The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

Bug With Global Variables?

Q I've found what looks like a bug in the compiler's treatment of global variables. On the global variables help page it states that they are initialised to zero by default. However, in a trivial application I can prove this to be incorrect (see Listing 1 and Figure 1). Is this a known bug?

A The section in the help that makes the statement referred to in the question has changed as Delphi versions have come and gone: in fact I cannot find any such statement in the help for Delphi 1 or 2. However, looking up *global variables* in Delphi 3, then choosing *Variable declarations* gives this text:

If a global variable declaration does not explicitly specify an initial value, the memory occupied by the variable will initially be set to zero.

➤ *Figure 1: Unexpected output from the program in Listing 1.*

Similarly, looking up global variables in Delphi 4 or later gives:

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Local variables, in contrast, cannot be initialized in their declarations and contain random data until a value is assigned to them.

After reading this information, and the short program in Listing 1, anyone would see the behaviour demonstrated in Figure 1 as a bug. However, this is not the view of Borland. According to Chief Delphi Architect, Chuck Jazdzewski, this behaviour is as designed. The logic goes as follows:

I and J, in this case, are considered local to the `begin..end` *block that follows and are allocated by the compiler as if they were declared to be local variables of that 'procedure'. If there was a real procedure declaration between the variable declaration and the* `begin..end`*, they would be treated as global variables.*

So, because the variables are only accessed by the main program block, and there are no other subroutines around, the compiler decides to treat it as a procedure with local variables. Given this extra information, it all becomes clear, but the online help certainly doesn't help us arrive at this conclusion for ourselves. Clearly, to fix (or rather to work around) the problem, you can define a dummy subroutine below the variables in the project source file.

➤ *Listing 1: A trivial project using a 'global' variable.*

```
program Project1;
{$OPTIMIZATION ON}
uses
  Dialogs, SysUtils;
var
  I: Integer; //should default to 0
  J: Integer = 1;
begin
  I := J-I; //I should be 1 now
  ShowMessage(IntToStr(I)); //should display 1 in a dialog, but won't
end.
```

Record Property Issues

Q I have a few record properties being used in my application and I've bumped into some anomalous compiler behaviour with them. I can assign a record to a record property as expected. I can also use a `with` clause and assign to the individual fields of the record property. However, I cannot make a direct assignment to any field of the record property. Why the difference in behaviour with a `with` and without a `with`?

A Let's have a good look at this problem to find out what's going on. Listing 2 shows a summary of what has been described in the question. Assigning a whole record to the record property is acceptable to the compiler: no surprises there. The next statement in the listing shows what the compiler will not accept. Attempting to write directly to a field of the record property is apparently forbidden, so let's look at why this is.

The first thing to remember is that a property is not the same as a variable. It has no storage space of its own, and you cannot pass a property to a subroutine's `var` parameter. A property is a mechanism that is defined in terms of a data type, and what happens when it's read from and written to. Any operation that reads from the property is substituted by the compiler with either a read from some data field specified in the property definition or, alternatively, a call to a specified function. Similarly, any operation that assigns a value to a property is substituted with either an assignment to the data field, or a

procedure call with the assigned value being passed as a parameter.

When you assign to a field of a record variable, the compiler will identify the offset that the field starts at, relative to the start of the variable, and write to the appropriately sized memory block starting at that address, thereby updating the appropriate portion of the record variable. There is no equivalent concept with properties.

When you place a property on the left side of an assignment statement, the idea is that you are writing a new property value. The property concept revolves around properties having a value assigned to them. There are no special cases where some types of properties can have values assigned to a small portion of them. After all, if the property has its write operation defined in terms of a procedure call, there is no way for the compiler to control what happens in that routine.

So, in short, when you have a record property, you can only assign a whole record to it. If you want to update a single field of the record, read the whole record into a temporary variable, update the field and write the whole record back.

