Dragging And Dropping Part 2: Enhanced VCL

by Brian Long

Last month we looked at basic drag and drop in Delphi applications, using the support built into the VCL. This month, we will progress onto custom drag objects and see what they can do for us.

Drag Control Objects

When a drag and drop application has a drop target control that can accept many different dragged source controls, the implementation of the `OnDragOver` and `OnDragDrop` event handlers can end up getting a little complex. Often, whilst there are a variety of dragged source controls that can be accepted, they will ultimately all provide the same sort of information, for example a file name.

To simplify this sort of scenario, you can use custom drag objects (sometimes called drag control objects), which were introduced in a very understated fashion in Delphi 2. The `OnStartDrag` event handler of all controls that can be dragged can each create an instance of a class inherited from `TDragObject`. This object is used to represent the information that is being transferred from the dragged control to the drop target.

The current drag control class hierarchy can be seen in Figure 1, with Figure 2 showing the original hierarchy. Whilst `TDragObject` is the key base class, you will typically be interested in inheriting from the more able `TDragControlObject` class. We will look into some of the capabilities of these two classes throughout the rest of this article. You can see in Figure 1 that this hierarchy also has a class that is used for drag and dock support, as introduced in Delphi 4. Dragging and docking support uses the same underlying mechanism to operate and we might come back to look at it in more detail in a future article.

To set the custom drag object up, you assign the created instance to the `DragObject var` parameter of the source control's `OnStartDrag` event handler, which defaults to `nil`. Having done this, when the source control is dragged over and dropped on a target control, one of the parameters of the target control's `OnDragOver` and `OnDragDrop` event handlers changes.

Specifically, the `Source` parameter now refers to the custom drag object, instead of the dragged control itself. Since the custom drag object is given to the target control in these event handlers, it can be interrogated for useful information, which could be any additional data fields or properties defined in the class.

If you are writing code that might be compiled in various versions of 32-bit Delphi, you must be careful about which class you inherit your custom drag object class from. In Delphi 2 and 3, the `Source` parameter would only represent your drag object if you did not inherit from `TDragControlObject`. Instead, you must inherit directly from `TDragObject`. Delphi 4 (and later) remedies this problem. You can inherit from any point in the hierarchy and `Source` will correctly represent your custom drag object.

The online help in Delphi 3, 4 and 5 claims that you do not need to free the drag object created in an `OnStartDrag` event handler (Delphi 2 neglected to describe it in the help). However, this information is *incorrect*. Delphi will only automatically free drag objects that it creates on your behalf when you do not create your own drag objects. This problem has been reported and *might* be fixed in a future web update of the VCL help.

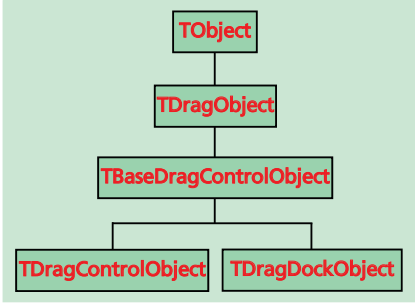
When using custom drag objects, you should verify in the `OnDragOver` event handler (and maybe also in the `OnDragDrop` event handler) that the `Source` parameter is actually a drag object before performing any typecasts. The normal Delphi way of doing this would involve using an expression like:

```
Source is TDragObject
```

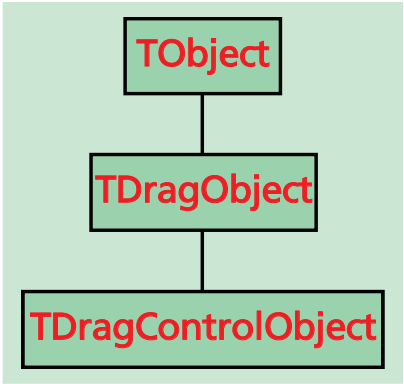
However, for reasons that will become clearer a little later, you should use this expression instead:

```
IsDragObject(Source)
```

In a normal application, the effect of these two expressions will be identical; however, `IsDragObject` caters for other scenarios that the



