Moving Up: Borland Pascal

by Dave Jewell

In this article, Dave looks at some of the issues involved in moving to Delphi from Borland Pascal

Not including Delphi itself, there are four different Windows programming languages which might be termed 'mainstream' development environments. These are C, C++, Pascal and Visual Basic. Yes, I know that C++ is a superset of C, but it's rather a dangerous over simplification to lump C and C++ together. I've heard it said recently that programmers can pick up C++ more easily if they haven't had any previous exposure to C, and from my own experience there's certainly some truth in that point of view!

In the coming months, I'll be discussing Delphi from the viewpoint of a developer coming from one of these four camps. This time we kick off with Pascal and in the next issue we'll be looking at Delphi from the viewpoint of a Visual Basic Developer.

Delphi For Pascal Programmers

You might find it surprising that I've included Pascal in the above list. After all, Delphi is just Pascal hiding behind a pretty user interface isn't it ? Well, no – not really.

For starters, Delphi uses Borland's Object Pascal, an object-oriented Pascal dialect that offers much of the power of C++ in a far simpler, more manageable language. If you happen to be a seasoned Borland Pascal developer, and know the Windows API like the back of your hand, then you're in good shape for getting into Delphi. You'll find, however, that Borland have made a number of changes and enhancements to the language, making the new Pascal dialect even more powerful than it was previously.

If you're at all familiar with the Borland Pascal development system, you should have little difficulty in getting to grips with Delphi. Beneath Delphi's friendly visual development environment lurks the same compiler that you're familiar with. If you want to merely use Delphi as a "straight" Pascal compiler, there's nothing to stop you doing so. You can use it to compile and build your existing Pascal projects. However, it goes without saying that this approach misses out on all the major productivity benefits that come from using a visual development tool and the feature-rich component library.

This article is primarily concerned with the language changes that Borland incorporated into Delphi's particular version of the Pascal compiler. Although backwards-compatible with existing code, the new compiler incorporates a number of important language enhancements that we'll be looking at here.

New Language Features

With Delphi, Borland have introduced a number of new language features, many of which relate to the Object Browser interface. These language features generate additional categories of run-time information which are read by the Object Browser and used to fill in the browser window with properties and events that relate to the currently selected object.

The Class Declaration

The single most important language enhancement in Delphi's Pascal implementation is the introduction of the `class` declaration. Let's take a look at the `class` declaration of the `Shape` component, reproduced in Listing 1.

You can see that, superficially, it looks very much like the old-style `object` declaration used in previous versions of the compiler and, in fact `object` declarations are still supported. You must, however, use the new style `class` declaration when creating Delphi components.

The declaration starts off with the name of the new class, an equals sign, ("="), the reserved word `class` and the name of the parent class in parentheses. Like `object` declarations, a `class` declaration can contain both `private` and `public` sections.

In essence, there are now four different levels of protection within Delphi Pascal. In order of increasing accessibility, these are:

Listing 1

```
TShape = class(TGraphicControl)
  private
    FShape: TShapeType;
    FReserved: Byte;
    FPen: TPen;
    FBrush: TBrush;
    procedure SetBrush(Value:
      TBrush);
    procedure SetPen(Value:
      TPen);
    procedure SetShape(Value:
      TShapeType);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner:
      TComponent); override;
    destructor Destroy; override;
  published
    procedure StyleChanged(
      Sender: TObject);
    property Brush: TBrush
      read FBrush write SetBrush;
    property DragCursor;
    property DragMode;
    property Pen: TPen
      read FPen write SetPen;
    property Shape: TShapeType
      read FShape write SetShape;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;
```

- `private` – Only accessible within the defining unit.
- `protected` – Accessible to derived classes
- `public` – Full run-time accessibility
- `published` – Available to the Object Inspector for design-time manipulation

Property Definitions

The part that's really interesting is the new `published` section. The `published` part of a `class` declaration effectively corresponds to the Object Browser interface for that object. This enables the Object Browser (or anyone else who's interested) to retrieve information on an object, its properties and event handlers. There's nothing magical about the way that the Object Browser does this, it simply makes use of the designer interface unit, DSGINTF. You can find the source code to this unit in the DSGNINTF.PAS source file.