Now let's look at the last statement in Listing 2. On the surface, the use of the `with` clause looks like syntactic sugar: you'd expect to get the same results (a failed compilation) as in the previous case without it. However, this time the compiler is fooled into thinking the statement is an acceptable request. The `with` clause causes the compiler to read the property first of all, and then assign the specified value to its `Field1` field.

Assuming the property read operation is implemented by a direct data field read, the read operation will return a reference to the record data field, and the assignment will assign directly to the `Field1` field. However, there are at least two scenarios where things will go bad and give unexpected results.

Firstly, if the property write operation is defined in terms of a procedure call, which validates the record before actually making the assignment to the underlying data field, this code will be skipped. This is because the compiler will be assigning directly to the underlying data field without going through the property writer routine.

Secondly, if the property read operation is defined in terms of a function call, no value will be assigned at all. When the compiler reads the property, the function call will return a temporary record, which is a copy of the real underlying data field. The assignment will update a field of this temporary record, leaving the original one completely untouched.

The fact that the `with` clause record property field update is accepted by the compiler is considered to be a bug. You should get a compiler error because of the chances of things going wrong, just as you do when not using a `with` clause. Unfortunately, this seems to be a difficult case to detect, and so fixing the problem is on hold for the moment (the bug is present in Delphi 6).

```
type
  TTestRec = record
    Field1: Integer;
  end;
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    FTestRec: TTestRec;
  public
    property TestRec: TTestRec read FTestRec write FTestRec;
  end;
...
procedure TForm1.FormCreate(Sender: TObject);
const
  Rec: TTestRec = (Field1: 10);
begin
  TestRec := Rec; //compiles
  TestRec.Field1 := 99; //gives error: Left side cannot be assigned to
  with TestRec do Field1 := 99; //compiles
end;
```

➤ *Listing 2: Record property handling inconsistency.*

Delphi ToolBar Problem

Q Sometimes, when Delphi 5 crashes, the IDE loses all of the customisation I have done to its toolbars. Then I have to manually put every icon and function back into my preferred locations, which is a tedious task to say the least.

Is there some way of saving and restoring the IDE toolbar custom format? Or can you tell me where that information is stored?

A Yes, you can save this information quite readily as it is stored in the Windows registry in its own special area. You should set up toolbars as you like in Delphi, then perhaps close Delphi just to be sure it has stored the changes in the registry. Next you should launch the registry editor (by running RegEdit.exe from the Windows `Run...` dialog). Navigate your way down the relevant branches of the registry until you reach

```
HKEY_CURRENT_USER\Software\
  Borland\Delphi\5.0\Toolbars
```

which is where the information is stored. You can now export this section of the registry by choosing `Registry | Export registry file`, which will prompt you to save a .REG file containing all this information. Keep this file somewhere handy and, if Delphi crashes again, you can simply double-click it in Explorer to restore all the settings as you want them (make sure Delphi is not running when you do this).

Of course, the usual caveats regarding messing with the registry apply here with respect to doom and gloom for your PC if you make a hash of it, but we're all grown-up programmers, aren't we? I generally tend to omit such warnings from this column.

Figure 2 shows all the IDE main window toolbars, menu bar and component palette stored in this registry key, and also shows some

of the information stored for one of the toolbars. You should just be able to make out that it is a streamed version of the toolbar. The toolbars are simply streamed into the registry as they are modified and streamed out of the registry as Delphi starts up.

AVs When Destroying Components

Q I am programmatically creating some objects, including speedbuttons, with a `TScrollBox` as its parent. When the user clicks the speedbuttons the `OnClick` event is fired and inside of it I destroy all the objects created (including the speedbutton that fired the event) and I create new objects. This happens several times before the program is closed.

The process mostly works fine, but sometimes an Access Violation occurs. Normally the AV occurs on the `TSpeedButton.UpdateTracking` method. I have checked and I can't see anything wrong. I don't know if there is a problem destroying the buttons inside the `OnClick` event, so could you clarify the situation for me?