➤ *Figure 1: The drag control object hierarchy in Delphi 4 and later.*



➤ *Figure 2: The drag control object hierarchy in Delphi 2 and 3.*

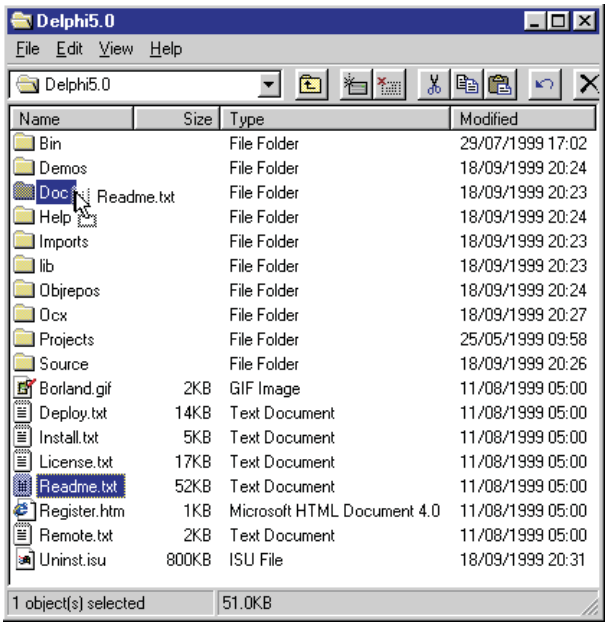
```
type
  TTextDragObject = class(TDragControlObject)
  public
    Data: String;
  end;
  TForm1 = class(TForm)
    ...
  private
    FDragObject: TTextDragObject;
  end;
...
procedure TForm1.Label1StartDrag(Sender: TObject;
  var DragObject: TDragObject);
begin
  FDragObject := TTextDragObject.Create(Sender as TLabel);
  FDragObject.Data := TLabel(Sender).Caption;
  DragObject := FDragObject;
end;
procedure TForm1.ListBox1StartDrag(Sender: TObject;
  var DragObject: TDragObject);
begin
  FDragObject := TTextDragObject.Create(Sender as TListBox);
  with TListBox(Sender) do
    FDragObject.Data := Items[ItemIndex];
  DragObject := FDragObject;
end;
procedure TForm1.SharedEndDrag(Sender, Target: TObject;
  X, Y: Integer);
begin
  // All draggable controls share this event handler
  FDragObject.Free;
  FDragObject := nil
end;
procedure TForm1.Panel1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  // It is tempting to write this...
  // Accept := Source is TTextDragObject
  // ...however we are advised to write this instead
  Accept := IsDragObject(Source)
end;
procedure TForm1.Panel1DragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  // The OnDragOver event handler verified we are dealing
  // with a drag object so there is no chance of getting a
  // normal control
  (Sender as TPanel).Caption := TTextDragObject(Source).Data
end;
```

➤ *Listing 1: Using drag control objects.*

functionality of the `is` operator does not.

The DragObjects.Dpr sample project tries to show the general idea. It has a number of controls on a form that can be dragged onto a panel, such as a listbox, a button, a combobox and a label. The plan is that the various controls all provide one piece of textual information, but the text is meant to come from different properties (the active listbox item, the button's caption, and so on). To try and help out, each control's `OnStartDrag` creates an instance of a class inherited from `TDragControlObject`.

➤ *Figure 3: A customised drag cursor showing the file being dragged.*



Because of this fact (in combination with the problem outlined above) the project will only work correctly in Delphi 4 and 5. An alternative project, which is called DragObjects2.Dpr, has another drag object class inherited from `TDragObject`, which will do the trick for Delphi 2 and 3.

The new class in the project DragObjects.Dpr is called `TTextDragObject` and has just one extra string data field called `Data`, which is given the piece of text as appropriate from each control.

The target panel's `OnDragOver` and `OnDragDrop` event handlers are therefore considerably simpler in implementation, as they just treat the `Source` parameter as a `TTextDragObject` and read the `Data` field. All controls that have an `OnStartDrag` event handler share an `OnEndDrag` event handler that frees the drag object. A number of these event handlers are shown in Listing 1.

You should be able to see that using custom drag objects allows a drag and drop application to be readily extensible. You can add more controls to a form which can be dragged from and, so long as they all create the same custom drag object and fill in the required data fields, the drop target need not be changed at all. It will continue to work regardless of the drag source, because it gets the information from the custom drag object, not the drag source itself.

