTS 879

What Is COM+? 880
Why COM? 880
Services 881
Transactions 881
Security 882
Just-In-Time Activation 888
Queued Components 888
Object Pooling 897
Events 898
Runtime 906
Registration Database (RegDB) 907
Configured Components 907
Contexts 907
Neutral Threading 907
Creating COM+ Applications 908
The Goal: Scale 908
Execution Context 908
Stateful Versus Stateless 909
Lifetime Management 910
COM+ Application Organization 910
Thinking About Transactions 911
Resources 912
COM+ in Delphi 912
COM+ Wizards 912
COM+ Framework 913
Tic-Tac-Toe: A Sample Application 916
Debugging COM+ Applications 934
Summary 935

19 CORBA Development 937

CORBA Features 938
CORBA Architecture 939
OSAgent 941
Interfaces 942
Interface Definition Language (IDL) 942
Basic Types 943
User-Defined Types 944
Aliases 944
Enumerations 944
Structures 944

Arrays 944
Sequences 944
Method Arguments 945
Modules 945
The Bank Example 946
Complex Data Types 958
Delphi, CORBA, and Enterprise Java Beans (EJBs) 965
A Crash Course in EJBs for Delphi Programmers 965
An EJB Is a Specialized Component 966
EJBs Live Within a Container 966
EJBs Have Predefined APIs 966
The Home and Remote Interfaces 966
Types of EJBs 967
Configuring JBuilder 5 for EJB Development 967
Building a Simple "Hello, world" EJB 968
CORBA and Web Services 975
Creating the Web Service 975
Creating the SOAP Client Application 977
Adding the CORBA Client Code to the Web Service 978
Summary 981

20 BizSnap Development: Writing SOAP-Based Web Services 983

What Are Web Services? 984
What Is SOAP? 984
Writing a Web Service 985
A Look at the TWebModule 985
Defining an Invokable Interface 986
Implementing an Invokable Interface 987
Testing the Web Service 989
Invoking a Web Service from a Client 991
Generating an Import Unit for the Remote Invokable Object 993
Using the THTTPRIO Component 994
Summary 995

21 DataSnap Development 997

Mechanics of Creating a Multitier Application 998
Benefits of the Multitier Architecture 999
Centralized Business Logic 999
Thin-Client Architecture 1000
Automatic Error Reconciliation 1000
Briefcase Model 1000
Fault Tolerance 1000
Load Balancing 1000

Typical DataSnap Architecture 1001
Server 1001
Client 1004
Using DataSnap to Create an Application 1007
Setting Up the Server 1007
Creating the Client 1009
More Options to Make Your Application Robust 1015
Client Optimization Techniques 1015
Application Server Techniques 1018
Real-World Examples 1027
Joins 1027
More Client Dataset Features 1039
Two-Tier Applications 1039
Classic Mistakes 1041
Deploying DataSnap Applications 1041
Licensing Issues 1042
DCOM Configuration 1042
Files to Deploy 1043
Internet Deployment Considerations (Firewalls) 1044
Summary 1046

PART VI Internet Development 1047

22 ASP Development 1049

Understanding Active Server Objects 1050
Active Server Pages 1050
The Active Server Object Wizard 1052
Type Library Editor 1055
ASP Response Object 1059
First Run 1060
ASP Request Object 1061
Recompiling Active Server Objects 1062
Running Active Server Pages Again 1063
ASP Session, Server, and Application Objects 1065
Active Server Objects and Databases 1066
Active Server Objects and NetCLX Support 1069
Debugging Active Server Objects 1071
Debugging Active Server Objects with MTS 1071
Debugging Using Windows NT 4 1073
Debugging Using Windows 2000 1074
Summary 1076

23 Building WebSnap Applications 1077

WebSnap Features 1078
Multiple Webmodules 1078
Server-side Scripting 1078
TAdapter Components 1078
Multiple Dispatching Methods 1079
Page Producer Components 1079
Session Management 1079
Login Services 1079
User Tracking 1080
HTML Management 1080
File Uploading Services 1080
Building a WebSnap Application 1080
Designing the Application 1080
Adding Functionality to the Application 1089
Navigation Menu Bar 1089
Logging In 1092
Managing User Preference Data 1095
Persisting Preference Data Between Sessions 1099
Image Handling 1101
Displaying Data 1103
Converting the Application to an ISAPI DLL 1107
Advanced Topics 1107
LocateFileServices 1108
File Uploading 1109
Including Custom Templates 1111
Custom Components in TAdapterPageProducer 1112
Summary 1114

24 Wireless Development 1115

Evolution of Development—How Did We Get Here? 1116
Pre-1980s: Here There Be Dragons 1116
Late 1980s: Desktop Database Applications 1117
Early 1990s: Client/Server 1117
Late 1990s: Multitier and Internet-Based Transactions 1117
Early 2000s: Application Infrastructure Extends to Wireless Mobile Devices 1117
Mobile Wireless Devices 1118
Mobile Phones 1118
PalmOS Devices 1118
Pocket PC 1119
RIM BlackBerry 1119

Radio Technologies 1119
GSM, CDMA, and TDMA 1119
CDPD 1119
3G 1120
GPRS 1120
Bluetooth 1120
802.11 1120
Server-Based Wireless Data Technologies 1121
SMS 1121
WAP 1121
I-mode 1132
PQA 1132
Wireless User Experience 1136
Circuit-Switched Versus Packet-Switched Networks 1137
Wireless Is Not the Web 1137
The Importance of Form Factor 1137
Data Entry and Navigation Techniques 1137
M-Commerce 1138
Summary 1138

Foreword

"Delphi 6—two years in the making; a lifetime of productivity."

I have been happily employed at Borland for more than 16 years now. I came to work here, in the summer of 1985, to 1) be a part of the new generation of programming tools (the UCSD Pascal System and command line tools just weren't enough), 2) help improve the process of programming (maybe even leaving a little more time for our families and friends), and 3) help enrich the lives of programmers (myself included). We been innovating and advancing developer technology for the past 18 years. I enjoy being a part of this great worldwide Borland community.

Turbo Pascal 1.0 changed the face of programming tools forever. It set the standard in 1983. Delphi also changed the face of programming once again. Delphi 1.0 focused on making object-oriented programming, Windows programming, and database programming easier. Later versions of Delphi focused on easing the pain of writing Internet and distributed applications. Even though we've added a host of features to our products over the years and written pages of documentation and megabytes of online help, there's still more information, knowledge, and advice that is required for developers to complete successful projects.

How do you top the award winning and universally praised Delphi 5? Didn't Delphi 5 already simplify the process of building Internet and distributed applications while also improving the productivity of Delphi programmers? Could the Delphi team push themselves again to meet the demands of today's and tomorrow's developers?

The Delphi team spent more than two years listening to customers, seeing how developers were using the product, looking at the pain points of programming in the new millennium. They focused their efforts on radically simplifying the process of developing next generation e-business Web applications, XML/SOAP based Web Services, B2b/B2C/P2P application integration, cross-platform applications, distributed applications including integration with AppServer/EJBs, and Microsoft Windows ME/2000 and Office 2000 applications.

Steve Teixeira and Xavier Pacheco have done it again. They have crafted their developer's guide so that you can take advantage of the depth and breadth of Delphi 6 programming.

I've known Steve Teixeira (some call him T-Rex) and Xavier Pacheco (some call him just X) for years as friends, fellow employees, speakers at our annual conference, and as members of the Borland community.

Previous versions of their developer's guides have been received enthusiastically by Delphi developers around the world. Here now is the latest version ready for everyone to enjoy.

Have fun, learn a lot. Here's hoping that all of your Delphi projects are enjoyable, successful, and rewarding.

David Intersimone (David I)
Vice President, Developer Relations
Borland Software Corporation
davidi@borland.com

About the Lead Authors

Steve Teixeira is the Director of Core Technology at Zone Labs, a leading creator of Internet security solutions. Steve has previously served as Chief Technology Officer of ThinSpace, a mobile/wireless software company, and Full Moon Interactive, a full-service e-business builder. As a research and development software engineer at Borland, Steve was instrumental in the development of Delphi and C++Builder. Steve is the best-selling author of four award-winning books and numerous magazine articles on software development, and his writings are distributed worldwide in a dozen languages. Steve is a frequent speaker at industry conferences and events worldwide.

Xavier Pacheco is the President and CEO of Xapware Technologies Inc, a software development and consulting company with a purpose of accelerating visions. Xavier is a frequent speaker at industry conferences and is a contributing author for Delphi periodicals. Xavier is an internationally known Delphi expert and member of Borland's select group volunteers—TeamB. He is the best-selling author of four award-winning books that are distributed worldwide in a dozen languages. Xavier lives in Colorado Springs with his wife Anne and children Amanda and Zachary.

About the Contributing Authors

Bob Swart (also known as Dr.Bob—`www.drbob42.com`) is a UK Borland Connections member and an independent technical author, trainer, and consultant using Delphi, Kylix, and C++Builder based in Helmond, The Netherlands. Bob writes regular columns for *The Delphi Magazine*, *Delphi Developer*, *UK-BUG Developer's Magazine*, as well as the DevX, TechRepublic, and the Borland Community Web sites. Bob has written chapters for *The Revolutionary Guide to Delphi 2*, *Delphi 4 Unleashed*, *C++Builder 4 Unleashed*, *C++Builder 5 Developer's Guide*, *Kylix Developer's Guide*, and now *Delphi 6 Developer's Guide* (for Sams Publishing).

Bob is a frequent speaker at Borland and Delphi/Kylix related seminars all over the world, and writes his own training material for Dr.Bob's Delphi Clinics (in The Netherlands and the UK).

In his spare time, Bob likes to watch video tapes of *Star Trek Voyager* and *Deep Space Nine* with his 7-year old son Erik Mark Pascal and 5-year old daughter Natasha Louise Delphine.

Dan Miser is an R&D Project Manager for the DSP group at Borland, where he spends most of his time researching emerging technologies. Dan also worked on the Delphi R&D team where his responsibilities included DataSnap development. Dan's major focus is finding ways to allow information to be shared across boundaries, and this has allowed him to work with a variety of distributed computing technologies, including MIDAS, SOAP, DCOM, RMI, J2EE, EJB, Struts, and RDS. He has also been involved with promoting Delphi by being a contributing author to the Delphi Developer's Guide series, acting as a technical editor, writing magazine articles, participating on the Borland newsgroups as a member of TeamB, and being a speaker at BorCon on topics such as COM and MIDAS.

David Sampson is an R&D engineer in the Borland RAD Tools Group and is responsible for the CORBA integration into the RAD products. He is long time Pascal, Delphi, and C++ developer, and is a frequent speaker at the Borland Developer's Conference. He lives in Roswell, GA with his wife and enjoys hockey, Aikido, and helping his wife with her pack of Basenjis.

Nick Hodges is a Senior Development Engineer with Lemanix Corporation in St. Paul, MN. He is a member of Borland's TeamB and a long time Pascal and Delphi developer. He serves on the Borland Conference Advisory Board, is a frequent speaker at the conference, and is a frequent writer for the Borland Community Site. He lives in St. Paul with his wife and two children and enjoys reading, running, and helping his wife homeschool their two children.

Ray Konopka is the founder of Raize Software, Inc. and the chief architect for CodeSite and Raize Components. Ray is also the author of the highly acclaimed Developing Custom Delphi Components books and the popular "Delphi by Design" column, which appeared in *Visual Developer Magazine*. Ray specializes in user interface design and Delphi component development, and is a frequent speaker at developer conferences around the world.

Dedication

This book is dedicated to the victims and heroes of September 11, 2001.

Thanks to my family, Helen, Cooper, and Ryan. Without their love, support, and welcome distractions, I'd likely never be able to finish a book, and I'd almost certainly go crazy trying.

—Steve

Thanks to my family, Anne, Amanda, and Zachary. Your love, patience, and encouragement, I cherish.

—Xavier

Acknowledgments

We need to thank those who, without whose help, this book would never have been written. In addition to our thanks, we also want to point out that any errors or omissions you find in the book are our own, in spite of everyone's efforts.

We'd first like to offer our enormous gratitude to our contributing authors, who lent their superior software development and writing skills to making *Delphi 6 Developer's Guide* better than it could have been otherwise. Mr. Component himself, Ray Konopka, wrote the excellent Chapter 13, "CLX Component Development." DataSnap guru Dan Miser pitched in by writing the brilliant Chapter 21, "DataSnap Development." Well-known CORBA expert, David Sampson, contributed Chapter 19, "CORBA Development." Thank you also to Robert "Dr. Bob" Swart, for bringing his considerable talents to bear on Chapter 22, "ASP Development." Last (but certainly not least!), Web wizard Nick Hodges is back in this edition of the book in Chapter 23, "Building WebSnap Applications."

Another large round of thank-yous to our technical reviewers (and all around great guys), Thomas Theobald and John Thomas. These guys managed to squeeze in their duties as uber-technical reviewers among their day jobs of helping Borland create great software.

While writing the Delphi Developer's Guide series, we received advice or tips from a number of our friends and coworkers. These people include (in alphabetical order) Alain "Lino" Tadros, Anders Hejlsberg, Anders Ohlsson, Charlie Calvert, Victor Hornback, Chuck Jazdzewski, Daniel Polistchuck, Danny Thorpe, David Streever, Ellie Peters, Jeff Peters, Lance Bullock, Mark Duncan, Mike Dugan, Nick Hodges, Paul Qualls, Rich Jones, Roland Bouchereau, Scott Frolich, Steve Beebe, and Tom Butt. We're certain there are others whose names we can't recall, and we owe you all a beer.

Finally, thanks to the gang at Pearson Technology Group: Carol Ackerman, Christina Smith, Dan Scherf, and the zillions of behind-the-scenes people whom we never met, but without whose help this book would not be a reality.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and authors' names as well as your name and phone or fax number. I will carefully review your comments and share them with the authors and editors who worked on the book.

Fax: 317-581-4770

E-mail: `feedback@samspublishing.com`

Mail: Michael Stephens
Executive Editor
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

You hold in your hands the fifth edition in the *Delphi Developer's Guide* series, and the product of literally thousands of man-hours over more than seven years of programming, writing, and refinement. Xavier and Steve were members of the original Delphi team at Borland, and this work is the outlet through which they can share their fifteen-plus years of combined experience developing software in Delphi. In *Delphi 6 Developer's Guide*, we have striven to hold true to the spirit that has made the Delphi Developer's Guide series perhaps the world's most read Delphi books and two-time winner of the Delphi Informant Reader's Choice award. This is a book by developers, for developers.

The intent of *Delphi 6 Developer's Guide* is to supplement and build on the Delphi Developer's Guide series. Ideally, we would have loved to include all the updated content form *Delphi 5 Developer's Guide* and all the new content in one book, but *Delphi 5 Developer's Guide* was already thick enough to stretch the technical limitations of modern book binding. In order to provide enough space to give proper coverage of the entire Delphi 6 feature set, we opted to publish a new book with new information.

Delphi 6 Developer's Guide contains a number of all-new chapters, many chapters that have been significantly enhanced from previous editions, and some of the favorite topics from *Delphi 5 Developers Guide*. The information in *Delphi 5 Developer's Guide* will not be lost, however. On the CD accompanying this book, you will find the entire contents of *Delphi 5 Developer's Guide*, with each chapter in a separate PDF file. On the inside front cover, we have also included the table of contents for *Delphi 5 Developer's Guide* so you can know at a glance where to find that programming tidbit. The end result for you, the reader, is essentially two books in one.

Delphi 6 Developer's Guide is divided into six sections. Part I, "Development Essentials," provides you with the foundation knowledge necessary to be an effective Delphi developers. Part II, "Advanced Techniques," highlights some common advanced development issues, such as threading and dynamic link libraries. Part III, "Database Development," discusses the many faces of Delphi's data access layers. Part IV, "Component-Based Development," takes you through the many manifestations of component-based development, from VCL to CLX to packages to COM and the Open Tools API. Part V, "Enterprise Development," is intended to give you the practical knowledge necessary to develop enterprise-grade applications with technologies such as COM+, CORBA, SOAP/BizSnap, and DataSnap. Finally, Part VI, "Internet Development," demonstrates the development of Internet and wireless applications in Delphi.

Who Should Read This Book

As the title of this book says, this book is for developers. So, if you're a developer, and you use Delphi, you need to have this book. In particular, however, this book is aimed at three groups of people:

- Delphi developers who are looking to take their craft to the next level.
- Experienced Pascal, C/C++, Java, or Basic programmers who are looking to hit the ground running with Delphi.
- Programmers who are looking to get the most out of Delphi by leveraging some of its more advanced and sometimes least obvious features.

Conventions Used in This Book

The following typographic conventions are used in this book:

- Code lines, commands, statements, variables, program output, and any text you see on the screen appear in a computer typeface.
- Anything that you type appears in a bold computer typeface.
- Placeholders in syntax descriptions appear in an italic computer typeface. Replace the placeholder with the actual filename, parameter, or whatever element it represents.
- Italics highlight technical terms when they first appear in the text and sometimes are used to emphasize important points.
- Procedures and functions are indicated by open and close parentheses after the procedure or function name. Although this isn't standard Pascal syntax, it helps to differentiate them from properties, variables, and types.

Within each chapter, you will encounter several Notes, Tips, and Cautions that help to highlight the important points and aid you in steering clear of the pitfalls.

You will find all the source code and project files on the CD-ROM accompanying this book, as well as source samples that we could not fit in the book itself.The CD also contains some powerful trial versions of third-party components and tools.

Delphi 6 Developer's Guide Web Site

Visit our Web site at `http://www.xapware.com/ddg` to join the *Delphi Developer's Guide* community and obtain updates, extras, and errata information for this book. You can also join the mailing list for our newsletter and visit our discussion group.

Getting Started

People sometimes ask what drives us to continue to write Delphi books. It's hard to explain, but whenever we meet with other developers and see their obviously well used, book marked, ratty looking copy of *Delphi Developer's Guide*, it somehow makes it worthwhile.

Now it's time to relax and have some fun programming with Delphi. We'll start slow but progress into the more advanced topics at a quick but comfortable pace. Before you know it, you'll have the knowledge and technique required to truly be called a Delphi guru.

This page intentionally left blank

PART I

Development Essentials

IN THIS PART

1 Programming in Delphi 7

2 The Object Pascal Language 35

3 Adventures in Messaging 129

This page intentionally left blank

Programming in Delphi

CHAPTER

1

IN THIS CHAPTER

- The Delphi Product Family 8
- Delphi: What and Why 10
- A Little History 15
- The Delphi IDE 19
- A Tour of Your Project's Source 24
- Tour of a Small Application 26
- What's So Great About Events, Anyway? 28
- Turbo Prototyping 29
- Extensible Components and Environment 25
- The Top 10 IDE Features You Must Know and Love 30

This chapter is intended to provide you with a high-level overview of Delphi, including history, feature sets, how Delphi fits into the world of Windows development, and general tidbits of information you need to know to be a Delphi developer. And just to get your technical juices flowing, this chapter also discusses the need-to-know features of the Delphi IDE, pointing out some of those hard-to-find features that even seasoned Delphi developers might not know about.