A Objects can be freed either by calling their `Free` method (which is definitely the way you should do it) or by calling their destructor, `Destroy`, which is generally not recommended. The reasons for this advice were covered in *The Delphi Clinic* way back in Issue 29 (January 1998).

In brief, `Free` checks that it is not being called through an obviously invalid object reference (with a nil value), before calling the destructor for you. `Free` is therefore a safer method to call than the destructor as it implicitly avoids some Access Violations. However, that doesn't answer the question.

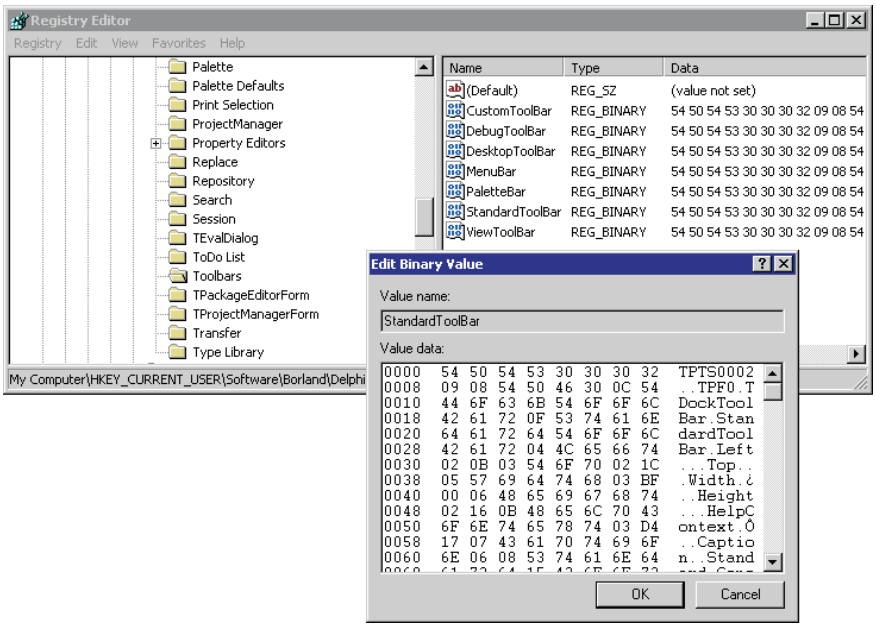
If you look up the `TObject.Free` method in the online help, you should see the following text, which explains the situation:

Warning: Never explicitly free a component within one of its own event handlers or free a component from the event handler of a component it owns or contains. For example, don't free a button in its `OnClick` *event handler or free the form that owns the button from the button's* `OnClick` *event.*

This text is in most versions of the Delphi help system (it got lost in Delphi 2 and 3, but was back in version 4) and strongly advises against what the questioner is doing. The problem is that a component's event handler is a routine called by some code in the component itself. When the event handler finished, the execution flow returns to the code in the component. If the component has been freed by the event handler, then all its instance data space will have been freed. Any attempt by the component to reference any of its instance data will therefore be invalid.

Depending on what has happened to the memory block that held the instance data, one of a number of things may happen. If the memory block is waiting to be re-used, this will probably give no evident symptom. If the memory block has already been re-used, it will trash the new data stored there. If the memory block has been completely freed back to the Operating System, then it will cause an Access Violation. Clearly, the questioner is finding no evident symptoms for a while, but ultimately is faced with an Access Violation.

➤ *Figure 2: Looking at one of the toolbars as it is stored in the registry.*



Word Automation Query

Q I have been reading many published articles (yours and many others) about Delphi and Word Automation. I was wondering if I may ask you a question on this particular topic. I have been able to automate just about all aspects of Word that I require, except the facility to insert a tab key. I am able to set a tab stop onto the ruler at any point, and I can do a full left indent that moves an entire paragraph, but I cannot automate a tab key to move text over to a set tab stop. How do I do it?