Although this business of creating drag objects is being introduced as if it is something over and above the normal VCL drag and drop support, in truth it is not. The VCL creates a drag object for each drag operation anyway. If you leave the `DragObject` parameter in the `OnStartDrag` event handler with its default `nil` value, or have no `OnStartDrag` event handler at all, the VCL creates a `TDragControlObject` instance to represent the drag operation. This behind-the-scenes drag object *will* be automatically freed by the VCL.

Customising The Drag Cursor Further

Drag objects can also be used as more flexible ways of specifying the drag cursor to be used when source controls are accepted or rejected, potentially using an image list component.

`TDragObject` has a protected virtual method `GetDragCursor` that takes a `Boolean var` parameter called `Accept`, and also the mouse co-ordinates. It is supposed to return a `TCursor` value depending on whether the target control accepts the dragged control or not. It is hard-coded to return either `crNoDrop` or `crDrag`.

Another protected virtual method is defined, called `GetDragImages`. The drag object can supply an image list containing an

```
type
  TTextDragObject = class(TDragControlObject)
  private
    FDragImages: TDragImageList;
  protected
    function GetDragCursor(Accepted: Boolean;
      X, Y: Integer): TCursor; override;
    function GetDragImages: TDragImageList; override;
  public
    Data: String;
    destructor Destroy; override;
  end;
...
destructor TTextDragObject.Destroy;
begin
  FDragImages.Free;
  inherited;
end;
function TTextDragObject.GetDragCursor(Accepted: Boolean;
  X, Y: Integer): TCursor;
begin
  if Accepted then
    Result := crPacMan
  else
    Result := inherited GetDragCursor(Accepted, X, Y)
end;
function TTextDragObject.GetDragImages: TDragImageList;
var
  Bmp: TBitmap;
  Txt: String;
begin
  if not Assigned(FDragImages) then
    FDragImages := TDragImageList.Create(nil);
  Result := FDragImages;
  Result.Clear;
  Bmp := TBitmap.Create;
  try
    // Make up some string to write on bitmap
    Txt := Format(' The control called %s says "%s" at %s',
      [Control.Name, Data,
      FormatDateTime('h:nn am/pm', Time)]);
    Bmp.Canvas.Font.Name := 'Arial';
    Bmp.Canvas.Font.Style :=
      Bmp.Canvas.Font.Style + [fsItalic];
    Bmp.Height := Bmp.Canvas.TextHeight(Txt);
    Bmp.Width := Bmp.Canvas.TextWidth(Txt);
    // Fill background with olive
    Bmp.Canvas.Brush.Color := clOlive;
    Bmp.Canvas.FloodFill(0, 0, clWhite, fsSurface);
    // Write a string on bitmap
    Bmp.Canvas.TextOut(0, 0, Txt);
    Result.Width := Bmp.Width;
    Result.Height := Bmp.Height;
    // Make olive pixels transparent,
    // whilst adding bmp to list
    Result.AddMasked(Bmp, clOlive)
  finally
    Bmp.Free;
  end
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  Screen.Cursors[crPacMan] :=
    LoadCursor(HInstance, 'PacMan');
  ControlStyle := ControlStyle + [csDisplayDragImage];
  for I := 0 to ControlCount - 1 do
    with Controls[I] do
      ControlStyle := ControlStyle + [csDisplayDragImage];
end;
```

➤ *Listing 2: Setting up a drag image list.*

image that will be merged with the drag cursor (from `GetDragCursor`) to produce a combined, possibly more informative, drag cursor. The implementation of `GetDragImages` in `TDragObject` simply returns `nil`.

You have probably seen the sort of effect that `GetDragImages` is designed for when dragging files in Windows Explorer. The drag cursor is enhanced by a faint representation of the item being dragged around (see Figure 3, where README.TXT is being dragged into a DOC directory).

`TDragControlObject` is the useful class that inherits from `TDragObject`. It keeps a reference to the control being dragged in the public `Control` property (although the original Delphi 2 implementation had the protected and public sections of the class the wrong way round, and so the `Control` property was actually protected).

By knowing which control is being dragged, the `GetDragCursor` method is overridden to return either `crNoDrop` or the control's `DragCursor` property (as opposed to being fixed to `crDrag`).

It also overrides the `GetDragImages` method and calls the control's `GetDragImages` method, rather than returning `nil`. Admittedly, the only controls that do anything in their `GetDragImages` methods are `TCustomTreeView` and `TCustomListView` (and their descendants), but the scope is there for controls to supply their own drag image list to enhance the drag cursor (as I will describe later). However, right now we are not interested in how components can supply custom image lists, but instead how the drag control object can do this.