This chapter isn't about providing an education on the very basics of how one develops software in Delphi. We figure you spent good money on this book to learn new and interesting things—not to read a rehash of content you can already find in Borland's documentation. True to that, our mission is to deliver the goods: to show you the power features of this product and ultimately how to employ those features to build commercial-quality software. Hopefully, our backgrounds and experience with the tool will enable us to provide you with some interesting and useful insights along the way. We feel that experienced and new Delphi developers alike will benefit from this chapter (and this book!), as long as new developers understand that this isn't ground zero for a Delphi developer. Start with the Borland documentation and simple examples. Once you've got the hang of how the IDE works and the general flow of application development, welcome aboard and enjoy the ride!

The Delphi Product Family

Delphi 6 comes in three flavors designed to fit a variety of needs: Delphi 6 Personal, Delphi 6 Professional, and Delphi 6 Enterprise. Each of these versions is targeted at a different type of developer.

Delphi 6 Personal is the entry-level version. It provides everything you need to start writing applications with Delphi, and it's ideal for hobbyists and students who want to break into Delphi programming on a budget. This version includes the following features:

- Optimizing 32-bit Object Pascal compiler, including a variety of new and enhanced language features.
- Visual Component Library (VCL), which includes over 85 components standard on the Component Palette.
- Package support, which enables you to create small executables and component libraries.
- An IDE that includes an editor, debugger, form designer, and a host of productivity features.
- IDE enhancements such as visual form inheritance and linking, object tree view, class completion, and Code Insight.

- Full support for Win32 API, including COM, GDI, DirectX, multithreading, and various Microsoft and third-party software development kits (SDKs).
- Licensing permits building applications for personal use only: No commercial distribution of applications built with Delphi 6 Personal is permitted.

Delphi 6 Professional is intended for use by professional developers who don't require enterprise development capabilities. If you're a professional developer building and deploying applications or Delphi components, this product is designed for you. The Professional edition includes everything in the Personal edition, plus the following:

- More than 225 VCL components on the Component Palette
- More than 160 CLX components for cross-platform development between Windows and Linux
- Database support, including DataCLX database architecture, data-aware VCL controls, dbExpress cross-platform components and drivers, ActiveX Data Objects (ADO), the Borland Database Engine (BDE) for legacy connectivity, a virtual dataset architecture that enables you to incorporate other database types into VCL, the Database Explorer tool, a data repository, and InterBase Express native InterBase components
- InterBase and MySQL drivers for dbExpress
- DataCLX database architecture (formerly known as MIDAS) with MyBase XML-based local data engine
- Wizards for creating COM/COM+ components, such as ActiveX controls, ActiveForms, Automation servers, property pages, and transactional components
- A variety of third-party tools and components, include the INDY internet tools, the QuickReports reporting tool, the TeeChart graphing and charting components, and NetMasters FastNet controls
- InterBase 6 database server and five-user license
- The Web Deployment feature for easy distribution of ActiveX content via the Web
- The InstallSHIELD MSI Light application-deployment tool
- The OpenTools API for developing components that integrate tightly within the Delphi environment as well as an interface for PVCS version control
- NetCLX WebBroker tools and components for developing cross-platform applications for the Internet
- Source code for the Visual Component Library (VCL), Component Library for Cross-platform (CLX), runtime library (RTL), and property editors
- License for commercial distribution of applications developed with Delphi 6 Professional

Delphi 6 Enterprise is targeted toward developers who create enterprise-scale applications. The Enterprise version includes everything included in the other two Delphi editions, plus the following:

- Over 300 VCL components on the Component Palette
- BizSnap technology for creating XML-based applications and Web services
- WebSnap Web application design platform for integrating XML and scripting technologies with Web-based applications
- CORBA support for client and sever applications, including version 4.0x of the VisiBroker ORB and Borland AppServer version 4.5
- TeamSource source control software, which enables team development and supports various versioning engines (ZIP and PVCS included)
- Tools for easily translating and localizing applications
- SQLLinks BDE drivers for Oracle, MS SQL Server, InterBase, Informix, Sybase, and DB2
- Oracle and DB2 drivers for dbExpress
- Advanced tools for building SQL-based applications, including SQL Explorer, SQL Monitor, SQL Builder, and ADT column support in grid

Delphi: What and Why

We're often asked questions such as "What makes Delphi so good?" and "Why should I choose Delphi over Tool X?" Over the years, we've developed two answers to these types of questions: a long answer and a short answer. The short answer is *productivity*. Using Delphi is simply the most productive way we've found to build applications for Windows. Of course, there are those (bosses and perspective clients) for whom the short answer will not suffice, so then we must break out the long answer. The long answer describes the combined qualities that make Delphi so productive. We boil down the productivity of software development tools into a pentagon of five important attributes:

- The quality of the visual development environment
- The speediness of the compiler versus the efficiency of the compiled code
- The power of the programming language versus its complexity
- The flexibility and scalability of the database architecture
- The design and usage patterns enforced by the framework

Although admittedly many other factors are involved, such as deployment issues, documentation, third-party support, and so on, we've found this simple model to be quite accurate in

explaining to folks why we choose Delphi. Some of these categories also involve some amount of subjectivity, but that's the point; how productive are *you* with a particular tool? By rating a tool on a scale of 1 to 5 for each attribute and plotting each on an axis of the graph shown in Figure 1.1, the end result will be a pentagon. The greater the surface area of this pentagon, the more productive the tool.

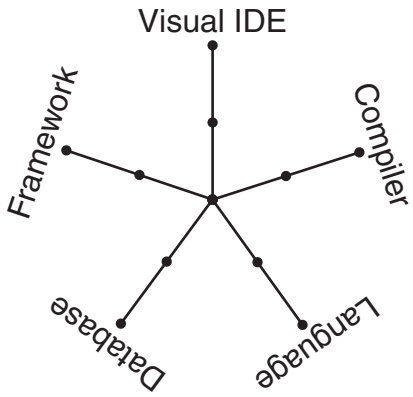

FIGURE 1.1
The development tool productivity graph.

We won't tell you what we came up with when we used this formula—that's for you to decide! Let's take an in-depth look at each of these attributes and how they apply to Delphi as well as how they compare with other Windows development tools.

The Quality of the Visual Development Environment

The visual development environment can generally be divided into three constituent components: the editor, the debugger, and the form designer. Like most modern *rapid application development (RAD)* tools, these three components work in harmony as you design an application. While you're working in the form designer, Delphi is generating code behind the scenes for the components you drop and manipulate on forms. You can add additional code in the editor to define application behavior, and you can debug your application from the same editor by setting breakpoints, watches, and so on.

Delphi's editor is generally on par with those of other tools. The CodeInsight technologies, which save you a lot of typing, are probably the best around. They're based on compiler information, rather than type library info like Visual Basic, and are therefore able to help in a wider variety of situations. Although the Delphi editor sports some good configuration options, I would rate Visual Studio's editor as more configurable.

Recent versions of Delphi's debugger have finally caught up with the debugger support in Visual Studio, with advanced features such as remote debugging, process attachment, DLL and package debugging, automatic local watches, and a CPU window. Delphi also has some nice IDE support for debugging by allowing windows to be placed and docked where you like during debugging and enabling that state to be saved as a named desktop setting. One very nice debugger feature that's commonplace in interpreted environments such as Visual Basic and some Java tools is the ability to change code to modify application behavior while the application is being debugged. Unfortunately, this type of feature is much more difficult to accomplish when compiling to native code and is therefore unsupported by Delphi.

A form designer is usually a feature unique to RAD tools, such as Delphi, Visual Basic, C++Builder, and PowerBuilder. More classical development environments, such as Visual C++ and Borland C++, typically provide dialog editors, but those tend not to be as integrated into the development workflow as a form designer. Based on the productivity graph from Figure 1.1, you can see that the lack of a form designer really has a negative effect on the overall productivity of the tool for application development.

Over the years, Delphi and Visual Basic have engaged in a sort of tug-of-war of form designer features, with each new version surpassing the other in functionality. One trait of Delphi's form designer that sets it apart from others is the fact that Delphi is built on top of a true object-oriented framework. Given that, changes you make to base classes will propagate up to any ancestor classes. A key feature that leverages this trait is *visual form inheritance* (VFI). VFI enables you to dynamically descend from any of the other forms in your project or in the Gallery. What's more, changes made to the base form from which you descend will cascade and reflect in its descendants. You'll find more information on this feature in the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this book in Chapter 3, "Application Frameworks and Design Concepts."

The Speediness of the Compiler Versus the Efficiency of the Compiled Code

A speedy compile enables you to develop software incrementally, thus making frequent changes to your source code, recompiling, testing, changing, recompiling, testing again, and so forth: a very efficient development cycle. When compilation speed is slower, developers are forced to make source changes in batch, making multiple modifications prior to compiling and adapting to a less efficient development cycle. The advantage of runtime efficiency is self-evident; faster runtime execution and smaller binaries are always good.

Perhaps the best-known feature of the Pascal compiler upon which Delphi is based is that it's fast. In fact, it's probably the fastest high-level language native code compiler for Windows.

C++, which has traditionally been dog-slow in terms of compile speed, has made great strides in recent years with incremental linking and various caching strategies found in Visual C++ and C++Builder in particular. Still, even these C++ compilers are typically several times slower than Delphi's compiler.

Does all this compile-time speed mean a tradeoff in runtime efficiency? The answer is, of course, no. Delphi shares the compiler back end with the C++Builder compiler, so the efficiency of the generated code is on par with that of a very good C++ compiler. In the latest reliable benchmarks, Visual C++ actually rated tops in speed and size efficiency in many cases, thanks to some very nice optimizations. Although these small advantages are unnoticeable for general application development, they might make a difference if you're writing computation-intensive code.

Visual Basic is a little unique with regard to compiler technology. During development, VB operates in an interpreted mode and is quite responsive. When you want to deploy, you can invoke the VB compiler to generate the EXE. This compiler is fairly slow and its speed efficiency rates well behind Delphi and C++ tools. At the time of this writing, Microsoft's next iteration, Visual Basic.NET, is in beta and promises to make improvements in this area.

Java is another interesting case. Top Java-based tools such as JBuilder and Visual J++ boast compile times approaching that of Delphi. Runtime speed efficiency, however, often leaves something to be desired because Java is an interpreted language. Although Java continues to make steady improvements, runtime speed in most real-world scenarios lags behind that of Delphi and C++.

The Power of the Programming Language Versus Its Complexity

Power and complexity are very much in the eye of the beholder, and this particular category has served as the guidon for many an online flame war. What's easy to one person might be difficult to another, and what's limiting to one might be considered elegant by yet another. Therefore, the following is based on the authors' experience and personal preferences.

Assembly is the ultimate power language. There's very little you can't do. However, writing even the simplest Windows application in assembly is an arduous and error-prone venture. Not only that, but it's sometimes nearly impossible to maintain an assembly code base in a team environment for any length of time. As code passes from one owner to the next to the next, design ideas and intents become more and more cloudy, until the code starts to look more like Sanskrit than a computer language. Therefore, we would score assembly very low in this category because, although powerful, assembly language is too complex for nearly all application development chores.

C++ is another extremely powerful language. With the aid of really potent features such as preprocessor macros, templates, operator overloading, and more, you can very nearly design your own language within C++. If the vast array of features at your disposal are used judiciously, you can develop very clear and maintainable code. The problem, however, is that many developers can't resist overusing these features, and it's quite easy to create truly horrible code. In fact, it's easier to write bad C++ code than good because the language doesn't lend itself toward good design—it's up to the developer.

Two languages that we feel are very similar in that they strike a very good balance between complexity and power are Object Pascal and Java. Both take the approach of limiting available features in an effort to enforce logical design on the developer. For example, both avoid the very object-oriented but easy-to-abuse notion of multiple inheritance in favor of enabling a class to implement multiple interfaces. Both lack the nifty but dangerous feature of operator overloading. Also, both make source files first-class citizens in the language rather than a detail to be dealt with by the linker. What's more, both languages take advantage of power features that add the most bang for the buck, such as exception handling, Runtime Type Information (RTTI), and native memory-managed strings. Not coincidentally, both languages weren't written by committee but rather nurtured by an individual or small group within a single organization with a common understanding of what the language should be.

Visual Basic started life as a language designed to be easy enough for programming beginners to pick up quickly (hence the name). However, as language features were added to address shortcomings over the years, Visual Basic has become more and more complex. In an effort to hide the details from developers, Visual Basic still maintains some walls that must be navigated around in order to build complex projects. Again, Microsoft's next-generation Visual Basic.NET is making significant changes in this area, albeit at the expense of backward compatibility.

The Flexibility and Scalability of the Database Architecture

Because of Borland's lack of a database agenda, Delphi maintains what we feel to be one of the most flexible database architectures of any tool. Out of the box, dbExpress is very efficient (although at the expense of advanced functionality), but the selection of drivers is rather limited. BDE still works and performs relatively well for most applications against a wide range of data sources, although it is being phased out by Borland. Additionally, the native ADO components provide an efficient means for communicating through ADO or ODBC. If InterBase is your bag, the IBExpress native InterBase components provide the most effective means to communicate with that database server. If none of this provides the data access you're looking

for, you can write your own data-access class by leveraging the abstract dataset architecture or purchase a third-party dataset solution. Furthermore, DataCLX makes it easy to logically or physically divide, into multiple tiers, access to any of these data sources.

Microsoft tools logically tend to focus on Microsoft's own databases and data-access solutions, be they ODBC, OLE DB, or others.

The Design and Usage Patterns Enforced by the Framework

This is the magic bullet or the holy grail of software design that other tools seem to be missing. All other things being equal, VCL is the most important part of Delphi. The ability to manipulate components at design time, design components, and inherit behavior from other components using object-oriented (OO) techniques it a critical ingredient to Delphi's level of productivity. When writing VCL components, you can't help but employ solid OO design methodologies in many cases. By contrast, other component-based frameworks are often too rigid or too complicated.

ActiveX controls, for example, provide many of the same design-time benefits of VCL controls, but there's no way to inherit from an ActiveX control to create a new class with some different behaviors. Traditional class frameworks, such as OWL and MFC, typically require you to have a great deal of internal framework knowledge in order to be productive, and they're hampered by a lack of RAD tool-like design-time support. Microsoft's .NET common library finally puts Microsoft on the right track in terms of component-based development, and it even works with a variety of their tools, including C#, Visual C++, and Visual Basic.

A Little History

Delphi is, at heart, a Pascal compiler. Delphi 6 is the next step in the evolution of the same Pascal compiler that Borland has been developing since Anders Hejlsberg wrote the first Turbo Pascal compiler more than 17 years ago. Pascal programmers throughout the years have enjoyed the stability, grace, and, of course, the compile speed that Turbo Pascal offers. Delphi 6 is no exception—its compiler is the synthesis of more than a decade of compiler experience and a state-of-the-art 32-bit optimizing compiler. Although the capabilities of the compiler have grown considerably over the years, the speed of the compiler has remarkably diminished only slightly. What's more, the stability of the Delphi compiler continues to be a yardstick by which others are measured.

Now it's time for a little walk down memory lane, as we look at each of the versions of Delphi and a little of the historical context surrounding each product's release.

Delphi 1

In the early days of DOS, programmers had a choice between productive-but-slow BASIC and efficient-but-complex assembly language. Turbo Pascal, which offered the simplicity of a structured language and the performance of a real compiler, bridged that gap. Windows 3.1 programmers faced a similar choice—a choice between a powerful-yet-unwieldy language such as C++ and an easy-to-use-but-limiting language such as Visual Basic. Delphi 1 answered that call by offering a radically different approach to Windows development: visual development, compiled executables, DLLs, databases, you name it—a visual environment without limits. Delphi 1 was the first Windows development tool to combine a visual development environment, an optimizing native-code compiler, and a scalable database access engine. It defined the phrase *rapid application development (RAD)*.

The combination of compiler, RAD tool, and fast database access was too compelling for scads of VB developers, and Delphi won many converts. Also, many Turbo Pascal developers reinvented their careers by transitioning to this slick, new tool. Word got out that Object Pascal wasn't the same as that language we had to use in college that made us feel like we were programming with one hand behind our backs, and many more developers came to Delphi to take advantage of the robust design patterns encouraged by the language and the tool. The Visual Basic team at Microsoft, lacking serious competition before Delphi, was caught totally unprepared. Slow, fat, and dumb, Visual Basic 3 was arguably no match for Delphi 1.

The year was 1995. Borland was appealing a huge lawsuit loss to Lotus for infringing on the 1-2-3 "look and feel" with Quattro. Borland was also taking lumps from Microsoft for trying to play in the application space with Microsoft. Borland got out of the application business by selling the Quattro business to Novell and targeting dBASE and Paradox to database developers, as opposed to casual users. While Borland was playing in the applications market, Microsoft had quietly leveraged its platform business to take away from Borland a vast share of the Windows developer tools market. Newly refocused on its core competency of developer tools, Borland was looking to do some damage with Delphi and a new release of Borland C++.

Delphi 2

A year later, Delphi 2 provided all these same benefits under the modern 32-bit operating systems of Windows 95 and Windows NT. Additionally, Delphi 2 extended productivity with additional features and functionality not found in version 1, such as a 32-bit compiler that produces faster applications, an enhanced and extended object library, revamped database support, improved string handling, OLE support, Visual Form Inheritance, and compatibility with 16-bit Delphi projects. Delphi 2 became the yardstick by which all other RAD tools are measured.

The year was 1996, and the most important Windows platform release since 3.0—32-bit Windows 95—had just happened in the latter part of the previous year. Borland was eager to make Delphi the preeminent development tool for that platform. An interesting historical note is that Delphi 2 was originally going to be called *Delphi32*, to underscore the fact that it was designed for 32-bit Windows. However, the product name was changed before release to Delphi 2 to illustrate that Delphi was a mature product and avoid what is known in the software business as the "1.0 blues."

Microsoft attempted to counter with Visual Basic 4, but it was plagued by poor performance, lack of 16-to-32-bit portability, and key design flaws. Still, there's an impressive number of developers who continued to use Visual Basic for whatever the reason. Borland also longed to see Delphi penetrate the high-end client/server market occupied by tools such as PowerBuilder, but this version didn't yet have the muscle necessary to unseat such products from their corporate perches.

The corporate strategy at this time was undeniably to focus on corporate customers. The decision to change direction in this way was no doubt fueled by the diminishing market relevance of dBASE and Paradox, and the dwindling revenues realized in the C++ market also aided this decision. In order to help jumpstart that effort to take on the enterprises, Borland made the mistake of acquiring Open Environment Corporation, a middleware company with basically two products: an outmoded DCE-based middleware that you might call an ancestor of CORBA and a proprietary technology for distributed OLE about to be ushered into obsolescence by DCOM.

