Subclassing

by Brandon Smith

Subclassing is a Windows API programming term that has nothing to do with the fact that almost every Delphi program consists of a one or more subclasses derived from the `TObject` class. Creating subclasses with new or altered characteristics is the essential business of Delphi programmers. So I was a bit confused the first time I came across a newsgroup message stating that "...subclassing is what you need to do..." This article will explain what subclassing is, when to do it, how to do it, and along the way provide some details from a couple of situations where subclassing is the required technique.

Subclassing in Windows API jargon is a very specific technique which to me, at least, ought to be called something like event handling takeover. It is definitely not another term for creating descendants of a class. Instead it is more like a way to bypass normal inheritance, a way for a child to alter how its parent behaves. Subclassing is not a way of setting up things at the declaration level with classes, but a technique for altering behavior of already instantiated objects.

In a Windows program, every instantiated class has a handle. If we think of each of these handles as an intersection in a busy city, the Window Procedure (`WNDPROC`) is the traffic cop standing in the middle of that intersection, directing traffic. Subclassing is when we jump in and substitute ourselves for that traffic cop. By contrast, within a Delphi program, a subclass, an object descended from `TWinControl`, builds a new intersection as well as putting in a new traffic cop. In other words, the Delphi VCL is to native Windows as a freeway cloverleaf is to Piccadilly Circus.

Analogy aside, Windows subclassing basically consists of substituting your own handler for the default handler, the `WNDPROC`, of an existing Windows class. In the Windows system, an essential ingredient of the instantiation of each window class is the `WNDPROC`, a function which serves as the central traffic cop for messages coming to that window. For a native Windows class, such as an edit box, Delphi simply points to the code inside the appropriate Windows DLL which implements the edit box's `WNDPROC`. This is accomplished inside the `CreateParams` method, the `WNDPROC` being specified inside the `WNDCLASS` structure.

The Delphi VCL puts Object Pascal wrappers around the vast majority of Windows classes, and surfaces event handler hooks for just about every message that can come to or from that kind of Window class. On top of that, you can override any Windows Message handler within a Delphi class that handles messages. Plus, inside the Delphi VCL there are component notification messages which can be handled within Delphi. So why would a Delphi programmer have to resort to subclassing?

When Is Subclassing Needed?

There are two kinds of situation where subclassing techniques become necessary. The first is when you want to do something which is not encapsulated by the Delphi VCL and cannot be directly controlled from a message handler. The second situation is when you want to control another application directly. In this article I'm going to describe in detail a few instances of the first situation, and only touch briefly on the second.

Subclassing an object in another application is the only way you can get at another application's data or functions when there are no OLE, DDE or other inter-process communication links available. It is fairly easy to do under Windows 3.x, since all the processes live in the same place. It is not so easy under Windows 95 or NT since each process has its own space and its own thread, but it can be done by someone who understands how to control threads. using `MUTEX` and related objects. I won't go into any more detail on this kind of subclassing. However, much of what is discussed in the rest of this article does apply when you are subclassing into another application.

When does the Delphi VCL not go far enough? One example is shown in the *Delphi 2 Developer's Guide* (by Xavier Pacheco and Steve Teixeira), Chapter 10: drawing a bitmap on the client surface of an MDI application. It is easy enough to draw a bitmap on the canvas of a `TForm`, but what makes subclassing necessary here is the fact that there is no convenient Delphi way to tell that canvas to refresh itself when a child window is moved or closed. Subclassing permits us to redefine how the client surface is created and in that redefinition, include message handling to keep the client surface updated.

The first of the two situations I found myself dealing with was a non visual component which needed to find the size and location of its owning form at runtime. The second situation involves a visual component, a descendent of `TCustomMemo` which turns scroll bars on and off according to whether they are needed. It turned out I needed to capture the `EN_CHANGE` notification message which is handled by its parent, a parent which is unknown at compile time, a parent which may be the form or any `TWinControl` on the form. Before dealing with these situations, however, let's go over the traditional method used to implement subclassing.

Traditional Subclassing Technique

You will find most Windows programming books describe

subclassing this way, albeit in C or C++. In Delphi terms, we need a far function to be our new WNDPROC, three variables, and code to turn on and turn off the subclassing activity.