A In case anyone else is interested in reading them, I have a couple of Automation articles on my website. *Writing And Controlling Automation Servers In Delphi* is the first one, and shows how you control Automation servers (including Microsoft Word) and also shows how to write them. This is then followed by *More Automation In Delphi*, which looks at how to include more advanced features in your own Automation servers. You can access both of these (and a bunch of others) by clicking on the Articles link from my home page (www.blong.com).

Back to the question (or more correctly, back to the answer). A tab character is equivalent to

character 9 in the ANSI character set (in the same way that a capital A is character 65). Knowing this makes the answer quite straightforward. Listing 3 shows some code that inserts two tab characters in a Word document, one using a local constant, and one that simply embeds character 9 in a string. After doing this it ensures that all non-printable characters can be seen, so you can verify the tab characters have been passed across successfully (see Figure 3).

```
uses
  ComObj;
...
procedure TForm1.Button1Click(Sender: TObject);
var
  W: Variant;
const
  vbTab = #9;
begin
  W := CreateOleObject('Word.Application');
  W.Visible := True;
  W.Documents.Add;
  W.Selection.TypeText('Hello'#9'world');
  W.Selection.TypeParagraph;
  W.Selection.TypeText('Delphi' + vbTab + '5');
  W.Selection.TypeParagraph;
  W.ActiveWindow.ActivePane.View.ShowAll := True
end;
```

➤ *Listing 3: Inserting a tab character in a Word document.*

Reading Excel Spreadsheets

Q I am stuck with a problem related to Automation, using Excel as an Automation server. I would appreciate it if you could help me out. All I want to do is to open a file and walk through a sheet cell by cell reading the cells' content. I tried two approaches to get the work done, firstly

```
CreateOleObject(
  'Excel.Application')
```

and secondly the `TExcelApplication` component from the `Servers` page on the component palette.

In both cases, sooner or later I came up against an error message: *Member not found*. As far as I can see I am following the syntax prescribed in the Excel Visual Basic online help documentation. What's wrong?

A There are a number of differences between the syntax used in VBA (as found in the VBA for Excel help file, VBAXL8.HLP in the case of Excel 97) and that used in Delphi. For example, strings are delimited by single quotes in Delphi and double quotes in VBA. In VBA, statements are implicitly relative to the `Application` object, whereas in Delphi the `Variant` that gets set by a call to `CreateOleObject` is the equivalent of the VBA `Application` object. In VB, properties are typically indexed by values being passed in parentheses (round brackets), whereas Delphi will expect square brackets. VBA doesn't require statement separators, but Delphi uses semicolons. The list goes on...

To look into this question, I made a simple spreadsheet containing the release dates of all Borland's RAD compiler products (Versions.xls). Then I wrote an application (ReadXLSheet.dpr) to load it and read all of the cell values. This version of the project uses `CreateOleObject` to connect to Excel and the return value is stored in a `Variant`, for late bound Automation.

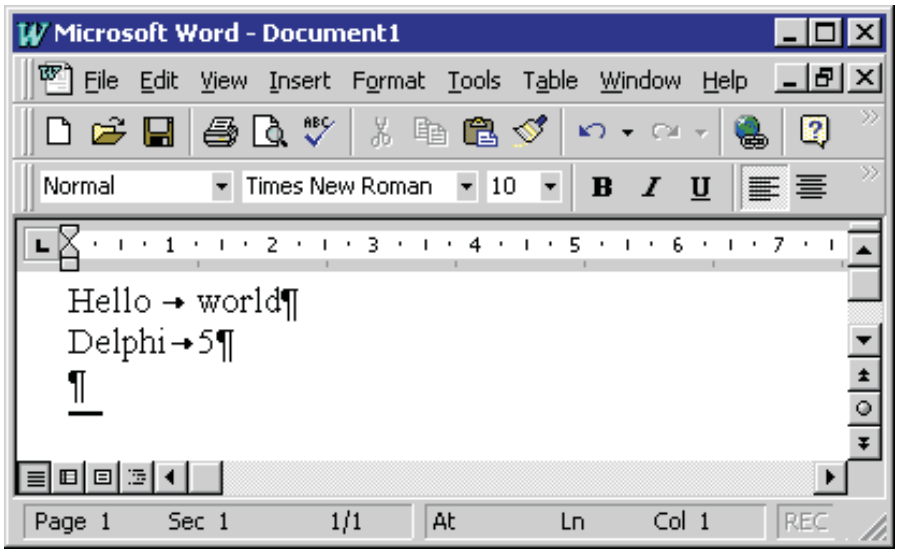
To open a spreadsheet file, you call the `Open` method of the `Application` object's `Workbooks` collection property. To read an individual cell value, the `Application` object has a property called `ActiveSheet`, which returns the active worksheet in the current spreadsheet file. `ActiveSheet` offers a `Cells` property, which is a `Range` object representing all the cells in the worksheet.