Delphi 3

During the development of Delphi 1, the Delphi development team was preoccupied with simply creating and releasing a groundbreaking development tool. For Delphi 2, the development team had its hands full primarily with the tasks of moving to 32 bit (while maintaining almost complete backward compatibility) and adding new database and client/server features needed by corporate IT. While Delphi 3 was being created, the development team had the opportunity to expand the tool set to provide an extraordinary level of breadth and depth for solutions to some of the sticky problems faced by Windows developers. In particular, Delphi 3 made it easy to use the notoriously complicated technologies of COM and ActiveX, World Wide Web application development, "thin client" applications, and multitier databases architectures. Delphi 3's Code Insight helped to make the actual code-writing process a bit easier, although for the most part, the basic methodology for writing Delphi applications was the same as in Delphi 1.

This was 1997, and the competition was doing some interesting things. On the low end, Microsoft finally started to get something right with Visual Basic 5, which included a compiler to address long-standing performance problems, good COM/ActiveX support, and some key

new platform features. On the high-end, Delphi was now successfully unseating products such as PowerBuilder and Forte in corporations.

Delphi lost a key member of the team during the Delphi 3 development cycle when Anders Hejlsberg, the Chief Architect, decided to move on and took a position with Microsoft Corporation. The team didn't lose a beat, however, because Chuck Jazdzewski, long time co-architect was able to step into the head role.

Delphi 4

Delphi 4 focused on making Delphi development easier. The Module Explorer was introduced in Delphi, and it enabled you to browse and edit units from a convenient graphical interface. New code navigation and class completion features enabled you to focus on the meat of your applications with a minimum of busy work. The IDE was redesigned with dockable toolbars and windows to make your development more convenient, and the debugger was greatly improved. Delphi 4 extended the product's reach into the enterprise with outstanding multitier support using technologies such as MIDAS, DCOM, MTS, and CORBA.

This was 1998, and Delphi had effectively secured its position relative to the competition. The front lines had stabilized somewhat, although Delphi continued to slowly gain market share. CORBA was the industry buzz, and Delphi had it and the competition did not. There was a bit of a down-side to Delphi 4 as well: After enjoying several years of being the most stable development tool on the market, Delphi 4 had earned a reputation among long-time Delphi users for not living up to the very high standard for solid engineering and stability.

The release of Delphi 4 followed the acquisition of Visigenic, one of the CORBA industry leaders. Borland changed its name to *Inprise* in an effort to better penetrate the enterprise, and the company was in a position to lead the industry to new ground by integrating its tools with the CORBA technology. To really win, CORBA needed to be made as easy as COM or Internet development had been made in past versions of Borland tools. However, for various reasons, the integration wasn't as full as it should have been, and the CORBA-development tool integration was destined to play a bit part in the overall software-development picture.

Delphi 5

Delphi 5 moved ahead on a few fronts: First, Delphi 5 continued what Delphi 4 started by adding many more features to make easy those tasks that traditionally take time, hopefully enabling you to concentrate more on what you want to write and less on how to write it. These new productivity features include further IDE and debugger enhancements, TeamSource team development software, and translation tools. Second, Delphi 5 contained a host of new features aimed squarely at making Internet development easier. These new Internet features include the

Active Server Object Wizard for ASP creation, the InternetExpress components for XML support, and new MIDAS features, making it a very versatile data platform for the Internet. Finally, Borland built time into the schedule to deliver the most important feature of all for Delphi 5: stability. Like fine wine, you cannot rush great software, and Borland waited until Delphi 5 was ready before letting it out the door.

Delphi 5 was released in the latter half of 1999. Delphi continues to penetrate the enterprise, whereas Visual Basic continues to serve as competition on the low end. However, the battle lines still appear stable. Inprise brought back the Borland name but only as a brand. The executive offices went through some turbulent times, with the company divisionalized between tools and middleware, the abrupt departure of CEO Del Yocam, and the hiring of Internet-savvy CEO Dale Fuller, who refocused the company back on software developers.

Delphi 6

Clearly the primary theme of Delphi 6 is compatibility with Borland's Kylix development tool for Linux. To this end, Borland developed the new Component Library for Cross-Platform (CLX), which includes VisualCLX for visual development, DataCLX client data-access components, and NetCLX Internet components. Applications written using only the CLX library and portable RTL elements will easily port between the Windows and Linux operating systems.

The new dbExpress set of components and drivers is one of the biggest breakthroughs to come out of the effort for Linux compatibility because it finally provides a real alternative for the BDE, which has really begun to show its age in recent years.

A secondary theme of Delphi 6 is essentially to embrace all things XML. This includes XML for database applications, Web-based applications, and SOAP-based Web services. Delphi developers have the tools they need to fully embrace the industry-wide trend toward XML, which provides great benefits in terms of applications that function across the traditional boundaries of different development tools, platforms, databases, and across the Internet.

Of course, in addition to all these improvements and additions, Delphi 6 brings the normal host of improvement you've come to expect between product versions in core areas like VCL, the IDE, the debugger, the Object Pascal language, and the RTL.

The Delphi IDE

Just to make sure that we're all on the same page with regard to terminology, Figure 1.2 shows the Delphi IDE and calls attention to its major constituents: the main window, the Component Palette, the toolbars, the Form Designer, the Code Editor, the Object Inspector, Object TreeView, and the Code Explorer.

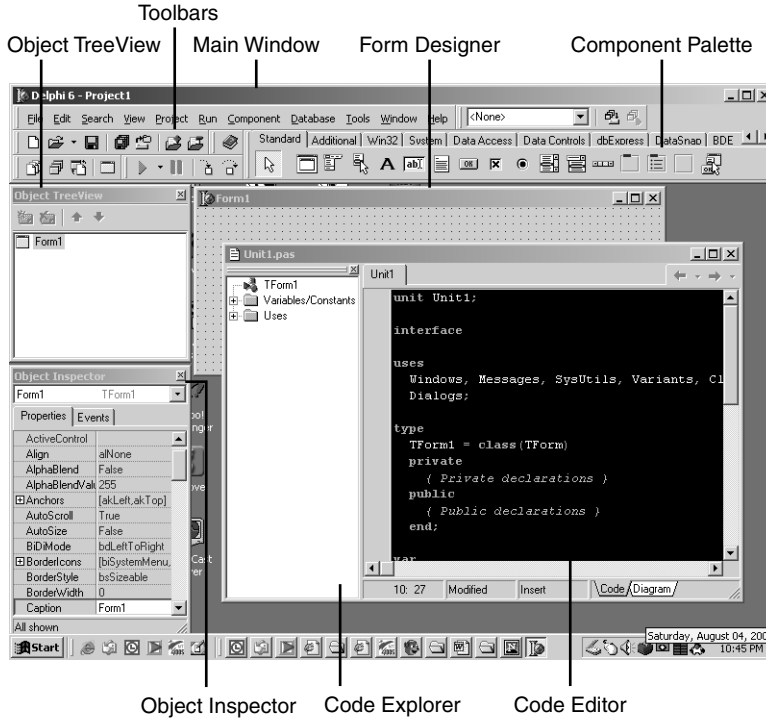

FIGURE 1.2
The Delphi 6 IDE.

The Main Window

Think of the *main window* as the control center for the Delphi IDE. The main window has all the standard functionality of the main window of any other Windows program. It consists of three parts: the main menu, the toolbars, and the Component Palette.

The Main Menu

As in any Windows program, you go to the main menu when you need to open and save files, invoke wizards, view other windows, modify options, and so on. Most items on the main menu can also be invoked via a button on a toolbar.

The Delphi Toolbars

The toolbars enable single-click access to some operation found on the main menu of the IDE, such as opening a file or building a project. Notice that each of the buttons on the toolbars offer a *tooltip* that contain a description of the function of a particular button. Not including the Component Palette, there are five separate toolbars in the IDE: Debug, Desktops, Standard,

View, and Custom. Figure 1.2 shows the default button configuration for these toolbars, but you can add or remove buttons by selecting Customize from the local menu on a toolbar. Figure 1.3 shows the Customize toolbar dialog box. You add buttons by dragging them from this dialog box and drop them on any toolbar. To remove a button, drag it off the toolbar.

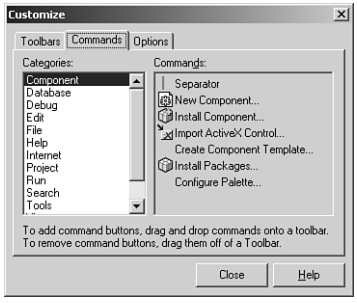

FIGURE 1.3
The Customize toolbar dialog box.

IDE toolbar customization doesn't stop at configuring which buttons are shown. You can also relocate each of the toolbars, the Component Palette, or the menu within the main window. To do this, click the raised gray bars on the left side of the toolbars and drag them around the main window. If you drag the mouse outside the confines of the main window while doing this, you'll see yet another level of customization: The toolbars can be undocked from the main window and reside in their own floating tool windows. Undocked views of the toolbars are shown in Figure 1.4.

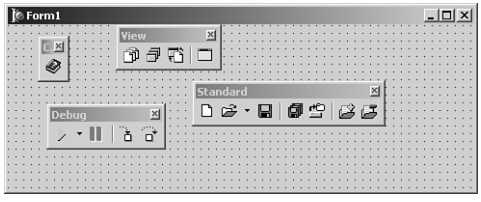

FIGURE 1.4
Undocked toolbars.

The Component Palette

The Component Palette is a double-height toolbar that contains a page control filled with all the VCL components and ActiveX controls installed in the IDE. The order and appearance of pages and components on the Component Palette can be configured via a right-click or by selecting Component, Configure Palette from the main menu.

The Form Designer

The Form Designer begins as an empty window, ready for you to turn it into a Windows application. Consider the Form Designer your artist's canvas for creating Windows applications; here is where you determine how your applications will be represented visually to your users. You interact with the Form Designer by selecting components from the Component Palette and dropping them onto your form. After you have a particular component on the form, you can use the mouse to adjust the position or size of the component. You can control the appearance and behavior of these components by using the Object Inspector and Code Editor.

The Object Inspector

With the Object Inspector, you can modify a form's or component's properties or enable your form or component to respond to different events. *Properties* are data such as height, color, and font that determine how an object appears onscreen. *Events* are portions of code executed in response to occurrences within your application. A mouse-click message and a message for a window to redraw itself are two examples of events. The Object Inspector window uses the standard Windows *notebook tab* metaphor in switching between component properties or events; just select the desired page from the tabs at the top of the window. The properties and events displayed in the Object Inspector reflect whichever form or component currently has focus in the Form Designer.

Delphi also has the capability to arrange the contents of the Object Inspector by category or alphabetically by name. You can do this by right-clicking anywhere in the Object Inspector and selecting Arrange from the local menu. Figure 1.5 shows two Object Inspectors side by side. The one on the left is arranged by category, and the one on the right is arranged by name. You can also specify which categories you would like to view by selecting View from the local menu.

One of the most useful tidbits of knowledge that you as a Delphi programmer should know is that the help system is tightly integrated with the Object Inspector. If you ever get stuck on a particular property or event, just press the F1 key, and WinHelp comes to the rescue.

The Code Editor

The Code Editor is where you type the code that dictates how your program behaves and where Delphi inserts the code that it generates based on the components in your application. The top of the Code Editor window contains notebook tabs, where each tab corresponds to a different source code module or file. Each time you add a new form to your application, a new unit is created and added to the set of tabs at the top of the Code Editor. The local menu in the Code Editor gives you a wide range of options while you're editing, such as closing files, setting bookmarks, and navigating to symbols.

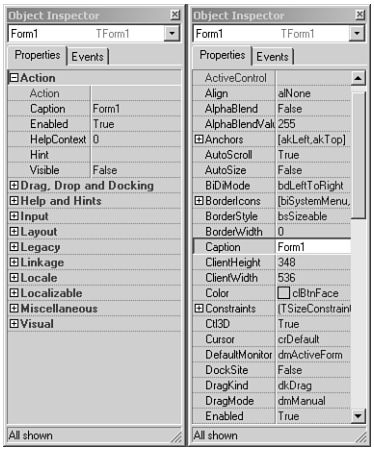

FIGURE 1.5
Viewing the Object Inspector by category and by name.

TIP

You can view multiple Code Editor windows simultaneous by selecting View, New Edit Window from the main menu.

The Code Explorer

The Code Explorer provides a tree-style view of the unit shown in the Code Editor. The Code Explorer allows easy navigation of units in addition to the ability to easily add new elements or rename existing elements in a unit. It's important to remember that there's a one-to-one relationship between Code Explorer windows and Code Editor windows. Right-click a node in the Code Explorer to view the options available for that node. You can also control behaviors such as sorting and filtering in the Code Explorer by modifying the options found on the Explorer tab of the Environment Options dialog box.

The Object TreeView

The Object TreeView provides a visual, hierarchical representation of the components placed on a form, data module, or frame. The tree displays the relationship between individual components, such as parent-child, property-to-component, or property-to-property relationships. In addition to being a means to view relationships, the Object TreeView also serves as a convenient means to establish relationships between components. This can be done most easily by

dropping one component from the palette or the tree on another in the tree. This will establish the relationship between two components that have a possibility of forming a relationship.

A Tour of Your Project's Source

The Delphi IDE generates Object Pascal source code for you as you work with the visual components of the Form Designer. The simplest example of this capability is starting a new project. Select File, New Application in the main window to see a new form in the Form Designer and that form's source code skeleton in the Code Editor. The source code for the new form's unit is shown in Listing 1.1.

LISTING 1.1 Source Code for an Empty Form

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation;

{$R *.dfm}

end.
```

It's important to note that the source code module associated with any form is stored in a unit. Although every form has a unit, not every unit has a form. If you're not familiar with how the Pascal language works and what exactly a *unit* is, see Chapter 2, "The Object Pascal Language," which discusses the Object Pascal language for those who are new to Pascal from C++, Visual Basic, Java, or another language.

Let's take a unit skeleton one piece at a time. Here's the top portion:

```
type
  TForm1 = class(TForm) ;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

It indicates that the form object, itself, is an object derived from `TForm`, and the space in which you can insert your own public and private variables is labeled clearly. Don't worry about what *class*, *public*, or *private* means right now. Chapter 2 discusses Object Pascal in more detail.

The following line is very important:

```
{$R *.dfm};
```

The `$R` directive in Pascal is used to load an external resource file. This line links the `.DFM` (which stands for *Delphi form*) file into the executable. The `.DFM` file contains a binary representation of the form you created in the Form Designer. The `*` symbol in this case isn't intended to represent a wildcard; it represents the file having the same name as the current unit. So, for example, if the preceding line was in a file called `Unit1.pas`, the `*.DFM` would represent a file by the name of `Unit1.dfm`.

NOTE

A nice feature of the IDE is the ability for you to save new DFM files as text rather than as binary. This option in enabled by default, but you can modify it using the New Forms As Text check box on the Preferences page of the Environment Options dialog box. Although saving forms as text format is just slightly less efficient in terms of size, it's a good practice for a few of reasons: First, it is very easy to make minor changes to text DFMs in any text editor. Second, if the file should become corrupted, it is far easier to repair a corrupted text file than a corrupted binary file. Finally, it becomes much easier for version control systems to manage the form files. Keep in mind also that previous versions of Delphi expect binary DFM files, so you will need to disable this option if you want to create projects that will be used by other versions of Delphi.

The application's project file; is worth a glance, too. A project filename ends in `.DPR` (which stands for *Delphi project*) and is really nothing more than a Pascal source file with a different file extension. The project file is where the main portion of your program (in the Pascal sense) lives. Unlike other versions of Pascal with which you might be familiar, most of the "work" of

your program is done in units rather than in the main module. You can load your project's source file into the Code Editor by selecting Project, View Source from the main menu. Here's the project file from the sample application:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

As you add more forms and units to the application, they appear in the `uses` clause of the project file. Notice, too, that after the name of a unit in the `uses` clause, the name of the related form appears in comments. If you ever get confused about which units go with which forms, you can regain your bearings by selecting View, Project Manager to bring up the Project Manager window.

NOTE

Each form has exactly one unit associated with it, and you can also have other "code-only" units that aren't associated with any form. In Delphi, you work mostly within your program's; units, and you'll rarely edit your project's `.DPR` file.

Tour of a Small Application

The simple act of plopping a component such as a button onto a form causes code for that element to be generated and added to the form object:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Now, as you can see, the button is an instance variable of the `TForm1` class. When you refer to the button in contexts outside `TForm1` later in your source code, you must remember to address it as part of the scope of `TForm1` by saying `Form1.Button1`. Scoping is explained in more detail in Chapter 2.

When this button is selected in the Form Designer, you can change its behavior through the Object Inspector. Suppose that, at design time, you want to change the width of the button to 100 pixels, and at runtime, you want to make the button respond to a press by doubling its own height. To change the button width, move over to the Object Browser window, find the `Width` property, and change the value associated with `Width` to `100`. Note that the change doesn't take effect in the Form Designer until you press Enter or move off the `Width` property. To make the button respond to a mouse click, select the Events page on the Object Inspector window to reveal the list of events to which the button can respond. Double-click in the column next to the `OnClick` event, and Delphi generates a procedure skeleton for a mouse-click response and whisks you away to that spot in the source code—in this case, a procedure called `TForm1.Button1Click()`. All that's left to do is to insert the code to double the button's width between the `begin..end` of the event's response method:

```
Button1.Height := Button1.Height * 2;
```

To verify that the "application" compiles and runs, press the F9 key on your keyboard and watch it go!

NOTE

Delphi maintains a reference between generated procedures and the controls to which they correspond. When you compile or save a source code module, Delphi scans your source code and removes all procedure skeletons for which you haven't entered any code between the `begin` and `end`. This means that if you didn't write any code between the `begin` and `end` of the `TForm1.Button1Click()` procedure, for example, Delphi would have removed the procedure from your source code. The bottom line here is this: Don't delete event handler procedures that Delphi has created; just delete your code and let Delphi remove the procedures for you.

After you have fun making the button really big on the form, terminate your program and go back to the Delphi IDE. Now is a good time to mention that you could have generated a response to a mouse click for your button just by double-clicking a control after dropping it onto the form. Double-clicking a component automatically invokes its associated component editor. For most components, this response generates a handler for the first of that component's events listed in the Object Inspector.

What's So Great About Events, Anyway?

If you've ever developed Windows applications the traditional way, without a doubt you'll find the ease of use of Delphi events a welcome alternative to manually catching Windows messages, cracking those messages, and testing for window handles, control IDs, `WParam` parameters, `LParam` parameters, and so on. If you don't know what all that means, that's okay; Chapter 3, "Adventures in Messaging," covers messaging internals.