A variation on the project DragObjects.Dpr is given in DragImage.Dpr. This project has a label and a listbox that can both be dragged onto a panel. The label gives its caption to the panel and the listbox gives its active item, and so a custom drag object is used to hold this information string. However, as well as adding the textual data field, this custom drag object class overrides both `GetDragImages` and `GetDragCursor`.

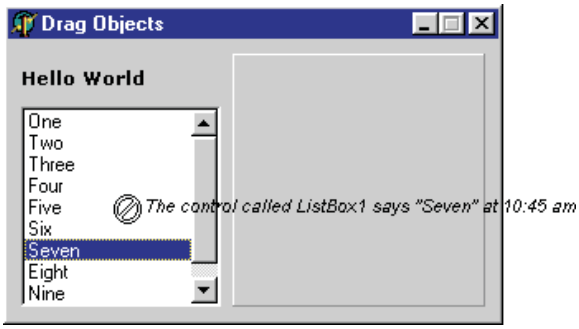
`GetDragCursor` is overridden to provide a custom drag cursor for all controls that make one of these drag objects. It uses the same custom cursor as was used in last month's article (the *PacMan* cursor). Notice that a custom drag object can be used to allow many drag source controls to have the same drag cursor, without having to set each control's `DragCursor` property.

`GetDragImages` is overridden to create an instance of a `TDragImageList`. This class, which was introduced in Delphi 3, inherits from `TCustomImageList` and is an ancestor of the `TImageList` component class. It provides enough functionality to cater for the requirements of drag cursor building.

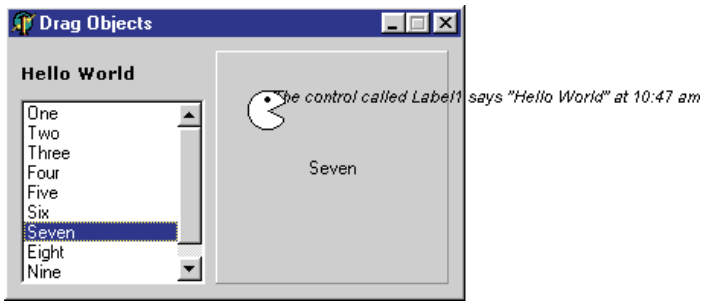
Listing 2 shows the custom drag object class with the two additional methods. The drag image list is stored in a private data field and the destructor ensures that it gets destroyed. You can see `GetDragCursor` returning the *PacMan* cursor when needed.

The `GetDragImages` method is a little more involved. If no drag image list has been created yet, one gets created by the method. Then a bitmap is set up, large enough to hold the image that is chosen to represent the dragged item. In this case the code simply makes a string describing the drag source, the dragged information and the time that the drag started. This information is written onto the bitmap, and the bitmap is added into the image list.

In order to get transparent areas in the image, the bitmap was initially flood-filled with olive. The `AddMasked` image list method was



➤ *Figure 4: A No Drop drag cursor enhanced by a drag image list.*



➤ *Figure 5: A custom drag cursor enhanced by a drag image list.*

used to add the bitmap to the image list whilst specifying that all olive pixels are to become transparent.

Unfortunately, whilst on first glances (and after checking with the help) this would seem enough to do the job, it is not. The `ControlStyle` property help is very misleading with respect to the `csDisplayDragImage` setting. It suggests that including this flag in a control's `ControlStyle` property will make the enhanced drag image for that control be used whenever and wherever the control is dragged. Unfortunately, the enhanced drag image will only be used when the mouse is over any control that has this setting, or when the mouse is not over any form in the project.

So the image list will only be used when the mouse is either not over a possible target (which means when the mouse is not over any form in the application) or when it is over a control that has the `csDisplayDragImage` value in its `ControlStyle` set property. Only treeviews and listviews include this member in their `ControlStyle` property, so the drag image list will only be used when the mouse is over a treeview or listview, or entirely off the form. This has been logged as a bug in the VCL, as opposed to a bug in the online help.

