



Cover Art By: Tom McKeith

Do the Strand

Delphi 2 Multithreading

ON THE COVER



9 Do the Strand — Joseph C. Fung
The 32-bit version of Windows features true multitasking through multithreading, a technology that can enhance the performance of your Windows 95 and Windows NT applications. Mr Fung explains how to implement multithreading capabilities into your Delphi 2 programs using clear, workable examples.

FEATURES



15 Informant Spotlight — Kevin Bluck
Most programmers use Paradox tables for their Delphi database applications. Depending on the application, however, Local InterBase may be the better tool. To help you decide which product best suits your needs, Mr Bluck provides a comprehensive analysis and comparison of Paradox and InterBase.



20 DBNavigator — Cary Jensen, Ph.D.
In addition to ranges, DataSets, and SQL queries, Delphi 2 provides filtering capabilities. Dr Jensen introduces the cast of characters — *TDataSet's TTable, TQuery, and TStoredProc* — you'll need to become familiar with to help your users "drill down" into their data with Delphi 2.



23 From the Palette — Ray Konopka
If you're interested in creating Delphi components, Mr Konopka's article is a great place to start. His *TRzAddress* component appears simple. Peel back its layers, however, and you'll see that this well-built control demonstrates the effective encapsulation of other components.



28 OP Tech — Keith Wood
There's good recursion and there's bad recursion. Mr Wood examines the positive side of this phenomenon in Object Pascal by exploring a number of interesting implementations, including: calculating factorials, building binary trees, and creating mind-bending fractals.



35 Dynamic Delphi — Andrew Wozniwicz
It's the final installment of a four-part series on DLLs. Mr Wozniwicz concludes by discussing: dynamically loading and releasing a DLL, using dynamically loaded DLLs, and accessing data in a library. He even provides a DLL summary for your quick reference.



40 Memory Monitor for Delphi
Product review by Robert Vivrette
Resource leaks are the bane of Windows programming. Fortunately, there's now a tool to help Delphi developers stop the bleeding. It's named MemMonD and our own Mr Vivrette puts it through its paces.



43 Developing Custom Delphi Components
Book review by Richard Wagner

DEPARTMENTS

- 2 Editorial** by Jerry Coffey
- 3 Delphi Tools**
- 6 Newsline**
- 44 File | New** by Richard Wagner



SYMPOSIUM

*Why does The Unabomber love Visual Basic?
Because he hates technology.*
— Anonymous



I love that one. It has all the elements of a great belly laugh. It's short, topical, and has that cruel edge a joke needs to really draw blood. Cruel because it rings true — as any VB-to-Delphi convert can attest.

The tone was only slightly more serious at the First Annual *Delphi Informant* Readers Choice awards dinner, where representatives of the best third-party Delphi tools were presented with lovely tokens of your affection. However, there is no affectionate term for the award itself — a handsome Lucite tower encapsulating the *DI* “Big D” and its happy electron satellites (a dire warning of a radiation leak or the first image in an “Our Friend the Atom” slideshow, depending on your mood). No candidate jumps to the forefront. For example, a “Dimmy” (from *Delphi Informant Magazine*) doesn't sound like something you'd really like to win. Obviously there's work to be done in this area. Or perhaps I should heed today's chic admonition and simply “not go there.”



Woll2Woll Software's Roy Woll graciously accepts the *Delphi Informant* Readers Choice Product of the Year award.

The important thing is that there *is* a happy, healthy, and still burgeoning third-party market for Delphi. This is especially remarkable given the fact that Delphi is just over a year old. Product of the Year was a runaway with Woll2Woll Software's InfoPower taking top honors. Other winners included Pacheco and Teixeira's *Delphi Developer's Guide* for Best Delphi Book, and Successware International's Apollo Rock-E-T for Best Delphi Add-In (see the April *DI* for all the winners).

And despite the fact the *DI* Readers Choice Awards weren't for them, *the* three top members of the Delphi development team — Delphi Chief Architect, Anders Hejlsberg; Director of Delphi Product Management, Zack Urlocker; and Director of Delphi Development, Gary Whizin — were kind enough to attend the festivities.

Actually, our little affair must have been a bit of a let down for the threesome, since earlier that evening they'd accepted the Jolt Cola award for Technical Excellence, beating an impressive array of competitors including white-hot Java, Symantec C++, and Visual Basic (okay, so *most* of the competitors were impressive).

But if it was anticlimactic for the Borland crew, they never let on. Come to think of it, it was probably just the thing after receiving the Jolt Cola award. The Delphi team took



Zack Urlocker, Director of Delphi Product Management, and Delphi creator, Anders Hejlsberg arriving at the *Delphi Informant* Readers Choice Awards. They'd just left the Jolt Cola awards where Delphi won for Technical Excellence.

home an industry award for themselves, and then looked on as the community they created was honored for its achievements. 'Course the fajitas and margaritas didn't hurt.

Thanks for reading,

Jerry Coffey, Editor-in-Chief

Internet: jcoffey@informant.com
CompuServe: 70304,3633
Fax: 916-686-8497
Snail: 10519 E. Stockton Blvd., Ste. 142, Elk Grove, CA 95624



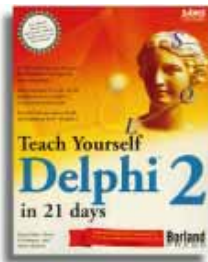
New Products
and Solutions



New Delphi Book

Teach Yourself Delphi 2
in 21 days

Dan Osier, Steve Grobman,
and Steve Batson
SAMS Publishing



ISBN: 0-672-30863-0
Price: US\$35 (706 pages)
Phone: (800) 428-5331

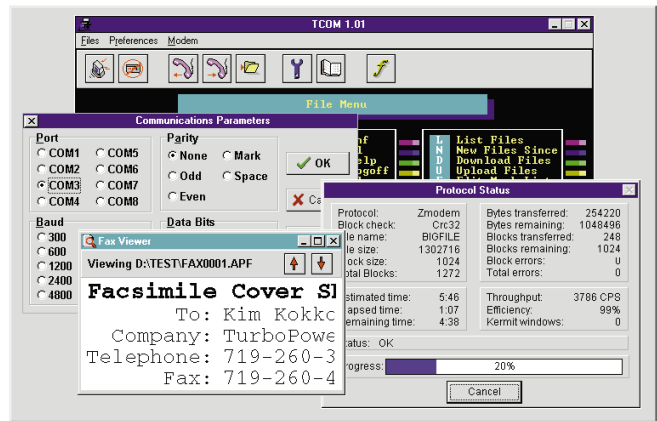
TurboPower Announces Async Professional 2.0 for Delphi

TurboPower Software Co., of Colorado Springs, CO, has announced *Async Professional 2.0 for Delphi* (APD), a serial communications library of native VCL components for Delphi.

Version 2.0 offers 32-bit support and fax components for converting files, sending and receiving fax files with fax modems, viewing and printing fax files, and converting print output to fax format. APD supports both 16- and 32-bit applications.

APD now includes TAPI devices, eliminating many modem configuration problems.

APD also features protocol status, modem selection, and dialing dialog boxes. It includes ANSI and VT100 emulators, ZModem, YModem, and XModem



protocols, debugging and tracing tools, modem database, and an event-driven dialing engine. File transfers and other operations run in the background.

A free trial version of APD is available on TurboPower's Web site and BBS.

The demonstration version has the full functionality of APD, but only runs when Delphi is operating.

Price: US\$199, includes full source code, printed documentation and online help file, free technical support by phone, e-mail, and fax, and a 60-day money-back guarantee.

Contact: TurboPower Software Co., PO Box 49009, Colorado Springs, CO 80949-9009

Phone: (800) 333-4160 or (719) 260-9136

Fax: (719) 260-7151

BBS: (719) 260-9726

CIS Forum: 60 PCVENB

Web Site: <http://www.tpower.com>

New Delphi Components for Internet Programming

Software Avenue, Inc. of Gilbert, AZ has announced the release of *The Internet Developer's Kit* for Delphi. The kit contains two new Delphi components, a document viewer, and 16- and 32-bit versions of the components. The two Delphi components, InternetClient and InternetServer, provide access to the Windows Socket library (WINSOCK).

The InternetClient component can be used to create various applications, from a Finger client to a World Wide Web Browser.

The InternetServer component can manage as many as 100 simultaneous client connections.

With this component, and an Internet SLIP account, Delphi developers can create Internet servers.

The kit also contains a series of documents collected from various Internet sites. This information describes the standards and protocols for common Internet services. The printed documentation provides installation instructions, a detailed description of both components, and programming examples. Context-sensitive online help is also available from within Delphi.

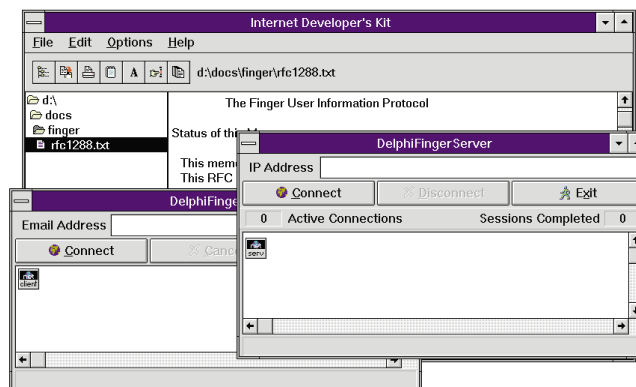
Price: US\$249.95, plus shipping and handling.

Contact: Software Avenue, Inc., PO Box 1324, Gilbert, AZ 85234

Phone: (800) 813-0876 or (602) 813-0876

E-Mail: CIS: 76455,3236

Web Site: <http://ourworld.computer.com:80/homepages/Software-Avenue/>



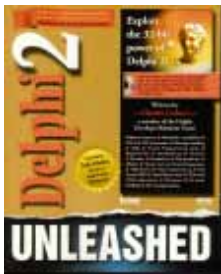
Delphi TOOLS

New Products
and Solutions



New Delphi Book

Delphi 2 Unleashed
Charles Calvert
SAMS Publishing



ISBN: 0-672-30858-4
Price: US\$59.99
(1,400 pages, CD-ROM)
Phone: (800) 428-5331

Game Tools for Delphi 2

Mobius Ltd. of Springfield, MA announced the availability of *Mobius Fast Sprites*. These native Delphi 2 components are designed to provide animation through the Windows 95 Games SDK and DirectDraw DLLs.

Fast Sprites enables Delphi 2 game developers to control video hardware using COM architecture. They allow developers to set screen resolutions, grab the CPU, and command the video card to create sprites and animation without code.

Fast Sprites also handles screen (buffer) flipping and updating in sync with the vertical blank, giving smooth tear-free graphics. Refresh rates have been clocked at over 70 FPS on an ATI Mach 64 Graphic



Xpression video card with a DX4 66 CPU. Fast Sprites are the first components of a complete suite; additional components are currently planned.

Price: Promotional offer US\$149, includes online help file and joystick

component. Registered users receive updates free for one year.

Contact: Mobius Ltd., 75 Dwight Rd., Springfield, MA 01108
Phone: (413) 827-9747
Fax: (413) 827-9747
Internet: CIS: 73563,533
Web Site: <http://www.xmission.com/~imagicom/mobius/mobius.html>

Shoreline Releases New Version of VisualPROS for Delphi 2

Shoreline Software of Vernon, CT has released version 2.0 of its *VisualPROS* for Delphi, an add-on component set for enhancing graphical user interfaces. The new version adds 16- and 32-bit versions of controls, optimizations, and full source code in Object Pascal.

Written in Delphi 2, VisualPROS controls typically add less than 20K-30K to an application. They use the existing Delphi 2 framework,

reducing the amount of code redundancy often found in VBXes and other components.

In this version, Shoreline has provided optimized components for bitmap management, drop-in management of both Windows 3.x .INI files and Registry elements of Windows 95, Windows NT, and online help.

They also enhanced the Tileback and HelpCloud components. The Tileback component now includes texture mapping for dialog box backgrounds with 25 texture choices, simultaneous display management of both gradient fills and bitmaps on form backgrounds, one component support for both MDI and regular form backgrounds with automatic detection, and nine directional gradient fill types.

The HelpCloud component now offers display, placement, and transparent support for

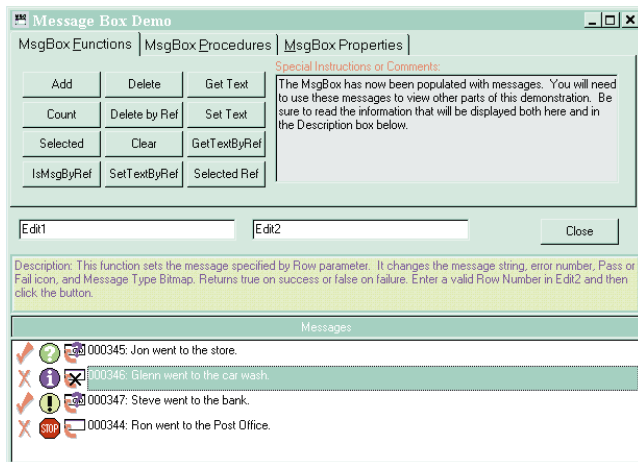
user-drawn help clouds.

VisualPROS supports any type of application created with Delphi 2. The 32-bit versions are not simple ports of the 16-bit controls, but are optimized for 32-bit execution.

VisualPROS' online help has been upgraded, and is now integrated with Delphi 2. Printed documentation is also available.

Currently VisualPROS is available from Breakthrough Technologies. For information call (800) 813-7909, (800) 323-1809, (602) 258-2715, or fax (602) 258-2805. An OCX version of the product is expected to ship in the second quarter of 1996.

Price: US\$149.95
Contact: Shoreline Software, 35-31 Talcottville Rd. #123, Vernon, CT 06066-4030
Phone: (860) 870-5707
Fax: (860) 870-5727
E-Mail: Internet: info@shoresoft.com
Web Site: <http://www.shoresoft.com>

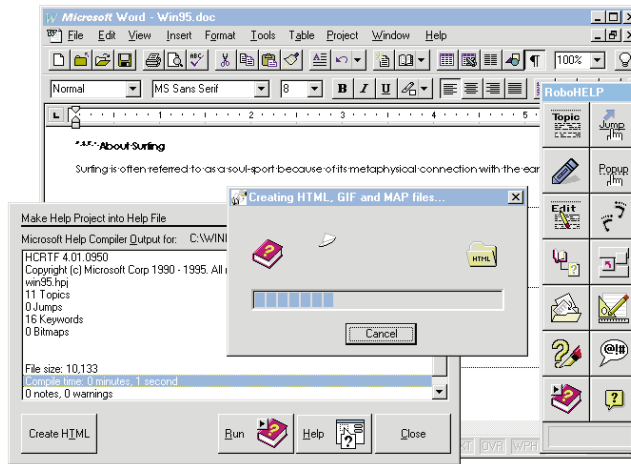


RoboHELP 95 HTML Edition Released by Blue Sky Software

Blue Sky Software Corp., of La Jolla, CA, has released *RoboHELP 95 HTML Edition*. This version allows users to create HTML and Windows Help files from the same source code. RoboHELP 95 HTML Edition turns Microsoft Word 7 for Windows 95 into an authoring tool capable of creating HTML files and Windows Help systems simultaneously.

The RoboHELP 95 HTML Edition recognizes the similarities between HTML and Windows Help, and provides a way to create the HTML Home page (including all HTML, GIF, and MAP files, and hyperlinks for a Web site and/or intranet).

The RoboHELP 95 HTML Edition can also be used to move information from Windows Help to



HTML, making it available on the Internet. In addition, the RoboHELP 95 HTML Edition is currently the only authoring tool providing simultaneous HTML and Help creation from single source code.

The RoboHELP 95 HTML Edition includes *Mastering HTML for Help Authors*, a guide to the

basics of HTML authoring.

Price: RoboHELP 95 HTML Edition, special promotional price, US\$699 (list price is US\$897).

Contact: Blue Sky Software Corp., 7777 Fay Ave., Suite 201, La Jolla, CA 92037

Phone: (800) 459-2356 or (619) 459-6365

Fax: (619) 459-6366

E-Mail: Internet: sales@blue-sky.com

Web Site: http://www.blue-sky.com

Delphi

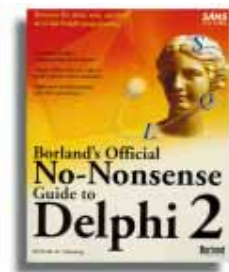
TOOLS

New Products and Solutions



New Delphi Book

Borland's Official
No-Nonsense Guide to Delphi 2
Michelle Manning
SAMS Publishing



ISBN: 0-672-30871-1
Price: US\$25 (386 pages)
Phone: (800) 428-5331

Wintertree Software Releases Updated Thesaurus Software

Wintertree Software Inc. has announced the release of the *ThesDB Thesaurus Engine, version 3.1*, a database of synonyms and a software library to access the database.

Developers can use ThesDB to add thesaurus and synonym-finding capabilities to their 16- and 32-bit applications. The thesaurus dialog box permits keyword searches, user-thesaurus maintenance, and synonym selection. The initial keyword and selected synonym can be set and retrieved by the application, replacing the selected word with a synonym. ThesDB also suggests replacements for misspelled or unknown words.

The new version includes 50,000 synonyms in over 3,100 word categories in the American English main thesaurus. Word categories in the main thesaurus are classified

according to parts of speech: adjectives, adverbs, nouns, and verbs. In addition, users can add custom word categories and synonyms.

ThesDB is available in three forms: the Win16 SDK for 16-bit Windows applications; the Win32 SDK for 32-bit development; and the Source SDK which includes the ANSI-C source code to the ThesDB engine and related software. The Source SDK is suitable for applications developed on non-Windows platforms, and includes Win16 and Win32 SDKs. All SDK types include a 50-page programmer's guide and a license to distribute applications royalty-free.

A demonstration version of ThesDB is available from Wintertree's CompuServe forum and Web page.

Price: Win16 SDK, US\$399 (CAN\$499); Win32 SDK, US\$399 (CAN\$499); and Source SDK, US\$1,499 (CAN\$1,899). The Win16 and Win32 SDKs are available bundled for US\$699 (CAN\$879).

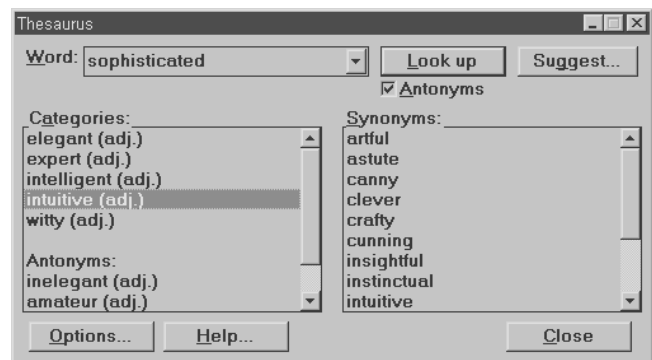
Contact: Wintertree Software Inc., 69 Beddington Ave., Nepean, Ontario, Canada, K2J 3N4

Phone: (800) 340-8803 or (613) 825-6271

Fax: (613) 825-5521

CIS Forum: GO WINSDK

Web Site: http://fox.nstn.ca/~wsi/



June 1996



Delphi 2 Help File Updated

Scotts Valley, CA — Borland has released an updated VCL.HLP file for Delphi 2. The file, VCL.ZIP, is available on Borland Online (<http://www.borland.com>) and Borland's new Delphi 2 CompuServe forum (GO BDELPHI32).

An improved, but interim version, this version fixes most of the broken jumps reported by the help compiler. It also amends the missing popup menus for the properties, methods, and events of *TTable*, *TQuery*, *TStoredProc*, and *TSession*.

Delphi 2 Takes First Place in NSTL Comparative Ratings Report

Scotts Valley, CA — In a product evaluation of client/server development tools conducted by National Software Testing Laboratories (NSTL), Delphi Client/Server Suite 2 received the highest overall rating out of four development packages.

Outperforming Powersoft's PowerBuilder Enterprise for Windows, Microsoft's Visual Basic Enterprise Edition, and

Gupta's SQLWindows, Delphi Client/Server Suite 2 was cited as "the fastest, most versatile, and easiest to use of the evaluated products."

NSTL's Software Digest and PC Digest Ratings Reports are used by many industry analysts and editors as a source of objective analysis of personal computer hardware and software, and also as buying guides for

Fortune 1000 companies and government agencies.

"Delphi's performance is head and shoulders above all its competitors," reported NSTL in the March Software Digest Ratings Report on Client/Server Development Tools (Volume 13, Number 3). "(Delphi) offers performance superiority across the board: database and non-database operations, browsing entire database tables, executing queries, or searching for data."

For a copy of the NSTL report, contact the NSTL Testing and Distribution Center at (610) 941-9600.

User-Based Pricing for Borland's InterBase

Scotts Valley, CA — Borland International has outlined its new user-based pricing scheme for InterBase. Under this program, the initial server purchase includes five user licenses. Pricing for additional users is identical across Intel platforms, including NT, NetWare, and SCO. For applications requiring more than five users, additional user license packs are available in single, 10, and 20 packs.

The InterBase 4.0, 5 User Starter Bundle includes a single copy of the media and documentation, and both client and server licenses for five users. This package is purchased for every new server, and is priced at US\$850. It is available for Windows NT, NetWare, and SCO OS 3.0.

The InterBase 4.0 Client/Server Access License Bundle, single user, contains one client and one server license so developers can add

additional users to an existing server. This license is not platform specific. Media and documentation are not included, but client libraries may be copied from the original server bundle diskettes. InterBase 4.0 Client/Server Access License Bundle for a single user is priced at US\$170.