A Delphi event is often triggered by a Windows message. The `OnMouseDown` event of a `TButton`, for example, is really just an encapsulation of the Windows `WM_xBUTTONDOWN` messages. Notice that the `OnMouseDown` event gives you information such as which button was pressed and the location of the mouse when it happened. A form's `OnKeyDown` event provides similar useful information for key presses. For example, here's the code that Delphi generates for an `OnKeyDown` handler:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin
end;
```

All the information you need about the key is right at your fingertips. If you're an experienced Windows programmer, you'll appreciate that there aren't any `LParam` or `WParam` parameters, inherited handlers, translates, or dispatches to worry about. This goes way beyond "message cracking" as you might know it because one Delphi event can represent several different Windows messages, as it does with `OnMouseDown` (which handles a variety of mouse messages). What's more, each of the message parameters is passed in as easy-to-understand parameters. Chapter 3 gets into the gory details of how Delphi's internal messaging system works.

Contract-Free Programming

Arguably the biggest benefit that Delphi's event system has over the standard Windows messaging system is that all events are contract free. What *contract free* means to the programmer is that you never are *required* to do anything inside your event handlers. Unlike standard Windows message handling, you don't have to call an inherited handler or pass information back to Windows after handling an event.

Of course, the downside to the contract-free programming model that Delphi's event system provides is that it doesn't always give you the power or flexibility that directly handling Windows messages gives you. You're at the mercy of those who designed the event as far as what level of control you'll have over your application's response to the event. For example, you can modify and kill keystrokes in an `OnKeyPress` handler, but an `OnResize` handler provides you only with a notification that the event occurred—you have no power to prevent or modify the resizing.

Never fear, though. Delphi doesn't prevent you from working directly with Windows messages. It's not as straightforward as the event system because message handling assumes that the programmer has a greater level of knowledge of what Windows expects of every handled message. You have complete power to handle all Windows messages directly by using the `message` keyword. You'll find out much more about writing Windows message handlers in Chapter 3.

The great thing about developing applications with Delphi is that you can use the high-level easy stuff (such as events) when it suits you and still have access to the low-level stuff whenever you need it.

Turbo Prototyping

After hacking Delphi for a little while, you'll probably notice that the learning curve is especially mild. In fact, even if you're new to Delphi, you'll find that writing your first project in Delphi pays immediate dividends in the form of a short development cycle and a robust application. Delphi excels in the one facet of application development that has been the bane of many a Windows programmer: user interface (UI) design.

Sometimes the design of the UI and the general layout of a program is referred to as *prototyping*. In a nonvisual environment, prototyping an application often takes longer than writing the application's implementation, or what is called the *back end*. Of course, the back end of an application is the whole objective of the program in the first place, right? Sure, an intuitive and visually pleasing UI is a big part of the application, but what good would it be, for example, to have a communications program with pretty windows and dialog boxes but no capacity to send data through a modem? As it is with people, so it is with applications; a pretty face is nice to look at, but it has to have substance to be a regular part of our lives. Please, no comments about back ends.

Delphi enables you to use its custom controls to whip out nice-looking UIs in no time flat. In fact, you'll find that after you become comfortable with Delphi's forms, controls, and event-response methods, you'll cut huge chunks off the time you usually take to develop application prototypes. You'll also find that the UIs you develop in Delphi look just as nice as—if not better than—those designed with traditional tools. Often, what you "mock up" in Delphi turns out to be the final product.

Extensible Components and Environment

Because of the object-oriented nature of Delphi, in addition to creating your own components from scratch, you can also create your own customized components based on stock Delphi components. For more details on this and other types of components, you should take a look at Part IV, "Component-Based Development."

In addition to allowing you to integrate custom components into the IDE, Delphi provides the capability to integrate entire subprograms, called *experts*, into the environment. Delphi's Expert Interface enables you to add special menu items and dialog boxes to the IDE to integrate some feature that you feel is worthwhile. An example of an expert is the Database Form Expert located on the Delphi Database menu. Chapter 17, "Using The Open Tools API," outlines the process for creating experts and integrating them into the Delphi IDE.

The Top 10 IDE Features You Must Know and Love

Before we can let you any further into the book, we've got to make sure that you're equipped with the tools you need to survive and the knowledge to use them. In that spirit, what follows is a list of what we feel are the top 10 IDE features you must learn to know and love.

1. Class Completion

Nothing wastes a developer's time more than have to type in all that blasted code! How often is it that you know exactly what you want to write but are limited by how fast your fingers can fly over the keys? Until the spec for the PCI-to-medulla oblongata bus is completed to rid you of all that typing, Delphi has a feature called *class completion* that goes a long way toward alleviating the busy work.

Arguably, the most important feature of class completion is that it is designed to work without being in your face. Simply type in part of a class declaration, press the magic Ctrl+Shift+C keystroke combination, and class completion will attempt to figure our what you're trying to do and generate the right code. For example, if you put the declaration for a procedure called `Foo` in your class and invoke class completion, it will automatically create the definition for this method in the implementation part of the unit. Declare a new property that reads from a field and writes to a method and invoke class completion, and it will automatically generate the code for the field and declare and implement the method.

If you haven't already gotten hooked on class completion, give it a whirl. Soon you'll be lost without it.

2. AppBrowser Navigation

Do you ever look at a line of code in your Code Editor and think, "Gee, I wish I knew where that method is declared"? Well, finding out is as easy as holding down the Ctrl key and clicking the name of the token you want to find. The IDE will use debug information assembled in the background by the compiler to jump to the declaration of the token. Very handy. And like a

Web browser, there's a history stack that you can navigate forward and back through using the little arrows to the right of the tabs in the Code Editor.

3. Interface/Implementation Navigation

Want to navigate between the interface and implementation of a method? Just put the cursor on the method and use Ctrl+Shift+up arrow or down arrow to toggle between the two positions.

4. Dock It!

The IDE allows you to organize the windows on your screen by docking together multiple windows as panes in a single window. If you have full window drag set in your windows desktop, you can easily tell which windows are dockable because they draw a dithered box when they're dragged around the screen. The Code Editor offers three docking bays on its left, bottom, and right sides to which you can affix windows. Windows can be docked side-by-side by dragging one window to an edge of another or tab-docked by dragging one window to the middle of another. Once you come up with an arrangement you like, be sure to save it using the Desktops toolbar. Want to prevent a window from docking? Hold down the Ctrl key while dragging it or right-click in the window and uncheck Dockable in the local menu.

TIP

Here's a cute hidden feature: Right-click the tabs of tab-docked windows, and you'll be able to move the tabs to the top, bottom, left, or right of the window.

5. The Object Browser

Delphi 1 through 4 shipped with essentially the same icky object browser. If you didn't know it was there, don't feel alone; many folks never used it because it didn't have a lot to offer. Delphi now comes equipped with an object browser that enables visual browsing of object hierarchies. Shown in Figure 1.6, the browser is accessible by selecting View, Browser in the main menu. This tool presents a tree view that lets you navigate globals, classes, and units and drill down into scope, inheritance, and references of the symbols.

6. GUID, Anyone?

In the small-but-useful category, you'll find the Ctrl+Shift+G keystroke combination. Pressing this keystroke combination will place a fresh new GUID in the Code Editor, which is a real timesaver when you're declaring new interfaces.

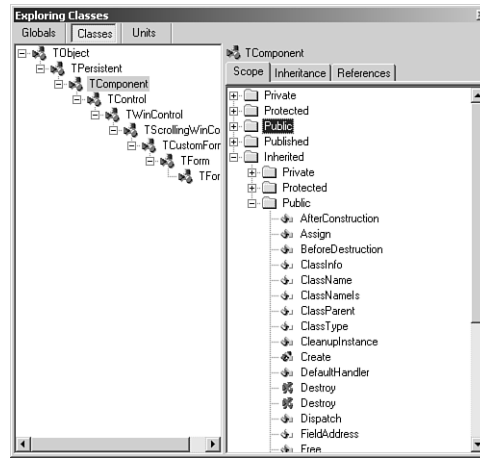

FIGURE 1.6
The new browser.

7. C++ Syntax Highlighting

If you're like us, you often like to view C++ files, such as SDK headers, while you work in Delphi. Because Delphi and C++Builder share the same editor source code, one of the advantages to users is syntax highlighting of C++ files. Just load up a C++ file such as a `.CPP` or `.H` module in the Code Editor, and it handles the rest automatically.

8. To Do. . .

Use the To Do List to manage work in progress in your source files. You can view the To Do List by selecting View, To Do List from the main menu. This list is automatically populated from any comments in your source code that begin with the token *`TODO`*. You can use the To Do Items window to set the owner, priority, and category for any To Do item. This window is shown in Figure 1.7, docked to the bottom of the Code Editor.

9. Use the Project Manager

The Project Manager can be a big timesaver when navigating around large projects—especially those projects that are composed of multiple EXE or DLL modules, but it's amazing how many people forget that it's there. You can access the Project Manager by selecting View, Project Manager from the main menu. There are a number of time saving features in the Project Manager, such as drag-and-drop copying and copy and paste between projects.

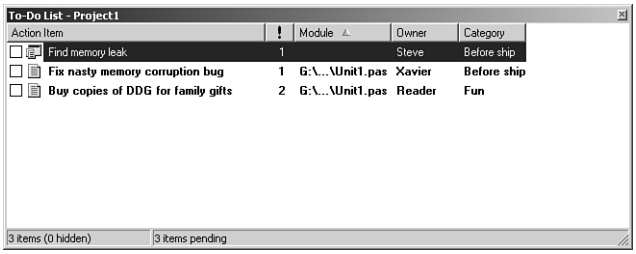

FIGURE 1.7
To Do Items window.

10. Use Code Insight to Complete Declarations and Parameters

When you type **`Identifier.`**, a window will automatically pop up after the dot to provide you with a list of properties, methods, events, and fields available for that identifier. You can right-click this window to sort the list by name or by scope. If the window goes away before you're ready, just press Ctrl+space to bring it back up.

Remembering all the parameters to a function can be a pain, so it's nice that Code Insight automatically helps by providing a tooltip with the parameter list when you type **`FunctionName(`** in the Code Editor. Remember to press Ctrl+Shift+space to bring the tooltip back up if it goes away before you're ready.

Summary

By now you should have an understanding of the Delphi 6 product line and the Delphi IDE as well as how Delphi fits into the Windows development picture in general. This chapter was intended to acclimate you to Delphi and to the concepts used throughout the book. Now the stage has been set for the really technical stuff to come. Before you move much deeper into the book, make sure that you're comfortable using and navigating around the IDE and know how to work with small projects.

This page intentionally left blank

The Object Pascal Language

CHAPTER

2

IN THIS CHAPTER

- Comments 36
- Extended Procedure and Function Features 37
- Variables 39
- Constants 41
- Operators 43
- Object Pascal Types 47
- User-Defined Types 75
- Typecasting and Type Conversion 87
- String Resources 88
- Testing Conditions 88
- Loops 90
- Procedures and Functions 93
- Scope 97
- Units 99
- Packages 101
- Object-Oriented Programming 103
- Using Delphi Objects 105
- Structured Exception Handling 119
- Runtime Type Information 126

This chapter sets aside the visual elements of Delphi in order to provide you with an overview of Delphi's underlying language—Object Pascal. To begin with, you'll receive an introduction to the basics of the Object Pascal language, such as language rules and constructs. Later on, you'll learn about some of the more advanced aspects of Object Pascal, such as classes and exception handling. Because this isn't a beginner's book, it assumes that you have some experience with other high-level computer languages such as Java, C/C++, or Visual Basic, and it compares Object Pascal language structure to that of those other languages. By the time you're finished with this chapter, you'll understand how programming concepts such as variables, types, operators, loops, cases, exceptions, and objects work in Pascal as compared to Java, C/C++, and Visual Basic.

NOTE

When we mention the C language in this chapter, we are generally referring to a language element that exists in both C and C++. Features specific to the C++ language are referred to as C++.

Even if you have some recent experience with Pascal, you'll find this chapter useful because this is really the only point in the book where you learn the nitty-gritty of Pascal syntax and semantics.

Comments

As a starting point, you should know how to make comments in your Pascal code. Object Pascal supports three types of comments: curly brace comments, parenthesis/asterisk comments, and double backslash comments. Examples of each type of comment follow:

```
{ Comment using curly braces }
(* Comment using paren and asterisk *)
// double backslash comment
```

The first two types of comments are virtually identical in behavior. The compiler considers the comment to be everything between the open-comment and close-comment delimiters. For double backslash comments, everything following the double backslash until the end of the line is considered a comment.

NOTE

You cannot nest comments of the same type. Although it is legal syntax to nest Pascal comments of different types inside one another, we don't recommend the practice. Here are some examples:

continues

```
{ (* This is legal *) }
(* { This is legal } *)
(* (* This is illegal *) *)
{ { This is illegal }: }
```

Extended Procedure and Function Features

Because procedures and functions are fairly universal topics as far as programming languages are concerned, we won't go into too much detail here. We just want to fill you in on a few unique or little-known features in this area. Where appropriate, we'll also point out the Delphi version in which various language features appeared to aid in porting or maintaining code compatible between various compiler versions.

Parentheses in Calls

Although it has been in the language since Delphi 2, one of the lesser-known features of Object Pascal is that parentheses are optional when calling a procedure or function that takes no parameters. Therefore, the following syntax examples are both valid:

```
Form1.Show;
Form1.Show();
```

Granted, this feature isn't one of those things that sends chills up and down your spine, but it's particularly nice for those who split their time between Delphi and languages such as C or Java, where parentheses are required. If you're not able to spend 100% of your time in Delphi, this feature means that you don't have to remember to use different function-calling syntax for different languages.

Overloading

Delphi 4 introduced the concept of function *overloading* (that is, the ability to have multiple procedures or functions of the same name with different parameter lists). All overloaded methods are required to be declared with the `overload` directive, as shown here:

```
procedure Hello(I: Integer); overload;
procedure Hello(S: string); overload;
procedure Hello(D: Double); overload;
```

Note that the rules for overloading methods of a class are slightly different and are explained in the section "Method Overloading." Although this is one of the features most requested by developers since Delphi 1, the phrase that comes to mind is "Be careful what you wish for." Having multiple functions and procedures with the same name (on top of the traditional ability

to have functions and procedures of the same name in different units) can make it more difficult to predict the flow of control and debug your application. Because of this, overloading is a feature you should employ judiciously. Not to say that you should avoid it; just don't overuse it.

Default Value Parameters

Also introduced in Delphi 4 were default value parameters (that is, the ability to provide a default value for a function or procedure parameter and not have to pass that parameter when calling the routine). In order to declare a procedure or function that contains default value parameters, follow the parameter type with an equal sign and the default value, as shown in the following example:

```
procedure HasDefVal(S: string; I: Integer = 0);
```

The `HasDefVal()` procedure can be called in one of two ways. First, you can specify both parameters:

```
HasDefVal('hello', 26);
```

Second, you can specify only parameter `S` and use the default value for `I`:

```
HasDefVal('hello');  // default value used for I
```

You must follow several rules when using default value parameters:

- Parameters having default values must appear at the end of the parameter list. Parameters without default values cannot follow parameters with default values in a procedure or function's parameter list.
- Default value parameters must be of an ordinal, pointer, or set type.
- Default value parameters must be passed by value or as `const`. They cannot be reference (`out`) or untyped parameters.

One of the biggest benefits of default value parameters is in adding functionality to existing functions and procedures without sacrificing backward compatibility. For example, suppose that you sell a unit containing a revolutionary function called `AddInts()`that adds two numbers:

```
function AddInts(I1, I2: Integer): Integer;
begin
  Result := I1 + I2;
end;
```

In order to keep up with the competition, you feel you must update this function so that it has the capability for adding three numbers. However, you're loathe to do so because adding a parameter will cause existing code that calls this function to not compile. Thanks to default parameters, you can enhance the functionality of `AddInts()` without compromising compatibility. Here's an example:

```
function AddInts(I1, I2: Integer; I3: Integer = 0);
begin
  Result := I1 + I2 + I3;
end;
```

Variables

You might be used to declaring variables off the cuff: "I need another integer, so I'll just declare one right here in the middle of this block of code." This is a perfectly reasonable notion if you're coming from another language such as Java, C, or Visual Basic. If that has been your practice, you're going to have to retrain yourself a little in order to use variables in Object Pascal. Object Pascal requires you to declare all variables up front in their own section before you begin a procedure, function, or program. Perhaps you used to write free-wheeling code like this:


```
void foo(void)
{
  int x = 1;
  x++;
  int y = 2;
  float f;
  //... etc ...
}
```

In Object Pascal, any such code must be tidied up and structured a bit more to look like this:

```
Procedure Foo;
var
  x, y: Integer;
  f: Double;
begin
  x := 1;
  inc(x);
  y := 2;
  //... etc ...
end;
```

NOTE

Object Pascal—like Visual Basic, but unlike Java and C—is not a case-sensitive language. Upper- and lowercase is used for clarity's sake, so use your best judgment, as the style used in this book indicates. If the identifier name is several words mashed

continues

together, remember to capitalize for clarity. For example, the following name is unclear and difficult to read:

```
procedure thisprocedurenamemakesnosense;
```

This code is quite readable, however:

```
procedure ThisProcedureNameIsMoreClear;
```

For a complete reference on the coding style guidelines used for this book, see the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this book.

You might be wondering what all this structure business is and why it's beneficial. You'll find, however, that Object Pascal's structured style of variable declaration lends itself to code that's more readable, maintainable, and less buggy than other languages that rely on convention rather than rule to enforce sanity.

Notice how Object Pascal enables you to group more than one variable of the same type together on the same line with the following syntax:

```
VarName1, VarName2: SomeType;
```

Remember that when you're declaring a variable in Object Pascal, the variable name precedes the type, and there's a colon between the variables and types. Note that the variable initialization is always separate from the variable declaration.

A language feature introduced in Delphi 2 enables you to initialize global variables inside a `var` block. Here are some examples demonstrating the syntax for doing so:

```
var
  i: Integer = 10;
  S: string  = 'Hello world';
  D: Double  = 3.141579;
```

NOTE

Preinitialization of variables is only allowed for global variables, not variables that are local to a procedure or function.

TIP

The Delphi compiler sees to it that all global data is automatically zero-initialized. When your application starts, all integer types will hold `0`, floating-point types will hold `0.0`, pointers will be `nil`, strings will be empty, and so forth. Therefore, it isn't necessary to zero-initialize global data in your source code.

Constants

Constants in Pascal are defined in a `const` clause, which behaves similarly to the C/C++'s `const` keyword. Here's an example of three constant declarations in C:

```
const float ADecimalNumber = 3.14;
const int i = 10;
const char * ErrorString = "Danger, Danger, Danger!";
```

The major difference between C constants and Object Pascal constants is that Object Pascal, like Visual Basic, doesn't require you to declare the constant's type along with the value in the declaration. The Delphi compiler automatically allocates proper space for the constant based on its value, or, in the case of scalar constants such as `Integer`, the compiler keeps track of the values as it works, and space never is allocated. Here's an example:

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString = 'Danger, Danger, Danger!';
```

NOTE

Space is allocated for constants as follows: Integer values are "fit" into the smallest type allowable (10 into a `ShortInt`, 32,000 into a `SmallInt`, and so on). Alphanumeric values fit into `Char` or the currently defined (by `$H`) `string` type. Floating-point values are mapped to the `extended` data type, unless the value contains four or fewer decimal places explicitly, in which case it's mapped to a `Comp` type. Sets of `Integer` and `Char` are of course stored as themselves.