To fix this problem in the application, the form's `OnCreate` event handler iterates through all its controls, adding `csDisplayDragImage` into the `ControlStyle` property. Once this is done, we get what we were after. When a control is dragged over something that does not accept it, it looks like Figure 4, and when it is over something that does accept it, it looks like Figure 5.

Note that this application has a simple form, where each control on the form has no child controls. In a more complex form, you will need to recursively loop through each control and its children, setting the `ControlStyle` property, as is done by the `FixControlStyles` procedure in Listing 3.

➤ *Listing 3: Generic solution to fix the ControlStyle problem.*

```
procedure FixControlStyles(Parent: TControl);
var
  I: Integer;
begin
  Parent.ControlStyle := Parent.ControlStyle + [csDisplayDragImage];
  if Parent is TWinControl then
    with TWinControl(Parent) do
      for I := 0 to ControlCount - 1 do
        FixControlStyles(Controls[I]);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FixControlStyles(Self);
  ...
end;
```

Custom Components And Drag Image Lists

The point was made earlier that all controls have a `GetDragImages` method that is automatically called by the drag objects that are created automatically by the VCL. Only listviews and tree views override this method (the Win32 Common Controls API supports setting up drag image lists automatically for these controls). These are also the only components to include `csDisplayDragImage` in their `ControlStyle` property.

You can write your own custom component classes that supply their own self-maintained image list by overriding the `GetDragImages` method. Two sample components are on the disk this month to show the idea. `TDragButton` inherits from `TButton`, and can be found in the DragButton.pas unit. `TDragEdit` is inherited from `TEdit`, and can be found in the DragEdit.pas unit.

These control classes do a number of similar things. They both automatically start a drag operation if the user `Ctrl`-clicks on them. This is done in an overridden version of the `MouseDown` method, which normally is only responsible for triggering the `OnMouseDown` event. They also both define a private data field called `FDragImages`, which is a `TDragImageList`. They add the `csDisplayDragImage` setting into their `ControlStyle` property inside the constructor. The other thing they do is to override the `GetDragImages` method to add some image into their drag image list.

In the case of the button component, it adds a bitmap that is a transparent representation of itself to the image list (see Figure 6). This is achieved by calling the button's `PaintTo` method, telling it to paint a copy of itself onto the bitmaps canvas. `AddMasked` is used to add the bitmap to the image list, specifying that all pixels that

```
type
  TDragButton = class(TButton)
  private
    FDragImages: TDragImageList;
  protected
    function GetDragImages: TDragImageList; override;
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;
...
constructor TDragButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle + [csDisplayDragImage]
end;
function TDragButton.GetDragImages: TDragImageList;
var
  Bmp: TBitmap;
begin
  if not Assigned(FDragImages) then
    FDragImages := TDragImageList.Create(Self);
  Bmp := TBitmap.Create;
  try
    Bmp.Width := Width;
    Bmp.Height := Height;
    Bmp.Canvas.Lock;
    try
      PaintTo(Bmp.Canvas.Handle, 0, 0);
    finally
      Bmp.Canvas.Unlock
    end;
    FDragImages.Width := Width;
    FDragImages.Height := Height;
    FDragImages.AddMasked(Bmp, clBtnFace);
    Result := FDragImages;
  finally
    Bmp.Free
  end
end;
procedure TDragButton.MouseDown(Button: TMouseButton; Shift:
  TShiftState; X, Y: Integer);
begin
  inherited;
  // Automatically start dragging on a Ctrl-click
  if ssCtrl in Shift then
    BeginDrag(True)
end;
```

➤ *Listing 4: A button component supplying a custom drag image list.*

match `clBtnFace` (the button's main colour) should be made transparent. Listing 4 shows how this is all achieved. Unfortunately, due to the lack of the canvas's `Lock` and `Unlock` methods in Delphi 2, the `PaintTo` method is ineffective in that version. Delphi 3 (and later) supports it fine, however.

The edit component tries something different. It loads a bitmap (of Athena) that is compiled in as a Windows resource (see Listing 5). Again, when adding the image to the image list, the common background colour (`clSilver`) is specified as the transparent colour. Clearly, having a copy of the large Athena bitmap hanging off the drag cursor is not very practical, but it does emphasise what you can achieve with custom drag images.