A `Range` object has a default indexed property that allows you to access any individual cell in the range by treating it as a two-dimensional array. Doing so returns another range for that cell, leaving you to read that range's `Value` property.

Listing 4 shows the code from the program looping through the cells reading the values into a string grid. The string grid has its size set based on the number of cells in the worksheet. You can see the `SpecialCells` method being used to activate the one-cell range, which is the last cell on the worksheet. The running program is shown in Figure 4.

This approach of reading individual cell values works fine for small worksheets, but will cause problems for larger ones. Each cell read requires the execution path to make two round trips to the Automation server and, obviously, as the number of cells increases,

➤ *Figure 3: Tab characters sent to Word through Automation.*



the time taken to do this will increase.

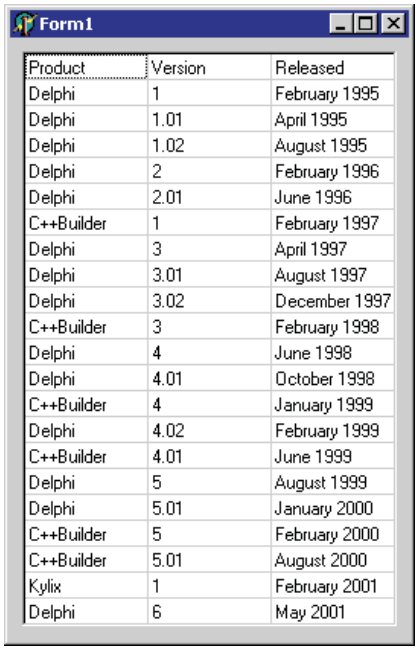
A better solution might be to read the whole cell range as a single `Variant` array value, and then loop through elements of the array in the client program. This means that no matter how many cells are being read, only one call is made (which equates to two round trips). ReadXLSheet2.dpr is a modified project that takes this more efficient approach, and the changes can be seen in Listing 5.

You can also achieve the goal using the `TExcelApplication` component from the component palette `Servers` page (see Listing 6). Notice, though, that the call to open the workbook requires a mass of parameters which are no longer optional (as they were when using a `Variant`). Fortunately, you can pass `EmptyParam` as a placeholder for all of them except the last one, which is a locale identifier (0 means use the default locale).

As well as a `TExcelApplication` component, I also dropped a `TExcelWorkSheet` component on the form with its `ConnectKind` property set to `ckAttachToInterface`. When the `TExcelApplication` component obtains a reference to a `WorkSheet` object, it can be represented by this new component thanks to a call to the `ConnectTo` method. Notice that the `ActiveSheet` property is actually defined as an `IDispatch` interface, rather than a `WorkSheet` interface (or `_WorkSheet` as it is defined), hence the use of the `as` operator to query for the correct interface.

