The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Memos And Scroll Bars

Q I have noticed that a TListBox automatically adds a scrollbar when there are too many items to fit. On the other hand, a TMemo component does not do this. How can I get a TMemo to add a scroll bar when there are more lines than can be drawn at any one time?

A The TListBox component does not really add a vertical scroll bar automatically. A TListBox is a wrapper around a Windows listbox control. It is the underlying control that is being helpful, not the Delphi component class.

Unfortunately, the Windows memo control is not so helpful in this regard. It is down to the programmer to choose which scroll bars to have. You can choose a vertical scroll bar, a horizontal scroll bar, both of these scroll bars or neither. The TMemo component has a ScrollBars property to surface these choices to the component user.

If you want a memo component to act in a similar way to a listbox, then you will need to check the details of the memo's contents whenever it is changed.

To customise a memo component that is in an application, you can make an OnChange event handler for it, and write some appropriate code that checks whether scroll bars are needed, setting the properties as appropriate. To act just like a listbox, only the vertical scroll bar should be taken into consideration.

Listing 1 has some code that will do this (from the MemoEg.Dpr project on the disk). The event handler first verifies that a memo triggered the event. Then it calculates how many whole lines of text can be displayed in the memo by dividing the memo's client height (which means that it discounts the border of the control) by the height of one line of text. The text height calculation is more complex than it could be (a TCanvas object has a TextHeight method) for reasons explained in the comments.

The position of the cursor (and the length of selected text, if any) are then recorded. If the vertical scrollbar is toggled, the underlying memo control will be destroyed and then recreated with appropriate flags, so we can then reposition the cursor where it was.

If there are more lines of text in the memo than can be displayed at one time, the scrollbar is added, otherwise it is removed. You can see that the memo's line count is calculated by sending a dedicated message to the memo, rather than by using the Count property of its TStrings object, represented by the Lines property. This is because the Count property will ignore blank lines at the end. In this application, however, any line that is outside the displayable area should cause a scrollbar to appear.

An alternative would be to make a new component that did this automatically. If we are writing a new component, then we should not use the OnChange event (which is intended for component users),