Let's look in detail at one of the property declarations in the above class definition:

```
property Brush: TBrush
  read FBrush write SetBrush;
```

This defines a property called `Brush`. Because the property is in the `published` section of the class, it will automatically be made available to the Object Inspector. Additionally, since it's a property, it can be manipulated just like any normal `public` element of the class – property elements are inherently `public`.

In the above declaration, the `Brush` property is defined as being of type `TBrush` – a handle to a Windows brush object. This is followed by information which tells the compiler how to access the property.

When reading the value of the `Brush` property, the compiler simply references the private `FBrush` field. However, when changing the value of the property, the `SetBrush` procedure (also a private element). This approach allows us to protect the private elements of a class from direct access, while at the same time presenting a convenient user interface that's just as convenient to use as if we had direct access to public elements.

Doing things in this way also allows us to perform other actions behind the scenes. We've just seen that the mere action of assigning to a `Shape` component's `Brush` property will invoke a routine called `SetBrush`. Internally, the `SetBrush` routine will not only store the new brush handle, but also typically redraw the component to reflect the brush change.

Exception Handling

With today's increasingly sophisticated applications, exception handling becomes less of an option and more of a must-have feature when designing any serious programming language. Exception handling allows you to localise error handling and recovery to one area and eliminates the need for repetitive checking for error conditions before and after every operation. Let's see how this works in practice.

The Try-Except Block

Consider this code:

```
function SafeDivide(
  A, B : Integer): Integer;
begin
  try
    {Point 1}
    SafeDivide := A div B;
  except
    {Point 2}
    on EDivByZero do
      SafeDivide := 0;
  end;
  {Point 3}
end;
```

The simple routine shown above is responsible for dividing two integers together and returning the result.

Pascal veterans will immediately spot two oddities here – the appearance of the `try` and `except` keywords. These new keywords are used to implement the exception handling mechanism.

In the case we're looking at, the `try` and `except` statements define a `try-except` block of code. Here, there's actually only one statement (the division statement) between these two keywords but there can potentially be many. Statements within this block of code execute completely normally, starting from Point 1, but if an exception occurs, control is immediately transferred to the `except` part of the block at Point 2. If no exception takes place, then once the `except` keyword is reached, execution continues at Point 3.

The net effect, of course, is that instead of the user being presented with a run-time error, this routine will silently return the value zero whenever a run-time error occurs.

In a more real-world situation, there would typically be a lot more code between the `try` and `except` keywords – code which would normally be full of lots of messy error-checking stuff.

By using an exception handling mechanism, the error checking can be done after the `except` keyword and things become very much neater.

This concept will perhaps be more familiar to Microsoft BASIC (including Visual Basic) programmers. BASIC provides a mechanism called `ON ERR`, which allows control to be transferred to a certain point in a routine whenever a run-time error takes place. The try-except mechanism is very similar in operation.

You'll also have noticed the `on` statement at Point 2 in the above source code. There are a considerable number of different exception types that can be tested for. In this case, we're testing for a divide by zero condition, but you can also test for floating point math errors, file I/O errors, and more.

The Try-Finally Block

In addition to `Try-Except` blocks, Delphi's Pascal language also provides `Try-Finally` constructs. These are particularly useful for Windows programming where it's often necessary to perform a certain amount of 'clearing up', such as de-allocating temporary memory buffers, deleting custom brushes and pens, closing files and so on. Here's how it works:

```
procedure TForm1.Button1Click(
  Sender: TComponent);
var
  pMem: Pointer;
begin
  GetMem (pMem, 2048);
  {...}
  FreeMem (pMem, 2048);
end;
```

In the above example, a 2Kb block of memory is allocated at the beginning of the routine and deallocated at the end. That's fine, but what would happen if an exception (such as a floating point error) were to occur before the `FreeMem` call was executed? In this case, the memory would remain allocated.