Optionally, you can also specify a constant's type in the declaration. This provides you with full control over how the compiler treats your constants:

```
const
  ADecimalNumber: Double = 3.14;
  I: Integer = 10;
  ErrorString: string = 'Danger, Danger, Danger!';
```

Object Pascal permits the usage of compile-time functions in `const` and `var` declarations. These routines include `Ord()`, `Chr()`, `Trunc()`, `Round()`, `High()`, `Low()`, and `SizeOf()`. For example, all of the following code is, valid:

```
type
  A = array[1..2] of Integer;

const
  w: Word = SizeOf(Byte);

var
  i: Integer = 8;
  j: SmallInt = Ord('a');
  L: Longint = Trunc(3.14159);
  x: ShortInt = Round(2.71828);
  B1: Byte = High(A);
  B2: Byte = Low(A);
  C: char = Chr(46);
```

CAUTION

The behavior of 32-bit Delphi type-specified constants is different from that in 16-bit Delphi 1. In Delphi 1, the identifier declared wasn't treated as a constant but as a preinitialized variable called a *typed constant*. However, in Delphi 2 and later, type-specified constants have the capability of being truly constant. Delphi provides a backward-compatibility switch on the Compiler page of the Project, Options dialog box, or you can use the `$J` compiler directive. By default, this switch is enabled for compatibility with Delphi 1 code, but you're best served not to rely on this capability because the implementers of the Object Pascal language are trying to move away from the notion of assignable constants.

If you try to change the value of any of these constants, the Delphi compiler emits an error explaining that it's against the rules to change the value of a constant. Because constants are read-only, Object Pascal optimizes your data space by storing those constants that merit storage in the application's code pages. If you're unclear about the notions of code and data pages,

see Chapter 3, "The Win32 API," in the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this, book.

NOTE

Object Pascal doesn't have a preprocessor as does C. There's no concept of a macro in Object Pascal and, therefore, no Object Pascal equivalent for C's `#define` for constant declaration. Although you can use Object Pascal's `$define` compiler directive for conditional compiles similar to C's `#define`, you cannot use it to define constants. Use `const` in Object Pascal where you would use `#define` to declare a constant in C.

Operators

Operators are the symbols in your code that enable you to manipulate all types of data. For example, there are operators for adding, subtracting, multiplying, and dividing numeric data. There are also operators for addressing a particular element of an array. This section explains some of the Pascal operators and describes some of the differences between their Java, C, and Visual Basic counterparts.

Assignment Operators

If you're new to Pascal, Delphi's assignment operator is going to be one of the toughest things to get used to. To assign a value to a variable, use the `:=` operator as you would use the `=` operator in Java, C, or Visual Basic. Pascal programmers often call this the *gets* or *assignment* operator, and, the expression

```
Number1 := 5;
```

is read either "`Number1` *gets* the value `5`" or "`Number1` *is assigned* the value `5`."

Comparison Operators

If you've already programmed in Visual Basic, you should be very comfortable with Delphi's comparison operators, because they're virtually identical. These operators are fairly standard throughout programming languages, so they're covered only briefly in this section.

Object Pascal uses the `=` operator to perform logical comparisons between two expressions or values. Object Pascal's `=` operator is analogous to the Java/C `==` operator, so a Java/C expression that would be written as

```
if (x == y)
```

would be written as this in Object Pascal:

```
if x = y
```

NOTE

Remember that in Object Pascal, the `:=` operator is used to assign a value to a variable, and the `=` operator compares the values of two, operands.

Object Pascal's "not equal to" operator is `<>`, and its purpose is identical to C's `!=` operator. To determine whether two expressions are not equal, use this code:

```
if x <> y then DoSomething
```

Logical Operators

Pascal uses the words `and` and `or` as logical "and" and "or" operators, whereas Java and C use the `&&` and `||` symbols, respectively, for these operators. The most common use of the `and` and `or` operators is as part of an `if` statement or loop, as demonstrated in the following two examples:

```
if (Condition 1) and (Condition 2) then
  DoSomething;

while (Condition 1) or (Condition 2) do
  DoSomething;
```

Pascal's logical "not" operator is `not`, which is used to invert a Boolean expression. It's analogous to the Java/C's `!` operator. It's also often used as a part of `if` statements, as shown here:

```
if not (condition) then (do something);   // if condition is false then...
```

Table 2.1 provides an easy reference of how Pascal operators map to corresponding Java, C, and Visual Basic operators.

TABLE 2.1 Assignment, Comparison, and Logical Operators

Operator	*Pascal*	*Java/C*	*Visual Basic*
Assignment	`:=`	`=`	`=`
Comparison	`=`	`==`	`=` or `Is`*
Not equal to	`<>`	`!=`	`<>`
Less than	`<`	`<`	`<`
Greater than	`>`	`>`	`>`

TABLE 2.1 Continued

Operator	*Pascal*	*Java/C*	*Visual Basic*
Less than or equal to	`<=`	`<=`	`<=`
Greater than or equal to	`>=`	`>=`	`>=`
Logical and	`and`	`&&`	`And`
Logical or	`or`	`||`	`Or`
Logical not	`not`	`!`	`Not`

**The* `Is` *comparison operator is used for objects, whereas the* `=` *comparison operator is used for other types.*

Arithmetic Operators

You should already be familiar with most Object Pascal arithmetic operators because they're generally similar to those used in Java, C, and Visual Basic. Table 2.2 illustrates all the Pascal arithmetic operators and their Java, C, and Visual Basic counterparts.

TABLE 2.2 Arithmetic Operators

Operator	*Pascal*	*Java/C*	*Visual Basic*
Addition	`+`	`+`	`+`
Subtraction	`-`	`-`	`-`
Multiplication	`*`	`*`	`*`
Floating-point division	`/`	`/`	`/`
Integer division	`div`	`/`	`\`
Modulus	`mod`	`%`	`Mod`
Exponent	*None*	*None*	`^`

You might notice that Pascal and Visual Basic provide different division operators for floating-point and integer math, although this isn't the case for Java and C. The `div` operator automatically truncates any remainder when you're dividing two integer expressions.

NOTE

Remember to use the correct division operator for the types of expressions with which you're working. The Object Pascal compiler gives you an error if you try to divide two floating-point numbers with the integer `div` operator or two integers with the floating-point `/` operator, as the following code illustrates:

continues

```
var
  i: Integer;
  r: Real;
begin
  i := 4 / 3;         // This line will cause a compiler error
  f := 3.4 div 2.3;   // This line also will cause an error
end;
```

Many other programming languages do not distinguish between integer and floating-point division. Instead, they always perform floating-point division and then convert the result back to an integer when necessary. This can be rather expensive in terms of performance. The Pascal `div` operator is faster and more specific.

Bitwise Operators

Bitwise operators enable you to modify individual bits of a given variable. Common bitwise operators enable you to shift the bits to the left or right or to perform bitwise "and," "not," "or," and "exclusive or" (`xor`) operations with two numbers. The Shift+left and Shift+right operators are `shl` and `shr`, respectively, and they're much like the Java/C << and >> operators. The remainder of Pascal's bitwise operators is easy enough to remember: `and`, `not`, `or`, and `xor`. Table 2.3 lists the bitwise operators.

TABLE 2.3 Bitwise Operators

Operator	*Pascal*	*Java/C*	*Visual Basic*
And	`and`	`&`	`And`
Not	`not`	`~`	`Not`
Or	`or`	`|`	`Or`
Xor	`xor`	`^`	`Xor`
Shift+left	`shl`	`<<`	*None*
Shift+right	`shr`	`>>`	*None*

Increment and Decrement Procedures

Increment and decrement procedures generate optimized code for adding or subtracting 1 from a given integral variable. Pascal doesn't really provide honest-to-gosh increment and decrement operators similar to the Java/C `++` and `--` operators, but Pascal's `Inc()` and `Dec()` procedures compile optimally to one machine instruction.

You can call `Inc()` or `Dec()` with one or two parameters. For example, the following two lines of code increment and decrement `variable`, respectively, by 1, using the `inc` and `dec` assembly instructions:

```
Inc(variable);

Dec(variable);
```

Compare the following two lines, which increment or decrement `variable` by 3 using the `add` and `sub` assembly instructions:

```
Inc(variable, 3);

Dec(variable, 3);
```

Table 2.4 compares the increment and decrement operators of different languages.

NOTE

With compiler optimization enabled, the `Inc()` and `Dec()` procedures often produce the same machine code as *`variable`*`:= variable + 1` syntax, so use whichever you feel more comfortable with for incrementing and decrementing variables.

TABLE 2.4 Increment and Decrement Operators

Operator	*Pascal*	*Java/C*	*Visual Basic*
Increment	`Inc()`	`++`	*None*
Decrement	`Dec()`	`--`	*None*

Do-and-Assign Operators

Not present in Object Pascal are handy do-and-assign operators like those found in Java and C. These operators, such as `+=` and `*=`, perform an arithmetic operation (in this case, an add and an multiply) before making the assignment. In Object Pascal, this type of operation must be performed using two separate operators. Therefore, this code in Java or C

```
x += 5;
```

becomes this in Object Pascal:

```
x := x + 5;
```

Object Pascal Types

One of Object Pascal's greatest features is that it's strongly typed, or *typesafe*. This means that actual variables passed to procedures and functions must be of the same type as the formal parameters identified in the procedure or function definition. You won't see any of the famous compiler warnings about suspicious pointer conversions that C programmers have grown to know and love. This is because the Object Pascal compiler won't permit you to call a function with one type of pointer when another type is specified in the function's formal parameters (although functions that take untyped `Pointer` types accept any type of pointer). Basically, Pascal's strongly typed nature enables it to perform a sanity check of your code—to ensure that you' aren't trying to put a square peg in a round hole.

A Comparison of Types

Delphi's base types are similar to those of Java, C, and Visual Basic. Table 2.5 compares and contrasts the base types of Object Pascal with those of these other languages. You might want to earmark this page because this table provides an excellent reference for matching types when calling functions in non-Delphi dynamic link libraries (DLLs) or object files (OBJs) from Delphi (and vice versa).

TABLE 2.5 A Pascal-to-Java-to-C-to-Visual Basic 32-bit Type Comparison

Type of Variable	*Pascal*	*Java*	*C/C++*	*Visual Basic*
8-bit signed integer	`ShortInt`	`byte`	`char`	None
8-bit unsigned integer	`Byte`	None	`BYTE`, `unsigned short`	`Byte`
16-bit signed integer	`SmallInt`	`short`	`short`	`Short`
16-bit unsigned integer	`Word`	None	`unsigned short`	None
32-bit signed integer	`Integer`, `Longint`	`int`	`int`, `long`	`Integer, Long`
32-bit unsigned integer	`Cardinal`, `LongWord`	None	`unsigned long`	None
64-bit signed integer	`Int64`	`long`	`__int64`	None

TABLE 2.5 Continued

Type of Variable	*Pascal*	*Java*	*C/C++*	*Visual Basic*
4-byte floating point	`Single`	`float`	`float`	`Single`
6-byte floating point	`Real48`	None	None	None
8-byte floating point	`Double`	`double`	`double`	`Double`
10-byte floating point	`Extended`	None	`long. double`	None
64-bit currency	`currency`	None	None	`Currency`
8-byte date/time	`TDateTime`	None	None	`Date`
16-byte variant	`Variant`, `OleVariant`, `TVarData`	None	`VARIANT`** `Variant`†, `OleVariant`†	`Variant`(Default)
1-byte character	`Char`	None	`char`	None
2-byte character	`WideChar`	`char`	`WCHAR`	
Fixed-length byte string	`ShortString`	None	None	None
Dynamic string	`AnsiString`		`AnsiString`†	`String`
Null-terminated string	`PChar`	None	`char *`	None
Null-terminated wide string	`PWideChar`	None	`LPCWSTR`	None
Dynamic 2-byte string	`WideString`	`String`**	`WideString`†	None
1-byte Boolean	`Boolean`, `ByteBool`	`boolean`	(Any 1-byte)	None
2-byte Boolean	`WordBool`	None	(Any 2-byte)	`Boolean`
4-byte Boolean	`BOOL`, `LongBool`	None	`BOOL`	None

†A proprietary Borland C++Builder class that emulates the corresponding Object Pascal type
***Not a language element proper, but a commonly used structure or class*

NOTE

If you're porting 16-bit code from Delphi 1, be sure to bear in mind that the size of both the `Integer` and `Cardinal` types has increased from 16 to 32 bits. Actually, that's not quite accurate: Under Delphi 2 and 3, the `Cardinal` type was treated as an unsigned 31-bit integer in order to preserve arithmetic precision (because Delphi 2 and 3 lacked a true unsigned 32-bit integer to which results of integer operations could be promoted). Under Delphi 4 and higher, `Cardinal` is a true unsigned 32-bit integer.

CAUTION

In Delphi 1, 2, and 3, the `Real` type identifier specified a 6-byte floating-point number, which is a type unique to Pascal and generally incompatible with other languages. In Delphi 4, `Real` is an alias for the `Double` type. The old 6-byte floating-point number is still there, but it's now identified by `Real48`. You can also force the `Real` identifier to refer to the 6-byte floating-point number using the `{$REALCOMPATIBILITY ON}` directive.

Characters

Delphi provides three character types:

- `AnsiChar`—This is the standard one-byte ANSI character that programmers have grown to know and love.
- `WideChar`—This character is two bytes in size and represents a Unicode character.
- `Char`—This is currently identical to `AnsiChar`, but Borland warns that the definition might change to `WideChar` in a later version of Delphi.

Keep in mind that because a character is no longer guaranteed to be one byte in size, you shouldn't hard-code the size into your applications. Instead, you should use the `SizeOf()` function where appropriate.

NOTE

The `SizeOf()` standard procedure returns the size, in bytes, of a type or instance.

A Multitude of Strings

Strings are variable types used to represent groups of characters. Every language has its own spin on how string types are stored and used. Pascal has several different string types to suit your programming needs:

- `AnsiString`, the default string type for Object Pascal, is comprised of `AnsiChar` characters and allows for virtually unlimited lengths. It's also compatible with null-terminated strings.
- `ShortString` remains in the language primarily for backward compatibility with Delphi 1. Its capacity is limited to 255 characters.
- `WideString` is similar in functionality to `AnsiString` except that it's comprised of `WideChar` characters.
- `PChar` is a pointer to a null-terminated `Char` string—like C's `char *` and `lpstr` types.
- `PAnsiChar` is a pointer to a null-terminated `AnsiChar` string.
- `PWideChar` is a pointer to a null-terminated `WideChar` string.

By default, when you declare a `string` variable in your code, as shown in the following example, the compiler assumes that you're creating an `AnsiString`:

```
var
  S: string;   // S is an AnsiString
```

Alternatively, you can cause variables declared as `string` types to be of type `ShortString` instead using the `$H` compiler directive. When the value of the `$H` compiler directive is negative, `string` variables are `ShortString` types; and when the value of the directive is positive (the default), `string` variables are `AnsiString` types. The following code demonstrates this behavior:

```
var
  {$H-}
  S1: string;  // S1 is a ShortString
  {$H+}
  S2: string;  // S2 is an AnsiString
```

The exception to the `$H` rule is that a `string` declared with an explicit size (limited to a maximum of 255 characters) is always a `ShortString`:

```
var
  S: string[63];    // A ShortString of up to 63 characters
```

The `AnsiString` Type

The `AnsiString` (or *long string*) type was introduced to the language in Delphi 2. It exists primarily as a result of widespread Delphi 1 customer demand for an easy-to-use string type without the intrusive 255-character limitation. `AnsiString` is that and more.

Although `AnsiString` types maintain an almost identical interface as their predecessors, they're dynamically allocated and garbage-collected. Because of this, `AnsiString` is sometimes referred to as a *lifetime-managed* type. Object Pascal also automatically manages allocation of string temporaries as needed, so you needn't worry about allocating buffers for intermediate results as you would in C/C++. Additionally, `AnsiString` types are always guaranteed to be null terminated, which makes them compatible with the null-terminated strings used by the Win32 API. The `AnsiString` type is actually implemented as a pointer to a string structure in heap memory. Figure 2.1 shows how an `AnsiString` is laid out in memory.

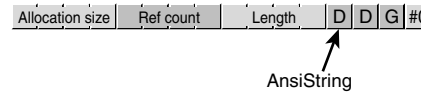

FIGURE 2.1
An `AnsiString` *in memory.*

CAUTION

The complete internal format of the long string type is left undocumented by Borland, and Borland reserves the right to change the internal format of long strings with future releases of Delphi. The information here is intended mainly to help you understand how `AnsiString` types work, and you should avoid being dependent on the structure of an `AnsiString` in your code.

Developers who avoided the implementation of details of string moving from Delphi 1 to Delphi 2 were able to migrate their code with no problems. Those who wrote code that depended on the internal format (such as the 0th element in the string being the length) had to modify their code for Delphi 2.

As Figure 2.1 illustrates, `AnsiString` types are *reference counted*, which means that several strings might point to the same physical memory. String copies, therefore, are very fast because it's merely a matter of copying a pointer rather than copying the actual string contents. When two or more `AnsiString` types share a reference to the same physical string, the Delphi memory manager uses a copy-on-write technique, which enables it to wait until a string is modified to release a reference and allocate a new physical string. The following example illustrates these concepts:

```
var
  S1, S2: string;
begin
  // store string in S1, ref count of S1 is 1
  S1 := 'And now for something... ';
  S2 := S1;          // S2 now references S1.  Ref count of S1 is 2.
  // S2 is changed, so it is copied to its own
  // memory space, and ref count of S1 is decremented


  S2 := S2 + 'completely different!';
```


Lifetime-Managed Types

In addition to `AnsiString`, Delphi provides several other types that are lifetime-managed. These types include `WideString`, `Variant`, `OleVariant`, `interface`, `dispinterface`, and dynamic arrays. You'll learn more about each of these types later in this chapter. For now, we'll focus on what exactly lifetime-managed types are and how they work.

Lifetime-managed types, sometimes called *garbage-collected types*, are types that potentially consume some particular resource while in use and release the resource automatically when they fall out of scope. Of course, the variety of resources used depends on the type involved. For example, an `AnsiString` consumes memory for the character string while in use, and the memory occupied by the character string is released when it leaves scope.

For global variables, this process is fairly straightforward: As a part of the finalization code generated for your application, the compiler inserts code to ensure that each lifetime-managed global variable is cleaned up. Because all global data is zero-initialized when your application loads, each lifetime-managed global variable will always initially contain a zero, empty, or some other value indicating the variable is "unused." This way, the finalization code won't attempt to free resources unless they're actually used in your application.