➤ *Listing 1: A memo component made to auto-detect the need for a scroll bar.*

```
type
  TWCAccess = class(TWinControl);
function TextHeight(Ctrl: TWinControl; const Msg: String):
  Integer;
var
  DC: HDC;
  OldFont: HFont;
  Size: TSize;
begin
  { Can't just ask a control for the font height, as Delphi }
  { caches the font and doesn't select it into the device }
  { context until some drawing is required. }
  { The memo may have a different font to its form and }
  { under those circumstances, you could get bad results. }
  { Access control's device context }
  DC := GetDC(Ctrl.Handle);
  try
    { Ensure font is selected into DC (saving old font) }
    OldFont := SelectObject(DC, TWCAccess(Ctrl).Font.Handle);
    try
      { Find text height }
     {$ifdef Win32}
      Win32Check(GetTextExtentPoint32(DC,PChar(Msg),1,Size));
     {$else}
      GetTextExtentPoint(DC, @(Msg[1]), 1, Size);
     {$endif}
      Result := Size.cy
    finally
      { Put old font back into memo }
      SelectObject(DC, OldFont)
    end;
  finally
    { Let the DC go }
    ReleaseDC(Ctrl.Handle, DC)
  end;
end;
procedure TForm1.Memo1Change(Sender: TObject);
var
  Memo: TMemo;
  MemoNumLines: Integer;
  OldSelStart, OldSelLength: Integer;
begin
  if Sender is TMemo then
    Memo := TMemo(Sender)
  else
    Exit;
  MemoNumLines :=
    Memo.ClientHeight div TextHeight(Memo, 'X');
  { Record where we were }
  OldSelStart := Memo.SelStart;
  OldSelLength := Memo.SelLength;
  { Would use the Count property of Lines, but }
  { this doesn't count a blank line at the end }
  { if Memo.Lines.Count > MemoNumLines then }
  if Memo.Perform(EM_GETLINECOUNT, 0, 0) > MemoNumLines then
    Memo.ScrollBars := ssVertical
  else
    Memo.ScrollBars := ssNone;
  { Go back to old position after memo control (possibly) }
  { recreated }
  Memo.SelStart := OldSelStart;
  Memo.SelLength := OldSelLength;
end;
```

```
type
  TDCMemo = class(TMemo)
  private
    FAutoScrollBar: Boolean;
    procedure SetAutoScrollBar(const Value: Boolean);
    function TextHeight(const Msg: String): Integer;
  protected
    procedure Change; override;
    procedure CheckScrollBar; virtual;
  published
    property AutoScrollBar: Boolean read FAutoScrollBar
      write SetAutoScrollBar default False;
  end;
...
procedure TDCMemo.Change;
begin
  inherited Change;
  CheckScrollBar
end;
procedure TDCMemo.SetAutoScrollBar(const Value: Boolean);
begin
  if FAutoScrollBar <> Value then begin
    FAutoScrollBar := Value;
    CheckScrollBar;
  end
end;
procedure TDCMemo.CheckScrollBar;
var
  MemoNumLines: Integer;
  OldSelStart, OldSelLength: Integer;
begin
  { Only proceed if the memo has a parent, and so is
    on-screen }
  if Parent = nil then
    Exit;
  MemoNumLines := ClientHeight div TextHeight('X');
  { Record where we were }
  OldSelStart := SelStart;
  OldSelLength := SelLength;
  if Perform(EM_GETLINECOUNT, 0, 0) > MemoNumLines then
    ScrollBars := ssVertical
  else
    ScrollBars := ssNone;
  { Go back to old position after memo control (possibly)
    recreated }
  SelStart := OldSelStart;
  SelLength := OldSelLength;
end;
```

➤ *Listing 2: A componentised version of Listing 1.*

but instead override and extend the polymorphic `Change` method (whose job is to call the `OnChange` event handler, and exists for component *writers*). Listing 2 shows a possible implementation, which includes a property that enables and disables this automatic scrollbar behaviour.

This code is in the `DCMemo` unit on the disk, which works in all versions of Delphi. You'll notice that I skipped the diverse calculation of the text height and went directly to the memo's canvas. This may be asking for trouble, bearing in mind what was written in the previous application's comments, but is fine if the memo's font is the same as the form's.

MIDAS Delta Packets

Q I have been trying for some time to get information out of the `Delta` property of a `TClientDataSet`, which holds information on modifications, inserts and deletes since updates were last applied.

I attended a talk at a recent conference about client datasets and the speaker implied it was possible to access the information in the `Delta` property without any other MIDAS components (providers) being involved. Is this possible?

As you might understand, I am trying to avoid the MIDAS licence fee for my application and still use the `TClientDataSet` component.

A Client dataset components are used in thin-client MIDAS applications to contain the data for displaying on the UI. They are populated by provider components in middle tier MIDAS applications, which may be using BDE components to get the data in the first place. They can also be used in normal BDE-less applications as a simple way of storing and manipulating a dataset, without the overhead of a database engine.

The client dataset has two prime properties for storing the data. The `Data` property is an `OleVariant` containing an array of bytes which represent the original data. `Delta` is another `OleVariant` array of bytes which records information about any changes that have been made to the data. These two properties are referred to as the data packet and the delta packet.

In a multi-tiered MIDAS application, changes in the delta packet are applied to the server using either the `Reconcile` or `ApplyUpdates` method. These send the delta packet back to the middle tier application to deal with as appropriate. In single tier applications (as per the question) changes are applied by calling `MergeChangeLog`. This merges the delta packet into the data packet.

In the help for *delta packets, editing* it shows how to use a provider's `OnUpdateData` event handler to iterate through the elements in a delta packet. `Delta` is passed to this event handler as a `TClientDataSet`, and you use the `UpdateStatus` property of the client dataset to ascertain what type of change each record represents.