App1main.pas on the disk illustrates the traditional approach. It uses an object independent function for the new handler, declared the way a WNDPROC has to be:

```
function NewFormFunction(
  handle: hWnd; msg, wParam :
  Word; lParam : LongInt):
  LongInt; stdcall;
```

Although Microsoft calls this a Window Procedure, and Delphi follows the convention by declaring several WindowProc related properties and methods, I like to include the word Function when I subclass so that I am reminded this is a function and Windows does expect a return code.

For our variables, we need one to point to the original handler, another one to point to our new handler and something to hold the handle of the window we are going to subclass. There are some situations where we won't need a variable for the handle, but we will always need the first two.

```
var
  NewWindowProc : TFarProc;
  OldWindowProcAddr : LongInt;
  TargetHandle : hWnd;
```

To turn on the subclassing, all we need to do is assign the correct values to the variables and fire the API call that changes where Windows will look for the handler to the class we are taking over. We do that in the OnCreate event for our form by calling TurnSubClassOn (Listing 1).

Whatever we need to do the subclassing for is accomplished in our new function, with a couple of important rules to keep in mind. The first rule is to remember that this new WNDPROC is going to grab *all* messages going to the instantiation of the Windows class whose handle we've taken over. Therefore, as with VCL objects, we need to be sure to pass control back to the original object to handle any messages we don't care about. The second rule is to remember that Windows thinks this is the handler for this class and may be expecting certain result codes, so we have to make sure an appropriate result is defined within our new Windows Procedure. The best way to obey both rules is to call the original handler at the end.

```
Procedure tmyForm.TurnSubClassOn;
begin
  TargetHandle := handle;
  OldWindowProcAddr := GetWindowLong(TargetHandle, GWL_WNDPROC);
  NewWindowProc := @NewFormFunction;
  SetWindowLong(TargetHandle, GWL_WNDPROC, LongInt(NewWindowProc));
end;
```

➤ *Listing 1*

```
function NewFormFunction(handle: hWnd; msg, wParam : Word; lParam : LongInt):
  LongInt;
begin
  case msg of
    WM_LBUTTONDOWN : MyForm.color := clLime;
    WM_RBUTTONDOWN : MyForm.color := clWhite;
  end;
  Result :=
    CallWindowProc(TfarProc(OldWindowProcAddr), Handle, Msg, wParam, lParam);
end;
```

➤ *Listing 2*

```
function AlternateNewFormFunction(handle: hWnd; msg, wParam : Word;
  lParam : LongInt): LongInt;
begin
  Result :=
    CallWindowProc(TfarProc(OldWindowProcAddr), Handle, Msg, wParam, lParam);
  case msg of
    WM_LBUTTONDOWN : MyForm.color := clLime;
    WM_RBUTTONDOWN : MyForm.color := clWhite;
  end;
end;
```

➤ *Listing 3*

What this subclassing function does is take over the mouse button handling. It turns the form's color green when the left button is pressed and white when the right button is pressed. I've provided a button and event handler in Listing 2 to illustrate how we can accomplish the same thing using a standard VCL event handler. By using yellow and red as the colors set when the VCL event handler is in charge, you can see that our subclassing activity is overruled by the VCL since the VCL activity is being called from the original WNDPROC which is called after our subclassing activity. The color changes made by our subclassing do occur, as you can verify with the debugger, but you won't see them on the screen. App1main has several buttons so you can see how the various combinations of subclassing and event handling work.

Hint: if you want to trace through this behavior in the integrated debugger, put your breakpoint inside the case statement: if you put a break point on the case line or the CallWindowProc line you will discover that there are a *lot* of messages being handled by the form's Window Procedure. So many that you'll never get to see the program running. This is one of the main reasons you are not running Windows on an 8088. It is also the reason why you need to keep processing to a minimum inside a subclass function.

What Happens If We Call The Original Handler First?

The AlternativeNewFormFunction and its activator, Button2, illustrate what happens when you change the guts of the subclassing function (Listing 3).