Whenever you declare a local lifetime-managed variable, the process is slightly more complex: First, the compiler inserts code to ensure that the variable is initialized to zero when the function or procedure is entered. Next, the compiler generates a `try..finally` exception-handling block, which it wraps around the entire function body. Finally, the compiler inserts code in the `finally` block to clean up the lifetime-managed variable (exception handling is explained in more detail in the section "Structured Exception Handling"). With this in mind, consider the following-procedure:

```
procedure Foo;
var
  S: string;
begin
  // procedure body
  // use S here
end;
```

Although this procedure looks simple, if you take into account the code generation by the compiler behind the scenes, it would actually look like this:

```
procedure Foo;
var
  S: string;
begin
  S := '';
  try
    // procedure body
    // use S here
  finally
    // clean up S here
   end;
end;
```

String Operations

You can concatenate two `strings` by using the `+` operator or the `Concat()` function. The preferred method of string concatenation is the `+` operator because the `Concat()` function exists primarily for backward compatibility. The following example demonstrates the use of `+` and `Concat()`:

```
{ using + }
var
  S, S2: string
begin
  S:= 'Cookie ':
  S2 := 'Monster';
  S := S + S2;   { Cookie Monster }
end.
```

```
{ using Concat() }
var
  S, S2: string;
begin
  S:= 'Cookie ';
  S2 := 'Monster';
  S := Concat(S, S2);  { Cookie Monster }
end.
```

NOTE

Always use single quotation marks (`'A String'`) when working with string literals in Object Pascal.

TIP

`Concat()` is one of many "compiler magic" functions and procedures (like `ReadLn()` and `WriteLn()`, for example) that don't have an Object Pascal definition. Such functions and procedures are intended to accept an indeterminate number of parameters or optional parameters, so they cannot be defined in terms of the Object Pascal language. Because of this, the compiler provides a special case for each of these functions and generates a call to one of the "compiler magic" *helper functions* defined in the `System` unit. These helper functions are generally implemented in assembly language in order to circumvent Pascal language rules.

In addition to the "compiler magic" string support functions and procedures, there are a variety of functions and procedures in the `SysUtils` unit designed to make working with strings easier. Search for "String-handling routines (Pascal-style)" in the Delphi online help system.

Furthermore, you'll find some very useful homebrewed string utility functions and procedures in the `StrUtils` unit in the `\Source\Utils` directory on the CD-ROM accompanying this book.

Length and Allocation

When first declared, an `AnsiString` has no length and therefore no space allocated for the characters in the string. To cause space to be allocated for the string, you can assign the string to a literal or another string, or you can use the `SetLength()` procedure, as shown here:

```
var
  S: string;         // string initially has no length
begin
  S := 'Doh!';       // allocates at least enough space for string literal
  { or }
  S := OtherString   // increases ref count of OtherString
                     // (assume OtherString already points to a valid string)
  { or }
  SetLength(S, 4);   // allocates enough space for at least 4 chars
end;
```

You can index the characters of an AnsiString like an array, but be careful not to index beyond the length of the string. For example, the following code snippet will cause an error:

```
var
  S: string;
begin
  S[1] := 'a';  // Won't work because S hasn't been allocated!
end;
```

This code, however, works properly:

```
var
  S: string;
begin
  SetLength(S, 1);
  S[1] := 'a';      // Now S has enough space to hold the character
end;
```

Win32 Compatibility

As mentioned earlier, AnsiString types are always null-terminated, so they're compatible with null-terminated strings. This makes it easy to call Win32 API functions or other functions requiring PChar-type strings. All that's required is that you typecast the string as a PChar. (Typecasting is explained in more detail in the section "Typecasting and Type Conversion.") The following code demonstrates how to call the Win32 GetWindowsDirectory() function, which accepts a PChar and buffer length as parameters:

```
var
  S: string;
begin
  SetLength(S, 256);           // important! get space for string first
  // call function, S now holds directory string
  GetWindowsDirectory(PChar(S), 256);
end;
```

After using an AnsiString in which a function or procedure expects a PChar, you must manually set the length of the string variable to its null-terminated length. The RealizeLength() function, which also comes from the StrUtils unit, accomplishes that task:

```
procedure RealizeLength(var S: string);
begin
  SetLength(S, StrLen(PChar(S)));
end;
```

Calling RealizeLength() completes the substitution of a long string for a PChar:

```
var
  S: string;
```

```
begin
  SetLength(S, 256);          // important! get space for string first
  // call function, S now holds directory string
  GetWindowsDirectory(PChar(S), 256);
  RealizeLength(S);           // set S length to null length
end;
```

CAUTION

Exercise care when typecasting a `string` to a `PChar` variable. Because strings are garbage-collected when they go out of scope, you must pay attention when making assignments such as `P := PChar(Str)`, where the scope (or lifetime) of `P` is greater than `Str`.

Porting Issues

When you're porting 16-bit Delphi 1 applications, you need to keep in mind a number of issues when migrating to `AnsiString` types:

- In places where you used the `PString` (pointer to a `ShortString`) type, you should instead use the `string` type. Remember, an `AnsiString` is already a pointer to a string.
- You can no longer access the 0th element of a string to get or set the length. Instead, use the `Length()` function to get the string length and the `SetLength()` procedure to set the length.
- There's no longer any need to use `StrPas()` and `StrPCopy()` to convert back and forth between strings and `PChar` types. As shown earlier, you can typecast an `AnsiString` to a `PChar`. When you want to copy the contents of a `PChar` to an `AnsiString`, you can use a direct assignment:

  ```
  StringVar := PCharVar;
  ```

CAUTION

Remember that you must use the `SetLength()` procedure to set the length of a long string, whereas the past practice was to directly access the 0th element of a short string to set the length. This issue will arise when you attempt to port 16-bit Delphi 1.0 code to 32, bits.

The ShortString Type

If you're a Delphi veteran, you'll recognize the ShortString type as the Delphi 1.0 string type. ShortString types are sometimes referred to as *Pascal strings* or *length-byte strings*. To reiterate, remember that the value of the $H directive determines whether variables declared as string are treated by the compiler as AnsiString or ShortString.

In memory, the string resembles an array of characters in which the 0th character in the string contains the length of the string, and the string itself is contained in the following characters. The storage size of a ShortString defaults to the maximum of 256 bytes. This means that you can never have more than 255 characters in a ShortString (255 characters + 1 length byte = 256). As with AnsiString, working with ShortString is fairly painless because the compiler allocates string temporaries as needed, so you don't have to worry about allocating buffers for intermediate results or disposing of them as you do with C.

Figure 2.2 illustrates how a Pascal string is laid out in memory.

FIGURE 2.2
A ShortString *in memory.*

A ShortString variable is declared and initialized with the following syntax:

```
var
  S: ShortString;
begin
  S := 'Bob the cat.';
end.
```

Optionally, you can allocate fewer than 256 bytes for a ShortString using just the string type identifier and a length specifier, as in the following example:

```
var
  S: string[45];  { a 45-character ShortString }
begin
  S := 'This string must be 45 or fewer characters.';
end.
```

The preceding code causes a ShortString to be created regardless of the current setting of the $H directive. The maximum length you can specify is 255 characters.

Never store more characters to a ShortString than you have allocated memory for. If you declare a variable as a string[8], for example, and try to assign 'a_pretty_darn_long_string' to that variable, the string would be truncated to only eight characters, and you would lose data.

When using an array subscript to address a particular character in a `ShortString`, you could get bogus results or corrupt memory if you attempt to use a subscript index that's greater than the declared size of the `ShortString`. For example, suppose that you declare a variable as follows:

```
var
  Str: string[8];
```

If you then attempt to write to the 10th element of the string as follows, you're likely to corrupt memory used by other variables:

```
var
  Str: string[8];
  i: Integer;
begin
  i := 10;
  Str[i] := 's';  // will corrupt memory
```

You can have the compiler link in special logic to catch these types of errors at runtime by selecting Range Checking in the Options, Project dialog box.

TIP

Although including range-checking logic in your program helps you find string errors, range checking slightly hampers the performance of your application. It's common practice to use range checking during the development and debugging phases of your program, but you should remove range checking after you become confident in the stability of your program.

Unlike `AnsiString` types, `ShortString` types aren't inherently compatible with null-terminated strings. Because of this, a bit of work is required to be able to pass a `ShortString` to a Win32 API function. The following function, `ShortStringAsPChar()`, is taken from the `STRUTILS.PAS` unit mentioned earlier:

```
func function ShortStringAsPChar(var S: ShortString): PChar;
{ Function null-terminates a string so it can be passed to functions }
{ that require PChar types. If string is longer than 254 chars, then it will }
{ be truncated to 254. }
begin
  if Length(S) = High(S) then Dec(S[0]); { Truncate S if it's too long }
  S[Ord(Length(S)) + 1] := #0;           { Place null at end of string }
  Result := @S[1];                       { Return "PChar'd" string }
end;
```

CAUTION

The functions and procedures in the Win32 API require null-terminated strings. Do not try to pass a `ShortString` type to an API function because your program will not compile. Your life will be easier if you use long strings when working with the API.

The `WideString` Type

The `WideString` type is a lifetime-managed type similar to `AnsiString`; they're both dynamically allocated, garbage collected, and even assignment compatible with one another. However, `WideString` differs from `AnsiString` in three key respects:

- `WideString` types are comprised of `WideChar` characters rather than `AnsiChar` characters, making them compatible with Unicode strings.
- `WideString` types are allocated using the `SysAllocStrLen()` API function, making them compatible with OLE `BSTR` strings.
- `WideString` types aren't reference counted, so assigning one `WideString` to another requires the entire string to be copied from one location in memory to another. This makes `WideString` types less efficient than `AnsiString` types in terms of speed and memory use.

As mentioned earlier, the compiler automatically knows how to convert between variables of `AnsiString` and `WideString` types, as shown here:

```
var
  W: WideString;
  S: string;
begin
  W := 'Margaritaville';
  S := W;  // Wide converted to Ansi
  S := 'Come Monday';
  W := S;  // Ansi converted to Wide
end;
```

In order to make working with `WideString` types feel natural, Object Pascal overloads the `Concat()`, `Copy()`, `Insert()`, `Length()`, `Pos()`, and `SetLength()` routines and the `+`, `=`, and `<>` operators for use with `WideString` types. Therefore, the following code is syntactically correct:

```
var
  W1, W2: WideString;
  P: Integer;
begin
  W1 := 'Enfield';
```

```
  W2 := 'field';
  if W1 <> W2 then
    P := Pos(W1, W2);
end;
```

As with the `AnsiString` and `ShortString` types, you can use array brackets to reference individual characters of a `WideString`:

```
var
  W: WideString;
  C: WideChar;
begin
  W := 'Ebony and Ivory living in perfect harmony';
  C := W[Length(W)];  // C holds the last character in W
end;
```


Null-Terminated Strings

Earlier, this chapter mentioned that Delphi has three different null-terminated string types: `PChar`, `PAnsiChar`, and `PWideChar`. As their names imply, each of these represents a null-terminated string of each of Delphi's three character types. In this chapter, we refer to each of these string types generically as `PChar`. The `PChar` type in Delphi exists mainly for compatibility with Delphi 1.0 and the Win32 API, which makes extensive use of null-terminated strings. A `PChar` is defined as a pointer to a string followed by a null (zero) value (if you're unsure of exactly what a pointer is, read on; pointers are discussed in more detail later in this section). Unlike memory for `AnsiString` and `WideString` types, memory for `PChar` types isn't automatically allocated and managed by Object Pascal. Therefore, you'll usually need to allocate memory for the string to which it points, using one of Object Pascal's memory-allocation functions. The theoretical maximum length of a `PChar` string is just under 4GB. The layout of a `PChar` variable in memory is shown in Figure 2.3.

TIP

Object Pascal's `AnsiString` type can be used as a `PChar` in most situations, so you should use this type rather than the `PChar` type wherever possible. Because memory management for strings occurs automatically, you greatly reduce the chance of introducing memory-corruption bugs into your applications if, where possible, you avoid `PChar` types and the manual memory allocation associated with them.

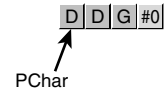

FIGURE 2.3
A `PChar` *in memory.*

As mentioned earlier, `PChar` variables require you to manually allocate and free the memory buffers that contain their strings. Normally, you allocate memory for a `PChar` buffer using the `StrAlloc()` function, but several other functions can be used to allocate memory for `PChar` types, including `AllocMem()`, `GetMem()`, `StrNew()`, and even the `VirtualAlloc()` API function. Corresponding functions also exist for many of these functions, which must be used to deallocate memory. Table 2.6 lists several allocation functions and their corresponding deallocation functions.

TABLE 2.6 Memory Allocation and Deallocation Functions

Memory Allocated with. . .	*Must Be Freed with. . .*
`AllocMem()`	`FreeMem()`
`GlobalAlloc()`	`GlobalFree()`
`GetMem()`	`FreeMem()`
`New()`	`Dispose()`
`StrAlloc()`	`StrDispose()`
`StrNew()`	`StrDispose()`
`VirtualAlloc()`	`VirtualFree()`

The following example demonstrates memory allocation techniques when working with `PChar` and `string` types:

```
var
  P1, P2: PChar;
  S1, S2: string;
begin
  P1 := StrAlloc(64 * SizeOf(Char));  // P1 points to an allocation of 63 Chars
  StrPCopy(P1, 'Delphi 6 ');          // Copy literal string into P1
  S1 := 'Developer''s Guide';         // Put some text in string S1
  P2 := StrNew(PChar(S1));            // P1 points to a copy of S1
  StrCat(P1, P2);                     // concatenate P1 and P2
  S2 := P1;                      // S2 now holds 'Delphi 6 Developer's Guide'
  StrDispose(P1);                     // clean up P1 and P2 buffers
  StrDispose(P2);
end.
```

Notice, first of all, the use of `SizeOf(Char)` with `StrAlloc()` when allocating memory for `P1`. Remember that the size of a `Char` might change from one byte to two in future versions of Delphi; therefore, you cannot assume the value of `Char` to always be one byte. `SizeOf()` ensures that the allocation will work properly no matter how many bytes a character occupies.

`StrCat()` is used to concatenate two `PChar` strings. Note here that you cannot use the `+` operator for concatenation as you can with long string and `ShortString` types.

The `StrNew()` function is used to copy the value contained by string `S1` into `P2` (a `PChar`). Be careful when using this function. It's common to have memory-overwrite errors when using `StrNew()` because it allocates only enough memory to hold the string. Consider the following example:


```
var
P1, P2: Pchar;
begin
  P1 := StrNew('Hello ');  // Allocate just enough memory for P1 and P2
  P2 := StrNew('World');
  StrCat(P1, P2);          // BEWARE: Corrupts memory!
  .
  .
  .
end;
```

TIP

As with other types of strings, Object Pascal provides a decent library of utility functions and procedures for operating on `PChar` types. Search for "String-handling routines (null-terminated)" in the Delphi online help system.

You'll also find some useful null-terminated functions and procedures in the `StrUtils` unit in the `\Source\Utils` directory on the CD-ROM accompanying this book.

Variant Types

Delphi 2 introduced a powerful data type called the `Variant`. Variants were brought about primarily in order to support OLE Automation, which uses the `Variant` type heavily. In fact, Delphi's `Variant` data type is an encapsulation of the variant used with OLE. Delphi's implementation of variants has also proven to be useful in other areas of Delphi programming, as you'll soon learn. Object Pascal is the only compiled language that completely integrates variants as a dynamic data type at runtime and as a static type at compile time in that the compiler always knows that it's a variant.

Delphi 3 introduced a new type called `OleVariant`, which is identical to `Variant` except that it can only hold Automation-compatible types. In this section, we initially focus on the `Variant` type and then we discuss `OleVariant` and contrast it with `Variant`.

Variants Change Types Dynamically

One of the main purposes of variants is to have a variable whose underlying data type cannot be determined at compile time. This means that a variant can change the type to which it refers at runtime. For example, the following code will compile and run properly:

```
var
  V: Variant;
begin
  V := 'Delphi is Great!';   // Variant holds a string
  V := 1;                      // Variant now holds an Integer
  V := 123.34;                 // Variant now holds a floating point
  V := True;                   // Variant now holds a boolean
  V := CreateOleObject('Word.Basic'); // Variant now holds an OLE object
end;
```

Variants can support all simple data types, such as integers, floating-point values, strings, Booleans, date and time, currency, and also OLE Automation objects. Note that variants cannot refer to Object Pascal objects. Also, variants can refer to a non-homogeneous array, which can vary in size and whose data elements can refer to any of the preceding data types (including another variant array).

The Variant Structure

The data structure defining the `Variant` type is defined in the `System` unit and is also shown in the following code:

```
  TVarType = Word;
  PVarData = ^TVarData;
  {$EXTERNALSYM PVarData}
  TVarData = packed record
    VType: TVarType;
    case Integer of
      0: (Reserved1: Word;
          case Integer of
            0: (Reserved2, Reserved3: Word;
                case Integer of
                  varSmallInt: (VSmallInt: SmallInt);
                  varInteger:  (VInteger: Integer);
                  varSingle:   (VSingle: Single);
                  varDouble:   (VDouble: Double);
                  varCurrency: (VCurrency: Currency);
                  varDate:     (VDate: TDateTime);
```

```
                varOleStr:   (VOleStr: PWideChar);
                varDispatch: (VDispatch: Pointer);
                varError:    (VError: LongWord);
                varBoolean:  (VBoolean: WordBool);
                varUnknown:  (VUnknown: Pointer);
                varShortInt: (VShortInt: ShortInt);
                varByte:     (VByte: Byte);
                varWord:     (VWord: Word);
                varLongWord: (VLongWord: LongWord);
                varInt64:    (VInt64: Int64);
                varString:   (VString: Pointer);
                varAny:      (VAny: Pointer);
                varArray:    (VArray: PVarArray);
                varByRef:  (VPointer: Pointer);
              );
            1: (VLongs: array[0..2] of LongInt);
          );
      2: (VWords: array [0..6] of Word);
      3: (VBytes: array [0..13] of Byte);
  end;
```

The `TVarData` structure consumes 16 bytes of memory. The first two bytes of the `TVarData` structure contain a word value that represents the data type to which the variant refers. The following code shows the various values that might appear in the `VType` field of the `TVarData` record. The next six bytes are unused. The remaining eight bytes contain the actual data or a pointer to the data represented by the variant. Again, this structure maps directly to 'COM's implementation of the variant type. Here's the code:

```
{ Variant type codes (wtypes.h) }

  varEmpty    = $0000; { vt_empty       }
  varNull     = $0001; { vt_null        }
  varSmallint = $0002; { vt_i2          }
  varInteger  = $0003; { vt_i4          }
  varSingle   = $0004; { vt_r4          }
  varDouble   = $0005; { vt_r8          }
  varCurrency = $0006; { vt_cy          }
  varDate     = $0007; { vt_date        }
  varOleStr   = $0008; { vt_bstr        }
  varDispatch = $0009; { vt_dispatch    }
  varError    = $000A; { vt_error       }
  varBoolean  = $000B; { vt_bool        }
  varVariant  = $000C; { vt_variant     }
  varUnknown  = $000D; { vt_unknown     }
//varDecimal  = $000E; { vt_decimal     } {UNSUPPORTED}
                       { undefined  $0f } {UNSUPPORTED}
```

```
  varShortInt = $0010; { vt_i1          }
  varByte     = $0011; { vt_ui1         }
  varWord     = $0012; { vt_ui2         }
  varLongWord = $0013; { vt_ui4         }
  varInt64    = $0014; { vt_i8          }
//varWord64   = $0015; { vt_ui8         } {UNSUPPORTED}

  { if adding new items, update Variants' varLast, BaseTypeMap and OpTypeMap }
  varStrArg   = $0048; { vt_clsid    }
  varString   = $0100; { Pascal string; not OLE compatible }
  varAny      = $0101; { Corba any }
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;
```

NOTE

As you might notice from the type codes in the preceding listing, a `Variant` cannot contain a reference to a `Pointer` or `class` type.