However, providers tend to imply use of MIDAS in a distributed application, requiring licence fees. So we need to look for another option to answer the question, which talks about non-licensed MIDAS applications (which are one tier applications that do not send data packets from one application to another).

Fortunately, the answer ends up being quite straightforward. To look at one client dataset's `Delta` property, you can assign it directly to another client dataset's `Data` property. The two properties are both `OleVariant` arrays of bytes, and are assignment compatible. Again, the `UpdateStatus` property of the second client dataset is used to identify what change happened to each record.

Each change made to the original client dataset is recorded in one or two records in the `Delta` property. If the record was modified, the delta packet holds a copy of the original unmodified record, as well as another record containing just the changed fields. If a record is modified more than once, the delta packet merges the new changes with the original change details.

An application on the disk called CDSDelta.Dpr shows how this works. It involves one client dataset that has been populated with some of the records from the BioLife.DB sample table and which is connected, via a datasource, to a `DBGrid`, `DBImage` and `DBMemo`.

The user is able to edit all fields in the table except the graphic field. The `AfterPost` event (triggered after a record is changed or added) and `AfterDelete` event (triggered after a record is deleted) share an event handler. The code in the event handler is minimal. It closes a second client dataset, then assigns the first client dataset's `Data` property to the second one's `Delta` property. Finally, it opens the second client dataset, which is connected to its own `DBGrid`, `DBImage` and `DBMemo` controls.

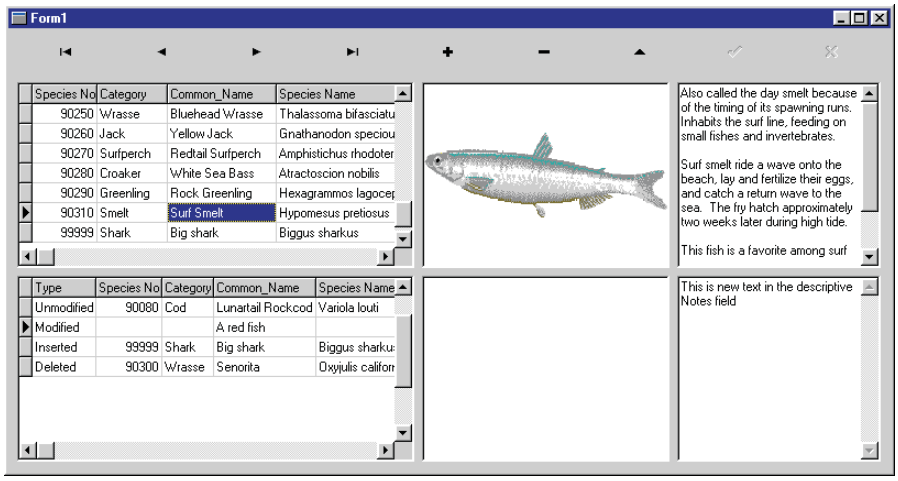
The property assignment and opening of the second client dataset are only executed if there are any changes represented in the original `Delta` property (the `ChangeCount` property is checked). This is because an error is produced if you attempt to read from an empty `Delta` property.

You might think that the test would be irrelevant, since the code only executes after a change actually occurs, but this is not so. If you make a change to a record and, for example, make another change that effectively sets the record back to its original state, the change is removed from the `Delta` packet. Consequently, it is possible to end up with an empty `Delta` property.

The only other code in the program is an `OnCalcFields` event handler for the second client dataset. A calculated field has been added to it to describe what form of change is represented by that record in the `Delta` packet (unmodified, modified, inserted or deleted).

You can see the code in Listing 3 and the program, after a few changes have been made, is shown in Figure 1. When changes are recorded in the `Delta` property, BLOb fields are omitted, unless they themselves have been modified. This explains why you do not see any fish pictures in the lower `DBImage` (the BLOb field is never modified by the program).