One additional change from before is that the `Range` interface returned by the `Cells` property

➤ *Figure 4: The spreadsheet content read into a Delphi string grid.*



Product	Version	Released
Delphi	1	February 1995
Delphi	1.01	April 1995
Delphi	1.02	August 1995
Delphi	2	February 1996
Delphi	2.01	June 1996
C++Builder	1	February 1997
Delphi	3	April 1997
Delphi	3.01	August 1997
Delphi	3.02	December 1997
C++Builder	3	February 1998
Delphi	4	June 1998
Delphi	4.01	October 1998
C++Builder	4	January 1999
Delphi	4.02	February 1999
C++Builder	4.01	June 1999
Delphi	5	August 1999
Delphi	5.01	January 2000
C++Builder	5	February 2000
C++Builder	5.01	August 2000
Kylix	1	February 2001
Delphi	6	May 2001

➤ *Listing 4: Using a Variant to read a spreadsheet's content.*

```
uses
  ComObj;
procedure TForm1.FormCreate(Sender: TObject);
var
  XL, XLSheet: Variant;
  X, Y: Integer;
const
  xlCellTypeLastCell = $B;
begin
  XL := CreateOleObject('Excel.Application');
  XL.Workbooks.Open('c:\Versions.xls');
  XLSheet := XL.ActiveSheet;
  XLSheet.Cells.SpecialCells(xlCellTypeLastCell).Activate;
  SG.ColCount := XL.ActiveCell.Column;
  SG.RowCount := XL.ActiveCell.Row;
  //String grid cells are numbered from 0
  //XL cells are numbered from 1
  //Also, XL cells are indexed as (row, column), not (column, row)
  for X := 0 to SG.ColCount - 1 do
    for Y := 0 to SG.RowCount - 1 do
      SG.Cells[X, Y] := XLSheet.Cells[Y + 1, X + 1].Value;
  XL.Quit
end;
```

➤ *Listing 5: A more efficient approach to reading multiple Excel cells.*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  XL, Range: Variant;
  X, Y: Integer;
const
  xlCellTypeLastCell = $B;
begin
  XL := CreateOleObject('Excel.Application');
  XL.Workbooks.Open('c:\Versions.xls');
  //Select last cell
  XL.ActiveSheet.Cells.SpecialCells(xlCellTypeLastCell).Activate;
  //Set string grid size
  SG.ColCount := XL.ActiveCell.Column;
  SG.RowCount := XL.ActiveCell.Row;
  //Read all cells from top left to last cell as a Variant array
  Range := XL.Range['A1', XL.Cells[SG.RowCount, SG.ColCount]].Value;
  XL.Quit;
  //String grid cells are numbered from 0, XL cells are numbered from 1
  //Also, XL cells are indexed as (row, column), not (column, row)
  for X := 0 to SG.ColCount - 1 do
    for Y := 0 to SG.RowCount - 1 do
      SG.Cells[X, Y] := Range[Y + 1, X + 1];
end;
```

➤ *Listing 6: Reading a spreadsheet using early bound Automation.*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  X, Y: Integer;
  Range: Variant;
const
  DefLocale = 0; //Default locale value
begin
  XL.Workbooks.Open('c:\Versions.xls', EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, DefLocale);
  XLSheet.ConnectTo(XL.ActiveSheet as _WorkSheet);
  //Select last cell
  XLSheet.Cells.SpecialCells(xlCellTypeLastCell,
    EmptyParam).Activate;
  //Set string grid size
  SG.ColCount := XL.ActiveCell.Column;
  SG.RowCount := XL.ActiveCell.Row;
  //Read all cells from top left to last cell
  //as a Variant array
  Range := XL.Range['A1', XL.Cells.Item[SG.RowCount,
    SG.ColCount]].Value;
  XL.Quit;
  XLSheet.Disconnect;
  XL.Disconnect;
  //String grid cells are numbered from 0
  //XL cells are numbered from 1
  //Also, XL cells are indexed as (row, column), not
  //(column, row)
  for X := 0 to SG.ColCount - 1 do
    for Y := 0 to SG.RowCount - 1 do
      SG.Cells[X, Y] := Range[Y + 1, X + 1];
end;
```

does not have a default array property defined in the type library (as far as Delphi sees) so we must explicitly refer to the `Item` array property this time. Apart from these changes, the code stays much the same as it was before. This COM version of the program is called ReadXLSheet3.dpr.