The 10-user pack for InterBase 4.0 Client/Server Access License Bundle is the same as the single user bundle, except it includes 10 client and 10 server access licenses. The 10-user pack is priced at US\$1,350.

InterBase 4.0 Client/Server Access License Bundle, 20-user pack, has 20 client and 20 server access licenses. Its features are identical to the single user bundle in all other respects, and it's priced at US\$2,400.

For more information, visit Borland Online at <http://www.borland.com>.

Borland Announces Java-Enabled InterBase InterClient

Scotts Valley, CA — Borland International Inc. has announced the InterBase InterClient. Written in Java, InterBase InterClient uses InterBase as a SQL database server to anonymously access Web databases.

Based on Borland's work with JavaSoft and the JDBC standard, the InterBase InterClient is designed to deliver the benefits of Internet technologies into corporate organizations. It will contain client and server components for interfacing with the Web.

The InterBase InterClient is expected to enable distributed transaction processing by the Java client and the database server, eliminate updating client database libraries, reduce overall traffic on the network, speed access to server data, provide automatic download of public Internet data to the client, and will not require developers to install client database libraries.

Software Development '96 West Update

San Francisco, CA — Amid a sea of vendors, over 12,000 attendees gathered at Moscone Center in San Francisco, CA last March for the ninth edition of Software Development '96 West. Over 100 new products were previewed at the three-day event, most addressing Internet and

intranet development.

Borland International, Informix, Microsoft, NeXT, Powersoft, Silicon Graphics, Sun Microsystems, and Sybase were among the key vendors adding Internet and intranet development tools to their product groups.

"SD '96 West: Internet Tools Unveiled" continued on page 7

"Borland Announces InterClient" continued on page 7

June 1996



Delphi Wins Jolt Cola Award for Best Development Tool

At Software Development '96 West, Delphi was presented with the "Jolt Cola" award for Technical Excellence. It dominated the competition which included Microsoft's Visual C++ and Visual Basic, and SunSoft's Java.

Borland Ships ReportSmith 3.0 for Windows 95 and Windows NT

Scotts Valley, CA — Borland International Inc. has begun shipping ReportSmith 3.0, a new release of its client/server reporting and query tool for Windows 95 and Windows NT.

ReportSmith 3.0 includes a set of tools for creating columnar, crosstab, and form reports using live data while working with local tables or client/server data-

bases. ReportSmith 3.0 features 32-bit functionality, updated native drivers for accessing 32-bit client libraries from Sybase, Oracle, Informix, Microsoft SQL Server, and Gupta SQLBase, as well as 32-bit ODBC drivers. It also has multitasking capabilities, support for PC and SQL databases, a new API layer for controlling ReportSmith

from other applications, and integration with Delphi 2.

ReportSmith 3.0 is included in Developer and Client/Server Suite versions of Delphi 2. The stand-alone package of ReportSmith 3.0 is available for US\$179.95. Previous owners of ReportSmith may upgrade for US\$129.95.

For more information, call Borland at (800) 233-2444.

SD '96 West: Internet Tools Unveiled (cont.)

Borland demonstrated how to develop Internet technologies in client/server applications with Delphi 2. They also demonstrated how Delphi can use Microsoft's Internet technologies, including the Internet Control Pack, ActiveX, and WinInet and ISAPI application programming interfaces (APIs), to create a Delphi Web browser client application running on Windows 95, and a Delphi Web server application running on Windows NT.

Delphi components that

support Microsoft technologies are now available to customers electronically on Borland Online (<http://www.borland.com>) and Borland's CompuServe forums. Borland also announced plans to add server support for Netscape's NSAPI as well as other popular Web server APIs.

In addition, Borland announced the release of their new C++ Development Suite 5.0 for Windows 95 and Windows NT, and previewed Latte, a RAD development tool for Java developers.

Sun Microsystems unveiled

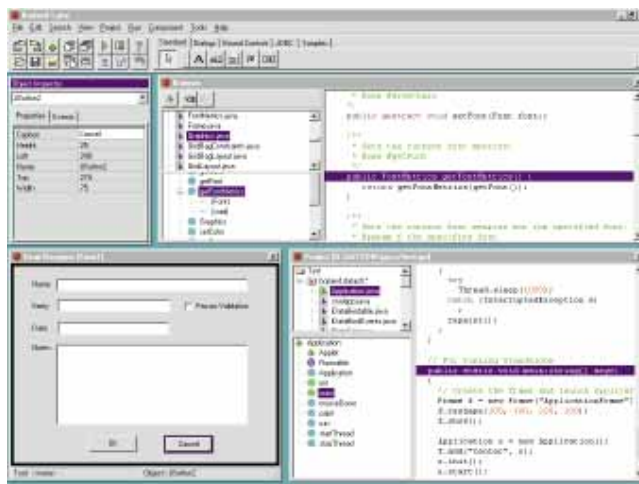
several new Java-enabled development tools, such as Java Products Everywhere (JOE), Internet Workshop, Solstice FireWall-1 2.0, and Solstice Messaging for Solaris (IMAP 4). Call Sun Microsystems at (415) 336-6483 for more information.

Software Development '96 East is scheduled for October 29-31, 1996 at the Washington Convention Center in Washington, DC.

Borland Announces InterClient (cont.)

The InterBase InterClient using JDBC will allow Java applets to be downloaded and run in a Web browser, bypassing traditional time-access methods such as a Common Gateway Interface (CGI). The benefits to IS managers include higher throughput speeds, lower traffic on the 'Net, and enhanced modular application design. InterBase InterClient is also designed to simplify intranet access by presenting a unified interface to all services and resources, without requiring additional software or hardware.

InterBase InterClient has been designed and written for the Windows and UNIX 32-bit platforms. Beta copies are expected to be available the second quarter of 1996.



On its way. Borland's Latte.

Borland's C++ Wins SD '96 Superbowl

San Francisco, CA — Borland International Inc. announced its newly released Borland C++ 5.0 won the C++ Superbowl at Software

Development '96 West. A team of five judges chose

"Borland's C++ Wins SD '96 Superbowl" continued on page 8

June 1996



Borland Announces C++ 5.0 and ObjectScripting Contest

San Francisco, CA — Borland International Inc. has begun shipping its C++ Development Suite 5.0, that combines five tools and Borland C++ 5.0.

Along with Borland C++, the suite features CodeGuard 32/16, PVC5 Version Manager, and InstallShield Express. It also includes the new AppAccelerator for Java, a just-in-time compiler.

Available separately or as part of the Suite, Borland C++ 5.0 includes a 32-bit hosted environment for targeting multiple platforms, including Windows 95, Windows NT, Windows 3.1, and DOS. It also includes ObjectWindows Library (OWL) 5.0, supporting the Windows 95-based common controls and 16-bit emulation of most Windows 95 common controls; Microsoft Foundation

Classes (MFC) compilation support; and Visual Database Tools (VDBT).

Borland C++ supports namespaces, the standard C++ library, OCXes, integrated 32-bit resource editing, and integrated 32-bit debugging. It also features free Java-compatible development tools, (including Sun's Java Development Kit), the Borland Debugger for Java, AppExpert, and color syntax highlighting for Java code.

Borland C++ Development Suite 5.0 is priced at US\$499.95, and Borland C++ 5.0 is priced at US\$349.95.

Upgrades are available on CD-ROM, and include online documentation; diskettes and printed documentation are available separately at an additional charge. For more information, call Borland at (800) 645-4559.

In addition, Borland has announced a new ObjectScripting contest, with the grand prize winner taking home a new laptop computer. Included in Borland C++ Development Suite 5.0 and Borland C++ 5.0, ObjectScripting allows developers to modify and configure Borland's integrated development environment (IDE).

Customers can submit scripts directly to Borland via Borland Online

(<http://www.borland.com>). The grand prize winner will win a laptop computer (estimated at US\$4,000) and be offered a speaking engagement at the 1996 Borland Developers Conference, scheduled for July in Anaheim, CA. Scripts awarded an Honorable Mention will be placed on a CD (or diskette). Visit Borland Online for more details.

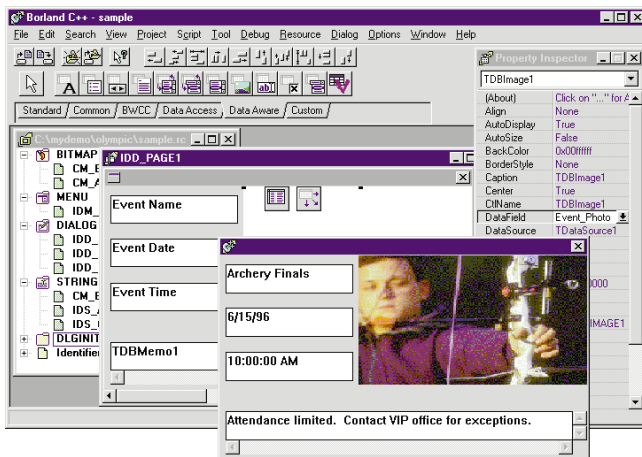
Fung Joins PCSI

Englewood, NJ — Professional Computer Solutions, Inc. (PCSI) announced Joseph C. Fung has been named Director of Technology and Tools.

Previously Fung was a principal of Farpoint Systems Corp., a New York/New Jersey-based consulting firm specializing in developing database applications with Delphi, ObjectPAL, and Visual Basic. The operations of Farpoint Systems are being merged into the operations of PCSI.

Fung currently writes for *Delphi Informant*, *Paradox Informant*, and other publications. He is the author of *Paradox for Windows Essential Power Programming* (Prima, 1995) and a co-author of *Delphi In-Depth* (Osborne/McGraw-Hill, 1996). He is the architect of ScriptView and AppExpert, perennial winners of the *Paradox Informant* Reader's Choice Award.

In addition, Fung has served as a guest lecturer for the Borland Paradox 5.0 World Tour, chaired an Advisory Board for the Borland Developer Conference '95, and served as advisory board member for the Borland International Conference '94.



Borland's C++ Wins SD '96 Superbowl (cont.)

Borland C++ 5.0 as the overall winner over Microsoft's Visual C++. The judges selected Borland C++ as the winner based on the total number of development problems Borland C++ programmers solved in an hour.

Borland recently announced two new versions of its C++ products:

Borland C++ Development Suite 5.0 and Borland C++ 5.0. For complete details, visit Borland Online at <http://www.borland.com>. Software Development is produced by Miller Freeman, Inc. For show information, visit Miller Freeman's Web site at <http://www.mfi.com>.





ON THE COVER

Delphi 2 / Object Pascal



By *Joseph C. Fung*

Do the Strand

An Introduction to Multithreading with Delphi 2

*Do the Strand-o, when you feel low
It's the new way, that's why we say
Do the Strand
— Roxy Music, "Do the Strand"
For Your Pleasure ...*

The Win32 API introduces the concept of multithreading to Windows 95 and Windows NT programs. Using multithreading, you can improve performance by partitioning an application into multiple paths of execution. Although multithreading is very powerful, it adds a whole new level of complexity to an application.

Fortunately, Delphi 2 provides the *TThread* class so that you can work with threads and the VCL (Visual Component Library) in a thread-safe manner.

Spindles and Spools

In Windows 95 and Windows NT (referred to here as *Win32*), an application consists of a *process* and one or more *threads*. A process is an instance of the application and has its own virtual address space, global variables, and operating system resources. By itself, a process does not execute. Instead, each process has a primary *thread* of execution that obtains *time slices* from the CPU. This primary thread executes the code in the application. When this thread terminates, the process ends.

Win32 allows you to create additional threads, or simultaneous paths of execution, within an application. Each of these threads shares the address space of the parent process so they have access to the same global variables and resources. Also, the operating system gives each of these additional threads time slices so they appear to execute concurrently. (In a multiprocessor machine under

Windows NT, each thread may even have its own dedicated CPU.)

Threads are useful because they let you more efficiently use the computer's CPU(s) by partitioning an application across threads, and, also, perform work or lengthy processing in the background while the user continues to interact with the application.

The Win32 API provides facilities for you to create and use additional threads in your application. However, these facilities alone do not permit you to use VCL components in a *thread-safe* manner without corrupting data. ("Thread safe" means that two or more simultaneous processes are designed in such a way that the code being executed in one thread does not interfere with the execution of the other.)

Fortunately, Delphi provides a *TThread* class that encapsulates the threading mechanism and works with the VCL. You can use the *TThread* class along with Win32 API functions to add powerful multithreading capabilities to your application.

When to Use Additional Threads

Here are some situations where multithreading can be very useful:

- Often a user will initiate some work that requires a lengthy process. With Windows 3.x and single-threaded applications, the user interface is unresponsive while the work is going on because the primary (and only) thread cannot process window messages while it's working on some other task. This leaves the user in an unproductive state while waiting for the task to complete. You can prevent this by creating an additional thread to finish the task in the background. This keeps the main user interface very responsive and available to handle user interaction while the work is being completed.
- In an MDI (multiple document interface) application where you have multiple documents represented by child forms, you can give each child form its own thread to perform any lengthy processing. If one of the child forms initiates a lengthy process, the user can move to one of the other child forms and continue working.
- Windows NT supports machines with one or more processors by distributing threads across the set of CPUs. When you run a single-threaded application on a machine with multiple processors, you may not be efficiently using the additional processing power. By partitioning the application using threads, you can balance the load and significantly increase the application's performance.
- With a Windows 3.x application, you frequently used a timer to perform work on a periodic basis by dividing a repetitive or lengthy task into logical units and executing the code when the timer event occurred. Any such code should be examined because it's a natural candidate for multithreading.
- Windows 95 supports both 16- and 32-bit Windows applications. To provide compatibility with 16-bit Windows applications and to maintain a good level of performance, Windows 95 contains a significant portion of the old 16-bit Windows API system code. This code is called by all Windows 3.x applications *and* by Windows 95 applications that make certain Win32 API calls, including those to the GDI and USER modules. This 16-bit code was never designed to be used simultaneously by multiple threads, so access to it is protected. Windows 95 internally uses *Win16Mutex*, a system-wide mechanism for guaranteeing exclusive access to the code.
- Essentially, whenever a thread makes a call to a protected 16-bit API function, any other thread attempting to call any other 16-bit API function waits until the first thread yields to Windows or releases control. If an application doesn't yield, or stops processing window messages for a long time, all these other applications will be tied up.

Unfortunately, this can leave Windows 95 applications unresponsive because many 32-bit API functions, including those that handle the user interface, actually call the 16-bit code base. To minimize the effects of this, you can create additional threads to perform the work in the background while the primary thread is waiting on the user interface.

Creating Threads Using the *TThread* Class

The Delphi 2 *TThread* class encapsulates the multithreading mechanism so that you don't have to rely solely on Win32 API function calls to create threads. More importantly, the *TThread* class provides a facility to work with VCL components in a thread-safe manner, and also supplies a simpler alternative to *thread local storage*, a way to associate data with individual threads.

You can work entirely with threads using the Win32 API. However, you will be working at a lower level and also must supply your own framework for synchronizing access to the VCL. If you do not synchronize your thread's access to VCL properties and methods, you run the risk of corrupting shared data and generating access violations.

The *TThread* class does not completely hide all the details of working with the Win32 API. Instead, it's a thin wrapper that removes the drudgery of VCL synchronization and thread local storage. When you instantiate a *TThread*, the class creates a new thread and stores its handle internally.

The *TThread* class is defined in the CLASSES unit, so be sure to include this unit in your `uses` statement. [Figure 1](#) lists the protected and public properties of the *TThread* class, while [Figure 2](#) lists the protected and public methods.

Using the *TThread* Class

The steps required to add multithreading to your Delphi 2 applications are fairly straightforward. For each new thread, you first derive a new *TThread* class from the base *TThread* class. Then you add code to instantiate the new object. You can define the new thread class by adding the *TThread* type declaration into an existing unit, or by creating a separate unit to hold the *TThread* definition from the Object Repository.

Property	Description
<i>FreeOnTerminate</i>	Specifies whether the VCL automatically destroys the thread upon termination. Default is <i>False</i> .
<i>Handle</i>	The thread handle.
<i>Priority</i>	Lets you set/get the thread's relative priority.
<i>OnTerminate</i>	An event property for the event handler that is executed when the thread terminates.
<i>ReturnValue</i>	The value returned by a thread.
<i>Suspended</i>	A <i>Boolean</i> variable that lets you set/get the thread's suspended state.
<i>Terminated</i>	A read-only <i>Boolean</i> that indicates if the thread should terminate.
<i>ThreadID</i>	The thread's ID.

Figure 1: *TThread* class properties.

Method	Description
Create	Create is the thread constructor.
Destroy	Destroy is the thread destructor.
DoTerminate	DoTerminate calls the OnTerminate event handler, if one exists. DoTerminate executes as part of the thread, as opposed to OnTerminate, which executes as part of the process. It is unusual to override this method.
Execute	Execute is a virtual, abstract method that you override to specify the thread's behavior. Do not call this method; it is automatically called by the constructor.
Resume	Resumes execution of a suspended thread.
Suspend	Suspends execution of a thread.
Synchronize	Synchronizes access to VCL properties or methods.
Terminate	Sets a flag that tells the thread to end.
WaitFor	Suspends execution until a specified thread is signaled.

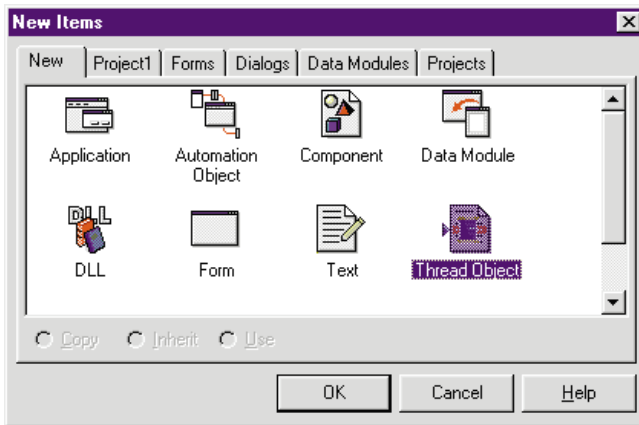


Figure 2 (Top): TThread class methods.
 Figure 3 (Bottom): The New Items dialog box.

To create a new thread unit for an existing project using the Object Repository, select **File | New**. Delphi displays the New Items dialog box, as shown in Figure 3.

When you select the Thread Object icon from this dialog box and click **OK**, the New Thread Object dialog box appears, prompting you to name the new class (see Figure 4).

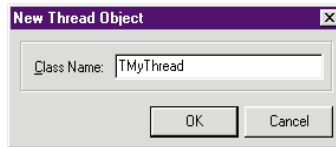


Figure 4: The New Thread Object dialog box.

After entering the name, a new unit appears containing a basic *TThread* class declaration. For example, if you have a new project, and then use the Object Repository to add a second unit for a thread class named *TMyThread*, your screen will resemble Figure 5.

After declaring the new class, you need to define the code that goes into the *TThread*'s member functions. At a minimum, you should place the thread's main worker code into the *TThread*'s *Execute* method, the place reserved for this code. For instance, if your code performs a background calculation, you should place this calculation in *Execute*.

Next, you should add code to store the *TThread* variable in a global or member variable so that you can reference it

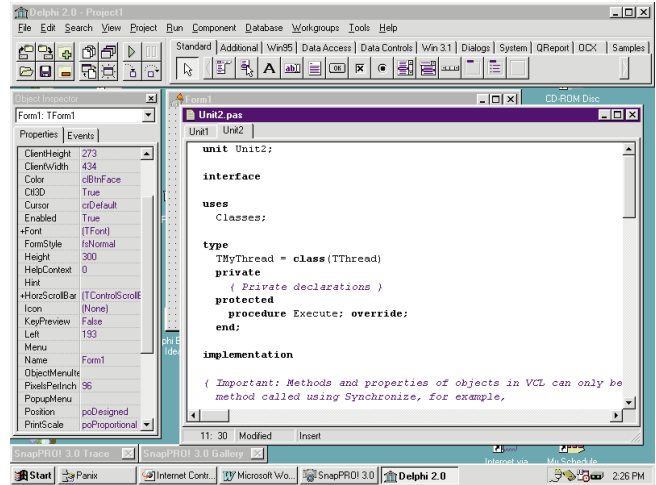


Figure 5: The default unit created for the new *TThread* type.

in your code. Finally, you should add the actual code to create the new *TThread* and work with it. This code may perform functions to suspend or resume execution of the thread or even terminate it. These steps are described in the following sections.

Creating a Thread

To create a new *TThread*, you declare a *TThread* class, a variable or member variable of this class, and then call its *Create* constructor. A constructor of the *TThread* type has the following syntax:

```
Create( Suspended : Boolean );
```

When you call the constructor you pass to it a single Boolean parameter, *Suspended*, that indicates if the thread should be created in a suspended state. Normally, you would pass a value of *False* so that the thread begins execution immediately.

Assuming that you have declared an object named *MyThread* of the type *TThread*, the following code segment would create and execute it:

```
MyThread.Create(False);
```

Placing the Code for the Background Work

The *TThread* class defines a virtual abstract method, *Execute*, that you override to implement the code for your background task. The constructor for *TThread* calls *Execute* immediately after creating the thread so you don't need to run it explicitly. When *Execute* is finished running, the thread terminates and sets a return value that you can query using the *ReturnValue* property.

The code that you place into *Execute* may perform some background task such as a lengthy calculation or query. If the code resides in a loop, the loop should contain some predefined condition that breaks out when it's *True*, such as when a certain number of iterations have passed. You should also place a test inside the loop to see if the *Terminated* property is *True*. If so, your code should immediately exit the

ON THE COVER

Execute method. (The reason why you need this additional piece of code will become clearer later.) The following is an example of the implementation code for an *Execute* method:

```
procedure TMyThread.Execute;
begin
  while MoreWork do begin
    CalculateSpreadsheet; { Perform some work }
    if Terminated then
      Exit;
  end;
end;
```

Using VCL Components from within a Thread

There are special considerations to make if you plan to access any of the VCL components from within your thread. This is because the VCL is not implicitly thread-safe. In other words, any unprotected access of a component from within a thread may potentially corrupt data or cause an access violation. This includes calling any methods, and/or reading or writing from any of a component's properties.

Access to the VCL must be protected because multiple threads may try to work with the same component simultaneously, possibly leading to data corruption or other unwanted side effects. Also, for performance and efficiency, the VCL caches Windows GDI (Graphic Device Interface) objects that handle screen painting.

Windows does not allow two or more threads to have simultaneous access to a GDI object. This may happen because the VCL stores and reuses GDI objects — in this case, you can get an access violation with multithreaded access.

Using *Synchronize* with the VCL. To address the issue of thread-safe VCL access, the *TThread* class provides a *Synchronize* method. If your thread needs to access a VCL component, the thread should not do it directly. Instead, you should put this code into a separate method, and then execute this method by calling *Synchronize*.

Synchronize has the following syntax:

```
Synchronize( Method : TThreadMethod );
```

When you call *Synchronize*, you send a method to it by value. *Synchronize* then executes this method in a thread-safe manner. In most cases, you will want the method that you pass to *Synchronize* to be a member of the same derived Thread object. Doing so provides the method with access to the object's private interface and variables.

The code fragment in [Figure 6](#) shows how your code might appear, and demonstrates how you can update a component's property from within a thread. A method named *UpdateDisplay* is implemented for the *TMyThread* class. Within this method, a Label component's *Caption* property is updated.

```
// This method accesses the caption of a Label component.
procedure TMyThread.UpdateDisplay
begin
  MyForm.ProgressBar.Caption :=
    'Up to ' + IntToStr(Count);
end;

procedure TMyThread.Execute;
begin
  while WorkToDo do begin
    DoSomeWorkMethod;

    Synchronize(UpdateDisplay);

    if Terminated then
      Exit;
  end;
end;
```

Figure 6: The *UpdateDisplay* method for the *TMyThread* object.

Within the *Execute* method for the Thread object, a *while* loop continues to execute so long as a variable named *WorkToDo* evaluates to *True*. Within this loop, a call is made to a method named *DoSomeWork*. Next, this thread needs to update the display of the Label component. To do this, it calls *Synchronize* and passes *UpdateDisplay* as an argument.

Suspending and Resuming Threads

The *TThread* class provides the *Suspend* and *Resume* methods to suspend and resume execution of a thread. When a thread is suspended, it's idle and no CPU cycles are given to it.

To suspend a thread, just call the *TThread* object's *Suspend* method from another thread (primary or otherwise). A thread can even suspend itself by calling its own *Suspend* method. You shouldn't do this, however, unless you expect another thread to reawaken the suspended thread. To resume execution, just call the *TThread*'s *Resume* method.

Thread Priority

The *TThread* class defines the *Priority* property so that you can dynamically change the *priority* of a thread. A thread's priority determines when and how often it's scheduled for execution by the operating system. By setting one thread's priority higher than another, you give it more CPU time. A thread's priority level ranges from 0 to 31, with 31 being the highest level. By default, a newly created thread adopts the same priority level as its parent process.

Setting a Thread's Priority. A thread's priority is measured *relative* to the priority of its process. The VCL defines seven priority levels that you can use to set the thread's relative priority. They are part of the *TThreadPriority* enumerated type.

The five main priority levels are:

- 1) *tpLowest*
- 2) *tpLower*
- 3) *tpNormal*
- 4) *tpHigher*
- 5) *tpHighest*

A thread with a relative priority level of *tpNormal* has a priority level equal to its process. With priority levels of *tpHigher* and *tpHighest*, the thread's relative priority is 1 higher and 2 higher than its process, respectively. With priority levels of *tpLowest* and *tpLower*, the thread's relative priority is 2 lower and 1 lower than its process.

There are also two special priority levels: *tpIdle* and *tpTimeCritical*. If a thread's priority is *tpIdle*, its priority is always set to 1, unless its process is 24 or higher (real-time). In this case, the thread's priority level is set to 16. If the thread's priority is *tpTimeCritical*, its priority is always set to 15, unless its process is 24 or higher. In this case, the thread's priority is set to 31.

Terminating a Thread

There are two recommended ways to terminate a thread when using the *TThread* class: you can call *Exit* from the *TThread*'s *Execute* method, or you can call the *TThread*'s *Terminate* method.

Calling *Exit* to Terminate a Thread. The bulk of the code responsible for performing your background task resides in the *Execute* method. When this code finishes running, it should exit the *Execute* method. This can happen implicitly as the last action of the method or if you call *Exit*. When *Execute* finishes, the thread terminates and the *TThread* class takes care of any cleanup.

Calling *Terminate* from Another Thread. A second way to terminate a thread is to call the thread's *Terminate* method. Typically, you do this when you want to terminate a specific thread from the main thread or from another thread.

Terminate does not actually terminate the thread, but sets the *Terminated* property to *True*. It's then up to your code to check the value of *Terminated* from inside the thread's methods, and to act appropriately to exit the *Execute* method. If you do not have any code that does this, calling *Terminate* is ineffective.

The following code segment illustrates how this code might appear:

```
procedure TMyThread.Execute;
begin
  while SomeCondition do begin
    { Do some work here }
    DoSomeWork;
    if Terminated then
      Exit;
  end;
end;
```

After performing a unit of work, the *if* statement checks the value of *Terminated*, and exits the method if it's *True*.

A Multithreading Example: Using Child Forms and Worker Threads

In an MDI application, or in an application that displays a form for each task, multithreading may be appropriate. With multithreading, you can let each form have its own thread. This way the user can initiate a background task in one form, and then continue to interact with other forms while the background work is progressing.

In such a scenario, the primary thread handles all the work of managing the user interface and responding to window messages. Each time the user initiates a new task, the primary thread creates a new form and gives it its own "worker" thread.

This next example illustrates how you can open new forms, each owning its own worker thread. The lengthy task in this example is represented by iterating through a long loop and updating a track bar to show the progress.

When you run the example in project WORK.DPR, the Worker Thread Examples form appears. Each time you press the **New Thread** button on this form, a new child form is created with its own thread, and immediately begins its work. Each new form has a unique caption that identifies the "worker" (Worker 1, Worker 2, etc.).

Each form also contains a **Suspend/Resume** button that allows you to suspend and resume the thread. Finally, if you close the form before the thread is finished, the form closes and the thread terminates.

Figure 7 depicts the main form surrounded by two worker forms. Listing One (on page 14) shows the .PAS file for this example.

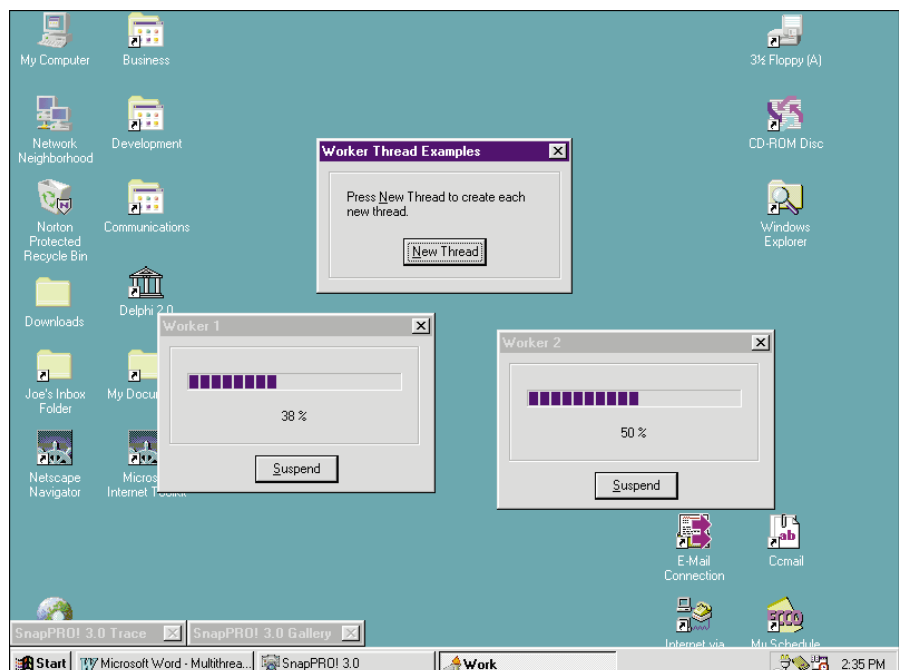


Figure 7: The demonstration project at work, showing the main form and two of its worker forms.

Conclusion

Multithreading is a welcome feature to Win32 (Windows 95 and Windows NT) applications, but can add complexity and unique problems of its own. The *TThread* class encapsulates the threading process, providing a facility for working with the VCL in a thread-safe manner, and offering a way to associate local data with each thread. This lets you pursue multithreading in earnest and add power and flexibility to your business applications. ▲

Portions of this article abridged and reprinted by permission from "Using Multithreading," *Delphi In-Depth* by Cary Jensen, Loy Anderson, Joseph C. Fung, Ann Lynnworth, Mark Ostroff, Martin Rudy, and Robert Vivrette, published by Osborne/McGraw-Hill Companies, Inc. Copyright 1996 by Osborne/McGraw-Hill Companies, Inc. ISBN: 0-07-882211-4.

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\JUNE\96\DI9606JF.

Joseph C. Fung is Director of Technology and Tools at PCSI, a leading client/server and Internet/Intranet consulting and development firm. He writes for *Delphi Informant* and *Databased Advisor*, and is the co-author of *Delphi In-Depth* and the author of *Paradox for Windows Essential Power Programming*. Mr Fung is the architect of AppExpert and ScriptView, perennial winners of the *Databased Advisor* Reader's Choice Award and *Paradox Informant* Reader's Choice Award. Recently, Mr Fung chaired an Advisory Board for the Borland Developer Conference.

PCSI, Professional Computer Solutions, Inc., is a national leader in developing SQL applications using Delphi, Visual Basic, and Access, and using Microsoft SQL Server, Sybase, and Oracle server technologies. PCSI is a Borland Connections Partner and Microsoft Solution Provider at the *Partner* level. The main number for PCSI is (201) 816-8002.

Begin Listing One — The WORKTHD.PAS file

```
unit workthd;

interface

uses
  Classes, Forms;

type
  TWorkerThread = class(TThread)
  private
    { Private declarations }
    MoreWork: Boolean;
    FOwnerForm: TForm;
  protected
    procedure Execute; override;
  public
    property OwnerForm:
      TForm read FOwnerForm write FOwnerForm;
    procedure UpdateDisplay;
  end;

implementation

uses WorkForm, Windows, SysUtils;

{ TWorkerThread }

procedure TWorkerThread.UpdateDisplay;
begin
  with (FOwnerForm as TWorkerForm) do begin
    with WorkProgressBar do

      if Position < Max then
        begin
          Position := Position + 1;
          WorkProgressLabel.Caption :=
            IntToStr(Position) + ' %';
          if Position = Max then
            begin
              MoreWork := False;
              WorkerButton.Caption := 'Done';
            end;
          end;
        end;
      end;

procedure TWorkerThread.Execute;
begin
  MoreWork := True;
  while MoreWork do begin
    { Do some work here }
    // Simulate work by putting thread to sleep 100 ms.
    // Actually, we're just putting the thread to sleep.
    Sleep(100);
    if not Terminated then
      Synchronize(UpdateDisplay)
    else
      Exit;
    end;
  end;
end.

End Listing One
```





INFORMANT SPOTLIGHT

InterBase / Paradox

By *Kevin Bluck*

InterBase vs. Paradox

Which Is Best for Your Application?

If you're like me, when you first bought Delphi you didn't pay much attention to the Local InterBase Server bundled in the package. Most likely, even if you were learning to build database applications, you amused yourself for weeks using nothing but Paradox tables. In fact, like me, you may have come from a Paradox background.

The products are so strongly associated that many people have had difficulty making the distinction. And rumors that Delphi would replace Paradox compounded the problem. Although it should be clear by now that this isn't the case, most developers still use Paradox tables for their Delphi database development. In short, I doubt many bought Delphi to get their hands on the Local InterBase Server.

Piqued Curiosity

But, programmers are a naturally curious lot, and many of them eventually began poking at Local InterBase. At first glance, it doesn't seem that different from Paradox. You access it using an alias, just as Paradox, and although they have different names, the field types are similar. You can even create InterBase tables using the Database Desktop. You use the same *TTable* and *TQuery* components that are used for Paradox tables. In short, the Borland Database Engine (BDE) interface creates a convincing illusion that InterBase tables behave like Paradox tables.

Except InterBase is supposed to be better somehow. It's an industrial-strength RDBMS, so it must be bigger and faster than Paradox, right?

For many developers, though, disillusionment soon sets in. After creating InterBase tables with the Database Desktop, they discover they can't casually change field definitions by simply restructuring as they can with Paradox tables. All searches and indexes are case-sensitive, unlike Paradox. Defining primary and foreign keys seems easy, but changing them seems near-

ly impossible. Worst of all, some operations are slower with InterBase than with Paradox. It quickly becomes apparent that InterBase isn't automatically better than Paradox.

The idea that InterBase isn't *better* than Paradox is absolutely true. At least, it's not always better. The two products are significantly different, and are intended to serve in different situations. The only thing they really have in common is that they both store data in tables. They diverge rapidly from that point.

Each is a system with strengths and weaknesses. The trick is deciding which is appropriate for a particular application. And once made, that decision fundamentally affects the subsequent development effort.

Paradox Is File Based

Paradox is a file-based database system. The data files contain data records that have a definite order. In other words, record number 106 will always be the same record until it's physically moved within the file, perhaps as a result of a sorting operation. Even more importantly, it will always follow record 105 and precede record 107, until that order is explicitly changed. This allows the records to be easily navigated by a cursor, since it's possible to identify a record by its position within a table without having to reference the data it contains.

This explicit physical ordering of records has some advantages. Moving back and forth through the data file is a simple matter, and the records are easily refreshed when the cur-

sor arrives at them. The concept of *browsing* is convenient for users and developers. It allows records to be handled one at a time, in a predictable order. This navigational behavior is one of the major Paradox concepts that is difficult to transplant into the InterBase world, and many Paradox developers have a difficult time making the transition.

InterBase Is Set Based

InterBase is a true, set-based relational database system. Tables aren't stored in individual files. More importantly, the records are not ordered. Mathematically speaking, sets are unordered. Order is "discovered" only when the set is physically represented, such as when querying a database. You can't count on the same record being record number 105 twice in a row, unless you explicitly impose a certain ordering on the query. Since you can't positively identify a record by its position within the table, you must refer to values within the record. Therefore, to positively identify a record, at least one field or combination of fields must contain unique values for each record. This is what's known as a *primary key*.

It's possible for more than one field or combination to provide unique values, in which case they form a pool of *candidate keys*. Since it's important for a database management system to identify individual records conclusively, the existence of a primary key is crucial. A table that has a unique primary key is called an *R-table*, and all data sets must be R-tables for the relational model to work.

The advantage of this set-based conceptualization of data is that sets and the operations that can be performed on them have the property of *closure*. This means that when you perform a set operation on a set, it always produces another set, which can then have another operation performed on it, producing another set, ad infinitum. This is a powerful logical concept, and if it's properly implemented, will remove the physical characteristics of storage from consideration in the application's design. This is the foundation of the relational model.

Cursor-based systems such as Paradox allow you to work only on one record at a time, repeating an operation when you want to process groups of records. Set-based systems allow you to manipulate a set of data as if it were a single entity, all components of which will share the same fate. This has the potential of increasing the simplicity of application design and vastly improving data integrity.

Strangely enough, the ANSI SQL specification, which purports to be a relational standard, doesn't require a primary key to be defined for each table. This means records within the data sets that are produced by SQL queries don't have to be uniquely identified by value.

Even worse, the results of SQL queries, even if they come from proper R-tables, don't have to be R-tables themselves. Because identification by value is the only reliable method of identifying individual records within a set, the primary key is

a fundamental cornerstone of the relational model, and its absence causes significant problems.

You will quickly discover how this rather incomprehensible omission by the ANSI committee will cause you problems while using Delphi and the BDE to develop SQL applications — unless you exercise the discipline to ensure that your queries always return correct R-tables. (We'll comprehensively cover the full implications of this in a future article.)

Physical Design and Its Impact on Speed

Paradox is a client-based system where the data is completely managed by the individual clients. Whenever data must be read or manipulated in any way, it must be transported to the Paradox application. Each application handles all processing itself. If multiple users are accessing data simultaneously on a network, each user's application transports the data it requires back to the user's machine.

Each instance of Paradox has no regard for the others. If an instance of Paradox needs to guarantee the stability of data for any reason, it must forcibly prohibit other instances from changing the data through a locking scheme.

When a Paradox application needs to search a table, the necessary raw data and indexes must be loaded into the client machine's memory, the physical activity of the search conducted, results produced, and the now-unneeded raw data discarded. This activity is repeated for every operation on every client. The file server holding the data files does absolutely nothing more than send the requested raw data over the network to the client machines, making absolutely no attempt to process the data. In essence, it acts as a remote hard disk.

What this usually means is that Paradox is speedy on a local hard drive, but slows down dramatically over a network. Network bandwidth can quickly become clogged with large volumes of unprocessed data being shipped repeatedly to the client machines. Even in a fairly small network environment, performance degrades rapidly as new users arrive.

InterBase is a server-based system. Instead of different processes physically manipulating the stored data, only one central process running on the server machine has direct access to the data. All the client applications make polite requests to the server process, which does the actual processing entirely on the server machine while the client waits.

When the server finishes, it passes only the result back to the client, which then goes on about its business. The most direct impact of this scheme is that the network doesn't need to be clogged with large volumes of redundant raw data being sent to the clients. Also, the often complex data processing tasks can be delegated from usually less powerful client machines to the usually more powerful server machine.

Notice that everything about InterBase's design implies a multi-user environment. InterBase was designed from the ground up as a multi-user system. Paradox, on the other hand, was designed primarily for a single user, with support for multi-user capabilities added on.

InterBase Isn't for Browsing

As mentioned earlier, despite the fact that InterBase is a true RDBMS, it performs some operations more slowly than does Paradox. For the most part, these are browsing operations — natural for Paradox — that the BDE is attempting to emulate with InterBase. The problem is most obvious if you have a rather large table, perhaps 100,000 records, attached to a *TTable*. If you call the *TTable*'s *Last* method, attempting to move the record pointer to the last record, the performance of the two systems will vary wildly.

The Paradox table's cursor will move almost instantly because it knows the physical location of the last record in the data file and can simply increment the file pointer to that location and retrieve the data. The InterBase table, however, is going to present some problems. First, because the data is unordered, there is some question about what constitutes "lastness." It must impose an order before it can decide which record is "last." By default, the last record will be considered to be the record whose primary key has the greatest value.

It's now necessary to determine the value of the maximum primary key. Once that value has been determined, the record matching that value can be retrieved. However, if the last few records must be retrieved, as would be the case in a grid display, the process has to be repeated.

Now, InterBase must find the greatest primary key value that is less than the maximum primary key value. For the third, it needs the next-to-next-greatest primary key value, and so forth until the required number of records is found. An operation that is a piece of cake for Paradox is a major pain for InterBase. In general, attempting to navigate backwards through SQL tables is inefficient. The BDE buffers groups of records to minimize this problem, but if you go backwards far enough, you'll have to pause for significant amounts of time in large tables.

Locking

Whenever two users attempt to change the same piece of data, problems can arise that threaten the integrity of the database. Paradox and InterBase address this problem in different ways.

Because Paradox has no direct knowledge of what other Paradox processes are doing, it uses a *pessimistic* locking scheme. As soon as a user attempts to change a record, the record is locked. No other user can change the record until the first user finishes editing, or cancels the changes. This is good, in that a user who successfully obtains a lock can definitely complete the editing operation. It's also bad, because a user can monopolize a record indefinitely. This would be a problem, for example, in a travel reservation system. An inde-

cise traveler can lock down a seat for a long time, causing others to believe the seat is taken, only to decide not to take the seat after all.

InterBase, on the other hand, handles all data manipulation by itself using an *optimistic* concurrency scheme. It's therefore in a better position to manage contention among users without resorting to Draconian locking tactics. The fact that one user may be in the process of changing a record will not prevent other users from also attempting to change it. Whenever a user begins to change a record, InterBase saves a copy of the original record. The user goes about his or her business, but other users are not prohibited from accessing the same record in any way.

When the editing user posts the changes, the original copy is compared to the current record. If the versions are different (most likely because another user beat them to the punch) the user's changes are rejected.

What this means is that individual users can't lock others out of records. In the above travel reservation scenario, the first traveler to commit a reservation gets the seat, even if several were considering it simultaneously. The downside, of course, is that the changes are not rejected until the record is posted, after the work of editing has been done. This can be mitigated, however, by refreshing the changed fields and resubmitting. *TTable* and *TQuery* do this transparently, for example, using the *UpdateMode* property. The post can be resubmitted if all fields match the originals, only the changed fields match, or only the primary key matches, whichever is appropriate to ensure integrity.

The two approaches reflect a basic difference in philosophy. The Paradox pessimistic model assumes that collisions will be common, and gives strong control of the record to whoever seizes it first. The optimistic InterBase model assumes that collisions will be rare, and maximizes the ability of users to share data without interfering with one another, while still maintaining integrity.

An important benefit of the InterBase model is that one user who wants to see a stable data set, perhaps to generate a series of reports that need to reflect the same snapshot of data, will not interfere with other users who want to change the same data. In Paradox, the report generator would have to place a write lock on the table to guarantee the data will not change, and nobody can update data in that table as long as the lock exists.

In InterBase, old versions of records are retained as long as a user is interested in them — so, other users do not have to be prevented from updating the records. This means that in InterBase, readers never prevent writers from succeeding, nor do writers compromise the results of readers. InterBase is the only SQL database that does this so transparently. When InterBase proponents are asked what the advantages of InterBase are, this record versioning is usually the first thing they mention.

Transaction Processing

As you'll recall, a basic premise of the set-based model is that sets of data can be treated as individual entities, regardless of the set's specific contents. Transaction processing is an extension of this idea. A *transaction* is a group of operations that must either all succeed or all fail. It's never acceptable only for some to succeed.

For example, your automated teller machine (ATM) performs database transactions. Whenever you withdraw cash, two operations must be performed for the bank to properly account for its assets: The balance of your account must be reduced, and the balance of cash on hand must also be reduced by the same amount. Obviously, the preferred situation is for both operations to succeed, but if the power goes off in the middle of the operation, it's absolutely not acceptable for one account to be updated and not the other — both operations must fail to maintain the proper accounting.

Transaction processing allows this to happen. The operations in a transaction are not permanent until the whole transaction is committed. Until that time, it may be rolled back to the starting point. A rollback can be explicitly triggered using the *Rollback* method, or it can occur automatically when a system failure occurs.

InterBase fully supports transactions. In fact, all operations occur within the context of a transaction. In the absence of explicit programmer control, the BDE automatically “wraps” every operation in its own transaction. For example, every time you post a record, a transaction is started and committed immediately after the post. Using the *TDatabase* component, you can explicitly control a single transaction and have it encompass as many operations as you like.

However, the BDE does not fully support InterBase's transaction capabilities. *TDatabase* methods can only be used against a single InterBase database, and only one transaction may exist at a time for each BDE alias. InterBase itself supports multiple simultaneous transactions per connection, and a transaction can also encompass more than one database, but the BDE doesn't surface these abilities. You'll have to make calls to the InterBase API to use these features.

Paradox doesn't support transactions. Whenever a record is posted, the changes are permanently written to the table. It requires another edit to manually change it back if a rollback is desired. In addition, the system will not guarantee that a group of operations will all either succeed or all fail. It's possible to *simulate* some of this capability through some tricky programming and temporary tables, but eventually the records must be modified one at a time in a batch, which leaves a window for failure. And there's no way you can program a Paradox application to recover from a system failure such as a power outage or disk crash.

Triggers and Procedures

A stored procedure is a piece of code that is stored in the database along with the data. It allows the server to perform complex manipulation of data entirely on the server.

The main advantages are that even more complex processing can be delegated to the server, and any number of different client applications can call the same procedures. If the procedure is modified on the server, none of the applications has to be rewritten as long as the procedure's interface remains the same.

A trigger is short for triggered procedure. It's a stored procedure that is not explicitly called by an application, but is executed in response to a data action, such as inserting a new record. Triggers allow you to perform extremely complex data validation, and are guaranteed to execute within the same transaction that performed the triggering operation. If any operation fails, all changes made by triggers associated with that operation are also rolled back.

InterBase supports stored procedures that return result sets, which can be treated exactly as read-only tables, as well as triggers that simply perform data transformations and don't return any results. It supports essentially unlimited numbers of triggers for each table, which can occur before or after inserts, updates, and deletes. If more than one trigger is associated with an operation, their order of execution can be specified. Triggers can make changes that execute other triggers, in a chain-reaction fashion, but all such cascading actions are still contained within a single transaction.

Paradox does not support either of these concepts. All data processing must be done at the client. Each application must contain the same code to maintain the data, and each application must be modified if the method of handling data must be changed.

There is no guarantee that an operation will be completed once it's started. For example, cascading the delete of a master record to its detail records can fail in midstream, leaving details undeleted. If this cascade were implemented as a trigger in InterBase, either all or none of the records would be deleted. Furthermore, the code to cascade the delete must be written into each different application that uses the Paradox data. Using InterBase, it only needs to be written once in a trigger. The application simply deletes the master record and InterBase takes care of deleting the detail records.

Making the Choice

Choosing between Paradox and InterBase has important implications for your project. Accordingly, it's essential to know what is important for your situation. [Figure 1](#) shows some general principles that can help you make a decision.

These are guidelines, not rules. Most of them assume a network is involved. If you are contemplating a single-user system, Paradox is usually the best choice. The Local InterBase Server can be deployed as a single-user system, but without concurrency issues, many InterBase features don't apply. If the ability to browse data is important, Paradox is also a good choice.

Paradox Is Better When ...	InterBase Is Better When ...
Primarily used by fewer than 10 concurrent users.	Primarily used by more than 10 concurrent users.
Data and data structures must easily be modified by end-users.	Data should be centrally maintained and protected.
Client machines are comparable in power to the server.	Server is much more powerful than the clients.
Plenty of network bandwidth.	Network is loaded.
Speed and convenience are more important than integrity.	Data integrity is crucial.
Little network and SQL expertise is available.	Skilled network and database administrators are available.
Only one application will routinely access the data.	Several applications may access the data.
Applications will be responsible for maintaining data integrity.	Database will enforce data integrity independently of applications.
Small to moderate amounts of data (< 100MB).	Moderate to large amounts of data (> 100MB).

Figure 1: Comparing Paradox and InterBase.

Conclusion

It's important to remember that Paradox and InterBase are substantially different systems, even though the BDE attempts to make them look similar. It's a seductive, but dangerous idea that converting an application from one to the other involves nothing more than changing an alias. They require significantly different design concepts, and a design that is efficient with one will likely not be with the other. Selecting which system to use is a crucial decision that must be made at the start of a project. It will profoundly impact your subsequent development effort. **Δ**

Kevin J. Bluck is an independent consultant based in Sacramento, CA. His specialty is database development with Borland products such as Delphi, Paradox, Borland C++, and InterBase. Kevin can be reached on CompuServe at 103447,3510, or on the Internet at kevinbluck@aol.com.





DB NAVIGATOR

Delphi 2 / Object Pascal



By Cary Jensen, Ph.D.

Strainless Filtering

Delphi 2's New and Powerful Filtering Capabilities

A common need in database applications is the ability to display or manipulate a subset of records. As described previously in this column, you can do this in Delphi 1 using ranges, linked DataSets, and SQL queries. Delphi 2 now offers you one more choice: filters. This month's DBNavigator takes a look at this new feature.

The Delphi 2 filter capability is available through the three descendants of *TDBDataSet*: *TTable*, *TQuery*, and *TStoredProc*. And there are four properties involved: *Filter*, *Filtered*, *FilterOptions*, and *OnFilterRecord* (an event property). Using these properties, you can instruct a DataSet to display fewer than all of its records. What makes filtering special, as opposed to using ranges and linked DataSets (the only other techniques that work with *all TDataSet* descendants), is that filters do not require an index. As a result, filters are more widely applicable than the other record-limiting techniques.

Although the primary application of filters is to limit the records displayed in a DataSet, there is an additional capability offered by filters that is unmatched by ranges, linked DataSets, and SQL queries. Specifically, filters permit you to display an entire database and still navigate only the records that match the filter. For example, you can set a filter to match records based on the state of California, but still display all records in the DataSet. Then, use the four new *DataSet* methods that let you move to the first, last, next, and previous record where the Customer resides in California, skipping over any records in between. The non-California records will still appear among the California records, but *Next* (for instance) will move to the next California record.

This article will demonstrate three applications of filters. The first application makes use of properties alone. The second demon-

strates the use of the new *OnFilterRecord* event handler. Finally, the use of filters to navigate records is demonstrated.

Filtering with Properties

There are two properties that — used together — produce a filtered DataSet. The properties are *Filter* and *Filtered*. *Filtered* is a Boolean property that you use to turn the filter on and off. If you want to filter a record, set *Filtered* to *True*; otherwise set *Filtered* to *False*.

When *Filtered* is set to *True*, the DataSet uses the value of the *Filter* property to identify which records to display. You assign to this property a string that contains at least one comparison operation involving at least one field in the DataSet. You can use any comparison operators, including =, >, <, >=, <=, and <>. As long as the field name doesn't include spaces, you can include the field name directly in the comparison without delimiters. For example, if your DataSet includes a field named *Country*, you can set the *Filter* property to the following value to filter only for customers in the US:

```
Country = 'US'
```

If the field named includes a space (field names in Paradox tables can include spaces), you must enclose the named field in brackets. For example, if your DataSet is a Paradox table, and you want to display only those records where the customer's last name is Jones, and the field name in the table is last

name, you would assign the following value to the *Filtered* property:

```
[last name] = 'Jones'
```

These examples have demonstrated only simple expressions. However, complex expressions can also be used. Specifically, you can combine two or more comparisons using the **and**, **or**, and **not** logical operators. In addition, more than one field can be involved in the comparison. For example, you can use the following *Filter* property value to limit records to those where the *City* field is San Francisco, and the last name is Martinez:

```
City = 'San Francisco' and [last name] = 'Martinez'
```

However, a value assigned to the *Filter* property does not automatically mean that records will be filtered. Only when the *Filtered* property is set to *True* does the *Filter* property produce a filtered DataSet. Furthermore, if no value appears in the *Filter* property, setting *Filtered* to *True* has no effect.

The use of the *Filter* and *Filtered* properties is demonstrated in the project FILTER.DPR, whose main form is shown in **Figure 1**. This project contains an Edit component, in which the user can type a filter string. The *OnExit* event handler for this property assigns the *Text* property of this Edit component to the *Filter* property of the DataSet. **Figure 2** shows this form after a filter string has been entered. Notice that only those records that match the *Filter* are displayed in the form's DBGrid.

The form in **Figures 1** and **2** also contains two CheckBox components. These components control a third filter-related property of DataSets, the *FilterOptions* property. This property is a set property, with two values: *foCaseInsensitive* and *foNoPartialMatch*. When *foCaseInsensitive* is included in the set, the filter is not case-sensitive. When *foNoPartialMatch* is included in the set, partial matches are also included in the filtered DataSet. Note, however, in my admittedly limited testing, I could find no affect of the *foNoPartialMatch* value in this set. In all my tests, all comparisons were complete. The following code, attached to the *OnChange* event handler for the **Case Sensitive** check box, demonstrates the run-time manipulation of the *FilterOptions* property:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if not CheckBox1.Checked then
    Table1.FilterOptions :=
      Table1.FilterOptions + [foCaseInsensitive]
  else
    Table1.FilterOptions :=
      Table1.FilterOptions - [foCaseInsensitive];
end;
```

Using the *OnFilterRecord* Event Handler

There is another — somewhat more flexible — way to define a filter. Instead of using the *Filter* property, you can attach code to the *OnFilterRecord* event handler for the DataSet. This event handler is passed by reference a Boolean property, named *Accept*, that you use to indicate whether the current

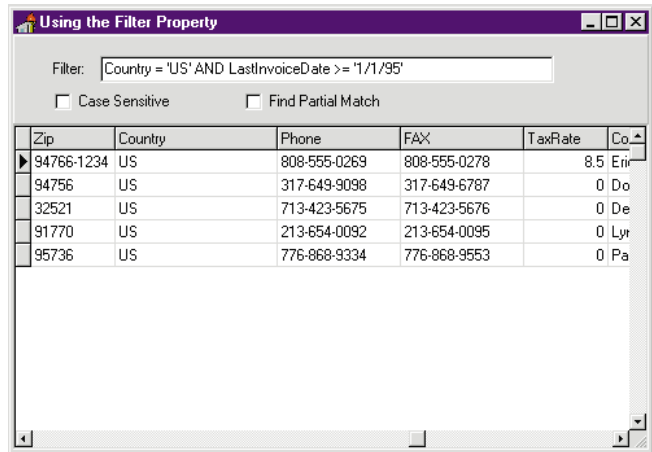
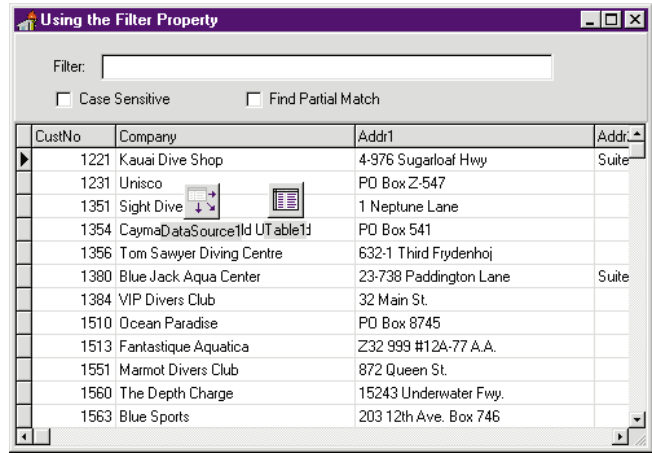


Figure 1 (Top): The main form for the FILTER.DPR project. This project permits the user to enter a filter statement at run time. **Figure 2 (Bottom):** The Customer table is being filtered. Only those companies that are located in the US, and whose last invoice date is later than or equal to January 1st, 1995, are displayed.

record should be included in the DataSet. You can perform almost any test you can imagine with this event handler. If, based on this test, you wish to exclude the current record from the DataSet, you set the value of the *Accept* parameter to *False*. Note that this parameter is *True* by default.

The use of *OnFilterRecord* is demonstrated in the project ONFILT.DPR (shown in **Figure 3**). When the user marks the check box labeled **Show orders for the following year only**, the *Filtered* property of the DataSet (*Table1* in *DataModule2*) is set to *True*. The year to be filtered is controlled by an UpDown control. In this case, the filtering is being performed in the *OnFilterRecord* event handler for the DataSet:

```
procedure TDataModule2.Table2FilterRecord(
  DataSet: TDataSet; var Accept: Boolean);
begin
  if (Table2.FieldByName('SaleDate').Value >=
    StrToDate('1/1/' +
      IntToStr(Form1.UpDown1.Position))) and
    (Table2.FieldByName('SaleDate').Value <=
      StrToDate('12/31/' +
        IntToStr(Form1.UpDown1.Position))) then
    Accept := True
  else
    Accept := False;
end;
```

This code uses the *Position* property of the UpDown control (on *Form1*) to accept only those records within the specified year. If the current record passes this test, the value of *Accept* is set to *False*.

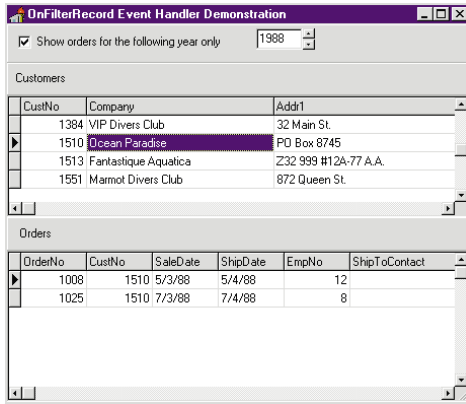


Figure 3: The ONFILTER.DPR project demonstrates the use of *OnFilterRecord*.

It's important to note that if you set the *Filter* property to a filter string, and also assign code to the *OnFilterRecord* property, both will be applied when *Filtered* is *True*. That is, only those records that match the filter string and those that are accepted by the event handler will appear in the DataSet.

Navigating Using a Filter

Whether you have set *Filtered* to *True* or not, you can still use a filter for the purpose of navigating selected records. For example, although you may want to view all records in a database, you may want to quickly move between records that meet specific criteria. For example, you may want to be able to quickly navigate between those records where an unpaid account balance exists.

In Delphi 2 the DataSet objects surface four methods for navigating using a filter. These methods are *FindFirst*, *FindLast*, *FindNext*, and *FindPrior*. When you execute one of these methods, the DataSet will locate the requested record based on the current *Filter* property or *OnFilterRecord* event handler. This navigation, however, does not require that the *Filtered* property be set to *True*.

The use of these special properties is demonstrated in the project FILTNAV.DPR, shown in **Figure 4**. As you navigate the project, it is constantly setting the filter to the state for the current record.

This is done by attaching the following code to the *OnDataChange* event handler for the DataSource that points to the CUSTOMER.DB table:

```

procedure TForm1.DataSource1DataChange(Sender: TObject;
    Field: TField);
begin
    Button5.Caption :=
        'First ' + Table1.FieldByName('State').AsString;
    Button6.Caption :=
        'Last ' + Table1.FieldByName('State').AsString;
    Button7.Caption :=
        'Next ' + Table1.FieldByName('State').AsString;
    Button8.Caption :=
        'Prior ' + Table1.FieldByName('State').AsString;
    Table1.Filter := '[State] = ' + #39 +
        Table1.FieldByName('State').AsString +
        #39;
end;
    
```

The first four lines of this event handler update the captions of the four buttons that the user navigates with using the filter. The last line sets the filter. Notice the use of the #39 character in this assignment statement. This is necessary to enclose the State string in the single quotation marks that identify string literals in Object Pascal.

The remainder of this project is very simple. The top four buttons shown in **Figure 4** call the table methods *First*, *Last*, *Next*, and *Prior*. The second set of buttons calls the new filtered navigation methods *FindFirst*, *FindLast*, *FindNext*, and *FindPrior*.

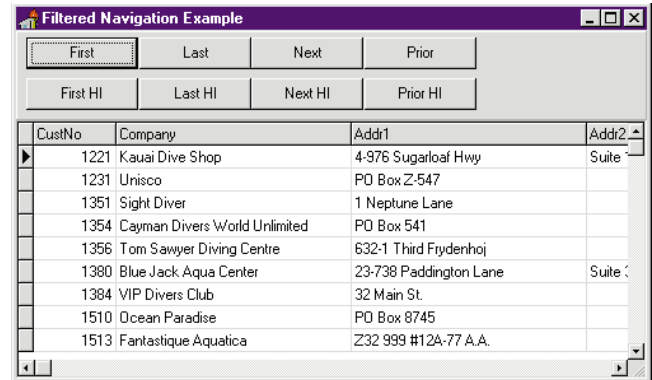


Figure 4: The FILTNAV.DPR project demonstrates how you can navigate a DataSet using a filter, even when the filter is not being actively applied.

Conclusion

The new filter feature of Delphi 2 provides you with a new set of tools for presenting users with subsets of records, as well as permitting them to navigate selected records, even while viewing the entire DataSet.

These valuable new filtering capabilities are welcome. However it's important to emphasize that these filtering operations do not make use of indexes. Therefore you should use these filtering techniques only on a subset (i.e. the result of a query, view, or range) of a large database. ▲

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\JUNE\96\DI9606CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including the upcoming *Delphi In-Depth* [Osborne/MacGraw-Hill, 1996]. He is also Contributing Editor of *Paradox Informant* and *Delphi Informant*, and is this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. You can reach Jensen Data Systems at (713) 359-3311, or via CompuServe at 76307,1533.





FROM THE PALETTE

Delphi 1/2 / Object Pascal



By *Ray Konopka*

Components & Sub-Components

Encapsulating Multiple Controls

The ability to build custom components is one of Delphi's most exciting features. It is also one of its more popular features judging by the number of components that are available on the World Wide Web, the Borland Delphi CompuServe forum (GO DELPHI), and of course, the *Delphi Informant* CompuServe forum (GO ICGFORUM). Aside from being Delphi components, most of the components available from these sources have one thing in common: they encapsulate a single control. While this is certainly the norm, it is by no means a restriction — the VCL is sufficiently rich to support encapsulating multiple controls. To illustrate the issues involved in this process, this article describes how to create a data-aware address component.

The RzAddress component is a single component that consists of separate sub-components representing the different fields in a typical US mailing address. The component comprises five DBEdit components, one DBComboBox, and six Label components. The arrangement of these components is illustrated in Figure 1, which shows the RzAddress component being used in an application.

Before getting into the details of the *TRzAddress* class, you may be wondering

why anyone would want to build a component like this. Aside from the ever popular answer, "Because you can," there are two principle reasons for encapsulating multiple controls in a single component: the first is to promote *reusability*; the second is to promote *consistency*.

For data entry applications, entering an address is a common task. Unfortunately, the forms used to enter address information generally capture additional information not relevant to the address. This extra information prevents the form from being reused in other applications. However, a component like RzAddress forces you to think about reusing a portion of the form, rather than the entire form.

The second benefit, consistency, is more of an issue with end users. Let's continue with the address example. Without a reusable component, every application that provides fields for entering an address will vary — at least slightly — from the others. The length of the last name field may be shorter in one application than another. The state field may be represented by an edit field in one program, and by a combo box in another. By creating a sin-

Last Name	First Name	Address	City	State	Zip
Davis	Jennifer	100 Cranberry St.	Well	MA	02181
Jones	Arthur	10 Hunnewell St	Los Altos	CA	94024
Parker	Debra	74 South St	Atherton	CA	98765
Sawyer	Dave	101 Oakland St	Los Altos	CA	94022
White	Cindy	1 Wentworth Dr	Los Altos	CA	94022

Figure 1: Editing an address the easy way!

gle component to represent the group of controls, consistency can be maintained between applications.

The TRzAddress Class

The RzAddress component is implemented in the *RzAddr* unit (see Listing Two beginning on page 25). The *TRzAddress* class descends from *TWinControl* because it essentially needs to be able to contain other controls within itself, and a window handle is needed to support this feature. However, *TWinControl* is not your only option — if you would like a border around the controls, the *TPanel* or *TGroupBox* classes serve equally well as ancestors.

The private section of this class contains an object field for each sub-component. The sub-components are created in the *Create* constructor. One-by-one, each sub-component is dynamically created and positioned within the RzAddress component. Note that the coordinates passed to the *SetBounds* method are relative to the main component's client area.

A few supporting methods are provided to make it easier to construct the sub-components. These include *CreateLabel*, *CreateCombo*, and *CreateEdit*. Each of these methods creates an object of the appropriate type, and then sets the *Parent* and *Visible* properties. The *CreateLabel* method then sets the caption of the label, while the *CreateCombo* method sets the *Sorted* property to *True*. The *CreateEdit* method finishes by assigning the *OnChange* event of the sub-component to point to the *TRzAddress.DoChange* method. We'll come back to the importance of this later in the article.

After all the sub-components are created, the internal *FStateList* is populated. *FStateList* is a string list object that is used to populate the *State* combo box. The combo box itself cannot be used to store the strings because it is dynamically created, and therefore does not provide persistent storage. A *StateList* property of *TRzAddress* could be displayed, but there is no real need to give the user direct access to this list.

Because the string list is created within the component, a destructor must be provided so the memory used by the string list can be freed. Speaking of destructors: who's responsible for destroying the sub-components? Well, actually we are. What I mean is, the component writer is responsible for making sure the sub-components are released when the main component is destroyed. However, you may have already noticed there is no code that specifically frees the sub-components.

When each sub-component is created, *Self* is passed to the *Create* constructor. This causes the sub-component to be placed into the *TRzAddress.Components* list. The *Components* list is defined in the *TComponent* class, and when the main component is destroyed, the inherited destructor from *TComponent* frees all the components in the *Components* list.

So, as long as a valid owner is passed to the constructor for each sub-component, we do not have to worry about cleaning up the sub-components.

Accessing Sub-Components through Properties

There are three basic ways of accessing a sub-component through the main component. All three involve properties, and each provides a different level of control. The first way is to provide the main component with a generic property that gets mapped to each sub-component. The *DataSource* property is an example of this type of access. Notice that the *TRzAddress* class does not maintain an internal field for holding the *DataSource* value. Instead, the sub-components themselves are used to manage the property. As you can see from the *SetDataSource* method, when the *TRzAddress.DataSource* is changed, the *DataSource* properties for all sub-components are updated with the new value.

The second way of providing access to sub-components is to provide individual properties that correspond to properties in each sub-component. *FirstNameField*, *LastNameField*, and *CityField* are examples of this type of access. Each one of these properties correspond to the *DataField* property of one of the sub-components. Unlike the *DataSource* property, these field names cannot be shared among the sub-components. Figure 2 shows the *FirstNameField* property being edited to link the corresponding sub-component to the appropriate table column. (As an aside, the *Field* properties also provide a good example of using indexed properties. All six properties are supported by the same access methods: *GetField* and *SetField*.)

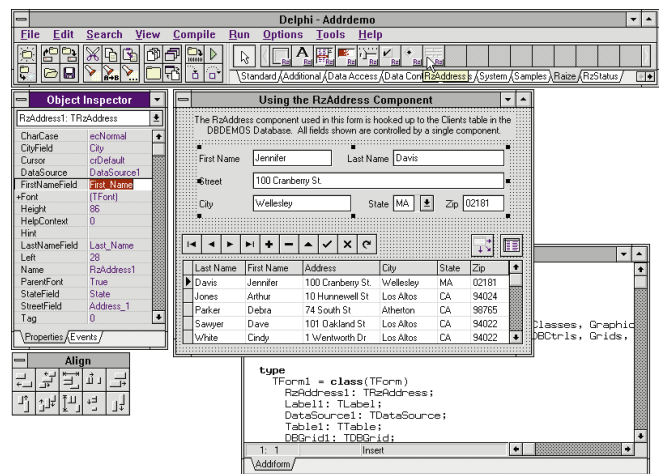


Figure 2: Editing the *FirstNameField* property.

The third way of providing access to sub-components is to expose a reference to the sub-component. Because this allows complete access to the sub-component, it is generally not wise to do. As an example, the *EdtFirstName* property provides a reference to the *FEdtFirstName* edit field. With this reference, the end user has access to the properties of *FEdtFirstName*, and therefore, could affect the way the entire RzAddress component behaves. For example, the edit control could be moved or resized. More dramatic is the problem of setting the *DataSource* property of the *FEdtFirstName* field directly. A sub-component reference can be used to bypass all other types of access.

Exposing Events that Occur in Sub-Components

Earlier, I mentioned that each edit field created gets its *OnChange* event assigned to the *DoChange* method. Like properties, events can be exposed individually, or shared; but because of the event architecture, a hybrid between the two can be achieved. Any time a change event occurs in one of the edit fields, the *DoChange* method is called. Since the *Sender* parameter identifies which component generated the message, this information can be passed on to the end user's event handler for the main component's *OnChange* event.

The *OnChange* event for the *RzAddress* component receives two parameters. The first parameter is an enumerated value indicating the sub-component that generated the event. For example, if the change event occurred in the *FEdtCity* edit field, the first parameter would have a value of *efCity*. The second parameter contains the current contents of the edit field. Using these two parameters, a user can create a single event handler to manage all of the *OnChange* events that occur within each of the sub-components.

Conclusion

You may have noticed that encapsulating multiple controls within a single component is very similar to dynamically creating components on a form at run time. This is not surprising since forms are part of the Visual Component Library. In essence, a form just happens to be a very intelligent multi-control component. Once the sub-components are created, building a multi-control component simply requires defining the interactions between the controls and providing sub-component access.

Portions of this article are adapted from material in Ray Konopka's, *Developing Custom Delphi Components* [The Coriolis Group, 1996]. [▲](#)

The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\JUNE\96\DI9606RK.

Ray Konopka is the author of *Developing Custom Delphi Components*, published by The Coriolis Group. Ray is also the founder of Raize Software Solutions, Inc., supplier of Delphi consulting services. Ray can be reached at Raize95@aol.com or 75107.2356@compuserve.com.

Begin Listing Two — The RzAddr Unit

```
{ RzAddr Unit. This unit implements the RzAddress
component which is comprised of the following edit
fields: FirstName, LastName, Street, City, and Zip. The
State field is actually a combo box which is populated
with the 50 states including the District of Columbia.
The edit fields are data-aware, and thus this component
can be hooked up to a DataSource.
Developing Custom Delphi Components — Ray Konopka }
```

```
unit RzAddr;
```

```
interface
```

```
uses
```

```
Classes, Controls, StdCtrls, DB, DBCtrls,
Graphics, ExtCtrls, RzCommon;
```

```
type
```

```
TEditField = (efFirstName, efLastName,
              efStreet, efCity, efZip);
TEditChangeEvent =
  procedure (Field : TEditField;
            Text : string) of object;
TRzAddress = class(TWinControl)
private
```

```
  FEdtFirstName : TDBEdit;
  FEdtLastName : TDBEdit;
  FEdtStreet : TDBEdit;
  FEdtCity : TDBEdit;
  FCbxState : TDBComboBox;
  FEdtZip : TDBEdit;
  { Internal List of State Abbreviations }
  FStateList : TStringList;
  { Common Change Event for all Edit Fields }
  FOnChange : TEditChangeEvent;
```

```
function GetCharCase : TEditCharCase;
procedure SetCharCase(Value : TEditCharCase);
function GetDataSource : TDataSource;
procedure SetDataSource(Value : TDataSource);
function GetField(Index : Integer) : string;
procedure SetField(Index : Integer; Value : string);
function CreateEdit : TDBEdit;
function CreateLabel(S : string) : TLabel;
function CreateCombo : TDBComboBox;
procedure CreateStateList;
procedure DoChange(Sender : TObject);
```

```
protected
```

```
procedure Change(Field : TEditField;
                 Text : string); dynamic;
procedure CreateWnd; override;
```

```
public
```

```
constructor Create(AOwner : TComponent); override;
destructor Destroy; override;
```

```
property EdtFirstName : TDBEdit
  read FEdtFirstName;
```

```
published
```

```
property CharCase : TEditCharCase
  read GetCharCase write SetCharCase;
```

```
property DataSource : TDataSource
  read GetDataSource write SetDataSource;
```

```
property FirstNameField : string
  index 1 read GetField write SetField;
```

```
property LastNameField : string
  index 2 read GetField write SetField;
```

```
property StreetField : string
  index 3 read GetField write SetField;
```

```
property CityField : string
  index 4 read GetField write SetField;
```

```
property StateField : string
  index 5 read GetField write SetField;
```

```
property ZipField : string
  index 6 read GetField write SetField;
```

FROM THE PALETTE

```
property OnChange : TEditChangeEvent
read FOnChange write FOnChange;

property Font;
property ParentFont;
end;

procedure Register;

implementation

uses
  DsgnIntf;

{ TRzAddress Methods }
constructor TRzAddress.Create(AOwner : TComponent);
var
  TempLbl : TLabel;
begin
  inherited Create(AOwner);

  { All labels are created using the TempLbl component
  because we do not need to reference these controls
  elsewhere. Clean up is handled when the TComponent
  ancestor class frees all components on the Components
  list. }
  TempLbl := CreateLabel('First Name');
  with TempLbl do SetBounds(0, 8, Width, Height);
  FEdtFirstName := CreateEdit;
  FEdtFirstName.SetBounds(67, 4, 97, 20);

  TempLbl := CreateLabel('Last Name');
  with TempLbl do SetBounds(182, 8, Width, Height);
  TempLbl.Alignment := taRightJustify;

  FEdtLastName := CreateEdit;
  FEdtLastName.SetBounds(240, 4, 137, 20);

  TempLbl := CreateLabel('Street');
  with TempLbl do SetBounds(0, 36, Width, Height);
  FEdtStreet := CreateEdit;
  FEdtStreet.SetBounds(67, 32, 310, 20);

  TempLbl := CreateLabel('City');
  with TempLbl do SetBounds(0, 64, Width, Height);
  FEdtCity := CreateEdit;
  FEdtCity.SetBounds(67, 60, 121, 20);

  TempLbl := CreateLabel('State');
  with TempLbl do SetBounds(200, 64, Width, Height);
  TempLbl.Alignment := taRightJustify;
  FCbxState := CreateCombo;
  FCbxState.SetBounds(240, 60, 50, 20);

  TempLbl := CreateLabel('Zip');
  with TempLbl do SetBounds(300, 64, Width, Height);
  TempLbl.Alignment := taRightJustify;
  FEdtZip := CreateEdit;
  FEdtZip.SetBounds(326, 60, 51, 20);

  CreateStateList;

  Width := 382;
  Height := 86;
end; { = TRzAddress.Create = }

destructor TRzAddress.Destroy;
begin
  FStateList.Free;
  inherited Destroy;
end;
```

```
function TRzAddress.CreateLabel(S : string) : TLabel;
begin
  Result := TLabel.Create(Self);
  Result.Parent := Self;
  Result.Visible := True;
  Result.AutoSize := True;
  Result.Caption := S;
end;

function TRzAddress.CreateEdit : TDBEdit;
begin
  Result := TDBEdit.Create(Self);
  Result.Parent := Self;
  Result.Visible := True;
  { Assign OnChange event of each Edit field
  to point to TRzAddress.DoChange method. }
  Result.OnChange := DoChange;
end;

function TRzAddress.CreateCombo : TDBComboBox;
begin
  Result := TDBComboBox.Create(Self);
  Result.Parent := Self;
  Result.Visible := True;
  Result.Sorted := True;
end;

procedure TRzAddress.CreateWnd;
begin
  inherited CreateWnd;

  { When CreateWnd is called, the Items list of FCbxState
  is cleared. Therefore, the contents of the FStateList
  are copied back into FCbxState. }
  FCbxState.Items.Assign(FStateList);
end;

procedure TRzAddress.CreateStateList;
begin
  FStateList := TStringList.Create;
  FStateList.Add('AK');
  FStateList.Add('AL');
  FStateList.Add('AR');
  FStateList.Add('AZ');
  FStateList.Add('CA');
  FStateList.Add('CO');
  FStateList.Add('CT');
  FStateList.Add('DC');
  FStateList.Add('DE');
  FStateList.Add('FL');
  FStateList.Add('GA');
  FStateList.Add('HI');
  FStateList.Add('IA');
  FStateList.Add('ID');
  FStateList.Add('IL');
  FStateList.Add('IN');
  FStateList.Add('KS');
  FStateList.Add('KY');
  FStateList.Add('LA');
  FStateList.Add('MA');
  FStateList.Add('MD');
  FStateList.Add('ME');
  FStateList.Add('MI');
  FStateList.Add('MN');
  FStateList.Add('MO');
  FStateList.Add('MS');
  FStateList.Add('MT');
  FStateList.Add('NC');
  FStateList.Add('ND');
  FStateList.Add('NE');
  FStateList.Add('NH');
  FStateList.Add('NJ');
```

```

FStateList.Add('NM');
FStateList.Add('NV');
FStateList.Add('NY');
FStateList.Add('OH');
FStateList.Add('OK');
FStateList.Add('OR');
FStateList.Add('PA');
FStateList.Add('RI');
FStateList.Add('SC');
FStateList.Add('SD');
FStateList.Add('TN');
FStateList.Add('TX');
FStateList.Add('UT');
FStateList.Add('VA');
FStateList.Add('VT');
FStateList.Add('WA');
FStateList.Add('WI');
FStateList.Add('WV');
FStateList.Add('WY');
end; { = TRzAddress.CreateStateList = }

procedure TRzAddress.Change(Field : TEditField;
                           Text : string);
begin
    if Assigned(FOnChange) then
        FOnChange(Field, Text);
end;

{ TRzAddress.DoChange. This method gets called if the
  OnChange event occurs for any of the edit fields
  contained in this component. The Change event dispatch
  method is called to surface those events to the user. }
procedure TRzAddress.DoChange(Sender : TObject);
var
    Field : TEditField;
begin
    if Sender = FEdtFirstName then
        Field := efFirstName
    else if Sender = FEdtLastName then
        Field := efLastName
    else if Sender = FEdtStreet then
        Field := efStreet
    else if Sender = FEdtCity then
        Field := efCity
    else
        Field := efZip;
    Change(Field, TDBEdit(Sender).Text);
end;

function TRzAddress.GetCharCase : TEditCharCase;
begin
    Result := FEdtFirstName.CharCase;
end;

procedure TRzAddress.SetCharCase(Value : TEditCharCase);
begin
    if Value <> FEdtFirstName.CharCase then
        begin
            FEdtFirstName.CharCase := Value;
            FEdtLastName.CharCase := Value;
            FEdtStreet.CharCase := Value;
            FEdtCity.CharCase := Value;
            FEdtZip.CharCase := Value;
        end;
end;

function TRzAddress.GetDataSource : TDataSource;
begin

```

```

    { Use FEdiFirstName to Get Current DataSource }
    Result := FEdtFirstName.DataSource;
end;

procedure TRzAddress.SetDataSource(Value : TDataSource);
begin
    if Value <> FEdtFirstName.DataSource then
        begin
            { Assign All Internal Controls to Same DataSource }
            FEdtFirstName.DataSource := Value;
            FEdtLastName.DataSource := Value;
            FEdtStreet.DataSource := Value;
            FEdtCity.DataSource := Value;
            FCbxState.DataSource := Value;
            FEdtZip.DataSource := Value;
        end;
end;

function TRzAddress.GetField(Index : Integer) : string;
begin
    case Index of
        1: Result := FEdtFirstName.DataField;
        2: Result := FEdtLastName.DataField;
        3: Result := FEdtStreet.DataField;
        4: Result := FEdtCity.DataField;
        5: Result := FCbxState.DataField;
        6: Result := FEdtZip.DataField;
    end;
end;

procedure TRzAddress.SetField(Index : Integer;
                               Value : string);
begin
    case Index of
        1: FEdtFirstName.DataField := Value;
        2: FEdtLastName.DataField := Value;
        3: FEdtStreet.DataField := Value;
        4: FEdtCity.DataField := Value;
        5: FCbxState.DataField := Value;
        6: FEdtZip.DataField := Value;
    end;
end;

{ Register Procedure }
procedure Register;
begin
    RegisterComponents(RaizePage, [ TRzAddress ]);
    { The following RegisterPropertyEditor calls instruct
      the Object Inspector to hold off accepting the text
      entered into the specified fields until the Enter key
      is pressed. }
    RegisterPropertyEditor(TypeInfo(string), TRzAddress,
        'FirstNameField', TStringProperty);
    RegisterPropertyEditor(TypeInfo(string), TRzAddress,
        'LastNameField', TStringProperty);
    RegisterPropertyEditor(TypeInfo(string), TRzAddress,
        'StreetField', TStringProperty);
    RegisterPropertyEditor(TypeInfo(string), TRzAddress,
        'CityField', TStringProperty);
    RegisterPropertyEditor(TypeInfo(string), TRzAddress,
        'StateField', TStringProperty);
    RegisterPropertyEditor(TypeInfo(string), TRzAddress,
        'ZipField', TStringProperty);
end;

end.
End Listing Two

```





By *Keith Wood*

Talking to Yourself

A Look at Recursion in Object Pascal

We've all seen those cereal boxes that have pictures of someone holding the same cereal box, which has another picture of someone, etc. This is an example of recursion, and it's a powerful tool for the programmer as well.

Recursion arises when something is defined in terms of itself. Pascal allows for recursive functions and procedures that are well suited for particular programming problems. To show how it all works, this article looks at recursion in general, and presents a recursive function, two recursive plotting procedures, and a binary tree class.

Recursion

Normal processing in a program is iterative. This means that part of the algorithm may be executed in a **for**, **while**, or **repeat** loop. In recursive processing, the entire algorithm is re-executed from the beginning on a smaller part of the problem.

Two things are necessary for using recursion in programming. First, the problem must be defined in terms of itself; second, it must have a terminating condition. Since the portion of the problem being solved at each level of recursion is smaller than the previous step, eventually we get to a very small or easy problem for which we can immediately provide the answer (the terminating condition). This answer can then be passed back up the levels for further manipulation before obtaining the final result.

For particular types of problems, recursive algorithms can provide a solution in a few steps that would be extremely complex using an iterative algorithm.

Within a function or procedure we make use of values held in variables. Won't these get

mixed up during recursion since we are calling the same function? No — Pascal provides separate variables each time the function is called. They are only valid within that particular instance of the function, and retain their values across any other recursive calls. Of course this does have the effect that the space available for variables (the "stack") is slowly eaten away as we go deeper into the recursive calls. If this continues indefinitely, the space is eventually exhausted and an error occurs.

Factorials

The factorial function in mathematics is a classic recursion example. This function computes the product of all the integers between one and a given positive value (the result gets very large, very quickly). It's denoted by the exclamation mark (!) and can be defined by the following:

$$1! = 1$$
$$n! = n * (n - 1)!$$

In other words, one factorial is equal to one, while n factorial — where n is any positive integer greater than one — is equal to the value n times the factorial of one less than n . The first part of the definition is the terminating condition and the second part is the recursive call.

To implement this in Pascal we define a function (see [Figure 1](#)) that takes one integer parameter and returns a long integer (remember that it grows very quickly). We

```
function Factorial(Value: Integer):LongInt;
begin
  { Check for invalid parameter }
  if Value < 1 then
    raise Exception.Create('Invalid value for Factorial');

  { Termination condition }
  if Value = 1 then
    Result := 1
  else
    { Call function again with smaller value }
    Result := Value * Factorial(Value - 1);
end;
```

Figure 1: A recursive factorial function.

need to check that a valid value has been entered initially: it cannot be zero or negative. If an incorrect value is found then an exception is raised to notify the program of the problem.

With a valid value, we check for the terminating condition (in this case the value being equal to one), stop the recursion, and return the predefined result if this is so. If we are not at the terminating condition, we need to call the function again with the next lowest value, and compute the product with the current value to obtain the final result.

Value	Result
5	5 * 4!
4	4 * 3!
3	3 * 2!
2	2 * 1!
1	1
2	2 * 1 = 2
3	3 * 2 = 6
4	4 * 6 = 24
5	5 * 24 = 120

Figure 2: Steps in computing 5!

When called with a value greater than one, the function halts its execution at the recursive call until the required value is returned from the next level down. Once the terminating condition is reached, the results are passed back through the preceding levels building up the final result as it goes. Figure 2 is a picture of this process.

To see this in “real life,” we can put a breakpoint on the result of the terminating condition (in this case, `Result := 1`) and then re-execute the program. When it stops, display the call sequence with the menu command `View | Call Stack`. The recursive function calls and their parameters are shown in the Call Stack window.

The factorial function is implemented in the FACT1.PAS unit that accompanies this article. The project, FACTRIAL.DPR (see Figure 3), allows you to select an input value for the factorial function between 0

and 12 (this being the limit for a long integer result). Note that using zero results in an exception being raised; this has been left to show what happens with invalid input.

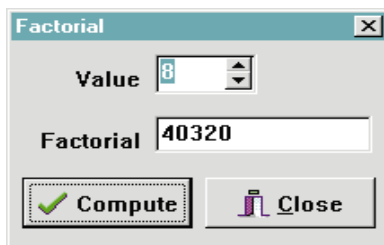


Figure 3: Our example project, FACTRIAL.DPR.

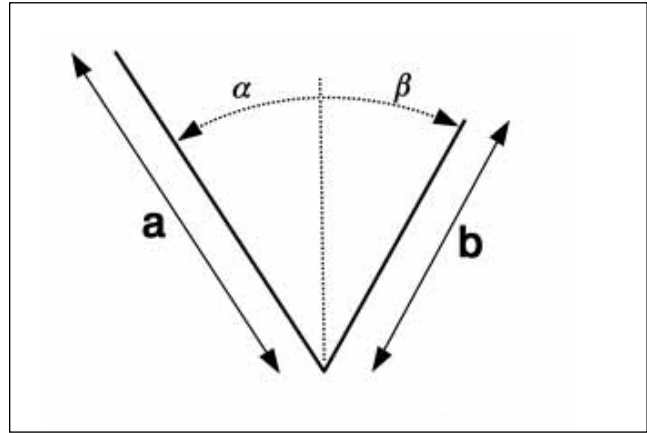


Figure 4: Definition of the tree structure.

Plotting Trees

A routine that allows us to plot images of trees based on the structure is shown in Figure 4. A tree consists of two branches from the one base point. These can have different lengths, a and b , and can be inclined at different angles, α (alpha) and β (beta), from the vertical.

To complete the figure, we then plot the same structure at the end of each branch on a reduced scale, using the direction of that branch as the new “vertical.” This sort of tree is effectively a fractal, with parts of it being repeated at an ever smaller scale.

The algorithm for this can be stated as:

If the length of the main branch is below a predefined threshold (the terminating condition) then exit the procedure without doing anything. Otherwise, given a point and a direction (the “vertical”), plot the two branches of the tree. The first is to the left of vertical by a specified angle and with a given length. The second is to the right of vertical by a , possibly, a different angle and with a length reduced from the first by a given amount. Then call the procedure recursively for each new endpoint and its corresponding angle, but with a reduced main branch length.

This process is captured in the code in Figure 5. We must pass numerous parameters to it, being the starting point, the current direction of the vertical (in degrees), the two angles by which the branches are offset, the length of the main branch and the amounts by which to reduce this to produce the length of the secondary branch, and the length of the main branch for the next level.

We must delve into some trigonometry to calculate the endpoints for the branches. The `mod` operator is used to ensure that the angles being used remain in the range of 0 to 359, preventing any possibility of overflow. Angles are defined in degrees, since this is easier for us to use, but must be converted to radians before Object Pascal can use them. Radians are used in all of Pascal’s trigonometric functions, and are calculated based on 180 degrees being equal to π radians. The conversion is done using a predefined value to reduce the amount of computation required in the plotting process.

```

{ Recursive plotting routine }
procedure TForm1.DrawImage(CurPoint: TPoint;
    CurAngle, LeftAngle,RightAngle: TAngle;
    Length: Integer; Ratio, Shrink: Real);
var
    Angle: Real;
    NewPoint: TPoint;
begin
    { Terminating condition - line too short }
    if (Length < MinLength) or Finish then
        Exit;

    { Let something else have a go }
    Application.ProcessMessages;

    { Calculate the new endpoint for the left side }
    Angle :=
        ((CurAngle - LeftAngle) mod 360) * ConvertToRadians;
    NewPoint.X := CurPoint.X + Round(Cos(Angle) * Length);
    NewPoint.Y := CurPoint.Y + Round(Sin(Angle) * Length);
    { And draw a line to it }
    TreeImage.Canvas.MoveTo(CurPoint.X, CurPoint.Y);
    TreeImage.Canvas.LineTo(NewPoint.X, NewPoint.Y);
    { Then call procedure again with reduced size from
    end of line }
    DrawImage(NewPoint, (CurAngle - LeftAngle) mod 360,
        LeftAngle, RightAngle, Round(Length * Shrink),
        Ratio, Shrink);

    { Calculate the new endpoint for the right side }
    Angle :=
        ((CurAngle + RightAngle) mod 360) * ConvertToRadians;
    NewPoint.X :=
        CurPoint.X + Round(Cos(Angle) * Length * Ratio);
    NewPoint.Y :=
        CurPoint.Y + Round(Sin(Angle) * Length * Ratio);
    { And draw a line to it }
    TreeImage.Canvas.MoveTo(CurPoint.X, CurPoint.Y);
    TreeImage.Canvas.LineTo(NewPoint.X, NewPoint.Y);
    { Then call procedure again with reduced size from end
    of line }
    DrawImage(NewPoint, (CurAngle + RightAngle) mod 360,
        LeftAngle, RightAngle,
        Round(Length * Ratio * Shrink), Ratio, Shrink);
end;

```

Figure 5: The recursive procedure for plotting trees.

The procedure contains some extra code to allow for additional processing. The first is a call to the *ProcessMessages* method of the *Application* object. This allows Windows to perform any other outstanding processing. The second is the alternate terminating condition, being when the flag *Finish* is set to *True*. This variable is defined outside the procedure and can be used to halt the drawing process if it's taking too long. To do this in our case, the **Draw** button on the form changes to a **Cancel** button once it has started the plotting process. A second click on it sets *Finish* to *True*, which it can do because of the call to *ProcessMessages*, and this causes the drawing procedure to terminate on its next cycle.

To see this procedure in action look in the TREES1.PAS unit and the TREES.DPR project that accompany this article. This program allows most of the parameters to the procedure to be set interactively before drawing the resultant tree (see [Figure 6](#)). The **Symmetric** check box disables the **Right Angle** and **L/R Ratio%** controls, tying the first to **Left Angle** and setting the second to 100 percent. This is merely a convenience to allow easier plotting of symmetric trees.

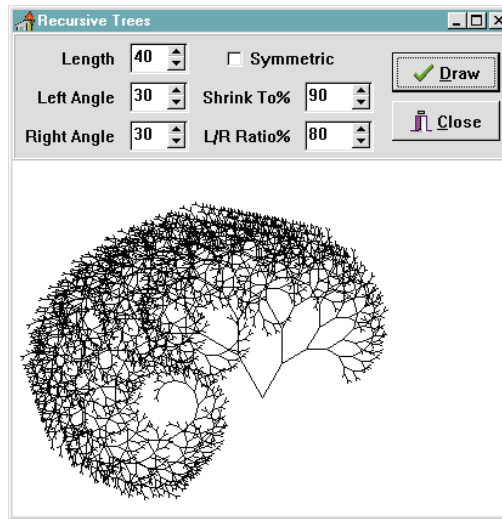


Figure 6: A recursive tree plotting program.

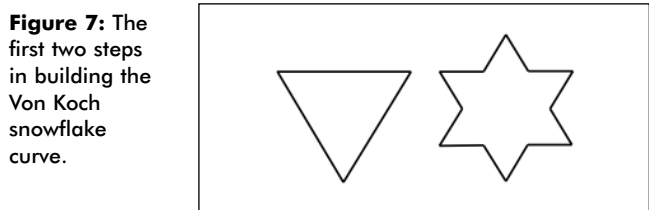


Figure 7: The first two steps in building the Von Koch snowflake curve.

Plotting a Fractal

For the third example of recursion we are drawing another fractal shape, this time the Von Koch snowflake curve. It's defined as consisting of an equilateral triangle, each side of which is divided into three equal parts, with the central part being replaced by two sides of a smaller equilateral triangle based on that part (see [Figure 7](#)). This structure is then applied repeatedly to each of the smaller triangles so created.

So the recursive algorithm that we are using can be defined as follows:

Given the two endpoints for this segment, determine its length. If this falls below some threshold (the terminating condition) then simply draw this segment on the screen. Otherwise, compute the three additional points that define the curve over this segment. These are the points one and two thirds of the way along the line and the midpoint at the top of the central equilateral triangle. Then call the procedure recursively for each of these component segments in turn.

To calculate the points along the segment is very simple for the two actually on the line. The third point — the midpoint — requires some trigonometry to determine its position. Basically we find the angle of the line segment (always from start to finish since this determines on which side the peak appears) and subtract 30 degrees, which is the angle that the midpoint makes to the original segment through the starting point. Then using trigonometry, find the x and y offsets to the point from the starting point.

The code for all of this is shown in [Figure 8](#). Note that two constants have been defined earlier to reduce the amount

```

{ Recursive fractal plotting routine }
procedure TForm1.DrawSegment(FromPoint, ToPoint: TPoint);
var
  X, Y: LongInt;
  Length, Angle: Real;
  MidPoint, FirstThird, SecondThird: TPoint;
begin
  if Finish then Exit;

  { Determine length of line segment }
  X := ToPoint.X - FromPoint.X;
  Y := ToPoint.Y - FromPoint.Y;
  Length := Sqrt(Y * Y + X * X);

  { Terminating condition }
  if Length < MinLength.Value then
  begin
    { Draw the line segment }
    FractalImage.Canvas.MoveTo(FromPoint.X, FromPoint.Y);
    FractalImage.Canvas.LineTo(ToPoint.X, ToPoint.Y);
    Exit;
  end;

  { Allow something else to have a go }
  Application.ProcessMessages;

  { Determine the angle of the line segment }
  if X = 0 then
  begin
    if Y > 0 then
      Angle := Pi / 2
    else
      Angle := - Pi / 2;
    end
  else
    Angle := ArcTan(Y / X);

  if X < 0 then
    Angle := Angle - Pi;

  { Compute intermediate points on this line }
  MidPoint.X := FromPoint.X +
    Round(Length / Root3 * Cos(Angle - Degrees30));
  MidPoint.Y := FromPoint.Y +
    Round(Length / Root3 * Sin(Angle - Degrees30));
  FirstThird.X := FromPoint.X + Round(X / 3);
  FirstThird.Y := FromPoint.Y + Round(Y / 3);
  SecondThird.X := FromPoint.X + Round(2 * X / 3);
  SecondThird.Y := FromPoint.Y + Round(2 * Y / 3);

  { And recursively draw each segment }
  DrawSegment(FromPoint, FirstThird);
  DrawSegment(FirstThird, MidPoint);
  DrawSegment(MidPoint, SecondThird);
  DrawSegment(SecondThird, ToPoint);
end;

```

Figure 8: The recursive procedure for plotting the Von Koch snowflake curve.

of calculation required. These are *Root3* which is the square root of three and *Degrees30* which is 30 degrees expressed as radians. Also note that we are making four recursive calls from this procedure, one for each component line segment.

This procedure can be found in the SNOWFLK1.PAS unit. The sample project, SNOWFLAK.DPR, allows the maximum segment length to be set and uses this as the threshold value when plotting the curve (see Figure 9). The form should plot different figures for values of 5, 10, 30, and 90. Note that some of the other values may produce slightly incomplete curves because of the closeness of the segment lengths to the threshold value. It also allows for the segments

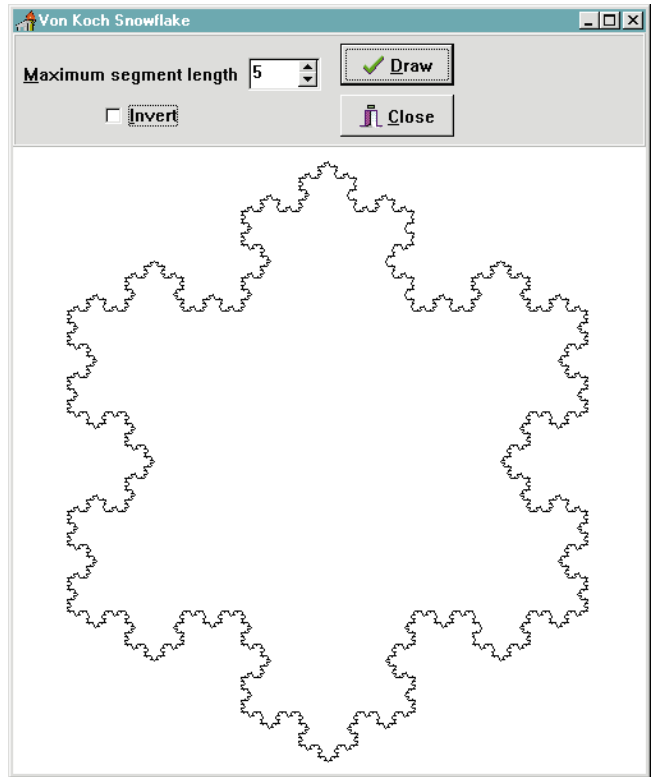


Figure 9: The recursive Von Koch snowflake plotting program.

to be plotted in the reverse direction, generating an inverted Von Koch snowflake. Because the plotting routine may take some time to complete, a call to the *ProcessMessages* method of the *Application* object allows other Windows programs to perform some processing of their own, and for this routine to be canceled, similar to the tree plotting program above.

Binary Tree

The final example of recursion involves creating a sorted binary tree object. This object maintains a group of other objects in a specified order, with searching and processing being on average faster than a straight array. A binary tree is built up from nodes. Each node has a value and two pointers (hence binary) to those nodes that have values less than this one and those that have values greater than this one.

Note that these pointers may be empty. The first node in the tree is referred to as the *root*, and the tree is usually depicted as growing down from the root, with node values increasing to the right (see Figure 10). Those nodes at the bottom of the tree with no descendants are referred to as *leaf* nodes. Note that the root is the only means of access to the entire tree. All processing must start from here.

As you can see, each node and its descendants form a sub-tree within the main tree. This gives us the self-definition required for recursion, with the leaf nodes or empty pointers being the terminating conditions. The processes that we must be able to perform on a binary tree include inserting a new value, removing an existing value, checking whether a value is present, and processing all the nodes in order.

To implement this in Delphi we can make use of the class structure and define a binary tree class, *TBinaryTree*,

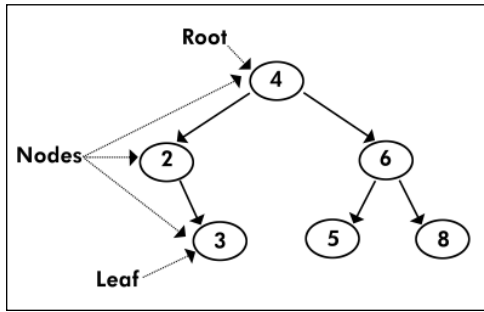


Figure 10: A sorted binary tree sample.

and its associated methods. It's derived from *TObject*, since it requires only the very basics of a class definition. It will have a single private variable, that points to the root of the tree, and several public functions and procedures to allow us to manipulate the tree. The tree's nodes are defined as a record, *TBinaryTreeNode*, each of which holds an object and two pointers to other nodes:

```

{ The binary tree nodes }
PBinaryTreeNode = ^TBinaryTreeNode;
TBinaryTreeNode = record
  NodeObject: TBinaryTreeObject;
  Left, Right: PBinaryTreeNode;
end;
  
```

To ensure that the objects to be stored in the binary tree can be properly manipulated by it, we create an abstract base class called *TBinaryTreeObject* that defines the minimum requirements for such objects. Within it are declared the comparison functions necessary to position an object in the tree, *IsEqualTo* and *IsLessThan*, and a procedure to act upon an object when processing all the nodes in order, *Process*. All these are declared as being virtual and abstract. The former means that they can be overridden in a class derived from this one, while the latter means that they must be overridden since they are not defined here.

Note that a reference to an object is really a pointer to that object and that it is allocated space somewhere in memory. This memory should be released for other uses when we have finished with an object. Since the tree holds objects of a class derived from *TBinaryTreeObject* of which we have no knowledge, we cannot create new instances of these objects to hold in the tree within *TBinaryTree* itself. Thus, we store the pointers to objects that were created outside the binary tree class and must take on the responsibility of releasing their resources once we are finished.

Grafting

To insert a new value into the tree we must start at the root (initially empty) and traverse the structure to see whether the value is already there. This is done by comparing the value with that of the current node. If they are equal then the value is already in the tree and nothing further needs to be done (a terminating condition).

If they are not equal then we need to determine which of the node's sub-trees to search to find it. If the value is less than the current node then we look in the left sub-tree, otherwise

```

{ Recursive procedure to insert an object into the tree }
procedure TBinaryTree.InsertInTree(obj: TBinaryTreeObject;
  var ptr: PBinaryTreeNode);
begin
  if ptr = nil then { Terminating condition - add }
  begin
    ptr := New(PBinaryTreeNode);
    ptr^.NodeObject := obj;
    ptr^.Left := nil;
    ptr^.Right := nil;
  end
  { Terminating condition - already there }
  else if obj.IsEqualTo(ptr^.NodeObject) then
  begin
    ptr^.NodeObject.Free; { Free previous object }
    ptr^.NodeObject := obj; { And assign new one }
  end
  { Check in appropriate half of tree }
  else if obj.IsLessThan(ptr^.NodeObject) then
    InsertInTree(obj, ptr^.Left)
  else
    InsertInTree(obj, ptr^.Right);
end;
  
```

Figure 11: The recursive binary tree insertion procedure.

we check the right. If we encounter an empty pointer in this direction then the value is obviously not in the tree and we must add it at this point, since this is where we should have found it (see Figure 11). This procedure is called by the publicly available *Insert* method, starting at the root.

The notation in this code includes references to pointers. Type *PBinaryTreeNode* is a pointer to a *TBinaryTreeNode* record. To refer to the contents of that record we use the pointer symbol (^) to access the object being pointed to. Thus ptr is a pointer to a node, while ptr^ is the node record itself (a process known as *de-referencing*). References to objects within Object Pascal are also pointers, but this is hidden by Object Pascal that automatically de-references them when necessary.

Since we only have a pointer declared for our use, we must use the *New* function to allocate space dynamically when we want to create a new instance of the record. Conversely, when we are finished with an object being pointed to, we must release its resources using the *Dispose* procedure.

The *var* directive in the parameter list for the procedure means that within its code we are working with the actual item passed in (in this case a pointer). Any changes to the parameter also update the original in the calling environment. Normally we would only have access to a copy of the item and would be unable to affect the original. We need to use the original pointer in our case to correctly position the new node in the tree.

In our implementation of the tree we are also replacing a node if it already exists in the tree. This allows for any dependent values of the object to be updated. Remember that we must free the original object with that value since we have taken over responsibility for it and we no longer require it.

Pruning

Inserting an object into the tree is fairly straightforward, but removing objects is much more complicated. As before, we

search the tree for the object to be removed, starting at the root, and recursively processing the left or right sub-tree as required. If we reach an empty pointer then we can stop since the object is not in the tree anyway.

Once the required node has been located it must be removed. This will involve restructuring the tree below this node since we need to maintain the order inherent in the tree. If this node is a leaf node then all we need to do is break the link from the previous node in the tree to this one. If the node has only one descendant then we can safely change the link from the previous node to point to this sub-tree.

If the node has two descendants then things become more involved since we cannot move both up to the previous level. We do not want to move large numbers of nodes around to get everything in the right order again. The solution turns out to be quite simple: find the largest node in the left sub-tree, or the smallest node in the right sub-tree, and replace the current node with that one.

This node is guaranteed to be able to replace the node being removed by the definition of the tree structure, with-

```
{ Recursive function to remove an object from the tree
and return a flag showing whether it was there }
function TBinaryTree.RemoveFromTree(obj: TBinaryTreeNode;
var ptr: PBinaryTreeNode): Boolean;
begin
  if ptr = nil then { Terminating condition - not there }
    Result := False
  { Terminating condition - remove }
  else if obj.IsEqualTo(ptr^.NodeObject) then
    begin
      ReorganiseInTree(ptr);
      Result := True;
    end
  { Check in appropriate half of tree }
  else if obj.IsLessThan(ptr^.NodeObject) then
    Result := RemoveFromTree(obj, ptr^.Left)
  else
    Result := RemoveFromTree(obj, ptr^.Right);
end;

{ Procedure to reorganise the tree from this
point - deleting current node }
procedure TBinaryTree.ReorganiseInTree(
var ptr: PBinaryTreeNode);
var
  ptrRemove: PBinaryTreeNode;

  { Recursive procedure to replace with minimum
  in right sub-tree }
  procedure ReplaceWithMin(var ptrRep: PBinaryTreeNode);
  begin
    { Terminating condition - found minimum }
    if ptrRep^.Left = nil then
      begin
        ptr := ptrRep; { Adjust pointers }
        ptrRep := ptrRep^.Right;
        ptr^.Left := ptrRemove^.Left;
        ptr^.Right := ptrRemove^.Right;
      end
    else { Recursive call to next level }
      ReplaceWithMin(ptrRep^.Left);
  end;
end;
```

out further changes. If we always used nodes from one sub-tree and the rate of deletions is relatively high, then the tree can degenerate into a linked list, which removes the advantages of the tree structure. To overcome this we randomly pick one of the sub-trees to process for each deletion.

In all cases where we found the object, its resources must be released and the node that contained it destroyed. The code for all of this is shown in Figure 12. The recursive searching function, *RemoveFromTree*, is initially called by the publicly available method *Remove* with the root as the starting point. Once the object is found, control is passed to the *ReorganiseInTree* procedure. This is separated from the searching code to make both easier to follow. It then determines whether or not the node has less than two descendants and invokes the recursive sub-tree replacement procedures as required.

Note that the replacement procedures are declared within the *ReorganiseInTree* procedure. This gives them access to the variables in that procedure, which we do need. A flag is passed back to the user to indicate whether or not the record existed in the tree.