One point to remember about single tier thin-client applications written using client datasets is that you must still distribute one support DLL from the BDE (even though they do not use any normal BDE components). In the case of Delphi 3 and 4, you must distribute DBCLIENT.DLL. Delphi 5 renames this to MIDAS.DLL.



➤ *Figure 1: Viewing a client dataset's Delta property.*

➤ *Listing 3: Code to assign a client dataset's Delta property to another client dataset.*

```
procedure TForm1.CDS1AfterChange(DataSet: TDataSet);
begin
  CDS2.Close;
  //Error arises if Delta is empty
  if CDS1.ChangeCount > 0 then begin
    CDS2.Data := CDS1.Delta;
    CDS2.Open
  end
end;
procedure TForm1.CDS2CalcFields(DataSet: TDataSet);
begin
  case CDS2.UpdateStatus of
    usUnmodified: CDS2Type.Value := 'Unmodified';
    usModified:   CDS2Type.Value := 'Modified';
    usInserted:   CDS2Type.Value := 'Inserted';
    usDeleted:    CDS2Type.Value := 'Deleted';
  end
end;
```

Drawing On Status Bars

Q Both Microsoft Word and Windows Explorer have icons displayed on their status bars under some circumstances. I can only get a status bar to display text. How do I draw pictures on it?

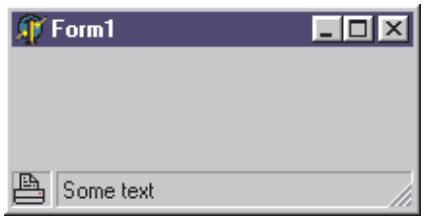
A A status bar can either display one simple text panel or, potentially, multiple panel sections. You set up the multiple panel sections (or status panels) by using the property editor for the `Panels` property. The property editor allows you to manufacture `TStatus- Panel` objects.

Using the `SimplePanel` property, you can dictate whether the status bar displays all the status panels (when `SimplePanel` is `False`) or just the one simple panel (when `SimplePanel` is `True`).

When the simple panel is displayed, the status bar's `SimpleText` property controls what is written. When multiple panels are displayed, they each have a `Text` property that controls what they display. At least, this is the case when the `TStatusPanel` object's `Style` property is set to `psText`. If you change this property to `psOwnerDraw` in any status panels, then you can take control of drawing them.

The code that draws these owner draw panels is placed in the status bar's `OnDrawPanel` event handler. This event handler is passed the status bar object, the status panel that needs drawing, and also a `TRect` record that describes the area in the status bar occupied by the panel.

A project on the disk shows an example of drawing into an owner drawn status bar panel, which runs in all 32-bit Delphi versions. It uses an image list set up with a few



➤ *Figure 2: A status bar with a picture on it.*

images in it, along with a timer component. Every second, the status bar (called `Bar`) is invalidated, causing it to redraw. This will trigger the `OnDrawPanel` event of the status bar which asks the image list to draw one of its images into the status panel. You can see the program running in Figure 2.

Accessing Application Icons

Q I am writing a program that will launch applications and I want it to be able to access the icon associated with any Windows application, so I can display it somewhere. Windows applications are not usually installed with a separate icon file, so how can I access it?

A Windows Explorer accesses application icons using a dedicated routine from the Windows shell API. `SHGetFileInfo` actually extracts many pieces of information about a specified file to enable Windows Explorer to describe the file in various ways to the user.