In this case it effectively eliminates the VCL event handler, the ability to change the form yellow and red. This is because the call to

```
case msg of
  WM_DESTROY : UnkownForm.MyComponent.SaveFormInfo;
  WM_ACTIVATE : UnkownForm.MyComponent.RestoreFormInfo;
  WM_CHILDACTIVATE : UnkownForm.MyComponent.RestoreFormInfo;
end;
```

➤ *Listing 4*

```
function NewFormFunction(handle: hWnd; msg, wParam : Word; lParam : LongInt):
  LongInt;
var ThisInstance : longint;
begin
  ThisInstance := getProp(handle, cMyID);
  if TAwkwardComp(ThisInstance).enabled then
    case msg of
      WM_LBUTTONDOWN : TAwkwardComp(ThisInstance).color := clLime;
      WM_RBUTTONDOWN : TAwkwardComp(ThisInstance).color := clWhite;
    end;
  Result := CallWindowProc(TFarProc(TAwkwardComp(ThisInstance).fOldWindowProc),
    Handle, Msg, wParam, lParam);
end;
```

➤ *Listing 5*

the original WindowProc fired the MouseDown event before arriving at our subclass case statement which takes over and uses green and white, again so quickly you can't even see the flicker.

Generally speaking, one can subclass as many times as one wants as long as an unbroken chain eventually gets the control back to the original object for the messages we are not intercepting. What I have found, however, is that there are situations where you need to turn off an existing subclass before turning on a different one. In this case, if you leave out the first line in TMyForm.Button2Click, you'll end up with a lot of disk activity followed by a stack overflow. If you've never managed to lock up your computer from within the Delphi IDE, I can safely predict that working with subclassing will provide you that experience.

Another hint: I discovered that when you get a fatal error while running a program within the IDE and you have the Close/Details box sitting there awaiting your attention, the best thing to do is to go up to the Delphi Run menu and click on Reset Program first, then click on the Close button. Clicking on the Close button seems to almost always lead to a reboot.

The Problem With Traditional Subclassing

This traditional, non-OOP, approach to subclassing has a major drawback, however. Suppose we need to subclass the parent of a component whose name we do not know at compile time? Specifically, I needed my WNDPROC subclass to do something similar to that shown in Listing 4.

This is the situation I faced when I was developing a non visual component to save the size and position of a form, as well as the location of user-moveable controls on that form such as splitter bars. In my save-the-window component, I needed to intercept the WM_DESTROY or WM_ACTIVATE messages sent to the form and save or retrieve that form's size and position data as well as iterate through the owned controls and save their size and position. Turning on the subclass from within my component was easy since my component has an Owner property, which is the form it was dropped onto. But the subclass function itself was not part of my component's class definition. So, when control passes to the WNDPROC, all I have at that point is a handle to the form, I no longer have anything which tells me which instance of my component this WNDPROC belongs to.

Delphi doesn't provide a function you can feed a handle to and get back a TForm, though I did consider trying to build one by starting at application level and iterating through all the forms until I found the one whose handle matched the handle in my subclassing WNDPROC, but this puts quite a bit of processing work into the WNDPROC, a function that gets called frequently. On top of that, once I knew which form I had, I'd still have to go through its components and look for mine.

A Windows API Solution

I came up with a workaround to this problem by using the Windows API SetProp and GetProp calls. For clarity, I have put together AwkMain.pas and AwkWorks.pas on the disk to illustrate this technique using a TCustomPanel as the component who needs to find out which form it belongs to.

SetProp lets one add some information to a particular instantiation of a Windows class at runtime. The information consists of a string or a pointer to a global atom and a longint. GetProp can then be used to retrieve that information. By executing the following code within the method my component used to turn on subclassing, I was able to add some useful information to the windows class instantiation of the form my component was dropped on.

```
SetProp(fTargetHandle, cMyID,
  longint(self));
```

Once the form has been branded with my component's information, GetProp will give me a pointer to myself, to the instance of my component that is associated with the handle receiving message traffic to be processed (Listing 5).

The Delphi Solution

I think you can see from the complex typecasting why I've called this TAwkward. At the time I thought it rather elegant. Since then, however, I was able to upgrade my paper library from *Delphi Developer's Guide* to *Delphi 2 Developer's Guide* where I discovered that Borland has provided a simple and elegant solution to this problem: MakeObjectInstance.

