Things You Can Do With Standard Controls: The ListBox

by Brian Long

In the rush to develop new components here there and everywhere, sometimes it's easy to forget that the standard controls included with Delphi are really quite powerful. This month, I'll take a look at what we can do with the humble `TListBox`. Note that all of this is applicable to both Delphi 1 and 2.

Displaying Non-Textual Items

To display things other than strings requires us to use the owner-draw facilities of a listbox. Many Windows controls offer owner-draw facilities, but not all have them surfaced by their respective Delphi VCL components. The term owner-draw basically means that you as the developer take over the responsibility of supplying code to do the drawing of the control, but we will look at this in more detail in my next article on this topic.

Listboxes, combo boxes, buttons and menus support owner drawing inherently in Windows, but only `TListBox` and `TComboBox` components make it easy for us to take advantage of it. Other Delphi components which have no direct Windows control equivalent but offer support for developer supplied drawing code in a quite similar way include `TStringGrid`, `TDrawGrid`, `TDBGrid` and `TOutline`.

Information about owner-draw listboxes is fairly easy to obtain. The first three Delphi books I picked off my shelf cover the subject, although the next two didn't. The ones that do are *Instant Delphi Programing* by Dave Jewell, *Delphi Programming Problem Solver* by Neil Rubenking and *Delphi Developer's Guide* by Steve Teixeira and Xavier Pacheco. I won't mention the ones that didn't as they might actually discuss the subject, but not list it obviously in the index.

To lay down the law, the rules for custom listbox drawing involve setting the `Style` property to either `lbOwnerDrawFixed` (for listbox items that will be all the same height, as in a normal listbox) or `lbOwnerDrawVariable` (for variable height items). Two other properties that come into play when `Style` is set to `lbOwnerDrawFixed` are `ItemHeight` and `IntegralHeight`. If each item is to be of a fixed size, `ItemHeight` specifies how high each one will be. If `IntegralHeight` is `True`, the listbox alters its own height to ensure that no item is only partially visible, otherwise the height remains as specified by the `Height` property and some of the last entry may be seen.

The drawing code goes in the `OnDrawItem` event handler in either case. If the `lbOwnerDrawVariable` style was chosen then the `OnMeasureItem` event handler must also be used to identify the height of each item that needs to be drawn.

Let's see a couple of examples that draw a bitmap and some text in each listbox item. The example program CUSTOM.DPR on this month's disk has two owner drawn listboxes and an owner drawn combobox is thrown in for good measure. The program allows you to choose a directory and press a button (or the `Enter` key) to load all the bitmaps in that directory into a `TStringList` called `Bmps`. The names are added and the bitmaps are loaded as objects using the `AddObject` method (see Listing 1 for the routine that gets called to do this). This `TStringList` is assigned to the `Items` properties of the components in question. The properties of the fixed height listbox (`LstFixed`) are shown in Listing 2 and Listing 3 has the listbox's `OnDrawItem` event handler.

Notice that one of the first operations ensures that the entirety of the item gets covered in the event handler. This ensures that no rubbish is left over from previous

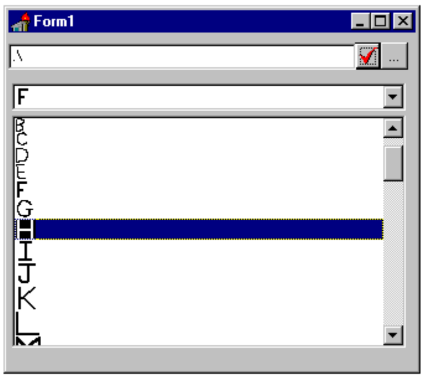
➤ *Listing 2*

```
object LstFixed: TListBox
  Left = 0
  Top = 32
  Width = 305
  Height = 186
  IntegralHeight = True
  ItemHeight = 13
  Sorted = True
  Style = lbOwnerDrawFixed
  TabOrder = 3
  OnDrawItem = LstFixedDrawItem
end
```

➤ *Listing 1*

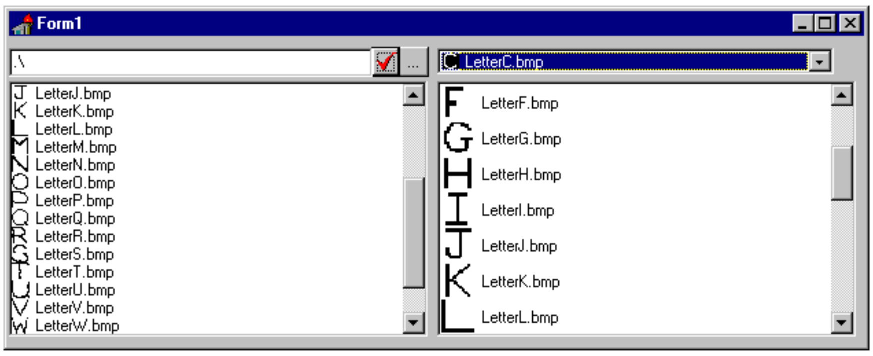
```
procedure LoadBitmaps(const Path: String; S: TStrings);
var
  Bmp: TBitmap;
  SearchRec: TSearchRec;
begin
  EmptyBmpList(S);
  with S do begin
    BeginUpdate;
    if FindFirst(Path + '*.bmp', faAnyFile, SearchRec) = 0 then
      try
        repeat
          Bmp := TBitmap.Create;
          Bmp.LoadFromFile(Path + SearchRec.Name);
          AddObject(SearchRec.Name, Bmp);
        until FindNext(SearchRec) <> 0;
      finally
        FindClose(SearchRec);
      end;
    EndUpdate;
  end;
end;
```

➤ *Figure 2*



items and is achieved by calling the `FillRect` method of the listbox's `Canvas`. This gets followed by a call to the canvas's `StretchDraw` method to draw an appropriately proportioned version of the bitmap into the specified area of the listbox. The `Rect` parameter of the event handler gives the dimensions of the area, whose the height is dictated by the `ItemHeight` property of the listbox.

➤ *Figure 1*



➤ *Listing 3*

```
function ScaleBmpToItemSize(const R: TRect; Bmp: TBitmap): TRect;
begin
  Result := R;
  with Result, Bmp do
    Right := Trunc(Left + (Bottom - Top) * Width / Height)
end;
procedure TForm1.LstFixedDrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var BmpRect, TextRect: TRect;
begin
  with (Control as TListBox), Items, Canvas, Rect do begin
    { Since we are only drawing over part of the item, make sure we
      blank out any old garbage with a foreground rectangle }
    FillRect(Rect);
    { Draw the bitmap, scaled to fit in the fixed height item }
    BmpRect := ScaleBmpToItemSize(Rect, TBitmap(Objects[Index]));
    StretchDraw(BmpRect, TBitmap(Objects[Index]));
    { Invert the item colours if it is selected }
    if odSelected in State then
      InvertRect(Handle, BmpRect);
    { Write the filename after the bitmap, vertically centred }
    with BmpRect do
      TextOut(Right + 4, (Top + Bottom + Font.Height - 1) div 2,
        Items[Index]);
  end
end;
```

➤ *Listing 4*

```
object LstVariable: TListBox
  Left = 3
  Top = 66
  Width = 318
  Height = 187
  Sorted = True
  Style = lbOwnerDrawVariable
  TabOrder = 3
  OnDrawItem = LstVariableDrawItem
  OnMeasureItem = LstVariableMeasureItem
end
```

The `State` parameter is a set that may contain any of the following values:

➢ `odSelected` means that the item has been selected: it would typically have the background displaying in blue.
➢ `odDisabled` implies that the listbox's `Enabled` property is `False`.
➢ `odFocused` suggests that there should be a focus rectangle around the item.

The event handler in Listing 3 checks to see if the item is selected, and if so inverts the bitmap image. It then writes the filename just to the right of the bitmap, using a simple formula to ensure the text is vertically centred. The result can be see in the two listboxes shown in Figure 1. The only difference between these listboxes is that the one on the right has an `ItemHeight` property value twice as large as the other. This results in the pictures being scaled to twice the size.

The combobox in Figure 1 is a fixed height owner drawn control and uses exactly the same logic to draw its contents. Another project called CUSTOM2.DPR is included on the disk. This features a variable height owner drawn listbox (and a combobox again). Figure 2 shows the program running and Listing 4 shows the listbox's properties. You can see that as well as an `OnDrawItem` event handler,we have an `OnMeasureItem` handler. Listing 5 shows what these look like.

The evident differences in the code here include a distinct lack of bitmap stretching. The idea is for the bitmaps to be displayed in their correct size. Additionally, the filename is not written in this project, to simplify the code.

The `OnMeasureItem` handler simply identifies the height of the appropriate bitmap that needs displaying (as specified in the `Index` parameter) and returns it in the `Height` parameter.

Incidentally, the differently sized bitmaps shown in the figures are aldo supplied on the disk.

Supporting Tab Characters

When you want to align columns of information in a listbox, some people will tell you to use a `TListView` component in Delphi 2. If you are writing code that is portable between 16-bit and 32-bit this is not an option. How do we do this with a standard TListBox? One way would be to use a monospaced, or fixed pitch, font and use a calculated number of spaces to align the secondary columns.

Apart from being tedious, this would look rotten. Antiquated versions of Windows forced us to use non-proportional fonts and it was a joy to behold when this restriction was removed. A better solution would be to employ the tab stop support that Windows listboxes already have. However, if you insert a tab character into a listbox data item, it gets drawn as an unsightly black blob (see Figure 3). Unfortunately the TListBox component does not surface this tab support and so a new TListBox derivative is called for: TTabListBox, as implemented in LBOXTAB.PAS on this month's disk.

In the component there is a private data field, FTabStopSupport, to store whether or not tab character support has been enabled (it defaults to False to act like a normal TListBox). This field is surfaced through the TabStopSupport property (not to be confused with the TabStop property, common to all windowed controls).