The function is defined in the `ShellAPI` unit as shown in Listing 5.

```
procedure TForm1.CDS1AfterChange(DataSet: TDataSet);
begin
  CDS2.Close;
  //Error arises if Delta is empty
  if CDS1.ChangeCount > 0 then begin
    CDS2.Data := CDS1.Delta;
    CDS2.Open
  end
end;
procedure TForm1.CDS2CalcFields(DataSet: TDataSet);
begin
  case CDS2.UpdateStatus of
    usUnmodified: CDS2Type.Value := 'Unmodified';
    usModified:   CDS2Type.Value := 'Modified';
    usInserted:   CDS2Type.Value := 'Inserted';
    usDeleted:    CDS2Type.Value := 'Deleted';
  end
end;
```

➤ *Listing 4: Code to show varying images on a status bar's panel.*

```
type
  _SHFILEINFOA = record
    hIcon: HICON;
    iIcon: Integer;
    dwAttributes: DWORD;
    szDisplayName: array [0..MAX_PATH-1] of AnsiChar;
    szTypeName: array [0..79] of AnsiChar;
  end;
  SHFILEINFOA = _SHFILEINFOA;
  SHFILEINFO = SHFILEINFOA;
  TSHFileInfoA = _SHFILEINFOA;
  TSHFileInfo = TSHFileInfoA;
function SHGetFileInfo(pszPath: PAnsiChar; dwFileAttributes: DWORD;
  var psfi: TSHFileInfo; cbFileInfo, uFlags: UINT): DWORD; stdcall;
```

➤ *Listing 5: SHGetFileInfo and its associated record type, as defined in Delphi 5.*

The Windows SDK help describes the record that `SHGetFileInfo` works with as being a `SHFILEINFO` record. Delphi defines this type (in version 4 and later) but also defines the more Delphi-esque `TSHFileInfo` record (in all 32-bit versions), allowing you a choice.

To get information, you pass the full path of a file as the first parameter to `SHGetFileInfo`. The second parameter is only used in very specific circumstances, so passing a zero for it will suffice. A record variable (whose type definition is shown in Listing 5) should be passed as the third parameter, and its size as the fourth. The record will be filled in with information by the function. The final parameter is a combination of flags that specify what information to retrieve.

There are a total of sixteen flags to choose from, but the more useful ones are shown in Table 1. A sample project ShellIcon.Dpr is on the disk, showing how to call the function. In fact it calls it several times to get several pieces of information.

The code is shown in Listing 6. An open dialog is used to choose a file and the resultant filename is passed to each `SHGetFileInfo` call. The first call requests the file's display name (returned in the record's `szDisplayName` field) and file type description (returned in the `szTypeName` field).

The next call passes `SHGFI_EXETYPE` as the only flag to `SHGetFileInfo`, causing it to return a 32-bit value that indicates the executable type. If the high word is zero and the low word contains the letters `MZ`, the file is a DOS batch file or executable. If the high word is zero and the low word contains `PE`, the file is a Win32 console application. If the high word contains `$300`, `$350` or `$400` and the low word is