I don't know whether this function is in the help files anywhere, I certainly haven't been able to find it. It is declared and implemented in the Forms unit. This function sets up code so that a specific kind of class method can be used as a

WNDPROC. The files Example.pas, Example2.pas and app2main.pas on the disk contain the code for this example.

I've followed Pacheco and Teixeira's (*ibid*, page 905) variable typing schema in this example, though I could have used the same types I used before with different typecasting. Whenever you deal with the Windows API, you will be typecasting at some point since Microsoft is in love with C++ and C++ programmers love to overload variables. For an excellent, detailed and reasonably non-biased discussion of the difference between Object Pascal and C++, I would recommend Bruce Eckel's introduction to Gary Entsminger's *The Way of Delphi*.

Using MakeObjectInstance to do subclassing still requires calling the original handler and being aware that this new WNDPROC will be called many times during program execution. The basic difference is that we are now working with a class method that knows who it is and has ready access to itself and its parent. Also, instead of having the window parameters listed as separate arguments, they are contained within a TMessage record (the declaration can be found in Controls.pas). The code snippets shown in Listings 6 to 8 are from App2main.pas and the component file example.pas on the disk.

The variables which support this OOP approach to subclassing can now be put in the private part of our component's class definition, keeping them out of public view and reducing the chance of name space confusion.

Turning on this kind of subclassing uses essentially the same steps as the non-OOP approach. I followed Pacheco and Teixeira's shortcut here, using the fact that a successful

```
SetWindowLong(
  x, GWL_WNDPROC, y);
```

call returns the longint which points to the previous WindowProc.

Why not simply turn on the subclassing in the constructor? Pacheco and Teixeira did in their example, or more precisely, they subclassed the Application.Handle, an entity which exists before (almost) anything else is created. Unfortunately, the parent of a component cannot be determined within the Create constructor.

However, if you are subclassing the form on which a component has been dropped, you do have typecasted access to the Owner.Handle by using TwinControl(aOwner) or Tform (aOwner). In other words, if the entity whose event handling you need to take over is the form or the application, you can set up the subclassing in the component's Create constructor. I've used this technique in the file Awkworks.pas (which is also on the disk), turning on the subclassing during the create constructor as well as doing some error checking to make sure the subclassing worked.

```
procedure Texample.NewFormFunction(var msg : TMessage);
begin
  case msg.msg of
    WM_LBUTTONDOWN : color := clLime;
    WM_RBUTTONDOWN : color := clWhite;
  end;
  msg.Result := CallWindowProc(fOldWindowProc, fTargetHandle,
    msg.Msg, msg.wParam, msg.LParam);
end;
```

➤ *Listing 6*

```
type
  Texample = class(TCustomPanel)
  private
    fNewWindowProc : Pointer;  {pointer to our window function}
    fOldWindowProc : Pointer;  {previous window function }
    fTargetHandle : hWnd;
  Procedure NewFormFunction(var msg : tmessage);
...
```

➤ *Listing 7*

```
procedure texample.turnOnSubClassing;
begin
  fTargetHandle := parent.handle;
  fNewWindowProc := MakeObjectInstance(NewFormFunction);
  fOldWindowProc := Pointer(SetWindowLong(fTargetHandle, GWL_WNDPROC,
    LongInt(fNewWindowProc)));
end;
```

➤ *Listing 8*

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  with texample.create(Panel1) do begin
    parent := Panel1;
    top    := panel1.height div 2 - height div 2;
    Left   := panel1.width div 2 - width div 2;
    turnOnSubclassing;
  end;
end;
```

➤ *Listing 9*

The Parenting Problem And Solution

For visual components, however, you may well want to subclass something other than the form. You may need to subclass the parent of your component. And the parent, unlike the owner, can be any TWinControl descendent. In that case you will not be able to turn on the subclassing within the Create constructor because the parent is not known during construction. The event shown in Listing 9 from app2main.pas puts our example component on a panel and subclasses the panel's WNDPROC rather than the form's WNDPROC. The functional result is that clicking on the form will not trigger our new subclass actions, but clicking on the panel will.

It would hardly make you popular as a component maker if your users have to remember to use TurnOnSubclassing in order for