You'll notice from the `TVarData` listing that the `TVarData` record is actually a *variant record*. Don't confuse this with the `Variant` type. Although the variant record and `Variant` type have similar names, they represent two totally different constructs. Variant records allow for multiple data fields to overlap in the same area of memory (like a C/C++ union). This is discussed in more detail in the "Records" section later in this chapter. The `case` statement in the `TVarData` variant record indicates the type of data to which the variant refers. For example, if the `VType` field contains the value `varInteger`, only four bytes of the eight data bytes in the variant portion of the record are used to hold an integer value. Likewise, if `VType` has the value `varByte`, only one byte of the eight is used to hold a byte value.

You'll notice that if `VType` contains the value `varString`, the eight data bytes don't actually hold the string; instead, they hold a pointer to this string. This is an important point because you can access fields of a variant directly, as shown here:

```
var
  V: Variant;
begin
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 2;
end;
```

You must understand that in some cases this is a dangerous practice because it's possible to lose the reference to a string or other lifetime-managed entity, which will result in your

application leaking memory or other resources. You'll see what we mean by the term *garbage collected* in the following section.

Variants Are Lifetime Managed

Delphi automatically handles the allocation and deallocation of memory required of a `Variant` type. For example, examine the following code, which assigns a string to a `Variant` variable:

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := S;
  ShowMessage(V);
end;
```


As discussed earlier in this chapter in the sidebar "Lifetime-Managed Types," several things are going on here that might not be apparent. Delphi first initializes the variant to an unassigned value. During the assignment, it sets its `VType` field to `varString` and copies the string pointer into its `VString` field. It then increases the reference count of string `S`. When the variant leaves scope (that is, the procedure ends and returns to the code that called it), it's cleared and the reference count of string `S` is decremented. Delphi does this by implicitly inserting a `try..finally` block in the procedure, as shown. here:

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := Unassigned;  // initialize variant to "empty"
  try
    V := S;
    ShowMessage(V);
  finally
    // Now clean up the resources associated with the variant
  end;
end;
```

This same implicit release of resources occurs when you assign a different data type to the variant. For example, examine the following code:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  V := 34;
end;
```

This code boils down to the following pseudo-code:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  Clear Variant V, ensuring it is initialized to "empty"
  try
    V.VType := varString; V.VString := S; Inc(S.RefCount);
    Clear Variant V, thereby releasing reference to string;
    V.VType := varInteger; V.VInteger := 34;
  finally
    Clean up the resources associated with the variant
  end;
end;
```

If you understand what happens in the preceding examples, you'll see why it's not recommended that you manipulate fields of the `TVarData` record directly, as shown here:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 32;
  V := 34;
end;
```

Although this might appear to be safe, it's not because it results in the failure to decrement the reference count of string `S`, probably resulting in a memory leak. As a general rule, don't access the `TVarData` fields directly, or if you do, be absolutely sure that you know exactly what you're doing.

Typecasting Variants

You can explicitly typecast expressions to type `Variant`. For example, the expression

```
Variant(X)
```

results in a `Variant` type whose type code corresponds to the result of the expression `X`, which must be an integer, real, currency, string, character, or Boolean type.

You can also typecast a variant to that of a simple data type. For example, given the assignment

```
V := 1.6;
```

where V is a variable of type Variant, the following expressions will have the results shown:

```
S := string(V);    // S will contain the string '1.6';
// I is rounded to the nearest Integer value, in this case: 2.
I := Integer(V);
B := Boolean(V);   // B contains False if V contains 0, otherwise B is True
D := Double(V);    // D contains the value 1.6
```

These results are dictated by certain type-conversion rules applicable to Variant types. These rules are defined in detail in Delphi's Object Pascal Language Guide.

By the way, in the preceding example, it's not necessary to typecast the variant to another data type to make the assignment. The following code would work just as well:

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

What happens here is that the conversions to the target data types are made through an implicit typecast. However, because these conversions are made at runtime, there's much more code logic attached to this method. If you're sure of the type a variant contains, you're better off explicitly typecasting it to that type in order to speed up the operation. This is especially true if the variant is being used in an expression, which we'll discuss. next.

Variants in Expressions

You can use variants in expressions with the following operators: +, =, *, /, div, mod, shl, shr, and, or, xor, not, :=, <>, <, >, <=, and >=.

When using variants in expressions, Delphi knows how to perform the operations based on the contents of the variant. For example, if two variants, V1 and V2, contain integers, the expression V1 + V2 results in the addition of the two integers. However, if V1 and V2 contain strings, the result is a concatenation of the two strings. What happens if V1 and V2 contain two different data types? Delphi uses certain promotion rules in order to perform the operation. For example, if V1 contains the string '4.5' and V2 contains a floating-point number, V1 will be converted to a floating point and then added to V2. The following code illustrates this:

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100';  // A string type
  V2 := '50';   // A string type
  V3 := 200;    // An Integer type
  V1 := V1 + V2 + V3;
end;
```

Based on what we just mentioned about promotion rules, it would seem at first glance that the preceding code would result in the value `350` as an integer. However, if you take a closer look, you'll see that this is not the case. Because the order of precedence is from left to right, the first equation executed. is `V1 + V2`. Because these two variants refer to strings, a string concatenation is performed, resulting in the string `'10050'`. That result is then added to the integer value held by the variant `V3`. Because `V3` is an integer, the result `'10050'` is converted to an integer and added to `V3`, thus providing an end result of `10250`.

Delphi promotes the variants to the highest type in the equation in order to successfully carry out the calculation. However, when an operation is attempted on two variants of which Delphi cannot make any sense, an *invalid variant type conversion* exception is raised. The following code illustrates this:

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2;  // Raises an exception.
end;
```

As stated earlier, it's sometimes a good idea to explicitly typecast a variant to a specific data type if you know what that type is and if it's used in an expression. Consider the following line of code:

```
V4 := V1 * V2 / V3;
```

Before a result can be generated for this equation, each operation is handled by a runtime function that goes through several gyrations to determine the compatibility of the types the variants represent. Then the conversions are made to the appropriate data types. This results in a large amount of overhead and code size. A better solution is obviously not to use variants. However, when necessary, you can also explicitly typecast the variants so the data types are resolved at compile time:

```
V4 := Integer(V1) * Double(V2) / Integer(V3);
```

Keep in mind that this assumes you know the data types the variants represent.

Empty and Null

Two special `VType` values for variants merit a brief discussion. The first is `varEmpty`, which means that the variant has not yet been assigned a value. This is the initial value of the variant set by the compiler as it comes into scope. The other is `varNull`, which is different from `varEmpty` in that it actually represents the value `Null` as opposed to a lack of value. This distinction between no value and a `Null` value is especially important when applied to the field

values of a database table. In Part III of this book, "Database Development," you'll learn how variants are used in the context of database applications.

Another difference is that attempting to perform any equation with a variant containing a `varEmpty VType` value will result in an *invalid variant operation* exception. The same isn't true of variants containing a `varNull` value, however. When a variant involved in an equation contains a `Null` value, that value will propagate to the result. Therefore, the result of any equation containing a `Null` is always `Null`.

If you want to assign or compare a variant to one of these two special values, the `System` unit defines two variants, `Unassigned` and `Null`, which have the `VType` values of `varEmpty` and `varNull`, respectively.

CAUTION

It might be tempting to use variants instead of the conventional data types because they seem to offer so much flexibility. However, this will increase the size of your code and cause your applications to run more slowly. Additionally, it will make your code more difficult to maintain. Variants are useful in many situations. In fact, the VCL, itself, uses variants in several places, most notably in the ActiveX and database areas, because of the data type flexibility they offer. Generally speaking, however, you should use the conventional data types instead of variants. Only in situations where the flexibility of the variant outweighs the performance of the conventional method should you resort to using variants. Ambiguous data types beget ambiguous bugs.

Variant Arrays

Earlier we mentioned that a variant can refer to a nonhomogeneous array. Therefore, the following syntax is valid:

```
var
  V: Variant;
  I, J: Integer;
begin
  I := V[J];
end;
```

Bear in mind that, although the preceding code will compile, you'll get an exception at runtime because `V` does not yet contain a variant array. Object Pascal provides several variant array support functions that allow you to create a variant array. Two of these functions are `VarArrayCreate()` and `VarArrayOf()`.

VarArrayCreate()

`VarArrayCreate()` is defined in the `Variants` unit as

```
function VarArrayCreate(const Bounds: array of Integer;
  VarType: Integer): Variant;
```

To use `VarArrayCreate()`, you pass in the array bounds for the array you want to create and a variant type code for the type of the array elements (the first parameter is an open array, which is discussed in the "Passing Parameters" section later in this chapter). For example, the following code returns a variant array of integers and assigns values to the array items:

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 4], varInteger); // Create a 4-element array
  V[1] := 1;
  V[2] := 2;
  V[3] := 3;
  V[4] := 4;
end;
```

If variant arrays of a single type aren't confusing enough, you can pass `varVariant` as the type code in order to create a variant array of variants! This way, each element in the array has the ability to contain a different type of data. You can also create a multidimensional array by passing in the additional bounds required. For example, the following code creates an array with the bounds `[1..4, 1..5]`:

```
V := VarArrayCreate([1, 4, 1, 5], varInteger);
```

NOTE

The `Variants` unit was added to the RTL in Delphi 6 because the support for variants was migrated out of the `System` unit. Among other things, this physical separation of the variant support code helped to smooth compatibility with Borland Kylix and provided the ability to extend variants to support developer-specified data types.

VarArrayOf()

The `VarArrayOf()` function is defined in the `Variants` unit as

```
function VarArrayOf(const Values: array of Variant): Variant;
```

This function returns a one-dimensional array whose elements are given in the `Values` parameter. The following example creates a variant array of three elements with an integer, a string, and a floating-point value:

```
V := VarArrayOf([1, 'Delphi', 2.2]);
```

Variant Array Support Functions and Procedures

In addition to `VarArrayCreate()` and `VarArrayOf()`, there are several other variant array support functions and procedures. These functions are defined in the `Variants` System unit and are also shown here:

```
procedure VarArrayRedim(var A: Variant; HighBound: Integer);
function VarArrayDimCount(const A: Variant): Integer;
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
function VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
function VarArrayRef(const A: Variant): Variant;
function VarIsArray(const A: Variant): Boolean;
```

The `VarArrayRedim()` function allows you to resize the upper bound of the rightmost dimension of a variant array. The `VarArrayDimCount()` function returns the number of dimensions in a variant array. `VarArrayLowBound()` and `VarArrayHighBound()` return the lower and upper bounds of an array, respectively. `VarArrayLock()` and `VarArrayUnlock()` are two special functions, which are described in further detail in the next section.

`VarArrayRef()` is intended to work around a problem that exists in passing variant arrays to OLE Automation servers. The problem occurs when you pass a variant containing a variant array to an automation method, like this:

```
Server.PassVariantArray(VA);
```

The array is passed not as a variant array but rather as a variant containing a variant array—an important distinction. If the server expected a variant array rather than a reference to one, the server will likely encounter an error condition when you call the method with the preceding syntax. `VarArrayRef()` takes care of this situation by massaging the variant into the type and value expected by the server. Here's the syntax for using `VarArrayRef()`:

```
Server.PassVariantArray(VarArrayRef(VA));
```

`VarIsArray()` is a simple Boolean check, which returns `True` if the variant parameter passed to it is a variant array or `False` otherwise.

Initializing a Large Array: `VarArrayLock()` and `VarArrayUnlock()`

Variant arrays are important in OLE Automation because they provide the only means for passing raw binary data to an OLE Automation server (note that pointers aren't a legal type in OLE Automation, as you'll learn in Chapter 15, "COM Development"). However, if used incorrectly, variant arrays can be a rather inefficient means of exchanging data. Consider the following line of code:

```
V := VarArrayCreate([1, 10000], VarByte);
```

This line creates a variant array of 10,000 bytes. Suppose that you have another array (nonvariant) declared of the same size and you want to copy the contents of this nonvariant array to the variant array. Normally, you can only do this by looping through the elements and assigning them to the elements of the variant array, as shown here:

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  for i := 1 to 10000 do
    V[i] := A[i];
end;
```

The problem with this code is that it's bogged down by the significant overhead required just to initialize the variant array elements. This is because the assignments to the array elements must go through the runtime logic to determine type compatibility, the location of each element, and so forth. To avoid these runtime checks, you can use the `VarArrayLock()` function and the `VarArrayUnlock()` procedure.

`VarArrayLock()` locks the array in memory so that it cannot be moved or resized while it's locked, and it returns a pointer to the array data. `VarArrayUnlock()` unlocks an array locked with `VarArrayLock()` and once again allows the variant array to be resized and moved in memory. After the array is locked, you can employ a more efficient means to initialize the data by using, for example, the `Move()` procedure with the pointer to the array's data. The following code performs the initialization of the variant array shown earlier, but in a much more efficient manner:

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  P := VarArrayLock(V);
  try
    Move(A, P^, 10000);
  finally
    VarArrayUnlock(V);
  end;
end;
```

Supporting Functions

There are several other common support functions for variants that you can use. These functions are declared in the `Variants` System unit and are also listed here:

```
procedure VarClear(var V: Variant);
procedure VarCopy(var Dest: Variant; const Source: Variant);
procedure VarCast(var Dest: Variant; const Source: Variant; VarType: Integer);
function VarType(const V: Variant): Integer;
function VarAsType(const V: Variant; VarType: Integer): Variant;
function VarIsEmpty(const V: Variant): Boolean;
```

```
function VarIsNull(const V: Variant): Boolean;
function VarToStr(const V: Variant): string;
function VarFromDateTime(DateTime: TDateTime): Variant;
function VarToDateTime(const V: Variant): TDateTime;
```

The `VarClear()` procedure clears a variant and sets the `VType` field to `varEmpty`. `VarCopy()` copies the `Source` variant to the `Dest` variant. The `VarCast()` procedure converts a variant to a specified type and stores that result into another variant. `VarType()` returns one of the `var`*`XXX`* type codes for a specified variant. `VarAsType()` has the same functionality as `VarCast()`. `VarIsEmpty()` returns `True` if the type code on a specified variant is `varEmpty`. `VarIsNull()` indicates whether a variant contains a `Null` value. `VarToStr()` converts a variant to its string representation (an empty string in the case of a `Null` or empty variant). `VarFromDateTime()` returns a variant that contains a given `TDateTime` value. Finally, `VarToDateTime()` returns the `TDateTime` value contained in a variant.

OleVariant

The `OleVariant` type is nearly identical to the `Variant` type described throughout this section of this chapter. The only difference between `OleVariant` and `Variant` is that `OleVariant` only supports Automation-compatible types. Currently, the only `VType` supported that's not Automation-compatible is `varString`, the code for `AnsiString`. When an attempt is made to assign an `AnsiString` to an `OleVariant`, the `AnsiString` will be automatically converted to an OLE `BSTR` and stored in the variant as a `varOleStr`.

Currency

Delphi 2.0 introduced a new type called `Currency`, which is ideal for financial calculations. Unlike floating-point numbers, which allow the decimal point to "float" within a number, `Currency` is a fixed-point decimal type that's hard-coded to a precision of 15 digits before the decimal and four digits after the decimal. As such, it's not susceptible to round-off errors as are floating-point types. When porting your Delphi 1.0 projects, it's a good idea to use this type in place of `Single`, `Real`, `Double`, and `Extended` where money is involved.

User-Defined Types

Integers, strings, and floating-point numbers often are not enough to adequately represent variables in the real-world problems that programmers must try to solve. In cases like these, you must create your own types to better represent variables in the current problem. In Pascal, these user-defined types usually come in the form of records or objects; you declare these types using the `Type` keyword.

Arrays

Object Pascal enables you to create arrays of any type of variable (except files). For example, a variable declared as an array of eight integers reads like this:

```
var
  A: Array[0..7] of Integer;
```

This statement is equivalent to the following C declaration:

```
int A[8];
```

It's also equivalent to this Visual Basic statement:

```
Dim A(8) as Integer
```

Object Pascal arrays have a special property that differentiates them from other languages: They don't have to begin at a certain number. You can therefore declare a three-element array that starts at 28, as in the following example:

```
var
  A: Array[28..30] of Integer;
```

Because Object Pascal arrays aren't guaranteed to begin at 0 or 1, you must use some care when iterating over array elements in a `for` loop. The compiler provides built-in functions called `High()` and `Low()`, which return the lower and upper bounds of an array variable or type, respectively. Your code will be less error prone and easier to maintain if you use these functions to control your `for` loop, as shown here:

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // don't hard-code for loop!
    A[i] := i;
end;
```

TIP

Always begin character arrays at 0. Zero-based character arrays can be passed to functions that require `PChar`-type variables. This is a special-case allowance that the compiler provides.

To specify multiple dimensions, use a comma-delimited list of bounds:

```
var
  // Two-dimensional array of Integer:
  A: array[1..2, 1..2] of Integer;
```

To access a multidimensional array, use commas to separate each dimension within one set of brackets:

```
I := A[1, 2];
```

Dynamic Arrays

Dynamic arrays are dynamically allocated arrays in which the dimensions aren't known at compile time. To declare a dynamic array, just declare an array without including the dimensions, like this:

```
var
  // dynamic array of string:
  SA: array of string;
```

Before you can use a dynamic array, you must use the `SetLength()` procedure to allocate memory for the array:

```
begin
  // allocate room for 33 elements:
  SetLength(SA, 33);
```

Once memory has been allocated, you can access the elements of the dynamic array just like a normal array:

```
  SA[0] := 'Pooh likes hunny';
  OtherString := SA[0];
```

NOTE

Dynamic arrays are always zero-based.

Dynamic arrays are lifetime managed, so there's no need to free them when you're through using them because they'll be released when they leave scope. However, there might come a time when you want remove the dynamic array from memory before it leaves scope (if it uses a lot of memory, for example) To do this, you need only assign the dynamic array to `nil`:

```
SA := nil;  // releases SA
```

Dynamic arrays are manipulated using reference semantics similar to `AnsiString` types rather than value semantics like a normal array. A quick test: What is the value of `A1[0]` at the end of the following code fragment?