➤ *Table 1: A selection of SHGetFileInfo flags.*

SHGetFileInfo flag	Flag meaning
SHGFI_DISPLAYNAME	Get the display name for the file
SHGFI_TYPENAME	Get the file type description
SHGFI_EXETYPE	Get the executable type
SHGFI_ICON	Get normal icon, whose size is defined by system metric values
SHGFI_ICON or SHGFI_SMALLICON	Get small icon
SHGFI_ICON or SHGFI_LARGEICON	Get large icon
SHGFI_ICON or SHGFI_SELECTED	Get normal icon blended with system highlight colour

```
procedure TMainForm.btnChooseFileClick(Sender: TObject);
var
  FI: TSHFileInfo;
  ExeType: DWord;
const
  MZ = $5A4D; //"MZ"
  NE = $504E; //"NE"
  PE = $4550; //"PE"
begin
  if dlgOpen.Execute then begin
    //Get display name and type description
    SHGetFileInfo(PChar(dlgOpen.FileName), 0, FI, SizeOf(FI),
      SHGFI_DISPLAYNAME or SHGFI_TYPENAME);
    lblDisplayName.Caption := FI.szDisplayName;
    lblFileType.Caption := FI.szTypeName;
    //Get EXE type
    ExeType := SHGetFileInfo(PChar(dlgOpen.FileName), 0, FI,
      SizeOf(FI),SHGFI_EXETYPE);
    if ExeType = MZ then
      lblExeType.Caption := 'MS-DOS .EXE, .COM or .BAT file'
    else if ExeType = PE then
      lblExeType.Caption := 'Win32 console application'
    else if ((LoWord(ExeType) = NE) or
      (LoWord(ExeType) = PE)) and
      ((HiWord(ExeType) = $0300) or
      (HiWord(ExeType) = $0350) or
      (HiWord(ExeType) = $0400)) then
      lblExeType.Caption := 'Windows application'
    else
      lblExeType.Caption := 'Not an executable file';
    //Get large icon
    SHGetFileInfo(PChar(dlgOpen.FileName), 0, FI,
      SizeOf(FI), SHGFI_ICON or SHGFI_LARGEICON);
    imgLarge.Picture.Icon.Handle := FI.hIcon;
    //Get small icon
    SHGetFileInfo(PChar(dlgOpen.FileName), 0, FI,
      SizeOf(FI), SHGFI_ICON or SHGFI_SMALLICON);
    imgSmall.Picture.Icon.Handle := FI.hIcon;
    //Get selected icon
    SHGetFileInfo(PChar(dlgOpen.FileName), 0, FI,
      SizeOf(FI), SHGFI_ICON or SHGFI_SELECTED);
    imgSelected.Picture.Icon.Handle := FI.hIcon;
  end
end;
```

➤ *Listing 6: Getting file information with the Shell API.*

either NE or PE, the file is a Windows application. Otherwise, the file is not an executable.

The last three calls each get an icon associated with the file, these being a large icon, small icon and an icon that looks selected (by being blended with the system highlight colour). You can see the program showing these icons, and the other information gleaned, in Figure 3.

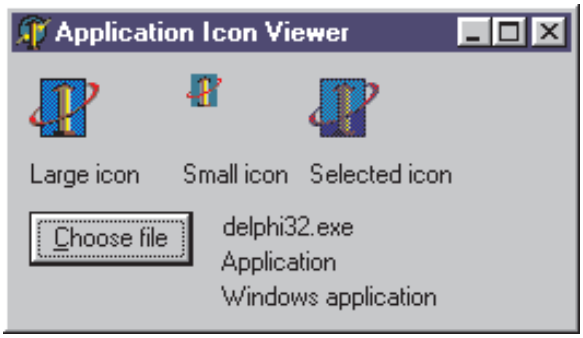
COM RTL Support

Q I know that there are two RTL routines that can be used to connect to an Automation object by specifying a ProgID (CreateOleObject and GetActiveOleObject). However, for connecting to COM objects using a ClassID there only appears to be one (CreateComObject). I would like to connect to a COM object that might be registered in the Running Object Table. How can I do this?

A When connecting to an Automation object (a COM object that implements an interface based on IDispatch) it is quite common to use CreateOleObject or GetActiveOleObject. GetActiveOleObject looks for an Automation object of the requested type that is registered as the active object in the Windows Running Object Table (ROT). If one exists, it returns a reference to its IDispatch interface. If one is not found, it raises an EOleSysError exception. CreateOleObject ignores any active instance of an Automation object and creates a new one (if possible). If one cannot be created for any reason, an EOleSysError is raised.

These routines use a ProgID (programmatic identifier) to identify the type of object that is required. A ProgID is a string made of two words separated by a full stop. The first word represents the program that implements the object. The second word represents the specific object required. This ProgID is defined by the object itself. For example, Microsoft Word 97 has two ProgIDs, Word.Application and Word.Basic, that allow access to the VBA Automation object and the WordBasic Automation object respectively.

To connect to an existing Automation object if one exists, or to a new instance if one does not exist, you use code like that shown in Listing 7.