Of course, if the run-time error resulted in the program's termination, there'd be no real problem since Windows would deallocate the memory anyway. However, if you were allocating large bitmaps, pens, or brushes, these items would remain allocated even after the program terminated. When programming with Delphi, the correct approach is to use a `Try-Finally` block, which looks something like this:

```
procedure TForm1.Button1Click(
  Sender: TComponent);
var
  pMem: Pointer;
begin
  GetMem (pMem, 2048);
  try
    {...}
  finally
    FreeMem (pMem, 2048);
end;
```

With this approach, the statement(s) following the `finally` keyword will be executed even if the routine terminates with a run-time error. This guarantees that the allocated resource will be freed no matter what happens.

The AS, IS And IN Keywords

The `as` and `is` keywords are used to implement run-time type checking and typecasting. For example, the following statement will determine whether an object, `xObj`, is of a given type:

```
if xObj is TForm then ....
```

This statement will return true if `xObj` is a Form component, ***or*** if it is of a type that's descended from a Form component.

Similarly, the `as` keyword can be used to perform run-time typecasting, like this:

```
with xObj as TForm do begin
  {...}
end;
```

In this example, the `xObj` object is treated as a Form component within the block. The `as` keyword will perform internal checking to ensure that it's valid to treat the `xObj` as if it were a Form component (specifically, that it is a Form component, or is derived from one). If not valid, then a `EInvalidCast` exception will be raised.

The `in` keyword will be familiar to most Pascal programmers as a test of set membership:

```
if TheInt in [1,3,5,7,9] then
  ...
```

However, this particular keyword now has a new meaning within the context of a `USES` clause inside Delphi project files:

```
uses
  Forms,
  Sdimain in 'SDIMAIN.PAS'
    {SDIAppForm},
  About in 'ABOUT.PAS'
    {AboutBox};
```

Changes To The Language

The version of Borland Pascal on which Delphi is based incorporates a number of useful language enhancements. In most cases, these are backwards-compatible. This means that they won't break any existing code.

However, there are a few pitfalls for the unwary so read the following sections with care.

The Result Variable

When developing a Pascal function, it's often useful to be able to "look" at the return result. Previously, it wasn't possible to do this, since specifying the name of the function in an expression was interpreted as a recursive call:

```
function GetFileHandle(
  fName: PChar): Integer;
begin
    GetFileHandle :=
      _lopen(fName, 0);
    if GetFileHandle = -1 then
      MessageBox(0,
        'Can''t open file',
        'Error', mb_ok);
end;
```

The reference to `GetFileHandle` in the `if` statement will be interpreted by the compiler as a recursive call which obviously isn't what's wanted. The compiler will fail to compile the code anyway, complaining that no arguments have been supplied for the (supposed) call to `GetFileHandle`. In order to get around this, most Pascal programmers use a local variable like this:

```
function GetFileHandle(
  fName: PChar): Integer;
var  fd: Integer;
begin
  fd := _lopen (fName, 0);
  if fd = -1 then
    MessageBox (0,
      'Can''t open file',
      'Error', mb_ok);
  GetFileHandle := fd;
end;
```

There's nothing wrong with this, of course, provided that you don't mind the unnecessary tedium of defining a local variable and (more importantly) remembering to set up the function result at the end!

The new `Result` variable does away with these considerations. It behaves as a predefined local variable but it also happens to correspond to the function result. Unlike the actual function name, you can use it anywhere in an expression without implying a recursive call. Here's how you'd recode the above example:

```
function GetFileHandle(
  fName: PChar): Integer;
begin
  Result := _lopen (fName, 0);
  if Result = -1 then
    MessageBox (0,
      'Can''t open file',
      'Error', mb_ok);
end;
```

This gives the best of both worlds; concise and elegant yet without irrelevant variables.

Note: There's an obvious caveat here. When porting old code to Delphi, it's a good idea to rename any local or global variables named `Result` or potential ambiguities may arise.