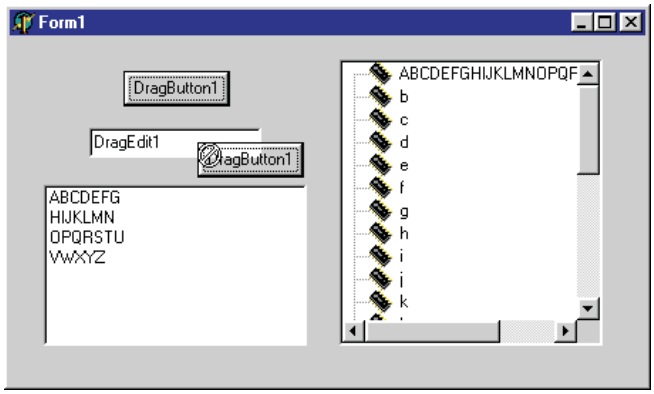
```
function TDragEdit.GetDragImages: TDragImageList;
var
  Bmp: TBitmap;
begin
  if not Assigned(FDragImages) then
    FDragImages := TDragImageList.Create(Self);
  Bmp := TBitmap.Create;
  try
    Bmp.LoadFromResourceName(HInstance, 'Athena');
    FDragImages.Width := Bmp.Width;
    FDragImages.Height := Bmp.Height;
    FDragImages.AddMasked(Bmp, clSilver);
    Result := FDragImages;
  finally
    Bmp.Free
  end
end;
```

➤ *Listing 5: An edit component supplying a custom image list.*

DLLs And Dragging

There is one more benefit of using custom drag objects that we should look into before leaving the subject. It involves dragging between a form created in a DLL and a form created in either a different DLL, or the main EXE. Incidentally, if you are using Delphi packages (a special type of DLL specific to Delphi and C++Builder) this issue does not arise: this only applies to normal DLLs.

A pair of projects are included on the disk which represent an executable and a DLL (ExeDrag.Dpr and DllDrag.Dpr respectively). The projects can be compiled and executed from any 32-bit version of Delphi. There is also a project group (ExeAndDllDragging.Bpg) containing these two projects that can be used in Delphi 4 or later.



➤ *Figure 6: Another enhanced drag cursor, this time managed by the component itself.*

The DLL contains a form class and exports a routine that displays it. In order for forms created in DLLs to display correctly, the DLL's `Application` object needs to have its `Handle` property assigned the value of the EXE's `Application.Handle`. The exported DLL routine takes a window handle (assumed to be the `Application` object handle) and assigns it to its own `Application.Handle`. Without this, each form from the DLL would have an extra icon on the task bar.

The form is destroyed when closed, thanks to the `OnClose` event handler assigning a value of `caFree` to its `Action` parameter (a `var` parameter), as shown in Listing 6.

The form in the DLL has a memo which can have selected text dragged from it (by dragging with the right mouse button). A problem with this is that `TDragObject` (which captures the mouse during a drag operation, and handles the resulting mouse messages) does not react to the right mouse button being released in Delphi 2 or 3. So dragging with the right mouse

button only works well in Delphi 4 or later (as do a number of other things relating to drag and drop, as we have seen). When you release the right mouse button, you must follow this with a click of the left mouse button, if running with the earlier versions of Delphi.

The form in the EXE has an edit control which is coded to accept anything dragged from a `TCustomEdit` (or any descendant of that class). Listing 7 shows both of these sections of code.

A `TMemo` is a descendant of `TCustomEdit` and so the code might be expected to work. However, because the memo lives in the DLL and the edit control's event handlers are in the EXE, things don't go according to plan. The edit appears not to want to accept anything from the memo.

The EXE's `is` expression will be asking the DLL's memo object whether its VMT (virtual method table) matches that of `TCustomEdit` (or some descendant), but will be referring to the implementation of `TCustomEdit` in the code compiled into the EXE. Since the memo inherits from `TCustomEdit` as compiled into the DLL, the VMTs of the two versions of `TCustomEdit` will be at different addresses and so `is` will return `False`.

It is probably a good idea that it fails, as `TCustomEdit` is a class that is quite far down the VCL hierarchy, and there is always the possibility that the DLL and EXE are compiled with different versions of Delphi. Each version of Delphi makes various changes around the VCL. Consequently, the internal layout of data fields and the content of the VMT could be rather different. Treating one `TCustomEdit` object (compiled with one version of Delphi) as if it were the other (compiled with a different version) could cause havoc.