```
Texample2 = class(TCustomPanel)
  private
    ...
    fSubClassDone : boolean;
    procedure WMPAINT(var Message: TMessage); message WM_PAINT;
...
procedure Texample2.WMPAINT(var Message: TMessage);
begin
  inherited;
  if not fSubclassDone then TurnOnSubclassing;
end;
```

➤ *Listing 10*

```
procedure tAutoMemo.SubClassParentFunc(var Msg : tmessage);
begin
  with msg do begin
    Result := CallWindowProc(OldWinProc, Parent.handle, Msg, wParam, lParam);
    if (msg = WM_COMMAND) and (lparam = handle) then
      if WParamHi = en_change then CheckSCrolling;
    if msg = WM_DESTROY then
      SetWindowLong(parent.handle, GWL_WNDPROC, longint(OldWinProc));
    end;
end;
```

➤ *Listing 11*

your component to work properly. Since we can't use our component's `Parent` property during its constructor, we have to turn on the subclassing at a later point. In the world of object-oriented event handling, as soon as Windows learns that a handle exists, as soon as the create constructor is complete, messages start buzzing around like bees in field of flowers. All we have to do is grab one and use it to do our work.

I suspect that it might be more efficient to use one of Delphi's component notification messages, such as `CM_INVALIDATE`, since these do not involve any Windows overhead. In any case, however, notice that I've introduced a flag, `fSubClassDone`, so that I can ensure the subclassing only gets done once. See `App2main` and example2.pas on the disk. `App2Main` has a number of buttons to activate these components and also illustrates multiple subclassing: we end up with two custom panels parented by `Panel1`, and both implement subclassing of `Panel1`'s `WNDPROC`.

I'm not sure if it is a good idea to use `WM_PAINT`, since this message seems to be very frequently used, and this does add a few machine cycles to every occurrence. On the other hand, this frequency of occurrence pretty much guarantees the subclassing will get turned on in a timely manner.

For our final foray into subclassing, consider the ubiquitous `TMemo`. How many times have you dropped a memo on a form and after seeing some live data in it, gone back and changed the `ScrollBar` property? I was frankly disappointed that Delphi's `TMemo` didn't automatically turn scroll bars on and off according to what's in the memo, following the reasonably intelligent behavior of `TListBox`.

I thought that simply using the `OnChange` event handler would take care of this: every time there is a change, run a scroll checking routine and modify the scroll bars. Unfortunately, `OnChange` doesn't get fired if you use a Windows message such as `WM_SETTEXT` to change the contents of the memo. App3Main.pas on the disk has an event handler that illustrates this failure. I considered the `OnEnter` and `OnClick` events, but I really wanted the memo to come up with the proper scroll bars whether the user clicked or entered or whatever.

Fumbling around in the Windows API Help files provided with Delphi, I eventually came across `EN_CHANGE`, a notification message that is sent to the parent of an edit control in a `WM_COMMAND` message after the contents of the edit control changes. And that's all it said, nothing about how to use it. When I found myself in a large city recently, I spent a number of hours in technical bookstores, trying to find a Windows book that had something more. About half of them listed `EN_CHANGE`, but none of them provided any more information.

Finally I came across one which had one additional sentence suggesting that one could subclass the parent of the edit control and use the `EN_CHANGE` notification message as a flag to perform additional actions when the contents of the edit control have changed.

When you work with a notification message, you have a `WM_COMMAND` message in which the notification code is found in `wParamHi` and the handle of the child control is found in the `LParam`.

AutoMemo.pas and its test bed, App3main.pas, on the disk present a condensed version of how I use the `EN_CHANGE` notification message to automate a memo's scroll bars. As with the previous example, I couldn't set up the subclassing in the constructor since a `Memo` can be dropped on a parent other than the form.

The subclassing `WNDPROC` used to capture and deal with `EN_CHANGE` required an additional safety measure (Listing 11).

Since the `EN_CHANGE` notification is sent after Windows makes the change, I wanted to take care of the original message handling first. However, this caused a side effect

Components which have been incorporated into the VCL (by using `CompLib` in Delphi 2 or packages in Delphi 3) automatically create themselves and establish parenthood when they are dropped on a form. When developing a component, one normally works with it initially outside the VCL by writing in an instance of it in an existing form's declaration and then creating it in response to button click. When working this way, one has to explicitly set the parent otherwise it will not be seen when you run the testing project. If you don't specify top and left, they will be 0,0.

whose explanation I won't even guess at. The literature is quite clear that one must reinstall the original WNDPROC when you are done subclassing. I had originally set up the TAutoMemo component to turn off the subclassing in the destructor, before the inherited destroy, similar to the way Pacheco and Teixeira did (*ibid*, p906) in the FormDestroy method. Apparently because we are here working with a component independent of the form, this did not work, in fact resulted in a GPF. I suspect the reasons have to do with the sequence of messages generated when a parent receives a WM_DESTROY message, but in any case I solved the problem by turning off the subclassing within the subclassing function when a WM_DESTROY message shows up.

Turning on the subclassing for the auto scrolling is also a bit more complex than previous examples (Listing 12).