Function Result Types

While on the subject of function results, Borland have relaxed the previous restrictions on permissible function result types. In the words of the on-line help documentation:

"Functions can now return any type, whether simple or complex, standard or user-defined, except old-style objects (as opposed to classes), and files of type text or 'file of'. The only way to handle objects as function results is through object pointers."

Open Array Construction

Some time ago, Borland introduced Open Array parameters, which allow you to pass an array type as a parameter to a function or procedure. Inside the called routine, you can use the built-in `Low` and `High` operators to obtain the lower and upper array bounds of the array. In this way, you could, for example, pass an arbitrarily large array to a function which would then return the average value of all the elements of the array.

Delphi's version of Pascal makes this facility even more flexible, by letting you build an array and pass it to a routine in a single operation:

```
Average := CalcAverage([5, 7,
  9, 14, 234, 86]);
```

The corresponding function declaration would be:

```
function CalcAverage(Nums:
  Array of Integer): Integer;
```

Delphi includes a new routine, `Format`, which takes a pointer to a destination character array, a format string, and an open array parameter. In essence, this gives all the power and flexibility of the C language's `sprintf` statement, something that's sure to be good news for Pascal programmers.

Note: Since the elements of the array are enclosed in square brackets, this can look just like a set. Take care not to confuse the two.

Case Statement Optimizations

Borland have made two changes to the way `case` statements operate. Firstly, it's no longer possible to have overlapping ranges in a `case` statement. For example:

```
case Errcode of
  7: Writeln(
    'Disk is write protected');
  1..100: Writeln(
    'Unknown error');
end;
```

This code will compile fine under previous versions of the Pascal compiler but won't be accepted by Delphi since 7 obviously overlaps with the range 1..100.

The second change concerns the way in which the compiler generates code for `case` statements. Basically, if the various `case` constants are sorted in ascending order, then the compiler converts the `case` statement into a number of jumps.

On the other hand, a non-sorted ordering of `case` constants will result in multiple calculations being carried out. It's therefore better to sort your `case` constants into ascending order if possible. For example:

```
case ErrCode of
  1: Writeln("This is case 1");
  2: Writeln("This is case 2");
  5: Writeln("This is case 5");
  {...}
```

Using Your Old Code

Delphi is perfectly capable of using your old code, integrating it into a new-style Delphi project. If the old code is in the form of a DLL, then you can just call the DLL from Delphi. Existing units can also be easily integrated into Delphi programs. Of course, old source code won't have any knowledge of Delphi's component library and VCL framework, but provided that the DLL has been well structured, it should be relatively easy to move it across.

But what about OWL, I hear you cry? Well, admittedly, this could be something of a problem. You can certainly use Delphi to compile all your existing OWL library source code and applications if you wish to continue using the OWL application framework. It should go without saying, though, that you can't readily mix OWL code with the new VCL library. At the time of writing, there's been no commitment from Borland as regards the implementation of a 32-bit OWL library. (Because of the differences between the Win32 and Win16 APIs, it's not just a simple matter of recompiling OWL with a 32-bit compiler).

My personal advice would be to bite the bullet and port your applications to VCL. Not only will you be able to use all Delphi's user interface components, (giving your program a much nicer user interface), but you'll also be assured of portability to the world of 32-bits, be it Windows/NT or Windows 95.

This is probably a good place to point out the importance of 'decoupling' the user interface of an application from the nuts and bolts of the program code. Whatever sort of application you're writing, always make a clear distinction between what the program ***does*** and what the program ***displays*** on the screen. If you always bear this in mind, then you can put the essence of your program into units or even DLLs, completely distinct from whatever user interface and application framework you might be using. If you've adopted this sort of approach with your OWL applications, then you will have greatly simplified the job of moving across to Delphi and the VCL library.

This article is based on an extract from Dave's new book, "Instant Delphi", published by Wrox Press. Dave Jewell is a freelance consultant/programmer, specialising in systems-level work under Windows and DOS. You can contact Dave on the internet as djewell@cix.compulink.co.uk