So, the way around this problem is to use custom drag objects to represent the information being dragged across, in conjunction with the aforementioned `IsDragObject` function. Drag objects are instances of quite shallow classes, not far from `TObject` in the VCL hierarchy. Things are less likely to change in these classes from one version to the next as they are with component classes, although they still do. Consequently, it is still important to ensure that the DLL and EXE are compiled with the same version of Delphi.

`IsDragObject` does not use `is` to find out if the object in question (passed as the `Source` parameter to `OnDragOver` and `OnDragDrop`) inherits from `TDragObject`. Instead, it compares the class name of the given object against the class name of `TDragObject`. If there is no match, it goes back to the ancestor of the supplied object and tries again. Eventually, it will either find a match or it won't, so the function will return `True` or `False`.

Clearly you could write a similar routine that would do the same job for edit controls or memos, but the fact that `IsDragObject` exists already suggests that it is easiest to use custom drag objects when dragging between forms from different binary modules.

Assuming `IsDragObject` returns `True`, you can then apply a static

```
procedure ShowForm(ApplicationHandle: HWnd); stdcall;
begin
  // Set Application object window handle to match that in the EXE,
  // meaning we do not get another task bar button for the form
  Application.Handle := ApplicationHandle;
  TDLLForm.Create(Application).Show
end;
procedure TDLLForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  // The form frees itself when closed
  Action := caFree
end;
```

➤ *Listing 6: A routine exported from a DLL that creates and shows a form.*

```
// This code is from the DLL form
procedure TDLLForm.Memo1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  // Check for right mouse button, and no other buttons/keys
  if Shift = [ssRight] then
    (Sender as TCustomEdit).BeginDrag(True)
end;
// This code is from the EXE form
procedure TExeForm.Edit1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TCustomEdit
end;
procedure TExeForm.Edit1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  (Sender as TCustomEdit).Text := (Source as TCustomEdit).SelText
end;
```

➤ *Listing 7: Drag and drop code from both the DLL and the EXE.*

➤ *Listing 8: Drag and drop code that works between a DLL and an EXE.*

```
// This code is from the DLL form
procedure TDLLForm.Memo1StartDrag(Sender: TObject;
  var DragObject: TDragObject);
begin
  DragObject := TTextDragObject.Create;
  TTextDragObject(DragObject).Data := (Sender as TMemo).SelText;
  FDragObject := DragObject
end;
procedure TDLLForm.Memo1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  FDragObject.Free;
  FDragObject := nil
end;
// This code is from the EXE form
procedure TExeForm.Edit1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := IsDragObject(Source)
end;
procedure TExeForm.Edit1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  (Sender as TCustomEdit).Text := TTextDragObject(Source).Data
end;
```

typecast to `Source` to turn it into a reference to your `TDragObject` descendant. In this case, the custom drag object inherits from `TDragObject` directly (not `TDragControlObject`). This means that the code will work in all Delphi versions from 2 onwards, but it does mean that the `DragCursor` property values will be ignored if you set them. It also means that trying to use the application in Delphi 2 or 3 will show up the problem of dragging with the right mouse button.

Listing 8 shows the `OnStartDrag` and `OnEndDrag` event handlers for the memo from the DLL along with the `OnDragOver` and `OnDragDrop` event handlers of the edit control in the EXE, now that they have been fixed to work as required.

Since we are on the subject of DLLs at the moment it might be useful to mention that a drag object has an `Instance` method that returns the instance handle for the module that created it. An instance handle is the address at which that module was loaded, so for EXEs, the instance handle will always be `$400,000`, but will be different for all the DLLs in a given application's address space.

In addition, `TDragObject` defines a virtual method `GetName` that returns (by default) the object's class name as a string. This can be overridden in descendant classes.

Summary

That's it for VCL drag and drop. Next month we finish the series by looking at how to do drag and drop with other applications in the Windows environment. This will involve writing message handlers, calling Windows APIs, implementing COM interfaces and using custom clipboard formats, so if you need to do any background reading, start now!

Acknowledgements

Thanks go to Roy Nelson from Inprise/Borland's European Technical Team. It was Roy who first pointed me in the direction of how to get custom drag image lists operational.

Brian Long is a UK-based freelance consultant and trainer. He spends most of his time running Delphi and C++Builder training courses for his clients, and doing problem-solving work for them. You can reach Brian at brian@blong.com