➤ *Figure 3: Extracting file information.*

```
var
  Server: Variant;
...
try
  Server := GetActiveOleObject('ServerApp.AutoObj')
except
  Server := CreateOleObject('ServerApp.AutoObj')
end;
```

➤ *Listing 7: Connecting a running object, or failing that to a new object.*

Most commercial Automation objects will register their first instance in the ROT to allow multiple client applications access to the same instance of the Automation object, rather than each client getting a separate instance of the Automation object. Delphi Automation objects do not do this unless you add your own code in to do it. There was some discussion of this subject in *The Delphi Clinic* in Issue 46, and also in *COM Corner* in Issue 53.

Automation objects are just COM objects at the end of the day. They just happen to implement IDispatch and publish a ProgID. When talking to COM objects, it is common to refer to them using a ClassID. This is a GUID (**G**lobally **U**nique **ID**entifier), a 128-bit number, often displayed as a string (particularly in the Windows registry), used to uniquely identify a COM class. A ClassID is passed to CreateComObject in order to create a new instance of a COM object of a given class and get back a reference to its IUnknown interface.

However, if an instance of the COM class already exists and has

registered itself as the active object in the ROT, the Delphi RTL provides no COM equivalent to the `GetActiveOleObject` Automation support routine. The symmetrically equivalent routine you are looking for is `GetActiveComObject`, but unfortunately this routine does not exist in the RTL. However, you can implement one using the code in Listing 8 (`OleCheck` is defined in the `ComObj` unit).

This allows you to access the running (or new) COM object using code laid out similar to Listing 7, something like Listing 9.

More than likely, the reason the RTL is devoid of this routine is that COM objects are typically implemented in in-proc (in process) servers, which are DLLs loaded into the address space of the client application. COM object instances created by in-proc servers cannot be accessed by other applications. They must create their own instances through their own locally loaded copy of the in-proc server. On the other hand, Automation servers are typically implemented as out-of-proc (out of process) servers, which are EXE files. COM object instances created in an out-of-proc server can register themselves in the ROT and be accessed by other client applications.

Object Assignment

Q I have defined a class `TFoo`, which has a simple string attribute, and I create two instances of it (Listing 10). I understand that the statement:

```
A := B;
```

causes the two variables `A` and `B` to point to the same object (the one that `B` was pointing to). If I then do:

```
A.Data := 'Other value';
```

then `B.Data` also contains '`Other value`'. What I want to know is how I can copy the contents of `B` to the contents of `A` instead of copying just the address of one to the address variable of the other?

A I'm glad this question arrived recently because it also allows me to logically answer the next question which has been pending for a while (see below).

Because Delphi implements objects via pointers (object references) and dynamic memory management (which occurs during construction and destruction), this is a common problem. Assigning an object reference variable to another object reference variable copies the address of the objects on the heap, not the contents of the objects. Fortunately, the VCL has had a mechanism designed specifically for this problem since its inception in Delphi 1.

The `TPersistent` class, which inherits from `TObject`, has a public `Assign` method and a lesser known protected `AssignTo` method. These both take a parameter defined as a `TPersistent` object and are discussed across six pages in Danny Thorpe's book *Delphi Component Design*, now sadly out of print.

An application can call the `Assign` method of a destination object, passing a source object as a parameter to it. You can override `Assign` in your class and write code that copies the attributes of the source object into the destination object. The job of `Assign` is therefore to do whatever is necessary to clone the source object.

When implementing `Assign`, make sure you check for a `nil` parameter value. When `nil` is passed to `Assign`, you should set your instance data fields to their default initialised state. The `TFoo` class from Listing 10 has been given an `Assign` method in the code in Listing 11.

Because of the `TPersistent` parameter type, `Assign` can also cater for source objects that are instances of different classes, if that is appropriate. However, anything that you do not support should be passed onto the inherited version of `Assign`.

If the source object ends up being passed back to the original

➤ *Listing 8: How to implement the missing COM RTL routine.*

```
function GetActiveComObject(const ClassID: TGuid): IUnknown;
begin
  OleCheck(GetActiveObject(ClassID, nil, Result));
end;
```

➤ *Listing 9: Accessing a running (or new) COM object.*

```
const
  CLASS_AutoObj: TGUID = '{00DEFE03-E9DB-11D3-96EC-444553540000}';
...
var
  Server: IAutoObj;
...
try
  Server := GetActiveComObject(CLASS_AutoObj)
except
  Server := CreateComObject(Class_AutoObj)
end;
```

➤ *Listing 10: A simple class.*

```
TFoo = class
public
  Data: String;
  constructor Create(AData: String);
end;
...
constructor TFoo.Create(AData: String);
begin
  Data := AData
end;
...
var
  A, B: TFoo;
...
A := TFoo.Create('Object A');
B := TFoo.Create('Object B');
```

```
TFoo = class(TPersistent)
public
  Data: String;
  constructor Create(AData: String);
  procedure Assign(Source: TPersistent); override;
end;
...
procedure TFoo.Assign(Source: TPersistent);
begin
  if Source = nil then
    Data := ''
  else if Source is TFoo then
    Data := TFoo(Source).Data
  else
    inherited Assign(Source);
end;
...
var
  A, B: TFoo;
...
A := TFoo.Create('Object A');
B := TFoo.Create('Object B');
A.Data := 'Other value';
B.Assign(A);
```

➤ *Listing 11: A simple class that supports being copied from and to.*

implementation of `Assign` in `TPersistent`, then the whole operation gets turned around. The code calls the source object's `AssignTo` method, passing the destination object as the parameter. The implication here is that if the destination object is unaware of how to deal with the source object, then the source object *might* be aware of how to copy itself into the destination object.

The reason `AssignTo` was added to `TPersistent` was to allow new classes to be written, which can be effectively assigned to older classes without necessarily having to rewrite the code in those older classes.

Object Property Assignment

Q Delphi lets me assign from one object property directly to another one, without any complaint. In fact, I can assign from the same object property (say, the `Lines` property of a `TMemo`) to another object property (say, the `SQL` property of a `TQuery`) repeatedly, without any apparent problems occurring in my program. My concern arises because I know that Delphi uses pointers to refer to objects. Surely, a statement such as:

```
Query1.SQL := Memo1.Lines;
```

causes the address of the `Lines` object to be copied across and stored in the `SQL` pointer variable?

A That conclusion is exactly the one you might draw if you had just digested the previous question. Yes, Delphi uses object references, which contain the address of an object on the heap. However, no problem occurs in the statement offered by the questioner thanks to the way properties are implemented. Remember that the statement in question is dealing with two *properties*, not two *variables*.

A property itself has no storage space allocated for it by the compiler. A property is defined in terms of a type (so the compiler can validate values assigned to it, or variables it is assigned to), and what should happen when it is read from and written to.

Typically, when a property is read from (such as when it is assigned to something), a private or protected function method is called. That method will return a value that can be used. Sometimes, if there is little point in having such a method, the property can be defined to read directly from a data field in the class.

When a property is written to, typically the value assigned is passed as a parameter to a private or protected procedure method.

To understand why the property assignment goes according to plan, you need to look at the definition of the `SQL` property (or any similar property exposed by the many VCL components). The `SQL` property in the `TQuery` class is defined like this:

```
property SQL: TStrings
  read FSQL write SetQuery;
```

This means that the statement shown above actually gets implemented as:

```
Query1.SetQuery(Memo1.Lines);
```

So what looked like an assignment has changed into a method call. If you look into the implementation of the `TQuery.SetQuery` method, you will see that it effectively boils down to this:

```
Query1.Close;
Query1.UnPrepare;
Query1.SQL.Assign(Memo1.Lines);
```

So, ultimately, because of the camouflaging mechanism we call a property, the assignment look-alike turns into a call to the `Assign` method, which we looked at in the previous question.

As Danny Thorpe says in his book: '*Where C++ has copy constructors, implicitly created temporary instances, and assignment operators, Delphi has Assign methods and property write methods*'.