```
Procedure tAutoMemo.TurnOnSubclassing;
begin
  if HandleAllocated and (not fsubclassDone) then begin
    if NewWinProc <> nil then
      FreeObjectInstance(newWinProc);
    NewWinProc := MakeObjectInstance(SubClassParentFunc);
    OldWinProc := Pointer(setWindowLong(parent.handle, GWL_WNDPROC,
      longint(NewWinProc)));
    if OldWinProc = nil then
      raise eAutoMemoError.create('subclass failed');
    fSubClassDone := true;
  end;
end;
```

➤ *Listing 12*

```
Procedure tAutoMemo.CheckScrolling;
var
  tmpScroll : tScrollStyle;
function Translate(customScrollEnum : tSynMemoScrollEnum ): tscrollstyle;
begin
  case CustomScrollEnum of
    smAutoScroll : result := ssNone;
    smVertical   : result := ssVertical;
    smHoriz      : result := ssHorizontal;
    smBoth       : result := ssBoth;
    smNoScroll   : result := ssNone;
  end;
end;
begin
  tmpScroll := translate(fScrolling);
  if fScrolling = smAutoScroll then begin
    if not fSubClassDone then
      TurnOnSubClassing;
    if lines.count*lineheight > clientheight then tmpScroll := ssVertical
    else tmpScroll := ssNone;
    // no point in horizontal scrolling unless wordwrap = false
    if WordWrap = false then
      if (longestline > clientwidth) and (getLinesShowing > 1) then begin
        if TmpScroll = ssVertical then TmpScroll := ssBoth
        else TmpScroll := ssHorizontal;
      end else begin
        if TmpScroll = ssBoth then
          TmpScroll := ssVertical
        else
          if lines.count*lineheight > clientheight then TmpScroll := ssVertical
          else TmpScroll := ssNone;
      end;
    end else TurnOffSubClassing;
  case fSCrolling of
    smAutoSCroll : if scrollbars <> TmpScroll then scrollbars := tmpScroll;
    smVertical : if scrollbars <> ssVertical then scrollbars := ssVertical;
    smHoriz : if scrollbars <> ssHorizontal then scrollbars := ssHorizontal;
    smBoth  : if scrollbars <> ssBoth then scrollbars := ssboth;
    smNoScroll : if scrollbars <> ssNone then scrollbars := ssnone;
  end;
end;
```

➤ *Listing 13*

Most of what's been added is insurance. The new variable, fSubClassDone, is set to False in the component's constructor and again when subclassing is turned off. By setting it True after a successful subclass, we ensure we don't add any extra layers of processing. Checking to make sure we have our own handle allocated will not be necessary if we know for sure that the handle exists when we call on this procedure. But if the component user has access to the TurnOnSubClassing procedure, we can't rely on that. If, as I ended up doing in this component, it is possible for the subclassing to be turned on and off multiple times during the component's life, one needs to make sure any old method object instances are taken care of. Finally, if Windows decides it won't do the subclassing for us, it's a good idea to raise an exception rather than have the user think it's working when it's not.

I decided this time to use WM_ERASEBKGND as the trigger to turn on subclassing the first time, figuring this message won't be as frequent as WM_PAINT. However, to ensure the component user doesn't get ambushed, I also included a call to the TurnOnSubClassing method inside the CheckScrolling method when the scrolling property is set to autoscroll (Listing 13).

By using a temporary variable and a separate case statement at the end where I make sure a change is only made if a change is called for, I eliminate extra calls to RecreateWnd which is what TCustomMemo uses when it changes the scroll bar setting. I spent a couple of hours in a panic when I realized this, thinking that my subclassing couldn't possibly work if the handle of the component was being changed every time the ScrollBar property changed. I had been using the debugger and was more than a bit perturbed when I overrode the RecreateWnd component notification and discovered my component was getting a new handle whenever the ScrollBar property changed as well as every time the Alignment or WordWrap properties changed. Actually it seemed to be changing a lot more frequently than that, but I didn't pursue it when I found that Borland has taken care of things: whenever the handle changes, the windows support structure is updated so the subclassing points to the right object.

However, when I took the TAutoMemo to the next logical step, data-awareness, I discovered that the EN_CHANGE notification does not