For more information on reading and writing values to and from Excel spreadsheets using the COM interface wrapper component (`TExcelApplication`), there is an article in UNDU (the Unofficial Newsletter of Delphi Users) that looks at the subject. It is written by Christian Ebenegger and Thierry Revillard and can be found at

```
www.undu.com/Articles/
  010316c.html
```

This is where I picked up the tip about how to identify the last cell in the worksheet, and also the efficiency tip about returning a range of cell values as a `Variant`.

Incidentally, if you have not checked out the offerings of UNDU before, I recommend you visit www.undu.com soon. You will find many invaluable articles, tips and tricks.

Updates

This month there are a couple of updates to previous entries in *The Delphi Clinic*. The first one relates to something all the way back in Issue 33 (May 1998) where I covered how to update Windows system files that could well be in use, using the `MoveFileEx` API with the NT-specific `MoveFile_Delay_Until_Reboot` flag.

At the time, I didn't have access to a Windows NT machine and so wrote the text based on what the Microsoft documentation seemed to tell me. The example revolved around updating the Common Controls library, ComCtl32.dll, and you can see the suggested code in Listing 7.

This is supposed to replace ComCtl32.dll with the new file ComCtl32.new when NT restarts (in other words when the file is not in use). Thanks are due to Michel Lucas who tried this code out and found it didn't work. After a little research, Michel pointed out that the Microsoft documentation recommends deleting the old file before overwriting it. That changes Listing 7 to Listing 8, which works fine.

```
MoveFileEx('C:\Windows\System32\ComCtl32.New',
  'C:\Windows\System32\ComCtl32.Dll',
```

➤ *Listing 7: Erroneous code for updating a system file on Windows NT/2000.*

```
MoveFileEx('C:\Windows\System32\ComCtl32.Dll', nil, MoveFile_Delay_Until_Reboot);
MoveFileEx('C:\Windows\System32\ComCtl32.New',
  'C:\Windows\System32\ComCtl32.Dll', MoveFile_Delay_Until_Reboot);
```

➤ *Listing 8: Correct code for updating a system file.*

```
procedure TForm1.SendEmail;
var
 NMSMTP: TNMSMTP;
begin
  NMSMTP := TNMSMTP.Create(Application);
  try
    with NMSMTP do begin
      Host := 'TheMailServer';
      UserId := 'SMTPuserID';
      try
        Connect;
        ClearParams := True;
        SubType := mtPlain;
        EncodeType := uuMime;
        PostMessage.LocalProgram := Application.Title;
        PostMessage.FromAddress := 'MyEmailAddress';
        PostMessage.ToAddress.Add('DestinationEmailAddress');
        PostMessage.Body.Add('Some text');
        PostMessage.Body.Add('Some more text');
        PostMessage.Subject := 'Email subject';
        SendMail;
      except
        on E: Exception do
          ShowMessageFmt('Unable to send email to caller'#13#13'"%s"',
            [E.Message]);
      end;
    end;
  finally
    FreeAndNil(NMSMTP)
  end
end;
```

➤ *Listing 9: Sending an email with the FastNet TNMSMTP component.*

Another email came in from Neil Haughton who commented on previous coverage of ways of sending emails from Delphi programs (covered in *The Delphi Clinic* in Issue 60 and Issue 69). I have looked at using MAPI, automating Outlook and running a URL that starts off an email message. Neil points out that the `TNMSMTP` component on the Delphi `FastNet` component palette tab is very straightforward to use.

This component will log into an SMTP server, create a message and send it for you, and some simple code that does this is shown in Listing 9. Thanks Neil.