```
var
  A1, A2: array of Integer;
```

```
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

The correct answer is 26. The reason is because the assignment `A2 := A1` doesn't create a new array but instead provides `A2` with a reference to the same array as `A1`. Therefore, any modifications to `A2` will also affect `A1`. If you want instead to make a complete copy of `A1` in `A2`, use the `Copy()` standard procedure:

```
A2 := Copy(A1);
```

After this line of code is executed, `A2` and `A1` will be two separate arrays initially containing the same data. Changes to one will not affect the other. You can optionally specify the starting element and number of elements to be copied as parameters to `Copy()`, as shown here:

```
// copy 2 elements, starting at element one:
A2 := Copy(A1, 1, 2);
```

Dynamic arrays can also be multidimensional. To specify multiple dimensions, add an additional `array of` to the declaration for each dimension:

```
var
  // two-dimensional dynamic array of Integer:
  IA: array of array of Integer;
```

To allocate memory for a multidimensional dynamic array, pass the sizes of the other dimensions as additional parameters to `SetLength()`:

```
begin
  // IA will be a 5 x 5 array of Integer
  SetLength(IA, 5,  5);
```

You access multidimensional dynamic arrays the same way you do normal multidimensional arrays; each element is separated by a comma with a single set of brackets:

```
IA[0,3] := 28;
```

Records

A user-defined structure is referred to as a `record` in Object Pascal, and it's the equivalent of C's `struct` or Visual Basic's `Type`. As an example, here's a record definition in Pascal as well as equivalent definitions in C and Visual Basic:

```
{ Pascal }
Type
  MyRec = record
    i: Integer;
```

```
    d: Double;
  end;

/* C */
typedef struct {
  int i;
  double d;
} MyRec;

'Visual Basic
Type MyRec
  i As Integer
  d As Double
End Type
```

When working with a record, you use the dot symbol to access its fields. Here's an example:

```
var
  N: MyRec;
begin
  N.i := 23;
  N.d := 3.4;
end;
```

Object Pascal also supports *variant records*, which allow different pieces of data to overlay the same portion of memory in the record. Not to be confused with the `Variant` data type, variant records allow each overlapping data field to be accessed independently. If your background is C, you'll recognize variant records as being the same concept as a `union` within C `struct`. The following code shows a variant record in which a `Double`, `Integer`, and `char` all occupy the same memory space:

```
type
  TVariantRecord = record
    NullStrField: PChar;
    IntField: Integer;
    case Integer of
      0: (D: Double);
      1: (I: Integer);
      2: (C: char);
  end;
```

NOTE

The rules of Object Pascal state that the variant portion of a record cannot be of any lifetime-managed type.

Here's the C equivalent of the preceding type declaration:

```
struct TUnionStruct
{
  char * StrField;
  int IntField;
  union u
  {
    double D;
    int i;
    char c;
  };
};
```

Sets

Sets are a uniquely Pascal type that have no equivalent in Visual Basic, C, or C++ (although Borland C++Builder does implement a template class called `Set`, which emulates the behavior of a Pascal set). Sets provide a very efficient means of representing a collection of ordinal, character, or enumerated values. You can declare a new set type using the keywords `set of` followed by an ordinal type or subrange of possible set values. Here's an example:

```
type
  TCharSet = set of char;      // possible members: #0 - #255

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;  // can contain any combination of TEnum members

  TSubrangeSet = set of 1..10; // possible members: 1 - 10
  TAlphaSet = set of 'A'..'z'; // possible members: 'A' - 'z'
```

Note that a set can only contain up to 256 elements. Additionally, only ordinal types can follow the `set of` keywords. Therefore, the following declarations are illegal:

```
type
  TIntSet = set of Integer;  // Invalid: too many elements
  TStrSet = set of string;   // Invalid: not an ordinal type
```

Sets store their elements internally as individual bits, which makes them very efficient in terms of speed and memory usage. Sets with fewer than 32 elements in the base type can be stored and operated upon in CPU registers, for even greater efficiency. Sets with 32 or more elements (such as a set of `char`–255 elements) are stored in memory. To get the maximum performance benefit from sets, keep the number of elements in the set's base type under 32.

Using Sets

Use square brackets when referencing set elements. The following code demonstrates how to declare `set` type variables and assign them values:

```
type
  TCharSet = set of char;      // possible members: #0 - #255

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
  TEnumSet = set of TEnum;  // can contain any combination of TEnum members

var
  CharSet: TCharSet;
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10; // possible members: 1 - 10
  AlphaSet: set of 'A'..'z'; // possible members: 'A' - 'z'

begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];  // Empty; no elements
end;
```


Set Operators

Object Pascal provides several operators for use in manipulating sets. You can use these operators to determine set membership, union, difference, and intersection.

Membership

Use the `in` operator to determine whether a given element is contained in a particular set. For example, the following code would be used to determine whether the `CharSet` set mentioned earlier contains the letter `'S'`:

```
if 'S' in CharSet then
  // do something;
```

The following code determines whether `EnumSet` lacks the member `Monday`:

```
if not (Monday in EnumSet) then
  // do something;
```

Union and Difference

Use the `+` and `-` operators or the `Include()` and `Exclude()` procedures to add and remove elements to and from a set variable:

```
Include(CharSet, 'a');           // add 'a' to set
CharSet := CharSet + ['b'];      // add 'b' to set
Exclude(CharSet, 'x');           // remove 'z' from set
CharSet := CharSet - ['y', 'z']; // remove 'y' and 'z' from set
```

TIP

When possible, use `Include()` and `Exclude()` to add and remove a single element to and from a set rather than the `+` and `-` operators. Both `Include()` and `Exclude()` constitute only one machine instruction each, whereas the `+` and `-` operators require 13 + 6*n* (where *n* is the size in bits of the set) instructions.

Intersection

Use the `*` operator to calculate the intersection of two sets. The result of the expression `Set1 * Set2` is a set containing all the members that `Set1` and `Set2` have in common. For example, the following code could be used as an efficient means for determining whether a given set contains multiple elements:

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
  // do something
```

Objects

Think of objects as records that also contain functions and procedures. Delphi's object model is discussed in much greater detail later in the "Using Delphi Objects" section of this chapter, so this section covers just the basic syntax of Object Pascal objects. An object is defined as follows:

```
Type
  TChildObject = class(TParentObject);
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

Although Delphi objects aren't identical to C++ objects, this declaration is roughly equivalent to the following C++ declaration:

```
class TChildObject : public TParentObject
{
  int SomeVar;
  void SomeProc();
};
```

Methods are defined in the same way as normal procedures and functions (which are discussed in the section "Procedures and Functions"), with the addition of the object name and the dot symbol operator:

```
procedure TChildObject.SomeProc;
begin
  { procedure code goes here }
end;
```

Object Pascal's `.` symbol is similar in functionality to Visual Basic's `.` operator and C++'s `::` operator. You should note that, although all three languages allow usage of classes, only Object Pascal and C++ allow the creation of new classes that behave in a fully object-oriented manner, which we'll describe in the section "Object-Oriented Programming."

NOTE

Object Pascal objects aren't laid out in memory the same as C++ objects, so it's not possible to use C++ objects directly from Delphi (and vice versa). If you are interested in learning more about how this is done, you might want to browse Chapter 13, "Hard-core Techniques," in the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this book. That chapter shows a technique for sharing objects between C++ and Delphi.

An exception to this is Borland C++Builder's capability of creating classes that map directly to Object Pascal classes using the proprietary `__declspec(delphiclass)` directive. Such objects are likewise incompatible with regular C++ objects.

Pointers

A *pointer* is a variable that contains a memory location. You already saw an example of a pointer in the `PChar` type earlier in this chapter. Pascal's generic pointer type is called, aptly, `Pointer`. A `Pointer` is sometimes called an untyped pointer because it contains only a memory address, and the compiler doesn't maintain any information on the data to which it points. That notion, however, goes against the grain of Pascal's typesafe nature, so pointers in your code will usually be typed pointers.

NOTE

Pointers are a somewhat advanced topic, and you definitely don't need to master them to write a Delphi application. As you become more experienced, pointers will become another valuable tool for your programmer's toolbox.

Typed pointers are declared by using the `^` (or *pointer*) operator in the `Type` section of your program. Typed pointers help the compiler keep track of exactly what kind of type a particular pointer points to, thus enabling the compiler to keep track of what you're doing (and can do) with a pointer variable. Here are some typical declarations for pointers:

```
Type
  PInt = ^Integer;        // PInt is now a pointer to an Integer
```

```
  Foo = record              // A record type
    GobbledyGook: string;
    Snarf: Real;
  end;
  PFoo = ^Foo;              // PFoo is a pointer to a foo type
var
  P: Pointer;               // Untyped pointer
  P2: PFoo;                 // Instance of PFoo
```

NOTE

C programmers will notice the similarity between Object Pascal's `^` operator and C's `*` operator. Pascal's `Pointer` type corresponds to C's `void *` type.

Remember that a pointer variable only stores a memory address. Allocating space for whatever the pointer points to is your job as a programmer. You can allocate space for a pointer by using one of the memory-allocation routines discussed earlier and shown in Table 2.6.

NOTE

When a pointer doesn't point to anything (its value is zero), its value is said to be `nil`, and it is often called a *nil* or *null* pointer.

If you want to access the data that a particular pointer points to, follow the pointer variable name with the `^` operator. This method is known as *dereferencing* the pointer. The following code illustrates working with pointers:

```
Program PtrTest;

Type
  MyRec = record
    I: Integer;
    S: string;
    R: Real;
  end;
  PMyRec = ^MyRec;

var
  Rec : PMyRec;
begin
  New(Rec);      // allocate memory for Rec
  Rec^.I := 10;  // Put stuff in Rec. Note the dereference
  Rec^.S := 'And now for something completely different.';
```

```
  Rec^.R := 6.384;
  { Rec is now full }
  Dispose(Rec);  // Don't forget to free memory!
end.
```

When to Use `New()`

Use the `New()` function to allocate memory for a pointer to a structure of a known size. Because the compiler knows how big a particular structure is, a call to `New()` will cause the correct number of bytes to be allocated, thus making it safer and more convenient to use than `GetMem()` or `AllocMem()`. Never allocate `Pointer` or `PChar` variables by using the `New()` function because the compiler cannot guess how many bytes you need for this allocation. Remember to use `Dispose()` to free any memory you allocate using the `New()` function.

You'll typically use `GetMem()` or `AllocMem()` to allocate memory for structures for which the compiler cannot know the size. The compiler cannot tell ahead of time how much memory you want to allocate for `PChar` or `Pointer` types, for example, because of their variable-length nature. Be careful not to try to manipulate more data than you have allocated with these functions, however, because this is one of the classic causes of an Access Violation error. You should use `FreeMem()` to clean up any memory you allocate with `GetMem()` or `AllocMem()`. `AllocMem()`, by the way, is a bit safer than `GetMem()` because `AllocMem()` always initializes the memory it allocates to zero.

One aspect of Object Pascal that might give C programmers some headaches is the strict type checking performed on pointer types. For example, the variables `a` and `b` in the following example aren't type compatible:

```
var
  a: ^Integer;
  b: ^Integer;
```

By contrast, the variables `a` and `b` in the equivalent declaration in C are type compatible:

```
int *a;
int *b
```

Object Pascal creates a unique type for each pointer-to-type declaration, so you must create a named type if you want to assign values from `a` to `b`, as shown here:

```
type
  PtrInteger = ^Integer;  // create named type

var
  a, b: PtrInteger;       // now a and b are compatible
```

Type Aliases

Object Pascal has the capability to create new names, or *aliases*, for types that are already defined. For example, if you want to create a new name for an `Integer` called `MyReallyNiftyInteger`, you could do so using the following code:

```
type
  MyReallyNiftyInteger = Integer;
```

The newly defined type alias is compatible in all ways with the type for which it's an alias, meaning, in this case, that you could use `MyReallyNiftyInteger` anywhere in which you could use `Integer`.

It's possible, however, to define *strongly typed* aliases that are considered new, unique types by the compiler. To do this, use the `type` reserved word in the following manner:

```
type
  MyOtherNeatInteger = type Integer;
```

Using this syntax, the `MyOtherNeatInteger` type will be converted to an `Integer` when necessary for purposes of assignment, but `MyOtherNeatInteger` will not be compatible with `Integer` when used in `var` and `out` parameters. Therefore, the following code is syntactically correct:

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I :=  1;
  MONI := I;
```

On the other hand, the following code will not compile:

```
procedure Goon(var Value: Integer);
begin
  // some code
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);  // Error: M is not var compatible with Integer
```

In addition to these compiler-enforced type compatibility issues, the compiler also generates runtime type information for strongly typed aliases. This enables you to create unique property editors for simple types, as you'll learn in Chapter 12, "Advanced VCL Component Building."

Typecasting and Type Conversion

Typecasting is a technique by which you can force the compiler to view a variable of one type as another type. Because of Pascal's strongly typed nature, you'll find that the compiler is very picky about types matching up in the formal and actual parameters of a function call. Hence, you occasionally will be required to cast a variable of one type to a variable of another type to make the compiler happy. Suppose, for example, that you need to assign the value of a character to a `byte` variable:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := c;   // compiler complains on this line
end.
```

In the following syntax, a typecast is required to convert `c` into a `byte`. In effect, a typecast tells the compiler that you really know what you're doing and want to convert one type to another:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := byte(c);   // compiler happy as a clam on this line
end.
```

NOTE

You can typecast a variable of one type to another type only if the data size of the two variables is the same. For example, you cannot typecast a `Double` as an `Integer`. To convert a floating-point type to an integer, use the `Trunc()` or `Round()` functions. To convert an integer into a floating-point value, use the assignment operator: `FloatVar := IntVar`.

Object Pascal also supports a special variety of typecasting between objects using the `as` operator, which is described later in the "Runtime Type Information" section of this chapter.

String Resources

Delphi 3 introduced the capability to place string resources directly into Object Pascal source code using the `resourcestring` clause. *String resources* are literal strings (usually those displayed to the user) that are physically located in a resource attached to the application or library rather than embedded in the source code. Your source code references the string resources in place of string literals. By separating strings from source code, your application can be translated more easily by added string resources in a different language. String resources are declared in the form of *`identifier = string literal`* in the `resourcestring` clause, as shown here:

```
resourcestring
  ResString1 = 'Resource string 1';
  ResString2 = 'Resource string 2';
  ResString3 = 'Resource string 3';
```

Syntactically, resource strings can be used in your source code in a manner identical to string constants:

```
resourcestring
  ResString1 = 'hello';
  ResString2 = 'world';

var
  String1: string;

begin
  String1 := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

Testing Conditions

This section compares `if` and `case` constructs in Pascal to similar constructs in C and Visual Basic. We assume that you've used these types of programmatic constructs before, so we don't spend time explaining them to you.

The `if` Statement

An `if` statement enables you to determine whether certain conditions are met before executing a particular block of code. As an example, here's an `if` statement in Pascal, followed by equivalent definitions in C and Visual Basic:

```
{ Pascal }
if x = 4 then y := x;

/* C */
if (x == 4) y = x;

'Visual Basic
If x = 4 Then y = x
```

NOTE

If you have an `if` statement that makes multiple comparisons, make sure that you enclose each set of comparisons in parentheses for code clarity. Do this:

```
if (x = 7) and (y = 8) then
```

However, don't do this (it causes the compiler displeasure):

```
if x = 7 and y = 8 then
```


Use the `begin` and `end` keywords in Pascal almost as you would use `{` and `}` in C and C++. For example, use the following construct if you want to execute multiple lines of text when a given condition is true:

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

You can combine multiple conditions using the `if..else` construct:

```
if x =100 then
  SomeFunction
else if x = 200 then
  SomeOtherFunction
else begin
  SomethingElse;
  Entirely;
end;
```

Using `case` Statements

The `case` statement in Pascal works in much the same way as a `switch` statement in C and C++. A `case` statement provides a means for choosing one condition among many possibilities without a huge `if..else if..else if` construct. Here's an example of Pascal's `case` statement:

```
case SomeIntegerVariable of
  101 : DoSomething;
```

```
  202 : begin
      DoSomething;
      DoSomethingElse;
    end;
  303 : DoAnotherThing;
  else DoTheDefault;
end;
```

NOTE

The selector type of a `case` statement must be an ordinal type. It's illegal to use nonordinal types, such as strings, as `case` selectors.

Here's the C `switch` statement equivalent to the preceding example:

```
switch (SomeIntegerVariable)
{
  case 101: DoSomeThing(); break;
  case 202: DoSomething();
            DoSomethingElse(); break
  case 303: DoAnotherThing(); break;
  default: DoTheDefault();
}
```

Loops

A *loop* is a construct that enables you to repeatedly perform some type of action. Pascal's loop constructs are very similar to what you should be familiar with from your experience with other languages, so we don't spend any time teaching you about loops. This section describes the various loop constructs you can use in Pascal.

The `for` Loop

A `for` loop is ideal when you need to repeat an action a predetermined number of times. Here's an example, albeit not a very useful one, of a `for` loop that adds the loop index to a variable 10 times:

```
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do
    inc(X, I);
end.
```

The C equivalent of the preceding example is as follows:

```
void main(void) {
  int x, i;
  x = 0;
  for(i=1; i<=10; i++)
    x += i;
}
```

Here's the Visual Basic equivalent of the same concept:

```
X = 0
For I = 1 to 10
  X = X + I
Next I
```


CAUTION

A caveat to those familiar with Delphi 1: Assignments to the loop control variable are no longer allowed due to the way the loop is optimized and managed by the 32-bit compiler.

The `while` Loop

Use a `while` loop construct when you want some part of your code to repeat itself while some condition is true. A `while` loop's conditions are tested before the loop is executed, and a classic example for the use of a `while` loop is to repeatedly perform some action on a file as long as the end of the file isn't encountered. Here's an example demonstrating a loop that reads one line at a time from a file and writes it to the screen:

```
Program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // a text file
  s: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  while not EOF(f) do begin
    readln(f, S);
    writeln(S);
  end;
  CloseFile(f);
end.
```

Pascal's `while` loop works basically the same as C's `while` loop or Visual Basic's `Do While` loop.

repeat..until

The `repeat..until` loop addresses the same type of problem as a `while` loop but from a different angle. It repeats a given block of code until a certain condition becomes `True`. Unlike a `while` loop, the loop code is always executed at least once because the condition is tested at the end of the loop. Pascal's `repeat..until` is roughly equivalent to C's `do..while` loop.

For example, the following code snippet repeats a statement that increments a counter until the value of the counter becomes greater than 100:

```
var
  x: Integer;
begin
  X := 1;
  repeat
    inc(x);
  until x > 100;
end.
```

The Break() Procedure

Calling `Break()` from inside a `while`, `for`, or `repeat` loop causes the flow of your program to skip immediately to the end of the currently executing loop. This method is useful when you need to leave the loop immediately because of some circumstance that might arise within the loop. Pascal's `Break()` procedure is analogous to C's `break` and Visual Basic's `Exit` statement. The following loop uses `Break()` to terminate the loop after five iterations:

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do
  begin
    MessageBeep(0);          // make the computer beep
    if i = 5 then Break;
  end;
end;
```

The Continue() Procedure

Call `Continue()` inside a loop when you want to skip over a portion of code and the flow of control to continue with the next iteration of the loop. Note in the following example that the code after `Continue()` isn't executed in the first iteration of the loop: