WHITE PAPER

Delphi™

Migrating from Visual Basic® to Delphi™

An Overview for Programmers and Developers

by Mitchell C. Kerman

Table of Contents

Introduction

The Linux® operating system (OS) has recently enjoyed extraordinary amounts of media coverage. It has gained almost fanatical acceptance as an alternative to Microsoft® Windows®. As a result, programmers and developers worldwide have been investigating how they will develop systems for their end-users and customers that take advantage of the many benefits of Linux. Borland® Delphi™ does just that. Delphi has long been recognized as being one of the top application development tools for Windows. With Inprise's Kylix™ Project ("Delphi for Linux"), developers using Delphi are on the fast path to cross-platform Windows/Linux development.

Visual Basic® (VB), the flagship development product of Microsoft Corporation, is its most popular development platform for the Windows OS. VB is a Rapid Application Development (RAD) tool that traces its roots to the BASIC language. Currently, over seven million developers use VB worldwide. Delphi is also a RAD tool, and Visual Basic developers will find it to be remarkably similar to VB. Both tools enhance programmer productivity, resulting in shorter application development times. The key difference, however, is that versions of Delphi will be available for both Windows and Linux.

What is Linux?

Technically speaking, the name Linux only refers to the kernel, the core part of the OS. However, most people (and this paper) commonly use Linux in reference to the entire OS and its

Borland®

packaged applications, as an alternative operating system to Microsoft Windows or MacOS.

Linux is a freely-distributed Unix-like operating system originally created by Linus Torvalds with the assistance of programmers, hobbyists, and computer enthusiasts around the world. It was originally designed in 1991 for Intel's 80386 microprocessor, but it now runs on a variety of hardware, including Alpha, SPARC, PowerPC, and Intel's complete family of x86 microprocessors.

Linux is a modern OS that offers the following features:

- 32-bit architecture
- Preemptive multitasking
- Protected memory
- Multi-user support
- Rich networking support, including TCP/IP

A continually growing number of web sites and Internet Service Providers (ISPs) rely on Linux as their server operating system. Linux is also the development platform of choice for many C and C++ programmers around the world. Furthermore, Linux runs all of the applications expected of a typical Unix server. Just to name a few, these applications include:

- Web servers (e.g., Apache)
- Mail servers (e.g., Sendmail)
- Database servers (e.g., Oracle and Informix)
- Windows and window managers (e.g., X-Windows, GNOME, and KDE)
- Office productivity suites (e.g., Applixware, StarOffice, and KOffice)

Linux is distributed under the GNU General Public License (GPL), meaning that the source code for Linux is freely available to everyone. Anyone can modify the code, provided that the modifications are also freely distributed with the source code.

Making the Linux source code open to everyone (i.e., open source) offers several advantages:

- **Flexibility**. Linux is easy to customize, enabling it to operate on a variety of platforms, from handhelds to clusters of servers working in concert.
- **Reliability**. Linux has been thoroughly debugged, and each new version of the operating system is rapidly reviewed and tested by thousands of programmers worldwide.
- **Economy**. The startup cost to use Linux is extremely low in comparison to other operating systems. There are no licenses or associated fees. Customer support is available from a legion of open source programmers and commercial vendors.

Linux is truly a phenomenon of the Internet. The Internet is the means by which Linux was given birth and continues to grow by providing the necessary collaborative environment required for its development. Programmers around the world can write and share their code for improvements and additions to Linux.

Many development tools are currently available for Linux. However, there is a wide disparity in terms of their features, speed, cross-platform capabilities, and cost. Several popular Linux programming tools and tool providers are described below:

- **GNU: GCC/EGCS**. These are open source development tools that have served Linux well in the development of the OS and environment. As open source tools, they are freely available across the Internet. They do hold a tremendous starting market share, and most Linux tools have been written in GCC/EGCS. Much like Code Warrior (below), these tools have no RAD, Internet, database, or graphical user interface (GUI) development capabilities. Without these capabilities, they do not meet the application needs of the Enterprise. Also, the development of large-scale projects is complex and time-consuming since these tools are C/C++ based.

- **Cygnus: Code Fusion™ for Linux**. Cygnus created a Linux integrated development environment (IDE) for the GNU tools such as GCC. Released in August 1999, Code Fusion was one of the first tools for the Linux market. As with the GNU tools and Code Warrior, Code Fusion's weaknesses are in the areas of RAD, Internet, Database, and GUI capabilities. Cygnus has a strong reputation in the open source community and includes the latest Intel Optimizations in its tools, but the company has little brand recognition outside of its own user base.
- **Metrowerks: Code Warrior**. Code Warrior was the first commercially available Linux development tool, initially released in May 1999. It allows for multi-platform development with a product line that includes Windows, Solaris™, Linux, BeOS®, Palm®, WinCE, and Java®, but it holds the smallest market share for each of these OSs. Additionally, Code Warrior is not a RAD tool, and it contains no GUI or database capabilities.
- **Microsoft**. At this time, Microsoft has not announced any plans to enter into the Linux development tools arena. Since Microsoft uses its development tools to promote the Windows OS, it is unlikely that Microsoft will ever enter the Linux tools market.
- **Borland/Inprise**. Borland/Inprise currently offers one Linux development tool, Jbuilder™, and will soon have two more available, C++Builder™ and Delphi. JBuilder is Borland's Java development environment. At present, it is available under Windows, Linux, and Solaris. JBuilder is also a fully scalable database development tool that supports Java 2. C++Builder is a C++ development platform, and Delphi is a popular RAD development tool. C++Builder uses the C++ programming language, whereas Delphi uses Object Pascal. Both tools share a common IDE. This IDE is very intuitive, and much of your knowledge of VB's IDE directly applies to this one. Windows versions of both tools are available now, and the Linux versions are forthcoming. Kylix is Borland's development effort to port both C++Builder and Delphi to the Linux world.

Benefits of Delphi

Now that you have been introduced to the Linux OS, its features, and the various Linux development tools, let's turn our attention back to VB and Delphi. Compared to VB, Delphi offers a multitude of benefits, such as:

- **Cross-platform development**. As previously mentioned, Delphi is currently available for Windows, and the Linux version will be released very soon. The same code can be used under both OSs, but it may require minor modifications due to inherent OS differences.
- **Superior development environment**. The Delphi IDE provides all of the functionality expected of a RAD tool and more. The environment is intuitive and easy to use. Furthermore, the IDE is flexible, allowing the programmer to customize the environment to suit his needs and preferences.
- **Powerful components and controls**. Similar to the VB Toolbox, Delphi contains a Visual Component Library (VCL) of commonly used components and controls. The number and type of components included in this library depend upon which edition of Delphi is being used (Standard, Professional, or Enterprise). All library components are written in Object Pascal. Thus, the programmer has the ability to modify and extend this library.
- **True object-oriented programming**. While Microsoft claims that VB is object-oriented, we (as educated programmers) know that it is really just object-based. True object inheritance and polymorphism are unavailable in VB. The object model in Delphi is complete, providing encapsulation, inheritance, and polymorphism.

- **Pointers and dynamic variables**. VB does provide for dynamic variables, but it does not allow explicit pointer variables. How many times has this problem come between you and more efficient code or cleaner data structures? "Gee, we really need a tree structure to represent this data properly." While more adept VB programmers can overcome this dilemma by using VB object variables (which are really implicit pointers anyway), many programmers just work around the problem at the expense of algorithm and memory efficiency. Delphi's Object Pascal offers dynamic variables and explicit pointers, effectively dissolving most of your algorithm efficiency and data structure woes.
- **Promotes sound programming practices**. We now come to my pet peeve. As a textbook author and introductory programming instructor, one of my greatest challenges is to teach students to *always* declare their variables. VB gives the programmer the option of not requiring explicit variable declarations (similar to its predecessor, BASIC). "Please make sure that 'Option Explicit' appears at the top of your VB code," I would tell my students. Perhaps you do not agree with me on this point, but try to remember when you first learned how to program. While this option proves extremely powerful and time-saving for expert programmers, it provides just enough rope for novice programmers to hang themselves. Explicitly declaring variables encourages self-commenting variable names, improves code readability, provides greater control over memory space requirements, and avoids confusion between variables of the same name in different portions of the source code. Since Object Pascal always requires explicit variable declarations, it promotes this sound programming practice. Additionally, Object Pascal's syntax encourages structured and modular programming.
- **Strong typing rules**. Akin to sound programming practices, we come to a comparison of the language typing rules. VB is a weakly-typed language. For instance, a variable of data type Double (a double-precision floating-point number) can be assigned to a variable of type Integer without repercussion (or is it?) VB will automatically convert the double-precision value to an integer value. This automatic conversion is not without its problems: Does VB round or truncate the floating-point portion? Do you remember? For positive values, the answer is that VB rounds: It rounds down to the next lowest integer for floating-point portions less than 0.5 (essentially, truncating the floating-point portion) and it rounds up to the next highest integer for floating-point values greater than or equal to 0.5. Of course, we can avoid this problem and simultaneously improve the readability of our code by using a data type conversion function, such as VB's Fix or Int functions. Unlike VB, Delphi's Object Pascal is strongly-typed. A double-precision value cannot be assigned to an Integer variable without first performing the necessary data type conversion. Thus, the problem is entirely avoided.

These and other benefits will be explored in greater detail in the pages that follow.

Purpose of this Paper

This paper is written for programmers and developers who are familiar with VB and would like to learn more about Delphi. While it compares product and language features of both packages, it is by no means a complete comparison. The paper assumes a basic knowledge of RAD, including GUI design concepts, component properties and methods, and event-driven programming. Students, beginning programmers, and experienced programmers alike will benefit from reading the remainder of this document.

In the pages that follow, VB and Delphi are compared and contrasted. This comparison exists at four levels: the IDE, programming language, built-in debugger, and application deployment. Again, rather than act as a complete reference manual, this paper guides those familiar with VB through the process of learning Delphi by leveraging their existing knowledge.

Integrated Development Environment

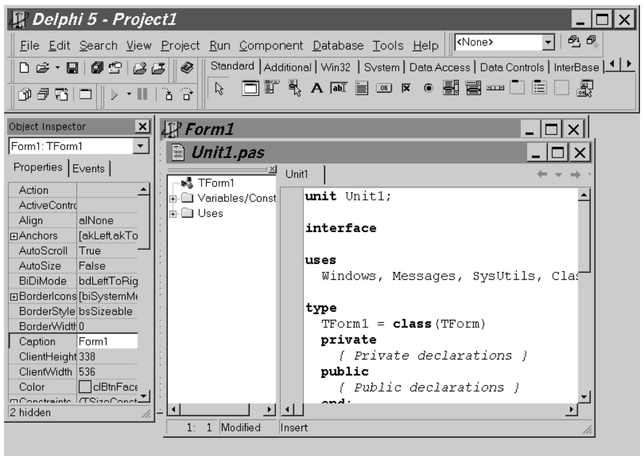
VB developers find a comfortable familiarity with Delphi's IDE. Many of the menus, toolbars, and windows have a design and purpose similar to those of VB. Delphi contains all of the tools that are required of modern RAD environments, and it is clear that these tools were created and organized with a great deal of forethought and effort on the part of Delphi's design team.

Both VB and Delphi contain windows with similar names and functionality. For instance, both contain windows in which you can modify control properties. Visual Basic is a Multiple Document Interface (MDI) development environment; all of its windows are fully contained within the main application window. Delphi, however, is a Single Document Interface (SDI) environment where all windows are free-floating. The following paragraphs introduce the elements of the Delphi IDE and compare and contrast its windows with their VB counterparts.

The default Delphi IDE consists of the following windows:

- A menu bar
- Six toolbars:
  1. Standard toolbar
  2. View toolbar
  3. Debug toolbar
  4. Custom toolbar
  5. Desktops toolbar
  6. Component Palette
- Smaller windows:
  1. Form window
  2. Object Inspector window
  3. Code Editor window
  4. Other windows

Figure 1 shows the default Delphi IDE layout.



Action	
ActiveContr	
Align	alNone
⊞Anchors	[akLeft,akTo
AutoScroll	True
AutoSize	False
BiDiMode	bdLeftToRig
⊞BorderIcons	[biSystemMe
BorderStyle	bsSizeable
BorderWidth	0
Caption	Form1
ClientHeight	338
ClientWidth	536
Color	clBtnFace

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Cla

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

Figure 1. *Delphi's default Integrated Development Environment*

Menu Bar

Like VB, Delphi contains a menu bar. The Delphi menu bar is a typical drop-down menu. Many of the menu options can be accessed directly through the shortcut key combinations that are listed on the right-hand side of the drop-down menu. The menu bar offers all of the functionality required for a developer to create an application. The menu bar and a sample drop-down menu appear in Figure 2.

Toolbars

Toolbars contain icons that allow quick access to common tasks. VB contains four separate toolbars that divide and organize tasks according to their purpose. These include the Standard, Debug, Edit, and Form Editor toolbars. By default, VB displays only the Standard toolbar. Delphi has six separate toolbars, all of which are displayed by default.

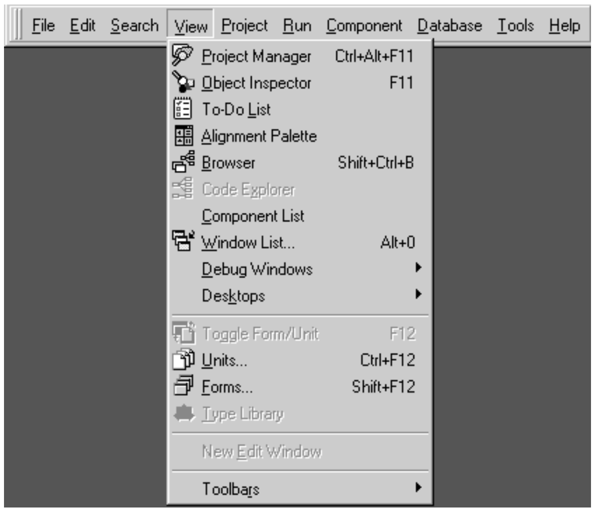


Figure 2. *Delphi menu bar with an activated drop-down menu*

Both VB and Delphi allow you to toggle the visibility of the various toolbars through the View menu. Alternatively, right-clicking on the Standard toolbar in VB opens a pop-up menu listing the available toolbars. This same action opens a similar pop-up menu in Delphi. Additionally, both VB and Delphi toolbars are customizable.

Standard Toolbar

The Standard toolbar (Figure 3) contains icons for common tasks, such as opening, saving, and creating Delphi projects and associated files.

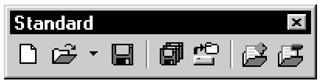


Figure 3. *Standard Toolbar*

View Toolbar

The View toolbar shown in Figure 4 contains icons for creating new forms, viewing forms and code units, and toggling between a form and its code unit. This toolbar allows you to quickly switch between windows in the Delphi IDE.

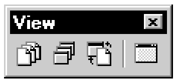


Figure 4. *View Toolbar*

Debug Toolbar

As in VB, the Debug toolbar (Figure 5) is used for interactive testing and debugging of your programs. It provides quick access to several Delphi debugger commands that are available on the Run menu. Like the VB debugger, the Delphi debugger is a design-time utility. It can only be used inside of the Delphi development environment while you are working on the source code.

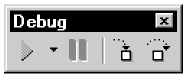


Figure 5. *Debug Toolbar*

Custom Toolbar

Figure 6 shows the Custom toolbar. By default, this toolbar contains a single button to access the Delphi online help facility.

Figure 6. *Custom Toolbar*

Desktops Toolbar

VB opens with all of its windows and toolbars in their previous positions (i.e., the last positions where they were located). The programmer does not have the ability to create several different desktop layouts. Conversely, a programmer can customize Delphi's desktop settings using the Desktops toolbar shown in Figure 7. This toolbar contains a pick list of the available desktop layouts and allows the programmer to load and save different layouts. A desktop layout includes the display, sizing, docking, and placement of windows in the IDE. A selected layout remains in effect for all projects and is used the next time Delphi is started. Again, VB has no equivalent of this toolbar.

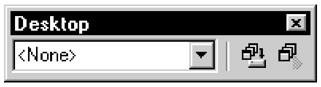


Figure 7. *Desktops Toolbar*

Component Palette

In VB, the Toolbox houses all of the ActiveX controls that are available to the current project. The equivalent window in Delphi is the Component Palette shown in Figure 8.

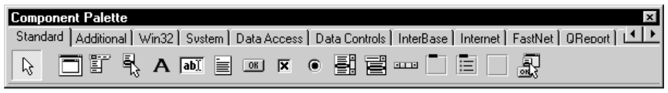


Figure 8. *Component Palette*

The first difference between VB's Toolbox and Delphi's Component Palette is that the Component Palette is tabbed. To alter this tab layout, right-click on the Component Palette and select Properties from the pop-up menu. This opens the Palette Properties window which allows you to customize the Component Palette. By default, the VB Toolbox is not tabbed. However, a programmer may customize the Toolbox and add tabs to organize the controls for quick and easy recognition, similar to Delphi's Component Palette.

Another difference between the two windows is that the VB Toolbox contains only the controls that are available to or used by the current project. On the other hand, the Component Palette always contains all of the controls. Furthermore, when you compile a VB application, each of the ActiveX® controls is still separate from the executable file, whereas Delphi compiles the required controls into the executable file.

Delphi also supports the use of ActiveX controls. When an ActiveX control is used in an application, it is "wrapped" in a set of code that allows is to be placed on the Component Palette and used within the Delphi IDE.

Smaller Windows

Smaller windows in Delphi's default IDE include the Form window, Object Inspector window, Code Editor window, and other windows.

Form Window

The Form window in Delphi (Figure 9) looks and acts like the Form window in VB. The major difference is the unit of measure. Instead of using twips like VB, Delphi uses pixels.

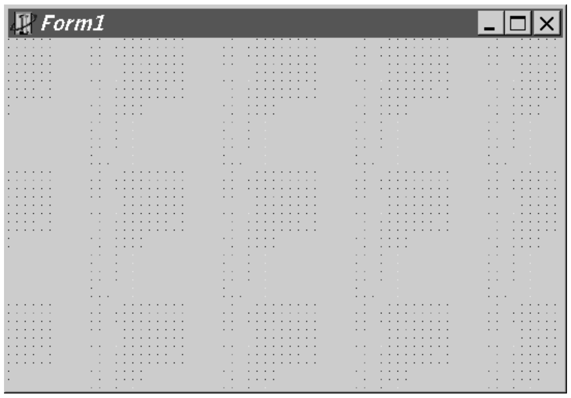


Figure 9. *Form Window*

As in VB, the grid dots on the form are used to align and size your controls. To change any of the Form designer's options, select Tools|Environment Options… from the menu bar and left-click the Preferences tab. The Form designer frame under this tab allows you to change the options summarized below:

- Display Grid — Controls whether or not the grid dots are displayed.
- Snap To Grid — Directs the alignment of the controls. When activated, all corners of the controls are aligned to the grid dots.
- Show Component Captions — For non-visual components, displays the name of the component underneath it on the form designer.
- Show Designer Hints — As you are sizing or moving a control with this option active, the size or position is displayed as a tooltip hint.
- New Forms As Text — Designates whether newly created forms are saved in text or binary format.
- AutoCreate Forms — Determines if new forms are automatically created when the application executes.

- Grid Size X, Grid Size Y — Determines the number of pixels between the grid dots.

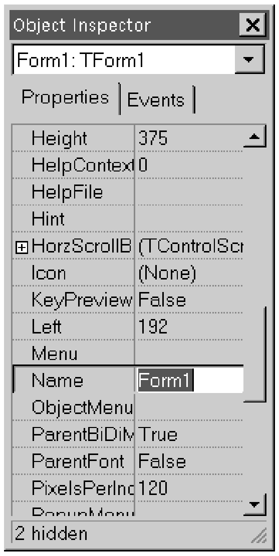
Like VB, Delphi offers several methods of placing a control on a form. First, double-clicking the desired control on the Component Palette places a control of the default size in the center of the form. A second method is to left-click the control on the Component Palette and then left-click on the form. This places a control of the default size on the form with its top left corner aligned to the location that you clicked. Lastly, you may single-click the control on the Component Palette and then click-and-drag to place this control on the form. This method allows the programmer to directly specify the size and position of the control.

Multiple controls of the same type can be placed on the form by holding down the shift key while selecting the control from the Component Palette. Once the control is selected, you can place controls of this type on the form by using one of the last two methods described above.

Object Inspector® Window

Figure 10 displays the Object Inspector® window. Delphi's Object Inspector is closely related to the Properties Window in VB. Both display a list of the available design-time properties for the currently selected object. By default, the Object Inspector displays this list alphabetically. If you are more accustomed to viewing properties by category, Delphi can accommodate you. Simply right-click the Object Inspector window and select Arrange|by Category from the pop-up menu.

There are four basic types of object properties in Delphi: Simple, Enumerated, Sets, and those containing property editors. Simple properties allow you to directly enter a property value by using the keyboard. Enumerated properties allow you to select from a valid list of property values. For instance, the BorderStyle property of a form is an Enumerated property. Set type properties are the only properties that allow you to assign multiple values. An example Set type property is the Style property within an object's Font property. Possible values for the Style property include Italics, Bold, Underline, Strikeout, or any combination of these values. As in VB, those properties that use Property Editors have ellipses (three dots) on their right-hand side in the Object Inspector. Left-click the ellipses to activate the Property Editor.



Height	375
HelpContext	0
HelpFile	
Hint	
⊞HorzScrollB	(TControlScr
Icon	(None)
KeyPreview	False
Left	192
Menu	
Name	Form1
ObjectMenu	
ParentBiDiM	True
ParentFont	False
PixelsPerInc	120

Figure 10. *Object Inspector Window*

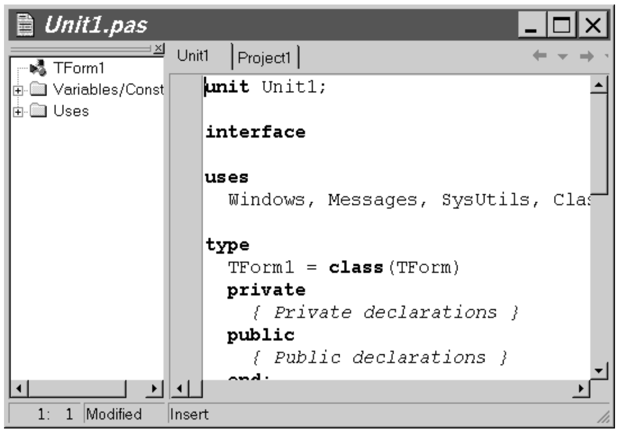
The Object Inspector not only displays the available design-time properties for an object, but it also contains a tab listing all of the events to which the object can respond. To see a list of all available events for an object in VB, you have to go to the Code Editor, select the object from the Object drop-down list, and then select the event from the Procedure drop-down list.

With Delphi, you can have multiple controls (or even different events) call the same event-handler. After writing the event-handler, use the Events tab of the Object Inspector to select this same event-handler for multiple controls (or different events). The drop-down list in the Object Inspector shows all event-handlers that have the same parameter list. You see that Delphi is very flexible and powerful in regard to objects and event-handlers. In order to do something similar in VB, one event-handler must call the other.

Code Editor Window

The VB Code Editor opens each module in a new window. In Delphi, the Code Editor is a single window as shown in Figure

11. This window contains a tab for each opened unit, or module. A word of caution: All too often, you will be tempted to close the Code Editor window after you finish editing your code. When you close a unit in Delphi, you are also closing the form that uses it. To close a single unit, and subsequently the form, right-click its tab in the Code Editor and select Close Page from the pop-up menu.



```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Cla

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

Figure 11. *Code Editor Window*

Delphi's Code Editor uses a color-coding similar to the VB editor. The colors can be customized by selecting Tools|Editor Options… and clicking the Colors tab. Then, select the element whose color you wish to change. Choose a color with the left mouse button to change the foreground color and the right mouse button to change the background color. You can also choose to have the element displayed with a bold or an italic font.

The keyboard shortcuts for the Code Editor include the standard Windows navigation keys:

Key	**Function**
Home	Beginning of the line
End	End of the line
Ctrl+Home	Beginning of the unit
Ctrl+End	End of the unit
PgUp	Previous screen
PgDn	Next screen
Ctrl+PgUp	Top of the screen
Ctrl+PgDn	Bottom of the screen

Similar to VB's IntelliSense® technology, Delphi contains a set of five tools known as Code Insight to aid the developer:

1. *Code Completion* displays a list of available data types when you declare a variable or a list of properties and methods when you use an object. As you type the data type, property, or method, Delphi performs an incremental search of the drop-down list. By default, this list is sorted by scope. To display it in alphabetical order, right-click the drop-down list and select Sort by Name from the pop-up menu. Once you have located the item that you wish to use, press the Enter key to select it and place it in your code.
2. *Code Parameters* displays a dialog of the names and types of the parameters for a function, method, or procedure. Thus, you can view the required arguments for a function, method, or procedure as you enter it into your code.
3. *Code Templates* is the most useful feature for your migration from VB to Delphi. Code Templates provides syntax templates for basic code constructs. Pressing Ctrl-J activates this feature and displays a pop-up menu of the available templates. Additionally, you can type in the beginning of a statement and then press Ctrl-J. If Delphi can resolve the statement, it will fill in the code with the applicable template. If Delphi is unable to resolve the statement, it displays a list of templates that most closely match the statement. To modify or add code templates, select the Code Insight tab under Tools|Editor Options…
4. *Tooltip Expression Evaluation* displays the value of a variable or expression as tooltip text during interactive debugging.
5. *Tooltip Symbol Insight* displays declaration information for any identifier in the Code Editor. A pop-up window displays the kind identifier (procedure,

function, type, constant, variable, unit, etc.) and the unit file and line number of its declaration.

Each open unit in Delphi has a separate tab in the Code Editor. If the unit that you wish to edit is not open, select either View|Units or View|Forms from the menu. The View|Units option displays a list of all of available units in the project. Similarly, View|Forms lists all available forms in the project. Opening a form opens its corresponding unit.

Another navigation feature in Delphi's Code Editor is bookmarks. Delphi permits ten (10) bookmarks in the Code Editor, numbered zero (0) through nine (9). To toggle a bookmark, position the cursor on the desired line of code, and then press Shift-Ctrl-*number*, where *number* is one of the number keys zero (0) through nine (9). To jump to a numbered bookmark, press Ctrl-*number*.

Other Windows

In addition to the three windows that are shown by default, Delphi has several other useful windows available. In Figure 11, the Code Explorer window is located in its default position, inside the Code Editor window to the left of the active editor page tabs. In other words, this window is docked on the left side of the Code Editor window. The Code Explorer allows the programmer to easily navigate through the unit files. It contains a tree diagram that shows all the types, classes, properties, methods, global variables, and global routines that are defined in the code unit that is currently being edited in the Code Editor window. It also lists the other units that are used by the unit currently being edited. Select View|Code Explorer to open the Code Explorer window if it is not visible.

The programmer can view the files that compose a Delphi project in the Project Manager window. In the Project Manager, Delphi projects may be arranged into project groups, where a project group consists of related projects or projects that function together as part of a multi-tiered application. In addition, this window allows the programmer to easily navigate among the various projects and each project's constituent files within a project group. Select View|Project Manager or press Ctrl-Alt-F11 to open the Project Manager window.

To successfully complete a program, especially a large program, there are many tasks that the programmer needs to perform. Delphi's To-Do List provides a built-in notepad for the programmer to organize these tasks. This list is extremely helpful in planning, programming, testing, and debugging large projects that are written by a team of programmers. Select View|To-Do List from the menu bar to open this window. You can add, edit, and delete list items by right-clicking on the To-Do List window.

The Alignment Palette window provides a rapid means of aligning components on a form. Select View|Alignment Palette from the menu bar to open this window.

The Project Browser lists the units, classes, types, properties, methods, variables, and routines that are declared or used in the current project. The Project Browser arranges this information in a tree diagram. The Project Browser is opened by selecting View|Browser or pressing Shift-Ctrl-B.

The Component List (or Components window) is opened by selecting View|Component List from the menu bar. This window shows an alphabetical listing of all of the components that are available in your version of Delphi. A component may be added to your Delphi program by selecting the component from this window with either the keyboard or the mouse. When using the mouse, the Component Palette provides a more rapid means of selecting components and placing them in your applications since it organizes the components according to their functions. Therefore, it is recommended to use the Component Palette in lieu of the Components window.

The Window List allows you to quickly switch between windows in the Delphi IDE. If you have many windows open, this is the easiest way to locate a window and make it active. Open the Window List by selecting View|Window List or pressing Alt-0. Then, select the desired window from the list and click the OK button.

Delphi contains several windows that are associated with its built-in debugger. Discussion of these windows is deferred until the debugging section of this paper.

Programming Language

This section discusses Delphi file types and the Object Pascal language syntax.

Delphi File Types

Similar to VB, a Delphi application (called a project) consists of several different file types. The three main file types are project files, unit files, and form files. The project file (.DPR) is the "main program;" it accesses the unit files and form files that compose the Delphi project. Thus, the project file ties together the files that are associated with a specific project. Typically, there is a one-to-one relationship between unit files and form files: Each unit file has an associated form file and vice-versa. The form file (.DFM) lists the objects on the form and the object property settings, and the unit file (.PAS) contains the source code associated with the form. Recall that VB combines the objects, property settings, and code into just the form file (.FRM).

The important point to remember is that you must save both the project and the forms for your Delphi application. When you save a form, both the unit file and form file are saved under the same name with the appropriate file extension. To save a form, select File|Save or File|Save As… from the menu bar, click the Save icon on the Standard toolbar, or press Ctrl-S. To save the project file, select File|Save Project As… from the menu bar. To rapidly save all files associated with a project, select File|Save All from the menu or click the Save All icon on the Standard toolbar.

Project File

Again, each application consists of a "main program" in the project file. This file links together all units associated with the application.

The general syntax of a project file is shown below:

```
program Project1;

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

{$R *.RES}

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

As in the above syntax, a project file consist of three elements:

1. a program heading
2. an (optional) uses clause
3. a block of declarations and statements between begin and end keywords

The program heading specifies a name for the program. The uses clause lists the unit files used by the program. The block (between begin and end keywords) contains declarations and statements that are executed when the program runs. The Delphi IDE expects to find these three elements in a single project file.

Finally, a word of caution: It is best not to manually alter the code generated by the IDE. This not only applies to project files, but unit files as well. In other words, let the IDE do the work for you. For instance, if you want to add a new form to a project, select File|New Form or click the New Form button on the View toolbar. Similarly, to add an event-handler to a unit file, select the object and event in the Object Inspector for the appropriate form, and then enter a name for the event-handler. Delphi automatically generates the heading and block for the event-handler.

Units

A Delphi unit consists of data types (including classes), constant and variable declarations, and subroutines (functions and procedures). Each unit exists in a separate unit file.

A unit file contains a heading, interface, implementation, initialization, and finalization sections. The initialization and finalization sections are optional. Like the project file, a unit file must conclude with the end keyword followed by a period. The general syntax of a unit file follows:

```
unit Unit1;

interface

uses {List of used units goes here}

const
  {Public constants go here}

type
  {Public types go here}

var
  {Public variables go here}

{Remainder of interface section goes here}

implementation

uses {List of used units goes here}

const
  {Private constants go here}

type
  {Private types go here}

var
  {Private variables go here}

{Remainder of implementation section goes
here}

initialization
{Initialization section goes here}

finalization
{Finalization section goes here}

end.
```

The interface section begins with the reserved word `interface` and continues until the beginning of the implementation section. The interface section declares constants, data types, variables, and subroutines that are available to clients (other units or programs) that use the unit where they are declared. Therefore, any entity that appears in the interface section has a public scope since a client can access it as if it were declared in the client itself.

The implementation section begins with the reserved word `implementation` and continues until the beginning of the initialization section (if one exists) or the end of the unit. The implementation section declares constants, data types, variables, and subroutines that are private to the unit; that is, these entities have a private scope and are inaccessible to clients.

The interface declaration of a procedure or function includes only the routine's heading. The block of the routine follows in the implementation section. Thus, procedure and function declarations in the interface section act like forward declarations.

The interface and implementation sections may include their own uses clauses which must appear immediately after the section headings (the interface and implementation keywords). The uses clause specifies the units that are used. The System unit is used automatically by every Delphi application and cannot be listed explicitly in the uses clause. The System unit implements routines for file input and output, string handling, floating-point operations, dynamic memory allocation, and so on. Other standard library units, such as SysUtils, must be included in the uses clause. In most cases, Delphi places all necessary units in the uses clause when it generates and maintains a source file.

Unit names must be unique within a project. Even if the unit files are located in different directories, two units with the same name cannot be used in a single program.

Elements of Programming

Now that you are familiar with the purpose and organization of the Delphi file types, it is time to focus on the elements of programming, those elements that are common to most high-level languages.

Comments

VB has two separate comment statements, the remark statement (Rem) and the apostrophe ('). Both of these statements are always ignored by the VB compiler. In Object Pascal, comments are also ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives. There are three ways to construct comments:

- `{Text between a left brace and a right brace constitutes a comment}`
- `(* Text between a left parenthesis followed by an asterisk and an asterisk followed by a right parenthesis constitutes a comment *)`
- `// Any text between a double forward slash and the end of the line constitutes a comment`

A comment that contains a dollar sign ($) immediately after the opening { or (* is a compiler directive. Compiler directives are non-executable statements within the code that alter compiler options. For instance, `{$WARNINGS OFF}` tells the Object Pascal compiler not to generate warning messages. In VB, the Option statement signifies a compiler directive.

Statement Termination

In VB, the end of a line terminates a statement unless the line continuation character (underscore) is used to continue the statement on the following line. Object Pascal does not require a line continuation character; that is, a statement may span over several lines. The semicolon (;) is the statement separator and terminator; it separates one statement from the next.

Identifiers

An Object Pascal identifier denotes a constant, variable, field, data type, property, procedure, function, program, unit, library, or package. An identifier can be of any length, but only the first 255 characters are significant. The first character of an identifier must be either a letter or an underscore. Any number of letters, digits, and underscores may follow the first character. Identifiers cannot contain spaces, and reserved words cannot be used as identifiers.

Object Pascal is case-insensitive, meaning that an identifier named FindItem can be written in a variety of ways, such as finditem, findItem, Finditem, and FINDITEM.

Data Types

A data type specifies the kind of data that a variable can contain. The predefined (built-in) data types for Object Pascal are summarized in Table 1. This table shows the valid range of values for each data type and their memory space requirements. Note that these data types apply to the Windows version of Delphi; the Linux version does not contain a Variant data type, as this data type is a Windows anomaly.

Logical and Numeric Data Types

Data Type	Range	Format or Size	Sig. Digits
Shortint	–128 to +127	signed 8-bit	
Smallint	–32768 to +32767	signed 16-bit	
Integer (or Longint)	–2147483648 to +2147483647	signed 32-bit	
Int64	-2^{63} to $+2^{63}-1$	signed 64-bit	
Byte	0 to 255	unsigned 8-bit	
Word	0 to 65535	unsigned 16-bit	
Longword (or Cardinal)	0 to 4294967295	unsigned 32-bit	
Boolean (or ByteBool)	True or False	1 byte	
WordBool	True or False	2 bytes	
LongBool	True or False	4 bytes	
Real48	2.9×10^{-39} to 1.7×10^{38}	6 bytes	11 to 12
Single	1.5×10^{-45} to 3.4×10^{38}	4 bytes	7 to 8
Real (or Double)	5.0×10^{-324} to 1.7×10^{308}	8 bytes	15 to 16
Extended	3.6×10^{-4951} to 1.1×10^{4932}	10 bytes	19 to 20
Comp	$-2^{63}+1$ to $+2^{63}-1$	8 bytes	19 to 20
Currency	–922337203685477.5808 to +922337203685477.5807	8 bytes	19 to 20

Character and String Data Types

Data Type	Maximum length	Memory required
Char (or AnsiChar)	1 ANSI character	1 byte
WideChar	1 Unicode character	2 bytes
ShortString	255 ANSI characters	2 to 256 bytes
String (or AnsiString)	2^{31} ANSI characters	4 bytes to 2 gigabytes
WideString	2^{30} Unicode characters	4 bytes to 2 gigabytes

Table 1. *Object Pascal Data Types*

Constants

A constant is a named item that retains a constant value throughout the execution of a program. It may be defined by any mathematical or string expression. At compile time, the compiler simply replaces the constant name with its associated value. A numeric constant refers to a number, or numeric literal, and a string constant is a string literal. The number 7, for example, is a numeric constant, and 'days per week' is a string constant.

In Object Pascal, the `const` statement is used to define constants. Like its VB counterpart, the `const` statement can declare a group of constants. The general form of this statement appears below:

```
const
  constantName = Expression;
  [constantName = Expression;]
```

In the general syntax above, the square brackets ([…]) are used to designate optional items that can appear any number of times. For instance, the following code fragment defines three constants:

```
const
  MINS_PER_HR  = 60;
  HRS_PER_DAY  = 24;
  DAYS_PER_WK  = 7;
```

In VB, these same constants are defined using the statement:

```
Const MINS_PER_HR = 60, HRS_PER_DAY = 24, _
      DAYS_PER_WK = 7
```

Variables

A variable is a named location in memory where values are stored. These values can be changed throughout a program's execution.

While VB uses the Dim (dimension) statement to declare variables, Object Pascal uses the `var` statement. The general syntax of the `var` statement is:

```
var
  variableName[, variableName]: DataType;
  [variableName[, variableName]: DataType;]
```

where *`DataType`* is any predefined or user-defined data type.

As an example, consider the following VB variable declarations:

```
Dim dollars As Integer, cents As Integer
Dim cost As Double
Dim myMessage As String
```

The equivalent variable declarations in Delphi follow:

```
var
  dollars:   Integer;
  cents:     Integer;
  cost:      Real;
  myMessage: String;
```

Operators

In VB, the equals sign (=) is an overloaded operator; it functions as both the assignment operator and the comparison operator for equality. However, the equals sign is only the comparison operator for equality in Object Pascal; it always compares the contents of the variables to determine equality. The Object Pascal assignment operator is a combination of two characters, the colon (:) immediately followed by the equals sign (=), or :=.

Now, consider the VB code fragment:

```
Dim value1 As Integer, value2 As Integer
Dim check As Boolean

value1 = 5
value2 = 7
check = (value1 = value2)
```

In this code, `value1` and `value2` are assigned two different values, 5 and 7, respectively. The Boolean variable `check` is assigned a value based upon whether `value1` equals `value2`. In this case, `check` is assigned False since the variable values are unequal. The equivalent Object Pascal code appears below:

```
var
  value1: Integer;
  value2: Integer;
  check:  Boolean;

begin
  value1 := 5;
  value2 := 7;
  check := (value1 = value2);
end;
```

Object Pascal's arithmetic and relational operators are presented in two tables. Table 2 shows the arithmetic operators and Table 3 displays the relational operators. The relational operators are the same as those of VB. The arithmetic operators are also the same as those of VB, except for integer division, modulo division, and exponentiation. In Object Pascal, integer division and modulo division have built-in operators, but exponentiation does not. To perform exponentiation, the programmer must call a function from the math library.

Operation	Operator	Operand Types	Result Type	Example
Sign Identity	+ (unary)	integer, real	integer, real	+x
Sign Negation	– (unary)	integer, real	integer, real	–x
Mutliplication	*	integer, real	integer, real	x * y
Division	/	integer, real	real	x / y
Integer Division	div	integer	integer	x div y
Modulo Division	mod	integer	integer	x mod y
Addition	+	integer, real	integer, real	x + y
Subtraction	–	integer, real	integer, real	x – y

Table 2. *Object Pascal Arithmetic Operators*

Relational Operator	Object Pascal	Mathematics
Less than	<	<
Less than or equal to	<=	≤
Greater than	>	>
Greater than or equal to	>=	≥
Equal to	=	=
Not equal to	<>	≠

Table 3. *Object Pascal Relational Operators*

Logical operators in Object Pascal include `and`, `or`, `not`, and `xor`. By default, Object Pascal performs short-circuited evaluations of `and` and `or` operations. That is, it only evaluates as much of the expression as required in order to determine the final value. To force complete evaluation of these expressions, select Project|Options… and then click on the Compiler tab. Next, click on "Complete boolean eval" under the Syntax options frame. Alternatively, put the `{$B+}` compiler directive in your code.

As in VB, the only string operation in Delphi is string concatenation (which combines strings together). VB has two different, but interchangeable operators that perform string concatenation, the ampersand (&) and the plus sign (+). In Object Pascal, only the plus sign (+) is used for string concatenation. Thus, the plus sign (+) is an overloaded operator in Object Pascal; it is used for sign identity, addition, and string concatenation.

While we are on the subject of strings, another difference between VB and Object Pascal is the string delimiter character. VB uses double quotes (") to delimit strings, but Object Pascal uses single quotes ('). An example line of code that uses strings and the string concatenation operator follows:

```
myName := 'Mitchell' + ' ' + 'Kerman';
```

This line of code is equivalent to:

```
myName := 'Mitchell Kerman';
```

Decision Structures

As in VB, Delphi's Object Pascal has two types of decision structures, `if` statements and `case` statements. These constructs are similar in both languages.

`if` Statements

Nearly every high-level language has some form of the `if` statement. The main difference between VB and Delphi `if` statements is that Object Pascal requires multiple lines of code under a condition to be in the form of a compound statement, where a compound statement is delimited by begin and end keywords. To make this simple, I *always* use a compound statement, even if the compound statement only consists of one statement. This just avoids several syntax problems in the long run. For instance, I won't inadvertently forget to add the begin and end keywords when I increase the number of statements under one of the conditions since these keywords are already in place. I highly encourage you to adopt the same convention for obvious reasons.

The general form of the Object Pascal `if` statement follows:

```
if condition1 then begin
  [statements1;]
end
else if condition2 then begin
  [statements2;]
end
      .
      .
      .
else if conditionN then begin
  [statementsN;]
end
else begin
  [statementsX;]
end;
```

An `if` statement may have any number of `else if` clauses, but may contain at most one `else` clause. In evaluating this `if` statement, we find that it operates in the same manner as its VB counterpart. *`statements1`* executes when *`condition1`* is True; *`statements2`* executes when *`condition1`* is False and *`condition2`* is True; *`statementsN`* executes when *`conditionN`* is True and all other preceding conditions (*condition1* through *condition{N-1}*) are False; finally, *statementsX* executes only if all conditions (*condition1* through *conditionN*) are False.

The following example code computes a golf handicap for your friend based upon your difference in scores:

```
difference := yourAverageScore -
              myAverageScore;
if (difference >= 10) then begin
  handicap := 5;
end
else if (difference >= 7) then begin
  handicap := 3;
end
else if (difference >= 4) then begin
  handicap := 2;
end
else begin
  handicap := 0;
end;
```

The VB equivalent of this code follows:

```
difference = yourAverageScore - myAverageScore
If (difference >= 10) Then
  handicap = 5
ElseIf (difference >= 7) Then
  handicap = 3
ElseIf (difference >= 4) Then
  handicap = 2
Else
  handicap = 0
End If
```

There are some important syntax differences to note here. In VB, ElseIf is a keyword, but the Object Pascal equivalent is two separate words, `else` followed by `if`. Also, there is no Object Pascal equivalent of VB's End If statement; it is not required since we use a semicolon (;) to terminate program statements. Now, about the semicolons, notice their locations in the Delphi syntax shown above. No semicolon directly precedes an `else` statement, as this would cause a syntax error.

`case` Statements

Delphi's `case` statement is very similar to the Select Case statement in VB. The main difference is that the VB Select Case statement can test strings and real numbers. Delphi's `case` statement is restricted to testing only ordinal data types, including integers and characters. If you need to test strings or real values, then you have to use an `if` statement in Delphi.

The syntax of the Object Pascal `case` statement is:

```
case selectorExpression of
  caseList1: begin
               statements1;
             end;
  caseList2: begin
               statements2;
             end;
            .
            .
            .
  caseListN: begin
               statementsN;
             end;
  else begin
    statementsX;
  end;
end;
```

In this syntax, *selectorExpression* is an expression that is compared to each *caseList* expression. *selectorExpression* must be an expression of an ordinal type, where the ordinal types include Integer, Char, and Boolean. Furthermore, each expression in a *caseList* must be an ordinal expression that can be evaluated at compile time. For instance, 12, True, 4 – 9 * 5, 'X', and Integer('Z') are valid *caseList* expressions, but variables and most function calls are not. A *caseList* may also be a subrange having the form *firstExpr..lastExpr*, where *firstExpr* and *lastExpr* are ordinal expressions with *firstExpr* ≤ *lastExpr*. Finally, a *caseList* may be a list in the form *expr1*, *expr2*, …, *exprN*, where each *expr* is an ordinal expression or a subrange as described above.

A `case` statement may have any number of *caseLists*, but at most one `else` clause. The execution of a `case` statement parallels that of the `if` structure. If *selectorExpression* matches any expression in a *caseList*, then the statements following that *caseList* are executed, and control then passes to the code following the `case` statement. If *selectorExpression* matches an expression in more than one *caseList*, only the statements following the first matched *caseList* expression are executed. If *selectorExpression* does not match an expression in any *caseList*, then the statements following the `else` clause, *statementsX*, are executed. Although an `else` clause is not required in a `case` statement, using one allows your code to handle unforeseen *`selectorExpression`* values. If *`selectorExpression`* does not match any *`caseList`* expression and there is no `else` clause, execution continues with the code following the `case` statement.

The golf handicap example is converted to a `case` statement in the code below. This code assumes that the maximum difference between your score and your friend's score is 126 strokes.

```
difference := yourAverageScore -
              myAverageScore;
case difference of
  4, 5, 6: begin
             handicap := 2;
           end;
  7, 8, 9: begin
             handicap := 3;
           end;
  10..126: begin
             handicap := 5;
           end;
  else begin
    handicap := 0;
  end;
end;
```

For comparison purposes, the VB equivalent of this code follows:

```
difference = yourAverageScore - myAverageScore
Select Case difference
  Case 4, 5, 6
    handicap = 2
  Case 7, 8, 9
    handicap = 3
  Case 10 To 126
    handicap = 5
  Case Else
    handicap = 0
End Select
```

Repetition Structures

Repetition structures (also known as loops) may be either definite or indefinite. A definite repetition structure is one in which the number of times the loop executes is known or can be computed. An indefinite repetition structure is one in which the number of times the loop executes is not necessarily known.

In VB, the For loop is a definite loop structure, whereas the Do loops are indefinite loop structures. VB's Do loop structures include the Do While…Loop, Do…Loop Until, Do Until…Loop, and Do…Loop While. Similarly, Object Pascal's `for` loop is a definite loop structure, and the `while` and `repeat` loops are indefinite loop structures. The following paragraphs discuss each of these loop structures.

`for` Loops

In VB, the loop control variable of a For loop may be any numerical data type, including integers and real numbers. In Object Pascal, the loop control variable of a `for` loop must be of an ordinal data type. Additionally, no step value may be specified in Object Pascal; a `for` loop always increments (or decrements, depending upon the syntax) to the next ordinal value.

The following VB code uses an incrementing For loop to compute the sum of the integers from 1 to 100:

```
Dim counter As Integer, sum As Integer

sum = 0
For counter = 1 To 100
  sum = sum + counter
Next counter
```

The general syntax of an *incrementing* `for` loop in Object Pascal follows:

```
for counter := start to finish do begin
  [statements;]
end;
```

So, translating the above VB code into Object Pascal syntax, we get:

```
var
  counter: Integer;
  sum:     Integer;

begin
  sum := 0;
  for counter := 1 to 100 do begin
    sum := sum + counter;
  end;
end;
```

We can just as easily compute the sum of the integers from 1 to 100 using a decrementing For loop as opposed to an incrementing one. The necessary VB code follows:

```
Dim counter As Integer, sum As Integer

sum = 0
For counter = 100 To 1 Step -1
```

```
  sum = sum + counter
Next counter
```

The general syntax of a *decrementing* `for` loop in Object Pascal is:

```
for counter := start downto finish do begin
  [statements;]
end;
```

Rewriting our decrementing loop structure in Object Pascal, we get:

```
var
  counter: Integer;
  sum:     Integer;

begin
  sum := 0;
  for counter := 100 downto 1 do begin
    sum := sum + counter;
  end;
end;
```

`while` and `repeat` Loops

Delphi's `while` and `repeat` loops are indefinite loop structures. The `while` loop is the equivalent of VB's Do While…Loop, and the `repeat` loop is equivalent to VB's Do…Loop Until. Note that there is no Delphi equivalent for VB's Do Until…Loop and Do…Loop While structures, but these structures may be easily converted to one of the other forms by negating the logical condition.

The `while` loop has the following general syntax:

```
while condition do begin
  [statements;]
end;
```

The `while` loop executes the body of the loop as long as *`condition`* evaluates to True. This type of loop is a top tested loop; if *`condition`* evaluates to False before the first loop execution, then the body of the loop never executes.

The following code sums the integers from 1 to 100 using a `while` loop:

```
sum := 0;
count := 1;
while (count <= 100) do begin
  sum := sum + count;
  count := count + 1;
end;
```

The VB equivalent of this code is:

```
sum = 0
count = 1
Do While (count <= 100)
  sum = sum + count
  count = count + 1
Loop
```

The `repeat` loop has the following syntax:

```
repeat
  [statements;]
until condition;
```

The `repeat` loop executes a block of code until *`condition`* becomes True. This is a bottom tested loop; the body of the `repeat` loop is guaranteed to execute at least once. Notice that no begin and end keywords are needed for this loop structure; the body of the loop is delimited by the `repeat` and `until` keywords.

The following code sums the integers from 1 to 100 using a `repeat` loop:

```
sum := 0;
count := 1;
repeat
  sum := sum + count;
  count := count + 1;
until (count > 100);
```

Again, the VB equivalent of this code is:

```
sum = 0
count = 1
Do
  sum = sum + count
  count = count + 1
Loop Until (count > 100)
```

`break` and `continue` Statements

VB allows a program to unconditionally exit out of a loop structure through the use of either the Exit Do or Exit For statements. Both of these statements transfer control to the statement following the loop structure. To accomplish this same task in Delphi, use the `break` statement. Delphi has another statement that is not available in VB, the `continue` statement. This statement transfers control to the beginning of the loop, skipping the remainder of the loop body.

Subprograms

Like most other high-level languages, Object Pascal has two different types of subprograms: Procedures and Functions.

Procedures and functions that are not built-in to Object Pascal are called user-defined because the programmer (the user of the compiler) must define them. A `procedure` in Delphi is analogous to a Sub procedure in VB. Similarly, a `function` in Delphi operates in the same manner as a VB Function.

In VB, event-handlers are actually Sub procedures. Similarly, event-handlers in Delphi are procedures that are automatically called when an event occurs on its associated object. For instance, the following VB event-handler finds and displays the square root of a user-entered value:

```
Private Sub cmdComputeSqrRt_Click()
Dim value As Double

  value = Sqr(Val(txtInputNumber.Text))
  picOutput.Print "The square root of " & _
                  txtInputNumber.Text & _
                  " is " & CStr(value)
End Sub
```

This same event-handler is written in Delphi as follows:

```
procedure TfrmSquareRoot.SquareRoot(Sender:
TObject);

var
  value:  Real;
  code:   Integer;
  result: String;

begin
  Val(edtInputNumber.Text, value, code);
  value := Sqrt(value);
  Str(value, result);
  memOutput.Lines.Add('The square root of ' +
                      edtInputNumber.Text +
                      ' is ' + result);
end;
```

The user interface for the Delphi code consists of the form (`frmSquareRoot`) and four components on the form: a label (`lblInputNumber`), an edit box (`edtInputNumber`), a button (`btnComputeSqrRt`), and a memo box (`memOutput`). This code simply computes the square root of a number that the user enters in `edtInputNumber` and displays the result in `memOutput`. Furthermore, this code is associated with the `OnClick` event of the `btnComputeSqrRt` button. In other words, when the mouse pointer is on the `btnComputeSqrRt` button and the left mouse button is clicked, this code executes.

Procedures

The general form for defining an Object Pascal procedure is:

```
procedure ProcedureName(param1: Type1;
                        param2: Type2; ...);

[localDeclarations;]

begin
  [statements;]
end;
```

A procedure is invoked by stating the procedure name along with any required arguments. The general syntax of a procedure call follows:

```
ProcedureName(argument1, argument2, ...);
```

As an example, a procedure called `Adder` is created to sum two numbers:

```
{Add num1 and num2 and store the result in
sum}
procedure Adder(num1: Real; num2: Real;
                var sum: Real);

begin
  sum := num1 + num2;
end;
```

`num1`, `num2`, and `sum` are the parameters of the `Adder` procedure. Parameters are merely placeholders for the information passed to a subprogram when it is invoked. The parameter list specified in the `Adder` procedure contains three elements, thereby informing the Delphi compiler that the `Adder` procedure requires three real numbers to be passed to it. Note that the `var` keyword precedes the `sum` parameter. We'll talk more about this later.

A simple test driver for the `Adder` procedure follows. When the user clicks the `btnAdd` button, the `AddNumbers` event-handler is invoked. This event-handler adds the numbers in edit boxes `edtNum1` and `edtNum2` by calling the `Adder` procedure and then stores the result in `edtResult`.

```
{Add two user-entered numbers}
procedure
TfrmAdderProcedure.AddNumbers(Sender:
TObject);

var
  firstNum:  Real;
  secondNum: Real;
  result:    Real;
  code:      Integer;
```

```
begin
  Val(edtNum1.Text, firstNum, code);
  Val(edtNum2.Text, secondNum, code);

  Adder(firstNum, secondNum, result);

  edtResult.Text := Format('%15.5f',
                           [result]);
end;
```

The variables `firstNum`, `secondNum`, and `result` are the arguments of the `Adder` procedure; these variables contain the values that are passed to the procedure. Note that an argument can be any expression that is of the same type as its corresponding parameter.

Functions

A function may take any number of arguments, but it always returns a single value to the calling routine. A procedure, on the other hand, does not automatically return a value. The general rule is to use a function if you need to return exactly one value to the calling routine. So, our `Adder` procedure is better written as a function:

```
{Return the sum of num1 and num2}
function Adder(num1: Real; num2: Real): Real;

begin
  Adder := num1 + num2;
end;
```

A test driver for the `Adder` function appears below:

```
{Add two user-entered numbers}
procedure TfrmAdderFunction.AddNumbers(Sender:
TObject);

var
  firstNum:  Real;
  secondNum: Real;
  result:    Real;
  code:      Integer;

begin
  Val(edtNum1.Text, firstNum, code);
  Val(edtNum2.Text, secondNum, code);

  result := Adder(firstNum, secondNum);

  edtResult.Text := Format('%15.5f',
                           [result]);
end;
```

From this example, we see the general form of a user-defined function:

```
function FunctionName(param1: Type1; ...):
  FunctionType;

[localDeclarations;]

begin
  [statements;]
  FunctionName := ReturnValue;
end;
```

Notice that a function is defined with a specific data type (*`FunctionType`*); this is the data type of the value that the function returns to the calling routine. To return a value to the calling routine, the return value (*`ReturnValue`*) must be assigned to the function name (*`FunctionName`*) somewhere within the function's code block, as shown in the line preceding the end keyword. Alternatively, the return value can be assigned to the parameter `Result`, an implicit parameter of every Object Pascal function. `Result` and *`FunctionName`* refer to the same value.

Similar to a procedure call, a function is invoked simply by writing the function name along with any required arguments. Unlike a procedure call, however, a function call returns a single value, and your program should do something with this value (store it in a variable, display it, etc.). The general form of a function invocation with the returned value assigned to a variable follows:

```
variableName := FunctionName(argument1,
                             argument2, ...);
```

Parameter Passing

An argument is a piece of information passed to a subprogram. A parameter is a placeholder for the information passed to a subprogram when it is invoked. Each argument has a corresponding parameter. With regard to arguments and parameters, there are several important points that apply to most high-level languages, including Object Pascal:

1. The number of arguments must equal the number of parameters.
2. Order is important. The first argument corresponds to the first parameter, the second argument to the second parameter, and so on.

3. The data type of each argument must match the data type of its corresponding parameter.
4. Names are not important. The name of an argument does not have to correspond to the name of its parameter.
5. Recognize the manner in which data is passed.

In Object Pascal, parameters that have the same data type and are passed in the same manner may be combined in the parameter list of a subprogram. For instance, we can rewrite the `Adder` function as follows:

```
{Return the sum of num1 and num2}
function Adder(num1, num2: Real): Real;

begin
  Adder := num1 + num2;
end;
```

When grouping parameters in the parameter list of a subprogram, make sure that you maintain the correct order.

Object Pascal, like VB, can pass parameters either by reference or by value. Passing a parameter by reference actually passes the memory location (address) of the argument to the subprogram instead of the argument's value. This allows the subprogram to access the actual variable. As a result, the variable's value can be changed by the subprogram. Passing a parameter by value, on the other hand, passes only the value of the argument to the subprogram. The subprogram accesses a copy of the variable, and the variable's actual value cannot be changed by the subprogram. Passing by value is the default method of parameter passing in Object Pascal; unless otherwise specified, Object Pascal passes parameters by value. To pass a parameter by reference, the parameter must be preceded by the `var` keyword in the subprogram heading. In VB, however, the default method of parameter passing is by reference, and the programmer must specify those parameters to be passed by value by preceding them with the ByVal keyword in the subprogram heading.

VB does not have the ability to pass parameters as constants, but Object Pascal does. This method of parameter passing uses the least amount of resources. When a parameter is passed as a constant, the subprogram is not allowed to alter its value. Any attempt to do so results in a compiler error. To pass a parameter as a constant, precede the parameter name with the `const` keyword in the subprogram heading.

Data Structures

Object Pascal contains a variety of built-in data types for storing data structures. Additionally, a programmer can define his own data types (i.e., create user-defined data types).

Arrays

An array is a set of sequentially indexed elements of the same intrinsic data type. In Object Pascal, an array variable is declared using the `array of` keywords. To declare a static array, use the syntax:

```
var
  arrayVariable: array[indexType1, ...,
                        indexTypeN] of BaseType;
```

`arrayVariable` is any valid variable name, and *`BaseType`* is the data type of each element in the array. Each *`indexType`* represents a separate index of the array and must be an ordinal data type. These are normally integer subranges.

To declare a dynamic array, use the `array of` statement without specifying indices. For instance, to declare a one-dimensional dynamic array, use the syntax:

```
var
  dynamicArrayVariable: array of BaseType;
```

Then, use the `SetLength` procedure to allocate memory for the dynamic array and set its size:

```
SetLength(dynamicArrayVariable, length);
```

Each element of an array acts as a separate variable that can be accessed using its unique index values. To access a specific array element, use either:

```
arrayName[indexValue1, …, indexValueN]
```

or

```
arrayName[indexValue1] … [indexValueN]
```

For instance, the following statement assigns the value 7 to the 4th element of `myArray`:

```
myArray[4] := 7;
```

Short Strings

A short string is a string whose length does not exceed 255 characters. To dimension a short string, use the following syntax:

```
var
  stringName: String[n];
```

where *n* is the length of the short string in characters. Note that the variable `stringName` occupies `n+1` bytes of memory (from 0 to `n`), where bytes 1 through `n` contain the characters in the string, the 0th byte contains the size of the string, and `n` is less than or equal to 255. The `ShortString` data type is equivalent to `String[255]`.

In Object Pascal, any string is just an array of characters. To access a particular character within a string variable, use the syntax `stringName[i]`, where `i` is the index of the character within the string (i.e., you want to access the `i`th character of the string), and the first character in a string has an index value of 1. Note that `stringName[i]` is of the Char (character) data type.

Enumerated Types

An enumerated type is an ordered set of values defined by the programmer. The values have no inherent meaning, but their ordinality follows the sequence in which they are listed. In Object Pascal, the `type` keyword allows the programmer to create user-defined data types. To declare an enumerated type, use the syntax:

```
type
  TypeName = (value1, …, valueN);
```

where `TypeName` and `value1` through `valueN` are valid identifiers. Consider the following example:

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

This code defines an enumerated type named `Days` whose possible values include the days of the week.

Sets

A set consists of a group of values of the same ordinal data type. The values have no inherent order, and it is not meaningful for a value to be included more than once within a set.

A set is defined using the `set of` keywords. The syntax is:

```
type
  SetName = set of BaseType;
```

`SetName` is the name of the set and must be a valid identifier. The possible values of `SetName` include all subsets of `BaseType`, including the empty set (denoted by `[]` in Object Pascal). Object Pascal restricts the size of `BaseType` to no more than 256 possible values.

Sets are usually defined using subranges, denoted by two periods (`..`). A set constructor consists of a list of comma-separated values or subranges within square brackets. Consider the following code fragment:

```
type
  LowercaseLetters = 'a'..'z';
  LowercaseSet = set of LowercaseLetters;

var
  myLetters: LowercaseSet;

begin
  myLetters := ['a'..'c', 'm', 'n', 'x'..'z'];
     .
     .
     .
end;
```

This code creates the `LowercaseSet` data type and the `myLetters` variable of this data type. The code block assigns `myLetters` the set consisting of the lowercase letters a, b, c, m, n, x, y, and z.

Records

As in VB, a record type declaration must specify a record type name as well as a name and data type for each field of the record. The Object Pascal syntax of a record type declaration follows:

```
type
  RecordTypeName = record
                     fieldList1: DataType1;
                     fieldList2: DataType2;
                          .
```

```
                         .
                         .
                       fieldListN: DataTypeN;
                     end;
```

Object Pascal, like VB, uses the dot-separator to access the fields of a record. Use the *recVarName.fieldName* format to access the *fieldName* field of record variable *recVarName*.

Pointers

A pointer is a variable that "points" to the memory location of another variable. So, a pointer is an indirect variable reference. VB does not allow explicit pointer variables, but Object Pascal does. In Object Pascal, the caret symbol (^) is used to both denote a pointer and *dereference* a pointer. The following syntax declares a pointer type:

```
type
  PointerTypeName = ^DataType;
```

For instance, to declare a pointer type named `IntPointer` that points to an integer value, use the following code:

```
type
  IntPointer = ^Integer;
```

A variable of type `IntPointer` contains the memory address of a variable that contains an integer value.

When the caret appears after a pointer variable, it dereferences the pointer variable and returns the value stored in the address contained by the pointer. The syntax *pointerVariable*^ dereferences *pointerVariable*. For the variable `ptr` of type `IntPointer`, for instance, `ptr^` returns an integer value or `nil`. `nil` is a reserved word and special constant that can be assigned to any pointer variable to reference "nothing," similar to VB's reserved word Nothing for objects.

The `New` and `Dispose` procedures create and destroy pointer variables, respectively. `New` allocates memory for a new dynamic variable and points the associated pointer variable to it. When an application finishes using a pointer, it should release the memory allocated for it using the `Dispose` procedure. The syntax follows:

```
New(pointerVariable);
Dispose(pointerVariable);
```

The value of *pointerVariable* is undefined after a call to the `Dispose` procedure.

File Input and Output

VB provides two different file types for file input and output (file I/O), sequential files and random-access files. Text files are the Delphi equivalent of VB sequential files, and binary files serve the same purpose as random-access files.

Text Files

To access a text file in Object Pascal, we must first create a variable capable of referencing a text file. Declaring a variable of the `TextFile` data type creates such a file reference variable. A file reference variable contains a file pointer. A file pointer is similar to the cursor in a text editor: A cursor indicates the position within a file in a text editor, and the file pointer indicates the position in an open file. For an input file, the file pointer indicates the next data item to be read from the file. For an output file, the file pointer indicates the position of the next data item written to the file.

Next, we must associate the file reference variable with a data file using the `AssignFile` procedure. The syntax follows:

```
AssignFile(fileRef, fileName);
```

fileRef is a file reference variable. *fileName* is a string expression containing any valid Windows file name and can specify the path of the file, where the path indicates the disk drive and subdirectory where the file resides.

Finally, before we can access a data file, it must be opened. A text file can be opened for either input or output, but not both simultaneously. The `Reset` procedure opens or reopens a text file for input, and the `Rewrite` and `Append` procedures open or reopen a text file for output. The syntax for these procedures follows:

```
Reset(fileRef);
Rewrite(fileRef);
Append(fileRef);
```

As before, `fileRef` is the file reference variable. `Reset` opens the existing data file whose name is associated with `fileRef` and sets the file pointer to the beginning of the file. If the file is already open, it is first closed and then reopened. A "file not found" run-time error results if no data file of the given name exists.

The `Rewrite` procedure creates a new data file whose name is associated with `fileRef` and sets the file pointer to the beginning of the file. If a data file of the same name already exists, it is deleted and a new, empty file is created in its place. If the file is already open, it is first closed and then re-created. In summary, the `Rewrite` procedure either creates a new file or overwrites an existing one.

To add data to the end of a file, use the `Append` procedure. `Append` opens the existing text file whose name is associated with `fileRef` and positions the file pointer at the end of the file. If the file is already open, it is first closed and then reopened. If no data file of the given name exists, a "file not found" run-time error results.

Once a `Reset` statement opens a text file for input, data can be read from the file using the `Read` statement:

```
Read(fileRef, variable);
```

This statement reads the next piece of data indicated by the file pointer from the input file associated with `fileRef`, stores this data in `variable`, and then moves the file pointer to the next character in the input file. The data type of `variable` should match the type of data that is being read from the input file. If the data file consists of integers, for example, the data should be read into Integer variables. Whitespace characters (spaces and tabs) delimit numerical data in text files.

Multiple variables can be read using one `Read` statement. So, the general syntax of the `Read` statement is:

```
Read(fileRef, variable1 [, variable2, …]);
```

While the `Read` statement reads data item by item from a text file, the `Readln` statement reads only a specified number of data items per line. The syntax to read one data item into a variable follows:

```
Readln(fileRef, variable);
```

This statement reads the next piece of data indicated by the file pointer from the input file associated with `fileRef`, stores this data in `variable`, and then moves the file pointer to the beginning of the next line of the input file.

The general syntax of the `Readln` statement is:

```
Readln(fileRef, variable1 [, variable2, …]);
```

When a file is opened for output with the `Rewrite` or `Append` statement, data may be written to the file using the `Write` procedure. The general syntax appears below:

```
Write(fileRef
      [, expression[:minWidth[:decPlaces]]]);
```

As before, `fileRef` is the file reference variable. `expression` is an expression of any simple or string data type, whereas `minWidth` and `decPlaces` are integer expressions. The optional `minWidth` parameter specifies the minimum number of characters in the output of expression. If the length of expression is less than `minWidth`, the `Write` procedure pads the left side of the output with blank spaces. All characters in expression are output when its length exceeds `minWidth`. For a real-type expression, the optional `decPlaces` parameter specifies the number of digits following the decimal point. Any number of expressions may be output with a single `Write` statement (including no expressions) by separating the expressions with commas. For example,

```
Write(myFile, 10:5, 10.47589:8:2);
```

outputs the following text to the file associated with `myFile`:

```
   10   10.48
```

The `Writeln` statement operates the same way as `Write`, except that it outputs a carriage return and line feed combination (<CR><LF>) after all of its expressions are output. For instance,

```
Write(myFile, 'Hello ');
Write(myFile, 'and Good-bye');
Writeln(myFile);        {Skip to next line}
Writeln(myFile, 'Hello ');
Writeln(myFile, 'and Good-bye');
```

outputs the following text:

```
Hello and Good-bye
Hello
```

```
and Good-bye
```

As with `Read` and `Readln`, `Write` statements can be combined but `Writeln` statements cannot. Thus, we can rewrite the above example as follows:

```
Write(myFile, 'Hello ', 'and Good-bye');
Writeln(myFile);        {Skip to next line}
Writeln(myFile, 'Hello ');
Writeln(myFile, 'and Good-bye');
```

Finally, when a program is finished working with a file, the file should be closed. The `CloseFile` statement ends the association between a file reference variable and a data file and returns these resources to the system. For an output file, the `CloseFile` statement also writes the end-of-file character before it closes the file. The syntax of the `CloseFile` statement follows:

```
CloseFile(fileRef);
```

You should be aware of two extremely important functions for working with text files: the `Eof` and `Eoln` functions. As in VB, the `Eof` (end-of-file) function returns a Boolean value that indicates whether the end of an input file is reached. `Eof(`*`fileRef`*`)` is True when the file pointer is beyond the last character of the file associated with *`fileRef`*. This function is useful in an indefinite loop structure to read data from an input file until the end-of-file is reached. The `Eoln` (end-of-line) function returns a Boolean value that indicates whether the file pointer is at the end of the current line. For an input file associated with *`fileRef`*, `Eoln(`*`fileRef`*`)` is True when the file pointer is at the end of the current line or `Eof(`*`fileRef`*`)` is True. The `Eoln` function is also useful in an indefinite loop structure for processing an input file character by character. VB does not contain a built-in function similar to `Eoln`.

Binary Files

Unlike text files, binary files can access data in any order; data can be read from or written to any location in the file. Thus, binary files are also known as random-access files. Furthermore, a binary file accesses an entire data structure at a time. For that reason, binary files present a better and faster method of storing and retrieving information contained within a known data structure.

While text files are stored in ASCII format, binary files are not. That is, if a binary file were to be read as a text file, not all of the characters in the file would be meaningful. To correctly access a binary file, a program must know the exact data structure contained in the file.

Object Pascal has two kinds of binary files, typed files and untyped files. Our discussion concentrates only on typed files, the most common of the two kinds. A typed file is an ordered file of elements of the same data type. You may notice the strong similarity in the definitions of an array and a typed file. Essentially, you can think of a typed file as an array in file form. As you will soon see, instead of using an array index to access a particular piece of data, you use a record number.
To define a typed file data type, use the `file of` syntax shown below:

```
type
  FileTypeName = file of DataType;
```

where *`FileTypeName`* is any valid identifier and *`DataType`* is a fixed-size data type. Since *`DataType`* is of a fixed size, both implicit and explicit pointer types are not allowed. In other words, a typed file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

As an example, consider the following code fragment:

```
type
  StudentRec = record
                 lastName:  String[30];
                 firstName: String[20];
                 ID:        String[12];
                 GPA:       Real;
                 crdtHrs:   Real;
               end;
  StudentDB =  file of StudentRec;

var
  studentFile: StudentDB;
```

This code fragment declares the `StudentDB` data type, a typed file of `StudentRec` records. `studentFile` is a typed file variable of type `StudentDB` whose associated file contains the names, ID numbers, grade point averages, and cumulative credit hours for the students at a particular school.

As with text files, the `AssignFile` procedure associates a file variable with an external binary file. The `Reset` and `Rewrite` procedures also work the same for binary files as they do for text files. By default, a binary file is capable of both input and output operations regardless of which of these two procedures is used. For a text file, recall that `Reset` accesses the file as read-only (for input) and `Rewrite` sets it to write-only (for output). Both procedures move the file pointer to the beginning of the file. The `Append` procedure is used exclusively for text files; it is not available for use with binary files.

The value of the global variable `FileMode` determines the access mode used when a binary file is opened using the Reset procedure. Valid values of `FileMode` are 0 for read-only access, 1 for write-only access, and 2 for read/write access. The default `FileMode` is 2. Assigning another value to `FileMode` causes all subsequent `Reset` calls to use that mode.

The `Read` procedure reads a data element from a binary file into a variable of a compatible data type. Similarly, the `Write` procedure writes the contents of a variable to a binary file of a compatible data type. Both operations (read and write) occur in the current location of the file pointer. After execution, both `Read` and `Write` automatically increment the file pointer to the next data element in the binary file. As with text files, multiple `Read` statements and multiple `Write` statements can be combined. So, the general syntax of the `Read` and `Write` procedures appears below:

```
Read(fileRef, dataVar [, dataVar2, …]);
Write(fileRef, dataVar [, dataVar2, …]);
```

Since binary files are not organized into lines (of text), a syntax error results from attempting to use the `Readln` or `Writeln` procedures. In short, `Readln` and `Writeln` are used exclusively for text files. Similarly, the `Eof` function works with binary files, but `Eoln` does not.

The `Seek` procedure moves the file pointer in a binary file to a specified record or data element. The syntax is:

```
Seek(fileRef, recNum);
```

`fileRef` is a binary file variable. *`recNum`* is a long integer representing the record number (or element number) in the file, where the first data element has a *`recNum`* of 0. The `FileSize` function returns the number of records (or elements) in a specified binary file. For the binary file corresponding to file variable *`fileRef`*, the values of *`recNum`* range from 0 to `FileSize(`*`fileRef`*`)-1`. To move the file pointer to the end of the file, use a statement of the form:

```
Seek(fileRef, FileSize(fileRef));
```

Performing a `Write` procedure immediately following the above statement expands the binary file by one data element. The `Truncate` procedure deletes all data elements in the binary file at and after the current position of the file pointer; the current file position becomes the end-of-file. When all file operations are complete, the `CloseFile` procedure is used to terminate the association between the binary file variable and external file.

As an example, the following code unit implements an address book program using records and binary file I/O. This program stores data in a binary file named Address.dat located in the home directory of drive C (C:\). Figure 12 displays the user interface for this program.

```
{---------------------------------------------
Address Book Program
---------------------------------------------}
unit Address;

interface

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

type
  TfrmAddressBook = class(TForm)
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtAddress: TEdit;
    edtCity: TEdit;
    edtState: TEdit;
    edtZip: TEdit;
    edtPhoneNumber: TEdit;
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblAddress: TLabel;
    lblCity: TLabel;
    lblState: TLabel;
    lblZip: TLabel;
    lblPhoneNumber: TLabel;
    btnAdd: TButton;
    btnClear: TButton;
    btnRemove: TButton;
    btnFind: TButton;
    procedure AddCard(Sender: TObject);
    procedure Initialize(Sender: TObject);
    procedure ClearForm(Sender: TObject);
    procedure Terminate(Sender: TObject;
                var Action: TCloseAction);
    procedure RemoveCard(Sender: TObject);
    procedure FindCard(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  AddressCard = record
                  firstName:   String[20];
                  {First Name}
                  lastName:    String[20];
                  {Last Name}
                  address:     String[30];
                  {Street Address}
                  city:        String[20];
                  {City}
                  state:       String[20];
                  {State}
                  zipCode:     String[15];
                  {Zip Code}
                  phoneNumber: String[20];
                  {Telephone Number}
                end;

{
GLOBAL VARIABLES
These global variables make the coding of this
Program sufficiently easier.
}
var
  frmAddressBook: TfrmAddressBook;
  dataFile:       File of AddressCard;

implementation

{$R *.DFM}

{Return the record number of the address card
that matches the first and last names.  The
search is not case sensitive.  If a matching
address card is not found, Find returns -1.}
function Find(first, last: String): Integer;

var
  addrCard: AddressCard;
  findCard: AddressCard;
  found:    Boolean;

begin
  Reset(dataFile);
  found := False;
  findCard.firstName :=
    Trim(UpperCase(first));
  findCard.lastName := Trim(UpperCase(last));
  while not(Eof(datafile) or found) do begin
    Read(dataFile, addrCard);
    found := (UpperCase(addrCard.firstName) =
              findCard.firstName) and
             (UpperCase(addrCard.lastName) =
              findCard.lastName);
  end;
  if found then begin
    Find := FilePos(dataFile) - 1;
  end
  else begin
    Find := -1;
  end;
end;

{Add the address card to the database}
procedure TfrmAddressBook.AddCard(Sender:
TObject);

var
  addrCard: AddressCard;

begin
  with addrCard do begin
    firstName := Trim(edtFirstName.Text);
    lastName := Trim(edtLastName.Text);
    address := Trim(edtAddress.Text);
    city := Trim(edtCity.Text);
    state := Trim(edtState.Text);
    zipCode := Trim(edtZip.Text);
    phoneNumber := Trim(edtPhoneNumber.Text);
  end;
  Seek(dataFile, FileSize(dataFile));
  Write(dataFile, addrCard);
  Application.MessageBox(
    PChar('Address card added!'),
    'ADD', MB_OK);
  ClearForm(Sender);
end;

{Remove the address card from the database}
procedure TfrmAddressBook.RemoveCard(Sender:
TObject);

var
  addrCard: AddressCard;
  pos:      Integer;
  recNum:   Integer;

begin
  recNum := Find(edtFirstName.Text,
              edtLastName.Text);
  if (recNum >= 0) then begin
```

```
    {Display the address card}
    Seek(dataFile, recNum);
    Read(dataFile, addrCard);
    edtFirstName.Text := addrCard.firstName;
    edtLastName.Text := addrCard.lastName;
    edtAddress.Text := addrCard.address;
    edtCity.Text := addrCard.city;
    edtState.Text := addrCard.state;
    edtZip.Text := addrCard.zipCode;
    edtPhoneNumber.Text :=
      addrCard.phoneNumber;

    {Remove the address card from the
     database}
    for pos := recNum to (FileSize(dataFile) -
                      2) do begin
      Seek(dataFile, pos + 1);
      Read(dataFile, addrCard);
      Seek(dataFile, pos);
      Write(dataFile, addrCard);
    end;
    Seek(dataFile, FileSize(dataFile) - 1);
    Truncate(dataFile);

    Application.MessageBox(
      PChar('Address card removed!'),
      'REMOVE', MB_OK);
  end
  else begin
    Application.MessageBox(
      PChar('Address card NOT found!'),
      'REMOVE', MB_OK);
  end;
end;

{Find and display the address card that
matches the first and last names}
procedure TfrmAddressBook.FindCard(Sender:
TObject);

var
  addrCard: AddressCard;
  recNum:   Integer;

begin
  recNum := Find(edtFirstName.Text,
              edtLastName.Text);
  if (recNum >= 0) then begin
    Seek(dataFile, recNum);
    Read(dataFile, addrCard);
    edtFirstName.Text := addrCard.firstName;
    edtLastName.Text := addrCard.lastName;
    edtAddress.Text := addrCard.address;
    edtCity.Text := addrCard.city;
    edtState.Text := addrCard.state;
    edtZip.Text := addrCard.zipCode;
    edtPhoneNumber.Text :=
      addrCard.phoneNumber;
  end
  else begin
    Application.MessageBox(
      PChar('Address card NOT found!'),
      'FIND', MB_OK);
  end;
end;

{Initialize the program by opening the Address
Book database -- File Name: c:\Address.dat}
procedure TfrmAddressBook.Initialize(Sender:
TObject);

begin
  AssignFile(dataFile, 'c:\Address.dat');
  try
    Reset(dataFile);
  except
    Rewrite(dataFile);
  end;
end;

{Clear the edit boxes on the form}
procedure TfrmAddressBook.ClearForm(Sender:
TObject);

begin
  edtFirstName.Clear;
  edtLastName.Clear;
  edtAddress.Clear;
  edtCity.Clear;
  edtState.Clear;
  edtZip.Clear;
  edtPhoneNumber.Clear;
end;

{Close the database file and terminate the
program}
procedure TfrmAddressBook.Terminate(Sender:
TObject;
  var Action: TCloseAction);
begin
  CloseFile(dataFile);
end;

end.
```

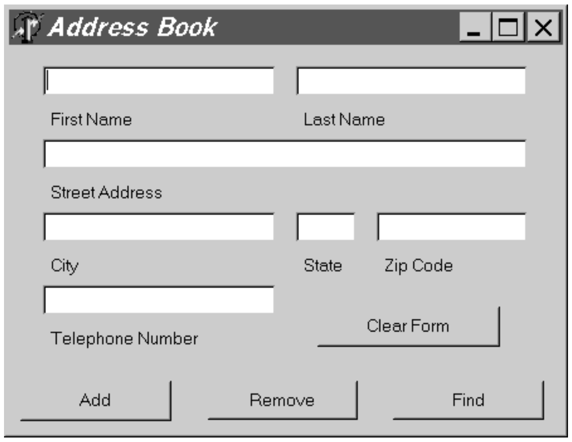


Figure 12. *User Interface for Address Book Program*

As a final note, Delphi contains a `TFileStream` class for object-oriented file I/O. This offers the programmer a portable, high-level approach to file I/O. The next section introduces object-oriented programming in Delphi.

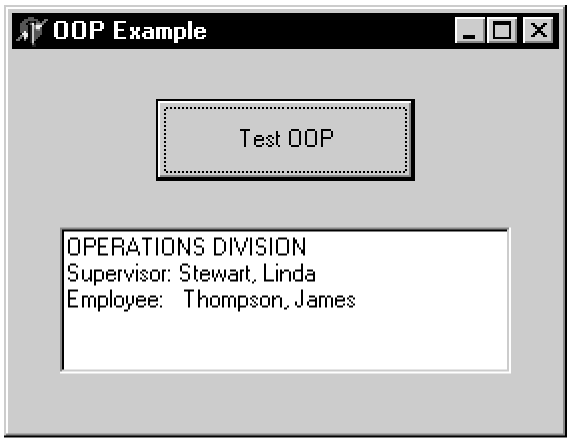
Object-Oriented Programming

As previously discussed, Delphi has a complete object model. All components and controls are ancestors of `TObject`, the base object class. Thus, all components and controls are fully extensible through object-oriented programming (OOP). VB, however, does not have a complete object model. It does not offer true object inheritance and polymorphism like Delphi.

Rather than extol the virtues of OOP and detail its syntax in Object Pascal, I have opted to provide an example. The following Delphi unit file defines two classes, `Employee` and `Supervisor`. The `Supervisor` class is a descendant of the `Employee` class. A sample execution of this code appears in Figure 13.

```
unit OOPEx;

interface

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

type
  TfrmOOPEx = class(TForm)
    btnTest: TButton;
    memOutput: TMemo;
    procedure DoTest(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

type
  NameStr    = String[25];
  IDStr      = String[10];
  DivType    = (Operations, Production,
               Maintenance);
  Employee   = class
    lastName:  NameStr;
    firstName: NameStr;
    division:  DivType;
    procedure SetAll(lname, fname:
                NameStr; dv: DivType);
    procedure SetLastName(lname: NameStr);
    function GetLastName: NameStr;
    procedure SetFirstName(fname: NameStr);
    function GetFirstName: NameStr;
    procedure SetDivision(dv: DivType);
    function GetDivision: DivType;
  end;
  Supervisor = class(Employee)
    managerID: IDStr;
    procedure SetAll(lname, fname: NameStr;
                     dv: DivType;
                     id: IDStr);
    procedure SetID(id: IDStr);
    function GetID: IDStr;
  end;

var
  frmOOPEx: TfrmOOPEx;

implementation

{$R *.DFM}

procedure Employee.SetLastName(lname:
                               NameStr);
begin
  lastName := lname;
end;

function Employee.GetLastName: NameStr;
begin
  GetLastName := lastName;
end;

procedure Employee.SetFirstName(fname:
                                NameStr);
begin
  firstName := fname;
end;

function Employee.GetFirstName: NameStr;
begin
  GetFirstName := firstName;
end;

procedure Employee.SetDivision(dv: DivType);
begin
  division := dv;
end;

function Employee.GetDivision: DivType;
begin
  GetDivision := division;
end;

procedure Employee.SetAll(lname, fname:
                          NameStr;
                          dv: DivType);
begin
  Self.SetLastName(lname);
  Self.SetFirstName(fname);
  Self.SetDivision(dv);
```

```
end;

procedure Supervisor.SetID(id: IDStr);
begin
  managerID := id;
end;

function Supervisor.GetID: IDStr;
begin
  GetID := managerID;
end;

procedure Supervisor.SetAll(lname, fname:
                            NameStr; dv:
                            DivType;
                            id: IDStr);
begin
  Self.SetLastName(lname);
  Self.SetFirstName(fname);
  Self.SetDivision(dv);
  Self.SetID(id);
end;

procedure TfrmOOPEx.DoTest(Sender:
                           TObject);

var
  emp: Employee;
  mgr: Supervisor;

begin
  emp := Employee.Create;
  mgr := Supervisor.Create;
  emp.SetAll('Thompson', 'James',
             Operations);
  mgr.SetAll('Stewart', 'Linda',
             Operations, '003685');
  memOutput.Clear;
  memOutput.Lines.Add('OPERATIONS DIVISION');
  memOutput.Lines.Add('Supervisor: ' +
                      mgr.GetLastName + ', '
                      + mgr.GetFirstName);
  memOutput.Lines.Add('Employee:   ' +
                      emp.GetLastName + ', '
                      + emp.GetFirstName);
  emp.Free;
  mgr.Free;
end;

end.
```



```
OPERATIONS DIVISION
Supervisor: Stewart, Linda
Employee:   Thompson, James
```

Figure 13. *Object-Oriented Programming Example*

Linux Compatibility Issues

When developing for Windows, we sometimes rely upon some technologies that, in all likelihood, will not be ported to Linux, such as ActiveX and Open Database Connectivity (ODBC). Although the use of ActiveX controls will not be ported to Linux, the other aspects of Delphi will be. Any operations that can be performed with the native components, including creating new components, will be available in Kylix. As for ODBC, drivers will be available to provide access to several popular databases, including MySQL™, InterBase®, and many others.

If you are presently using ActiveX Data Objects (ADO) or Component Object Model/Distributed Component Object Model (COM/DCOM) in your applications, you may wish to do some further investigation as to their availability on Linux. At this time, I know of no efforts on Microsoft's part, with the exception of COM/DCOM, to release these technologies for Linux.

For component developers, there are substantial differences between the Windows and Linux versions of Delphi, but application developers will find very subtle differences. Some key differences follow:

- Most component and property names are the same, but there are some new properties as well as some missing ones.
- The Linux file system is different from that of DOS/Windows. Drive access is different and file names are case sensitive.
- As previously stated, ActiveX is not supported under Linux. Additionally, Object Linking and Embedding (OLE) is unavailable. Thus, the ComObj, ComServ, ActiveX, and Windows units do not exist in Kylix.

Built-in Debugger

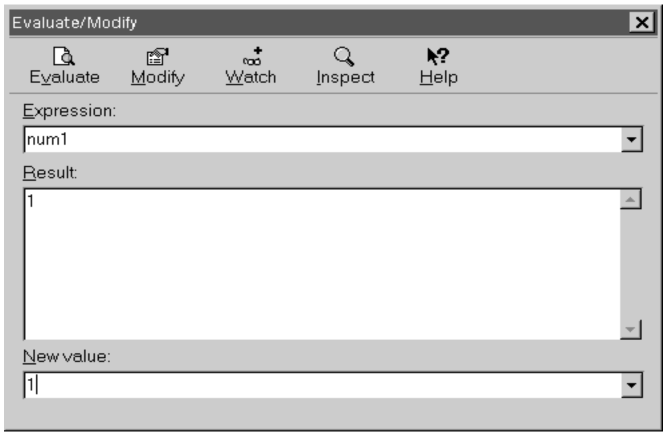
Like VB, Delphi contains a built-in debugger to assist the programmer in tracing code and locating errors. To use the Delphi debugger, the integrated debugging option must be enabled. To enable integrated debugging, select Tools|Debugger Options… from the menu, check the Integrated debugging box at the bottom left-hand side of the Debugger Options window, and click the OK button.

The debugger commands are available through the Run menu and the Debug toolbar. Additionally, debugger commands can be quickly accessed through the Code Editor window. Right-click anywhere inside the Code Editor window to open a pop-up menu. The Debug option on this pop-up menu lists the available debugger commands.

The Delphi debugger provides a semi-automatic method of locating errors. It enables a programmer to watch specific variables or expressions without modifying the program code. Additionally, the debugger can stop the program execution at designated breakpoints or execute the program code step-by-step. Note that the debugger is a design-time utility; none of the debugger commands may be used in an executable module at run-time outside of the Delphi IDE.

Source breakpoints are toggled on and off at specific lines of code designated by the programmer. Breakpoints can be set only on executable lines of code. Blank lines, declaration statements, and comments cannot have breakpoints. When a breakpoint is encountered, program execution is temporarily halted until the programmer selects Run from the Run menu, presses F9, or left-clicks the Run button on the Debug toolbar. The Delphi debugger also allows address, data, and module load breakpoints.

While a program is halted, the programmer can immediately evaluate and modify expressions in the Evaluate/Modify window (Figure 14). An expression may also be viewed and changed in the Inspector window shown in Figure 15. The Inspector window provides the programmer with a better view of objects that have advanced data structures.

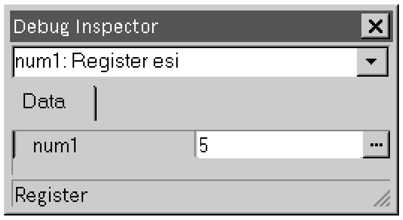


Expression: num1

Result: 1

New value: 1

Figure 14. *Evaluate/Modify Window*



num1: Register esi

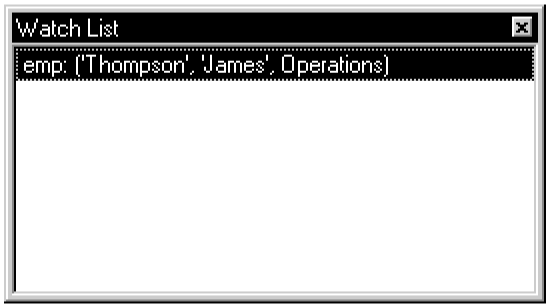
num1	5

Register

Figure 15. *Inspector Window*

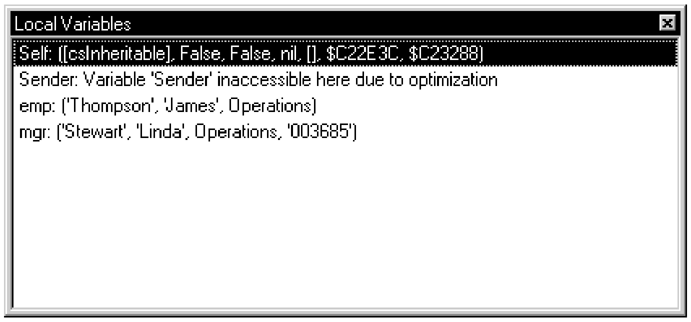
The Delphi debugger has two different stepping operations. Trace Into executes code one statement at a time. For example, if the statement is a call to a subprogram, the next statement displayed is the first statement in the subprogram. Step Over executes a subprogram call as a single unit, and then steps to the next statement in the current subprogram. Thus, in every situation, Step Over moves to the next statement in the current subprogram.

A watch expression is a user-defined expression that allows the programmer to observe its behavior. Watch expressions appear in the Watch window (Figure 16), and their values are automatically updated in break mode. Furthermore, the Local Variables window automatically displays the values of all declared variables (all local variables) in the current subprogram as shown in Figure 17.



```
emp: ('Thompson', 'James', Operations)
```

Figure 16. *Watch Window*



```
Self: ([csInheritable], False, False, nil, [], $C22E3C, $C23288)
Sender: Variable 'Sender' inaccessible here due to optimization
emp: ('Thompson', 'James', Operations)
mgr: ('Stewart', 'Linda', Operations, '003685')
```

Figure 17. *Local Variables Window*

Application Deployment

This section discusses how to compile Delphi projects and distribute the final application.

Building Executable Files

Much like C, C++, and other high-level languages, generating an executable file with Delphi is a two-step process. The first step is to compile the project. This step checks the syntax of each unit in the project and produces an object file for the unit. A unit object file has a DCU extension, meaning Delphi Compiled Unit.

Next, these object files must be linked. The linker takes each of the unit object files in the project and then combines them to produce a single executable file.

The Project menu in Delphi contains the commands for creating the executable file. The Compile option compiles each unit in the project, but it does not link them. The Build menu item compiles the units, if necessary, and then links them together, creating the final executable file.

Distributing the Application

If your project does not access a database, third-party Dynamic Link Library (DLL), or ActiveX control, then the executable file that is generated is a stand-alone executable. You can simply copy this executable file to a floppy disk or other removable media and run it on any machine that uses its target operating system. All of the Delphi controls used in the project are compiled into the resulting executable file. There is nothing extra that needs to be included with the application.

As with VB, if you use ActiveX controls in a project, the distribution of the application becomes more complicated. You must distribute the OCX file for each ActiveX control used in the project. Be sure to read the documentation for third-party ActiveX controls; some require additional DLLs.

For Windows, an elegant method of deploying an application is to create a setup program. If you create your own, be sure to register all ActiveX controls used by the application. This is accomplished by running the RegSvr32.exe utility.

The preferred method of creating a setup program is to use one of the many commercially available installation packages. VB includes the Package and Deployment Wizard for this purpose. Similarly, Delphi ships with InstallShield Express. This is a menu-driven installation utility that allows you to create a setup program for application deployment.

Additional References

I am the lead co-author of a VB textbook, and I have written a forthcoming Delphi textbook. While both of these texts are designed for introductory programming courses, they also prove to be invaluable references for advanced programmers. Both of these textbooks are published by Addison Wesley Longman (AWL). See the AWL web site (`www.awl.com/cs`) for more information regarding the texts.

Computer Programming Fundamentals with Applications in Visual Basic 6.0, by Mitchell C. Kerman and Ronald L. Brown

An Introduction to Computer Science: Programming and Problem Solving with Delphi (tentative title), by Mitchell C. Kerman

Additionally, there are several companies that offer seminar or classroom training on Delphi. RayTech Software, Inc. (certified as both a VB and Delphi trainer), offers training courses and consulting services. RayTech recently developed a full one-day seminar specifically for training VB developers in Delphi. RayTech courses range from certified classes that use standardized materials to custom courses tailored to meet your specific needs. For more information, see the RayTech Software web site, `www.raytech-software.com`.

If textbooks or training classes won't suit your needs, I recommend going to the source. I use the Microsoft and Inprise developer's web sites, online help utilities, and printed documentation. If you must use another printed reference, choose a book that comes directly from the source: Microsoft Press for Microsoft products and Borland Press through Macmillan Publishing.

Conclusions

Delphi is a fully capable RAD tool that harnesses the power of the Object Pascal language. It offers a variety of readily available components and tools, a hierarchical component design, true object-orientation, and an intuitive IDE. Furthermore, Delphi is currently available for Windows and the Linux version will be released soon.

While no single document can truly expound the benefits of Delphi, I hope that I have at least intrigued you with the ease with which you can migrate from VB to Delphi and become a proficient Delphi developer.

Enjoy your future with Delphi for Windows and Linux. Inprise Corporation has truly let the genie out of the bottle for Windows and Linux development, its name is Delphi, and you are granted an infinite number of wishes.

Inprise Corporation
100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com

Borland®