```
{ Recursive procedure to replace with maximum
in left sub-tree }
procedure ReplaceWithMax(var ptrRep: PBinaryTreeNode);
begin
  { Terminating condition - found maximum }
  if ptrRep^.Right = nil then
    begin
      ptr := ptrRep; { Adjust pointers }
      ptrRep := ptrRep^.Left;
      ptr^.Left := ptrRemove^.Left;
      ptr^.Right := ptrRemove^.Right;
    end
  else { Recursive call to next level }
    ReplaceWithMax(ptrRep^.Right);
end;

begin
  { Remember current node for later disposal }
  ptrRemove := ptr;
  if (ptr^.Left = nil) and (ptr^.Right = nil) then
    { Terminating condition - no more branches }
    ptr := nil
  { Terminating condition - move right tree up }
  else if ptr^.Left = nil then
    ptr := ptr^.Right
  { Terminating condition - move left tree up }
  else if ptr^.Right = nil then
    ptr := ptr^.Left
  else
    { Replace current object with min in right branch
    or max in left branch - this is done randomly
    to reduce distortions to the tree structure }
    begin
      if Random(2) = 0 then
        ReplaceWithMin(ptr^.Right)
      else
        ReplaceWithMax(ptr^.Left);
    end;
  { Free object at node to be deleted }
  ptrRemove^.NodeObject.Free;
  { And destroy the node }
  Dispose(ptrRemove);
end;
```

Figure 12: Recursive binary tree deletion function and procedures.

Easy Processing

Processing the entire tree in order is very simple with a recursive algorithm. At each node, simply process all the nodes in the left sub-tree (all less than this one), then process this node and finally those in the right sub-tree (all greater than this one).

When we reach an empty pointer we can halt the process. To process all the nodes in reverse order, we just reverse the algorithm: processing right, then the current one, then left. This is shown in [Figure 13](#). Again it's called by a public method, *ProcessAll*, starting at the root. Remember that the *Process* method of the objects in the tree must be defined in the class derived from the base binary tree object class for this to work.

The binary tree class also defines some other functions: whether an object exists in the tree, and returning the minimum and maximum values in the tree, but these are left for the reader to work through.

To make use of the binary tree class we must first derive a binary tree object to be stored in it. In the example program we derive a class that just contains an integer value, *TBinaryTreeInteger*, although it could contain anything that we wanted. We must then override the three abstract methods with real definitions. For example, the *IsEqualTo* function is defined as:

```
{ Equality for a binary tree integer - values equal? }
function TBinaryTreeInteger.IsEqualTo(
  other: TBinaryTreeObject): Boolean;
begin
  Result := (Value = (other as TBinaryTreeInteger).Value);
end;
```

Note that the parameter must be a *TBinaryTreeObject*, since this is what was defined in the base class, but that it can be cast as our derived class to gain access to its contents. The *Process* method will simply display the contents of each node.

The example form itself in the BINARYTREE.DPR project (see [Figure 14](#)) allows us to manipulate the binary tree. The value to be used is entered in the spin edit control, and one of the command buttons is pressed to initiate an action, the results of which are displayed in the memo component.

For each action, except **Print Tree**, a new *TBinaryTreeInteger* object is created and passed to the binary tree for processing. If the binary tree does not incorporate it into itself, then we must free that object after it has been used. In the case of finds within the tree, we are returned a pointer to the actual object in the tree, if it exists. This object must *not* be freed since the tree still has a reference to it (the original object that we created was freed by the tree before it returned the found one). The binary tree class is defined separately in the BINTREE.PAS file so that it can easily be incorporated into other projects.

Conclusion

Recursion involves defining a problem in terms of itself. Each time a smaller part of the problem is worked on, until we reach a position where we can easily compute the answer.

```
{ Recursive procedure for processing all objects in tree
  Can be done in ascending or descending order using flag }
procedure TBinaryTree.ProcessInTree(descending: Boolean;
  ptr: PBinaryTreeNode);
begin
  if ptr = nil then { Terminating condition }
    Exit;
  { Process appropriate half of tree before this object }
  if descending then
    ProcessInTree(descending, ptr^.Right)
  else
    ProcessInTree(descending, ptr^.Left);
  { Call process method of the object }
  ptr^.NodeObject.Process;
  { Process appropriate half of tree after this object }
  if descending then
    ProcessInTree(descending, ptr^.Left)
  else
    ProcessInTree(descending, ptr^.Right);
end;
```

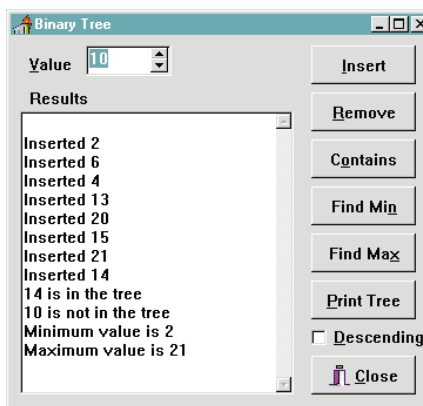


Figure 13 (Top): Recursive binary tree processing in order.
Figure 14 (Left): The example form for our binary tree example.

The examples presented in this article show how recursion can be used to implement various algorithms within Object Pascal. It's a powerful tool when applied to the right sort of programming problem, and frequently produces concise and elegant algorithms as a solution. Fortunately Object Pascal makes it easy to use this tool, without which we might need to perform some quite complex steps to achieve the same results.

Some other examples where recursion can be of great use are in most problems involving fractals, in backtracking algorithms, where a path is followed until it proves fruitless before backing up and trying another path, in traversing tree structures (not just binary ones), and any other algorithm that is defined in terms of itself. In fact, have a look under recursive loop in the *Object Pascal Language Guide*. If you want to practice using recursion, try to program a recursive function that reverses the order of characters in a string. Remember to define the problem in terms of itself and then implement this in the code.

Now if I can just get you back to the start of this article for another look ... ▲

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM-JUNE96\DI9606KW.

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. You can reach him via e-mail at kwood@nla.gov.au or by phone (Australia) 6 291 8070.





DYNAMIC DELPHI

Delphi 1 / Object Pascal

By *Andrew J. Wozniwicz*



DLLs: Part IV

Wrapping Up Dynamic Link Libraries in Delphi

In this series, we've discovered some of Delphi's formidable capabilities to create and use dynamic link libraries (DLLs). In this final installment, we'll cover dynamically loading a DLL, releasing a DLL, using dynamically loaded DLLs, and accessing data in a library. The end of the article features an overview for the use of DLLs for your reference.

Dynamically Loading a DLL — *LoadLibrary*

The keys to explicitly loading a DLL are a pair of standard Windows API functions that are always used in conjunction: *LoadLibrary* and *GetProcAddress*. First, you must issue a call to *LoadLibrary*, which is declared in the *WinProcs* unit, as follows:

```
function LoadLibrary(  
  LibFileName: PChar): THandle;
```

You can use this function by passing it a string value, for example:

```
var  
  ALibrary: THandle;  
begin  
  ...  
  ALibrary := LoadLibrary('dllfirst.dll');  
  ...  
end
```

This example, however, hides a number of difficulties and issues that arise when you attempt to use the *LoadLibrary* function. The first issue is making sure the *WinProcs* unit is in the *uses* clause of the unit or module in which you intend to use *LoadLibrary*. The second issue is that the function takes a single parameter, *LibFileName*, which is of type *PChar*.

So far, you have avoided dealing with the non-Pascal string types, known as *null-terminated strings*. The standard string type was sufficient for most purposes. Note, however, that strings are limited to 255 characters, while null-termin-

ated strings are not and may run up to 64KB characters. A *PChar* is essentially a null-terminated string (the issue of pointers deliberately is avoided here) for all intents and purposes.

Null-terminated strings are native to the Windows API, but Delphi shields you from them most of the time. When you need to make a direct call to an API subroutine, however, there is no "protection" — you are dealing with Windows directly and must use the data types that Windows expects.

A null-terminated string is an array of characters. Unlike Pascal strings, which have the length byte at the beginning, null-terminated strings do not explicitly store their lengths, but instead mark their ends with the null-character (ASCII 0) — hence, their name. A null-terminated string can be much longer than a Pascal string, up to the maximum limit of 64KB.

The *LoadLibrary* function expects you to pass the file name of the library as a null-terminated string. The example in this section looks deceptively simple because it hides the fact that the string passed as the actual parameter is a null-terminated string:

```
'dllfirst.dll'
```

This line is treated by the compiler as the native Pascal string or a null-terminated string, depending on the context. In this case, because

DYNAMIC DELPHI

the function expects a null-terminated string, the compiler ensures that the literal constant is treated as such. This is possible only because the call uses a literal constant. Otherwise, without the additional information from the context in which the literal constant is used, you would not be able to tell whether 'dllfirst.dll' refers to a Pascal string or a null-terminated string. Both look exactly the same when written as true constants.

Typically, however, you would not use literal constants for the file name with the *LoadLibrary* function. After all, you can import the library of interest implicitly by using static constants. The advantage of using *LoadLibrary* is the ability to specify a variable as its parameter, thereby filling the actual value of the string at run time. This is where the incompatibility problem between Pascal strings and null-terminated Windows strings creeps in. The compiler does not accept a Pascal string variable in place of the actual parameter for the *LoadLibrary* function.

Note that you must provide the complete file name in a call to the *LoadLibrary* function. At a minimum, this involves providing the file name and the extension. Because file name extensions other than the default .DLL are possible, no default extension is assumed. You have to explicitly supply the extension, if any, even if it is DLL.

Fortunately, you don't have to worry too much about null-terminated strings to make the *LoadLibrary* call. Just remember that it's not a straightforward Pascal string. Typically, you must translate between a *String* variable in which you likely will have the file name of the library stored, and what the *LoadLibrary* function requires.

Calling LoadLibrary

Without getting into too much detail, **Figure 1** illustrates the steps to issue the *LoadLibrary* call. This code shows how to use the *LoadLibrary* function in a generic situation, when the file name of the library to use for the call is stored as a Pascal string, as it typically would be.

First, the **uses** clause ensures that the required subroutines are visible. You need *WinProcs* to access the *LoadLibrary* function, and *SysUtils* to access *StrPCopy* (the conversion routine that translates between a Pascal string and a null-terminated string). You also need *WinTypes* to use the *HINSTANCE_ERROR* constant defined there when checking for the result of the *LoadLibrary* call.

Then, the necessary variables are declared:

```
var
  AFileName: string;
  ABuffer: array[0..255] of Char;
  ALibrary: THandle;
```

AFileName is the *String* variable in which you store the file name originally. Here, for simplicity, the *AFileName* variable gets its value from a straightforward constant assignment. However, this can be replaced by a more elaborate scheme, such as getting the value from a user, reading it from an .INI file, and so on.

```
uses
  ..., WinTypes, WinProcs, SysUtils;

var
  AFileName: string;
  ABuffer: array[0..255] of Char;
  ALibrary: THandle;

begin
  ...
  AFileName := 'dllfirst.dll';
  ...
  StrPCopy(ABuffer, AFileName);
  ...
  ALibrary := LoadLibrary(ABuffer);
  ...
  if ALibrary <= HINSTANCE_ERROR then
    { There was a problem! }
  else
    { It is safe to use the library! }
  ...
end;
```

Figure 1: Explicitly loading a DLL via a *LoadLibrary* call.

After you have the name of the library file to load, you can proceed with the conversion to a null-terminated string. The *ABuffer* variable serves as the storage for the file name string after the conversion occurs. The *StrPCopy* subroutine converts between a Pascal string and a null-terminated string. The first parameter to *StrPCopy* is the destination buffer where the null-terminated equivalent will be stored. The second parameter is the Pascal string you want to convert.

After the conversion, you are ready to call *LoadLibrary*. The return value of this function is assigned to the variable *ALibrary*, declared as a *THandle*. (We'll discuss this more later.) In a nutshell, it's a "token" through which you can refer to the library after it has been loaded. You will need to check the value of the returned token, however, because a value below the predefined *HINSTANCE_ERROR* indicates an error condition. The last three lines of code determine if the call to *LoadLibrary* was successful.

The result of the *LoadLibrary* function call is a value of type *THandle*. This value is a token, or an "abstract value," that enables you to identify the library to Windows after it has been loaded successfully. It's important to realize that, as long as the *LoadLibrary* call was successful, the numeric value of the handle it returns is of no importance to you directly. You simply supply whatever value *LoadLibrary* returned to other functions, such as *GetProcAddress*, that require it.

The only time you need to actually look at the value returned by *LoadLibrary* is directly after the call, because the return value may indicate an error condition. You can check whether a call to *LoadLibrary* was successful by comparing it with a predefined constant declared in the *WinTypes* unit: *HINSTANCE_ERROR*. If *LoadLibrary* returns a value that is less than or equal to this predefined constant, the call succeeded in actually loading the library, and it's not safe to use any of the functions that need the library to be loaded, such as *GetProcAddress*.

A call to *LoadLibrary* does not necessarily result in the library being loaded from disk. If the library is already loaded (i.e. it's already being used by another application or by another instance of the same application), the DLL's "usage counter" maintained by Windows is incremented. Only one copy of the library itself resides in memory at any time. This, after all, is the reason for having DLLs: to be able to share code.

Releasing a DLL

Assuming that the call to *LoadLibrary* was successful, you need to ensure that you free the library once you no longer need it. This is again different from the situation of implicitly importing the DLL, where Windows itself takes the responsibility for both loading and unloading the library when necessary. Once you have taken over the responsibility to explicitly load the library, you also must ensure that it's possible for Windows to unload it when it's no longer needed.

To tell Windows that your application is no longer interested in the library, you must issue a call to the standard Windows *FreeLibrary* procedure. It's declared inside *WinProcs* as follows:

```
procedure FreeLibrary(LibModule: THandle);
```

As you can see, *FreeLibrary* takes a single parameter, *LibModule*, of type *THandle*, that identifies the library to be released. This is the same "token" handle that *LoadLibrary* returns. In fact, a call to *FreeLibrary* does not necessarily result in the library being unloaded immediately. It all depends on who else is using the same library at the time. If there are other applications also using the library in question, the call to *FreeLibrary* merely decrements a usage counter maintained by Windows. Only after the usage counter reaches zero is the library unloaded.

It's important to remember that if you used *LoadLibrary* to access a DLL, you must use a corresponding *FreeLibrary* to release it. Otherwise the usage counter is never decremented to zero and the library remains loaded even if it's no longer being used. When your program terminates, normally or abnormally, it's your responsibility to issue the *FreeLibrary* call.

Using Dynamically Loaded DLLs

Assuming the call to *LoadLibrary* was successful, you can take steps to retrieve the addresses of the subroutines you want to use from the library. The most convenient way of doing this is by declaring a subroutine type variable, the value of which you fill at run time with the address retrieved from a DLL via a call to *GetProcAddress* (you'll see an illustration of this shortly).

GetProcAddress is declared in *WinProcs* as follows:

```
function GetProcAddress(Module: THandle;
    ProcName: PChar): TFarProc;
```

This declaration takes two parameters. The first, *Module*, is the library handle previously returned from the call to *LoadLibrary*. The second parameter, *ProcName*, is a null-terminated string that contains the name of the subroutine for which you want

```
uses
    WinTypes, WinProcs;

var
    TheLib: THandle;
    FillStr:      function (C : Char; N : Byte): string;
    UpCaseFirstStr: function (const S: string): string;
    LTrimStr:     function (const S: string): string;
    RTrimStr:     function (const S: string): string;
    StripStr:     function (const S: string): string;

begin
    ...
    { Initialize the variables }
    @FillStr      := nil;
    @UpCaseFirstStr := nil;
    @LTrimStr     := nil;
    @RTrimStr     := nil;
    @StripStr     := nil;
    ...
    { Load the library dynamically }
    TheLib := LoadLibrary('DLLFIRST.DLL');
    if TheLib > HINSTANCE_ERROR then try
        ...
        { Retrieve the subroutine addresses }
        if TheLib > HINSTANCE_ERROR then begin
            @FillStr := GetProcAddress(TheLib, 'FillStr');
            @UpCaseFirstStr :=
                GetProcAddress(TheLib, 'UpCaseFirstStr');
            @LTrimStr := GetProcAddress(TheLib, 'LTrimStr');
            @RTrimStr := GetProcAddress(TheLib, 'RTrimStr');
            @StripStr := GetProcAddress(TheLib, 'StripStr');
        end;
        ...
        { Use routines here, almost same as before. e.g.: }
        if Assigned(LTrimStr) then
            S := LTrimStr(' This is fun! ');
            { S now equals 'This is fun! ' }
        S := StripStr(' Teach Yourself Delphi Now! ');
        { If StripStr = nil, an exception occurs and control
          jumps to the finally block; if no exception occurs,
          S now equals 'TeachYourselfDelphiNow!' }
        ...
    finally
        { Release the library once you are done }
        FreeLibrary(TheLib);
    end;
    ...
end.
```

Figure 2: Dynamically loading the sample DLLFirst library.

to obtain the address. (As before, you may need to translate the Pascal string where you stored the procedure name into a null-terminated string required by the *GetProcAddress* call.)

Note that Windows does not give you any way of retrieving any information about the parameter lists and return types of the DLL subroutines you are accessing. In other words, you must know the *signature*, or the declaration of the subroutine you want to use beforehand. You can dynamically retrieve the run-time address of the subroutine in a DLL, but not run-time type information about it.

Loading and Calling

Now take a look at how you would go about dynamically loading and calling subroutines in the DLLFirst library we've developed in this article series. Figure 2 provides a template for using any DLL. Simply substitute the definitions and names specific to DLLFirst in the listing with ones pertaining to the specific library you want to access.

DYNAMIC DELPHI

The code in [Figure 2](#) gives you an idea of what's involved when using an explicitly loaded DLL. The example DLL-FIRST.DLL is used here as an illustration of the steps needed.

The key to successfully using the subroutines in the library is to provide a set of variables that can hold the values of run-time addresses of the library subroutines. In the `var` section, a set of variables is declared, conveniently named the same as the function addresses of which they will be storing.

The library handle, *TheLib*, is retrieved via a call to *LoadLibrary*. A `try..finally` exception-handling block is set up to protect the “working” code from the possibility that the code executed within the `try` block may have failed. If an attempt to call a subroutine from the library within the `try` block results in a failure, an exception is thrown. The `finally` block is guaranteed to execute the *FreeLibrary* call, releasing the resources taken by the library.

In other words, after the initial determination that the library call was successful is made by the `if` statement, the `finally` clause of the `try..finally` block ensures that the library is unloaded properly, even if an exception occurs during the execution of the statements inside the `try` part.

Assuming that the *LoadLibrary* operation was successful, the actual run-time addresses of the subroutines imported from the DLLFirst library module are retrieved.

The `@` operator in front of the variable names is meant to deliberately circumvent the strong type-checking mechanism of Object Pascal and, for a short moment, to treat the procedural variables as if they are pointer variables (variables storing memory addresses of an unspecified type). Otherwise the compiler would complain about the incompatibility between the two sides of these assignment statements. Observe that *GetProcAddress* returns a *THandle* type, while the left-hand side variables are declared as function-typed variables.

The subroutine variables, after the address values are assigned to them, can be used to call the appropriate subroutines. For example, the following line of code appears exactly as if a call to a normal subroutine is being made:

```
S := LTrimStr(' This is fun! ');
```

The function *LTrimStr* is being called and returns a value as before. The interesting point here is that *LTrimStr* is not a function per se, but a reference to a functional variable, the value of which was determined at run time. The difference is that you must check the variable's value for validity, because unlike statically linked or implicitly imported subroutine addresses, the subroutine's address may not be available. The determination as to whether it's safe to make the call is done with this code:

```
if Assigned(LTrimStr) then
```

This `if` statement is an example of the traditional approach to error-checking. A run-time error is prevented by making sure

that the functional variable contains a valid address. A different approach to error-checking is taken with this statement:

```
S := StripStr(' Teach Yourself Delphi Now! ');
```

The new style of programming makes use of the exception-handling mechanism built into Delphi. Remember that the `try..finally` block you set up earlier is in effect here:

```
S := StripStr(' Teach Yourself Delphi Now! ');
```

If the call to *StripStr* is unsuccessful, such as when the value of *StripStr* is `nil`, a General Protection Fault (GPF) exception occurs. Instead of performing any useful action, the control of execution immediately jumps to the `finally` block. The `finally` block traps the exception and, in this case, makes sure that the library is properly unloaded:

```
finally  
    FreeLibrary(TheLib);  
end;
```

After you are done using the dynamic library that you have loaded explicitly, be sure to release it with a call to *FreeLibrary*. The code in [Figure 2](#) ensures that the library is eventually unloaded by making a call to *FreeLibrary*:

```
FreeLibrary(TheLib);
```

Before the call is made, however, the code checks if the library has been successfully loaded in the first place.

The remainder of our article points out another important issue that makes using DLLs different from using regular units: accessing data inside the library.

Accessing Data in a Library

An important point when you are considering implementing DLLs is that there's no way of directly exporting data from them. Unlike in the case of a simple, statically linked unit, where a variable declared in the `interface` section is potentially visible to, and accessible from, any other unit or Pascal module in a project, variables declared inside a library remain “external” and private to that library. The only interface available to the users of the library is the subroutine interface: procedures and functions.

You are free to declare global variables inside a library module, and declaring them is just as easy as doing so in a program module. However, the variables you declare inside a DLL are private to that DLL. You must provide a procedural interface to allow the applications using the library to access the variable (or variables) declared inside it when needed.

Warning! Always remember DLLs are shared resources. Global variables in a DLL are shared across all clients of the DLL. If one client application using the DLL changes its value through a procedural interface to the DLL, all other clients will also see the new, changed value. This may