```
procedure TsynCustomMemo.CMRecreateWnd(var Message: TMessage);
var oldhandle : hwnd;
begin
  oldHandle := handle;
  inherited;
  if oldHandle <> handle then begin
    lines.add('old = '+intTostr(oldhandle));
    lines.add('new = '+IntToStr(handle));
  end;
end;
```

➤ *Listing 14*

get fired when we go from one record to the next. To implement data-awareness, I simply copied the code for TDBMemo out from DBCTRLS.PAS and changed its inheritance from TCustomMemo to TCustomAutoMemo (not shown here: simply a recasting of TAutoMemo where all the properties are protected rather than published). Rather than jump in and look for another Windows message to use to turn on the auto scrolling, I studied the TDBMemo code and finally decided to put a call to CheckScrolling at the end of the DataChange event handler and that took care of it.

The TAutoMemo example on the disk does not include several additional items I found necessary for a decent component, but not necessary to explain how subclassing works: for example, the CheckScrolling method, when it changes the scroll bars, will always recreate the memo with the top line at the top and the cursor at 0,0. A 'production' version of TAutoMemo, called TSynMemo and with more features, was included on the Issue 27 disk and is also available on *The Delphi Magazine* website (www.itecuk.com).

A final comment before summarizing. TControl has a property, WindowProc, which the Help says is to be used "to temporarily replace or subclass the window procedure of the control." There is also a TControl.WndProc method, which is the procedure itself, virtual, so it can be overridden. However, it would appear these animals are designed to support altering the behavior of your own control, not the behavior of another control which happens to be the parent of your control. If a reader can show me how these guys can simplify what I've described above, I'd be happy to use them.

In summary, the main points to keep in mind when considering the use of Windows subclassing are: Don't subclass if a VCL event handler or a message override or component notification override will do the trick. Subclassing bypasses the numerous object oriented safety features built into Delphi as well as introducing a whole raft of ways to crash your program.

Be sure you understand the difference between the parent and the owner, and that the parent is not in scope during the create constructor. Therefore, be aware that turning on subclassing of anything other than the main application WNDPROC or a TForm's WNDPROC will require some planning and experimentation.

Inside your new WNDPROC, you are replacing the traffic cop in the middle of a very busy intersection. Like that cop, your toes are in danger of being run over and if you wave your arms the wrong way, you will cause at least a traffic jam and at worst an accident with fatalities. In other words, Windows won't work right if your WNDPROC doesn't behave properly.

When you are done subclassing, don't leave the scene until the original traffic cop is back on the job. Sometimes this works OK, you can leave out the call to SetWindowLong which reinstalls the old WNDPROC when you destroy your component or the form it resides on. But that's luck, not the way the system normally works. Sometimes, you have to take special care to insure the old WNDPROC is reinstated before your component is destroyed, rather than as part of the destruction process.

Brandon Smith lives and works in Mansfield, MO, USA and can be emailed at Synature@aol.com