```

library ExtStr;
var
  AString: string;

function GetValue: string; export;
begin
  Result := AString;
end;

procedure SetValue(AValue: string); export;
begin
  AString := AValue;
end;

exports
  GetValue index 11,
  SetValue index 12;

begin
end.

```

Figure 3: Exporting a string variable from a DLL via a procedural interface.

be useful sometimes, but often is unwanted, unexpected, and may be disastrous. Writing multi-user servers as DLLs is a tricky issue that is beyond the scope of this series.

The library presented in [Figure 3](#) indirectly exports a string variable, *AString*, by providing two access subroutines: *GetValue* and *SetValue*. The code accomplishes the goal of exporting a data element from a library module. The data element “exported” by the DLL, *AString*, is declared early.

The *AString* variable is not visible outside the module, however. To enable applications using the library to obtain and change the value of the variable, two access subroutines are defined:

- The *GetValue* function, to retrieve the current value of the variable.
- The *SetValue* procedure, to change the value of the variable.

These subroutines actually are exported via an **exports** clause, and the client applications can operate effectively on the value of the variable without “seeing” the variable directly. This is similar to the concept of access methods for a class property. In both cases, the access subroutines shield the using code from directly manipulating the value, and may introduce side-effects, as well as validation and checking.


Conclusion

This series has discussed the following issues:

- Like application modules, DLLs are executable modules, but they are not directly executable.
- DLLs make it possible for many running applications, or many instances of the same application, to share code and binary resources.
- DLLs are created in Delphi by replacing the keyword **program** with the keyword **library** in the main Delphi project file, and making some additional changes to the standard project file generated by Delphi.
- The **export** directive makes a subroutine exportable — capable of being exported by a DLL and used by an application external to that DLL.

- The **exports** clause lists the subroutines actually exported from a DLL. All the exported subroutines must have been declared with the **export** directive.
- The most convenient way of accessing the subroutines inside a DLL is by creating an implicit **import** unit, declaring the headers of the subroutines, and binding them to the corresponding routines inside a DLL via the **external** directive.
- There are many ways of binding the subroutines declared as external to the actual subroutines implemented inside a DLL. The subroutines can be bound by their declared or assumed name, or by an ordinal number.
- A *LoadLibrary* standard API function can be used to provide a greater degree of control over when a particular DLL is loaded. There must be a matching call to *FreeLibrary* for each invocation of the *LoadLibrary* function.
- Before you can use the subroutines inside a library explicitly loaded with a call to *LoadLibrary*, you must retrieve their addresses via a call to *GetProcAddress*.
- You can store the values retrieved by *GetProcAddress* in procedural-type variables, which can later be used to call the subroutines.
- You cannot export data elements directly from a DLL. To make data available to applications using a DLL, you must provide a procedural interface, consisting of a *GetXXXX* function and a *SetXXXX* procedure, to retrieve and change the value of a particular variable in question.

This discussion of DLLs has only scratched the surface of the issues involved. Be sure to consult other references when you are considering making heavy use of dynamic linking. The examples given here are just a foundation for knowledge.

Now go build some DLLs! 

This article was adapted from material for *Teach Yourself Delphi in 21 Days* [SAMS, 1995], by Andrew Wozniwicz and Namir Shamas.

The example DLLFIRST.DLL and its associated files are available on the Delphi Informant Works CD located in INFORM\JUNE\96\DI9606AW.

Andrew J. Wozniwicz is president and founder of Optimax Development Corporation (<http://www.webcom.com/~optimax>), a Chicago-based consultancy specializing in Delphi and Windows custom application development, object-oriented analysis, and design. He has been a consultant since 1987, developing primarily in Pascal, C, and C++. A speaker at international conferences, and an early and vocal advocate of component-based development, he has contributed articles to major computer industry publications. Andrew can be contacted on CompuServe at 75020,3617 and on the Internet at optimax@optidevl.com.





NEW & USED

Delphi 1 Utility

By *Robert Vivrette*

Memory Monitor for Delphi

Plug Your Delphi Application Resource Leaks

Programming for an environment such as Microsoft Windows is full of pitfalls. Fortunately, many of the development languages available today (such as Delphi) take great strides to protect us from these hazards. However, a programmer can still get into trouble fairly easily.

One significant problem is resource leakage. All of you who have worked with Windows 3.x or Windows for Workgroups are familiar with the problem. A program uses pieces of memory from two small supplies called the GDI and USER heaps. Each of these is limited to 64KB, so when a program takes some of this memory and doesn't give it back, less is available for the next application. Eventually, enough badly behaved programs bring Windows to its knees, forcing the user to restart the machine.

This has given rise to a multitude of resource monitoring programs that display

the state of each of these memory heaps. Fortunately, Windows 95 takes great strides to alleviate much of the basic restrictions on the use of these memory segments. A determined, ill-behaved program, however, can still wreak havoc.

As useful as these resource monitoring programs are, they are — after all — merely *monitors*. They do nothing to help uncover the programming flaw that generated the leak in the first place. In addition, they typically only monitor the relatively small GDI and USER heaps. Leaks relating to the global memory pool are much more difficult to track.

What would you think of an oil company that expended huge amounts of money to create a device that measured the amount of oil that leaked from improperly designed hull plates of a super-tanker? Waste of money, right? The oil company should spend that money designing better hull plates so the oil wouldn't leak in the first place.

Resource monitoring gauges are much like this. They track the amount of memory a program is leaking after it has already been shipped to thousands, or tens of thousands, of users. It does little to help a programmer fix the problem, but rather indicates to the user when the machine is about to crash.



Enter MemMonD

MemMonD (for Memory Monitor for Delphi) is *not* a typical memory monitoring tool. Rather it is a tool used by Delphi programmers to track down leaks before the application leaves the shop. It allows programmers to see the lines of source code that are causing memory leaks so they can modify the code and correct the problem.

Using MemMonD is about as simple as it gets. Before testing, the application needs to be compiled with the **Map file: Detailed** radio button selected on the Linker page of the Project Options dialog box (see Figure 1). This setting tells the Delphi compiler to generate a map file (*.MAP) during the linking stage that shows (among other things) memory addresses for the various pieces of code in the program.

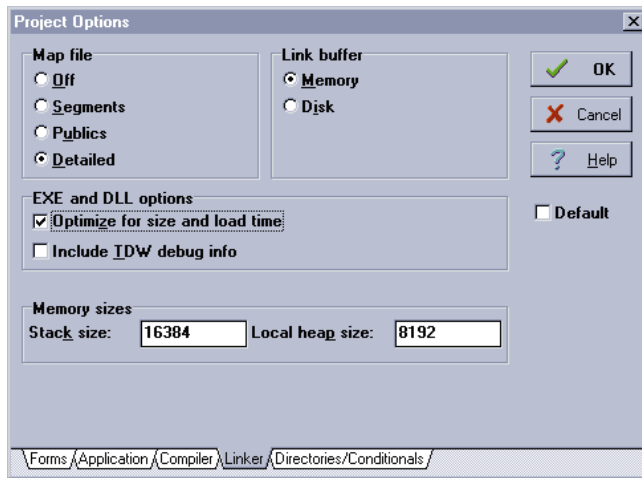


Figure 1: Delphi's Project Options dialog box.

Then it's a simple matter of firing up MemMonD, selecting the application to test, and running that application. Your program will run normally with MemMonD sitting in the background watching what your program is doing. (Its main window even has shifting eyes to indicate that it's at work.)

Exercise all portions of your test program, then shut it down. MemMonD will prepare a report that contains information about global memory and stack consumption, and a list of pointers that were never freed — each with a reference to the line of source code where the pointer was allocated.

It will also catch one of the more difficult bugs to track down in a Delphi program — namely, the use of the *FreeMem* procedure with a different size parameter from that used when the *GetMem* or *AllocMem* procedure was used to reserve the memory.

To demonstrate MemMonD's principal capabilities, let's look at a simple example.

An Example

The example application performs one simple task — and performs it incorrectly. It presents a form with a single

button (see Figure 2). When the button is clicked, a single *TBitmap* component is instantiated in memory:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    Bitmap1 := TBitmap.Create;
    Label1.Caption := 'Bitmap Created';
end;
```

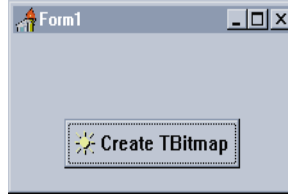


Figure 2: A simple "leaky" demonstration program.

Most of you will recognize the problem here. Aside from the fact that the program does nothing with the *TBitmap*, it also doesn't release it from memory. The object was instantiated, telling Delphi to create various internal structures for the *TBitmap*, yet it is

never removed from memory. Delphi has no way of flagging this type of logic error. My use of the *TBitmap* is syntactically correct — as far as it goes.

To see how MemMonD helps in this situation, let's run this sample program while MemMonD is watching. After launching MemMonD and selecting the program to monitor, you'll see something similar to

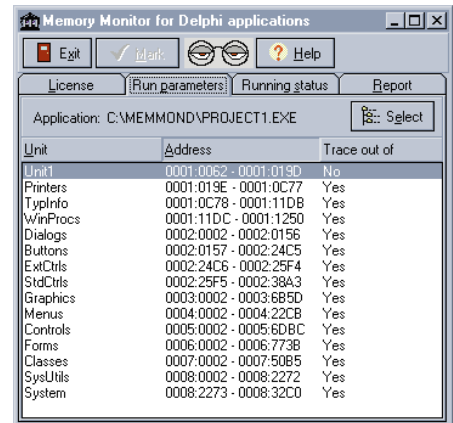


Figure 3: MemMonD at work, displaying each unit's address.

The list box shows all the units that have been compiled into the example project. (Note that the user has complete control over what memory issues will be reported. This is just one "mode.")

After running the sample program, clicking on the button (and creating the *TBitmap*), and then quitting, MemMonD will generate a report that uncovers our problem (Figure 4). As you can see, MemMonD noticed there were 2 pointers to memory, accessing a total of 44 bytes, that were not released. It also indicates that these pointers were allocated in the UNIT1.PAS file on line 29. If we go back to the source for our demonstration application, we see the following on line 29:

```
Bitmap1 := TBitmap.Create;
```

In this case MemMonD has spotted the culprit as being the creation of a bitmap that was never released. To remedy this problem, the programmer would then need to include a:

```
Bitmap1.Free;
```

statement at an appropriate spot in the application.

Keeping Perspective

Obviously, this sample program is about as trivial as you can get. Any programmer worth his or her salt would have spotted this one with both eyes

closed. However, MemMonD is just as capable with the most complex program you can throw at it. I've had the opportunity to test MemMonD with many applications where I work and it has paid for itself many times over.

Many leaks (including this one) could allow an application to run for hours or days with no apparent side effects. It is simply a matter of how much memory is leaking and from where. If it is a leak from one of the limited GDI or USER heaps, you could see nasty behavior quite early. However, if the leak is simply out of the global memory pool, it could be weeks before the program would misbehave.

In our example, the program is leaking 44 bytes every time the button is clicked. My machine has 32 megabytes of memory, so it would likely take quite a few mouse clicks to exhaust the system's available memory. However, as mentioned above, programs are never as simple as this example. If this leak were inside a loop that created hundreds of bitmaps, it would constitute a major resource leak.

Interestingly, MemMonD also points out some memory leaks in the Delphi VCL. Granted these leaks are minor, and Borland has commented that these leaks are negligible and were purposely left in for performance reasons. However, the registered version of MemMonD comes with patches that you can insert into the VCL to correct them if you wish.

But Wait, There's More

What I have described here are only the basic capabilities of MemMonD. It's also capable of providing profiling statistics on the most time-consuming routines, the most stack-consuming, or the most frequently-called routines. These are very handy features that allow you to see where your program is spending most of its time. Armed with this information, you can optimize those sections of code. In addition, when you register

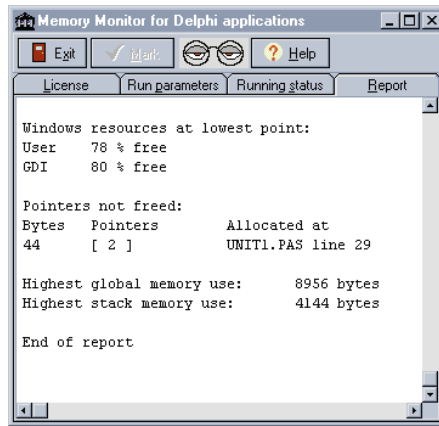


Figure 4: MemMonD identifies the problem — two pointers that were never freed — and the line of Object Pascal code that created the pointers.

MemMonD, you get a user license file that enables its stack check hooking and block overwrite options.

Currently, MemMonD only works with the 16-bit version of Delphi. However the author has indicated that a Delphi 2 compatible version is in the works and should be available by the time you read this.

MemMonD was created by Per Larsen (CompuServe 75470,1320) and costs US\$49 for a single-user license, US\$147 for an unlimited site-license. There is no written documentation, but the included Help files are sufficient to learn how the package works. There is a shareware version of the package (without some of the extended capabilities) that can be found as MEMMND.ZIP in the "Debugger/Tools" library section of Borland's Delphi CompuServe forum (GO DELPHI). Users can register the software via SWREG (ID 9253) and will obtain, by e-mail, a file that unlocks the extended capabilities of the package.

High Marks

MemMonD gets very high marks from me. It has already saved me untold effort — and embarrassment — tracking down memory leaks in some of the applications I've been working on. (Hmmm. Should I admit that I have written a leaky program?)

I honestly feel that no serious Delphi programmer should be without MemMonD. I eagerly await the 32-bit version — and have expressed this to the author on numerous occasions. It already has a permanent place in my suite of utilities, and when I die, they will have to pry it from my fingers. ▲

Robert Vivrette is a contract programmer for Pacific Gas & Electric and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached on CompuServe at 76416,1373.

INFORMANT FACT FILE

MemMonD detects and reports resource leaks in Delphi applications. It's also capable of generating profile statistics for the most time-consuming routines, the most stack-consuming, or the most frequently-called routines. The registered version features stack check hooking and block overwrite options. MemMonD should be in every Delphi developer's tool kit.

Price: Single-user license US\$49; unlimited site-license US\$147. A shareware version is available as MEMMND.ZIP in the "Debugger/Tools" library section of Borland's Delphi CompuServe forum (GO DELPHI). It's also available on the *Delphi Informant* Companion Disk and for download from the Informant Web page (<http://www.informant.com>) or the Informant Forum on CompuServe (GO ICGFORUM). Delphi 3rd Party Library. File name: MEMMND.ZIP.

Developer:
Per Larsen
CompuServe: 75470,1320



TEXTFILE



Developing Custom Delphi Components

If components are the heart of Delphi, Ray Konopka finds himself in the position of a pioneering surgeon.

Until now, there has been a paucity of substantive information about developing custom components. The Borland documentation is helpful, but woefully incomplete. And early third-party books fail to discuss the issue in any depth. Additionally, while the source code for the Visual Component Library (VCL) is invaluable, it is no substitute for a solid grounding in the basics.

Ray Konopka solves this problem — and many more — in his *Developing Custom Delphi Components*, published by Coriolis Group Books.

While not flawless, *Components* is the first book to address the weighty subject of Delphi component development, and it tackles the challenge with precision and focus. It's obvious — sometimes painfully so — that some books include topics solely so they can be included in the table of contents. Not so with *Components*. Even sections devoted to relatively minor subjects, such as exception handling and debugging, have considerable meat to them and stand up on their own. Add to this Konopka's easy-to-read writing style and the result is a book that any serious Delphi developer should add to his or her library.

Components begins by discussing the basics developers should know before building components. Included in

this section is an excellent overview of Delphi's object model. The author addresses many key object-oriented programming (OOP) issues — such as virtual methods, method pointers, class methods, virtual constructors, and run-time type information — that I haven't seen discussed in other third-party Delphi books.

Unfortunately, *Components* doesn't cover them in as much detail as one would ideally like. Additionally, if you are new to Object Pascal or OOP terminology, you may want to have a second reference handy. Terms such as *forward class*, *virtual*, and *abstract* are probably not adequately defined for the uninitiated.

The second part of the book introduces the reader to Delphi's component architecture. Konopka spends a chapter dissecting a component and follows with a look at the Visual Component Library. The actual process of building a component is then detailed in a cradle-to-grave manner, starting with creating the unit file with the Component Expert, writing the code, installing it onto the Component Palette, and concluding with a test of the Design-Time Interface.

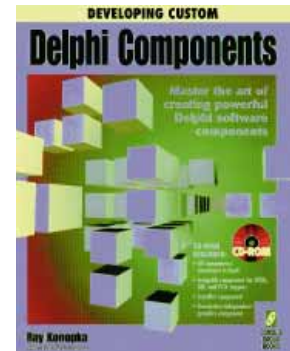
Part three — which for many readers will represent the core of the book — builds on this initial discussion in order to guide the reader through the process of building components in all the major categories: graphical, dialog, non-visual, and data-aware. It also covers related

issues such as building “wrapper” components and encapsulating multiple controls in a component. The final part of the book examines advanced topics related to component development, most notably a chapter on property editors and component editors.

Perhaps the most interesting chapter in the book for me was the one on building data-aware business components. This chapter is the first discussion I've seen on developing persistent business objects in Delphi. Konopka shows a technique that allows you to attach a “business component” to the normal data link (i.e. `TTable->TDataSource->TDBEdit`). This allows you to work with business objects in code, store their data persistently in a database, and even display object data using built-in data-aware controls.

While *Components* is the first book to address business objects in Delphi, the solution proposed is limited to simple objects, and would not stand up to a complex object model without modification. Nonetheless, at a minimum, this discussion provides a jumping off point for development of a more intricate real-world solution.

Developing Custom Delphi Components comes with a CD-ROM filled with source code for all components discussed. Among the many examples are: progress and slider bars, an application launcher, an e-mail mailer, a data status control, and an employee business component.



It should be noted that the book was written using Delphi 1, but that all example components compile without problem under Delphi 2. An appendix includes information on moving to the 32-bit version of Delphi.

In the past 15 months, the Delphi developer community has been forced to deal with a lack of information on advanced programming subjects. Now Ray Konopka's *Components* serves as a pioneering book to address this recognized need. However, what ultimately makes *Developing Custom Delphi Components* a winner isn't that it's the first to cover these advanced issues, but that it also covers them comprehensively, with clarity and insight.

— Richard Wagner

Developing Custom Delphi Components by Ray Konopka, Coriolis Group Books, 7339 East Acoma Dr., Suite 7, Scottsdale, AZ 85260, (800) 410-0192, <http://www.coriolis.com>.

ISBN: 1-883577-47-0
Price: US\$39.99
585 pages, CD-ROM



A Zero Sum Game?

If Mark Twain were alive today, and in our line of work, he would perhaps quip about Windows 95 and Windows NT by saying “reports of their death have been greatly exaggerated.” At various times over the past few years, each has been touted as being both the operating system (OS) of the future and as a failure. Andrew Schulman wrote in *Unauthorized Windows 95* [IDG Books, 1994]: “Products such as NT...speak to too small a niche to be interesting” and followed with: “Windows 95 will be the standard desktop OS for the next five years.” Recent columns now say the opposite, painting a gloomy picture for Windows 95 as a “home” operating system. If you believe everything you read, you may be quite confused. Forget the hyperbole. Neither OS is perfect for everyone, but each has convincing reasons for its use.

A Remarkable Similarity. We’re tempted to choose an operating system like we choose a spouse: you can only pick one. Microsoft is altering this age-old trend on the desktop by minimizing the differences in the two platforms for both users and developers. First, the popular Windows 95 user interface is now available in Windows NT 4.0 (Shell Update Release), giving users the same “look and feel” across platforms. Second, since applications — the lifeblood of any operating system — are designed for a specific environment, ISVs have always had to select the platform on which to put their development efforts. Quite ingeniously, Microsoft gently coerced vendors to support both Win32 platforms by requiring Win95 Logo applications to work on both systems. Therefore, although Windows 95 and Windows NT have fundamentally different architectures, they can be remarkably similar for both users and developers.

Making a choice. Even if both can coexist in the marketplace and in your company, which one is right for you and for users of your software? Let’s look at four environments most Delphi developers have to be concerned with — development, corporate, mobile, and small office/home office (SOHO) — and see how Windows 95 and Windows NT rate in them.

Development. Probably the single most important environment for you personally is the one on which you develop software. For those of us developing 32-bit Windows applications, there is no contest; everything

significant to developers is in Windows NT: stability, crash protection, preemptive multitasking capabilities (even with Win16 apps), and Windows 95 UI compliance. Developers will typically face the least number of hurdles in terms of hardware constraints, since many of you who use Delphi 2 already have a Pentium with at least 24MB of RAM. **Grade:** Windows NT (A+), Windows 95 (B-).

Corporate. If perception is reality, Windows 95 is doomed for the corporate desktop. While at one time it seemed inevitable that Windows 95 would be the next standard for the corporate desktop, today’s conventional wisdom says that Windows NT 4.0 will own this market in 12 months. Slower-than-expected corporate sales, Windows 95 bashing in the trade press, and an incoherent message from Redmond have left many companies holding off on Windows 95. Robustness, security, and client/server stability are proving key factors in convincing many medium- to large-sized companies to opt for Windows NT. **Grade:** Windows NT (A), Windows 95 (B).

Mobile. Which OS should reign on the notebook? Neither is perfect, so your decision depends fundamentally on where you stand on the “Ease of use vs. Security” issue. Touting strong PC card support, hot docking, and advanced power management, Windows 95 is perhaps the best OS ever created for the mobile user. However, many companies deploying mobile work forces have a more important concern: security. Letting sensitive

and proprietary information leave the confines of a secure office environment is a notion that gives most corporate MIS managers nightmares. With Windows 95, you have no way to fully protect data on the notebook. However, NT’s secure NTFS file system prevents users from gaining access to data unless they have logged into NT itself. **Grade:** Windows NT (B-), Windows 95 (B-).

SOHO. For most users in the SOHO category, Windows 95 is the only one of the two that makes sense. Critical factors, such as performance and Plug and Play, help get the job done quicker with the least amount of effort. In this “Just Do It” environment, you will want Windows 95. **Grade:** Windows NT (B-), Windows 95 (A).

Making a decision ... for now. In 1996, choosing a desktop operating system need not be an either/or proposition. The right decision depends on the context. For now, it seems clear that Microsoft is comfortable providing a two-tiered solution. But whether Redmond’s dual OS approach is short- or long-term remains unclear. I suspect the answer depends largely on the market’s ability to accept the notion that two desktop platforms can — in fact — coexist and complement each other. ▲

— Richard Wagner

Richard Wagner is the Chief Technology Officer of Acadia Software in Boston, MA. He welcomes your comments at rwagner@acadians.com.