There is a potential showstopper of a problem with this tab stop support business in that to enable it, the listbox control (ie the Windows interface element, rather than the TTabListBox object that represents it) must have been created at the API level with a certain listbox style flag (lbs_UseTabStops). Conversely, to disable it, the listbox must be created without this flag. This implies that to toggle the tab stop support we will need to destroy and then re-create the listbox Windows interface element on an as-needed basis. But if we do that, what happens to all the data in the listbox?

Fortunately we have no need to worry as it has all been taken care of by those nice people at Borland. Consider for a moment the BorderIcons property of a form, which allows you to remove the minimise and maximise buttons and the system menu at any arbitrary point. This again relies on the form window being destroyed and recreated with the new attributes, but all the controls on the form appear to remain intact when the property gets modified at run-time.

All TWinControl-derived controls have a method called RecreateWnd that will destroy and recreate the Windows interface element, preserving the data as it goes. RecreateWnd calls (indirectly) DestroyWnd and then CreateWnd, both of which are virtual routines. CreateWnd itself calls the virtual CreateParams routine to find the flags, styles and attributes to pass over to Windows in order to get the window created correctly.

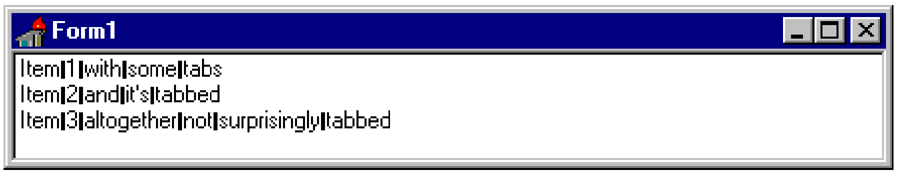
You can override CreateParams in order to choose appropriate styles to be used when the Windows interface element gets created. You can also override CreateWnd and DestroyWnd to save/restore any special data over and above what will be saved for you. By default, the TWinControl functionality simply saves the control's caption/text in the DestroyWnd virtual method and restores it when needed in the CreateParams method. The TListBox (or, to be correct, its ancestor TCustomListBox) class saves the data contained within the listbox using a TStringList in DestroyWnd and restores it all in CreateWnd.

To get our new class working, we need to override CreateParams and set the appropriate window style as suggested by the TabStopSupport property. When the property gets changed, we call RecreateWnd. These bits of code can be sen in Listing 6. When the TabStopSupport property is True, the listbox will interpret ASCII character 9, represented in Delphi either as Chr(9) or #9, as a tab stop character.

Tab Stop Metrics

By default, the listbox has tab stops every 32 horizontal dialog box units, where there are four horizontal dialog box units to every current dialog box basewidth unit. The number of pixels that make up one dialog box basewidth unit can be found by using LoWord(GetDialogBaseUnits): the value used is based on the System font, and is supposed to represent approximately one character position (bearing in mind a proportional font is probably being used in the dialog). So the default tab stops are set to an approximation of every eight characters.

➤ *Figure 3*

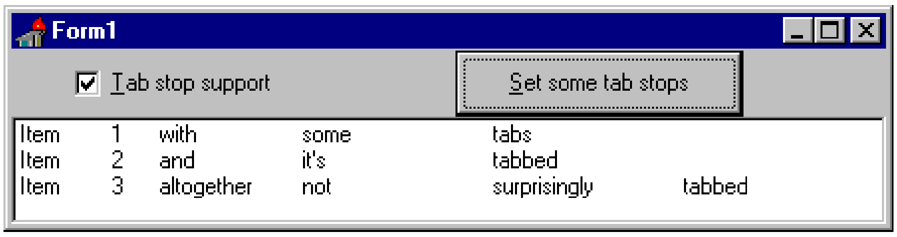


Item█1█with█some█tabs
Item█2█and█it's█tabbed
Item█3█altogether█not█surprisingly█tabbed

➤ *Listing 5*

```
procedure TForm1.LstVariableMeasureItem(Control: TWinControl;
  Index: Integer; var Height: Integer);
begin
  if Index >= 0 then with Bmps do
    if Assigned(Objects[Index]) then
      Height := TBitmap(Objects[Index]).Height;
end;
procedure TForm1.LstVariableDrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var Bmp: TBitmap;
begin
  with LstVariable, Items, Canvas, Rect do begin
    FillRect(Rect);
    Bmp := TBitmap(Objects[Index]);
    Draw(Left, Top, Bmp);
    if odSelected in State then
      InvertRect(Handle, Bounds(Left, Top, Bmp.Width, Bmp.Height))
  end
end;
```

➤ *Figure 4*



```
Item	1	with	some	tabs
Item	2	and	it's	tabbed
Item	3	altogether	not	surprisingly	tabbed
```

➤ *Listing 6*

```
unit LBoxTab;
interface
uses
  StdCtrls, Controls, Classes;
type
  TTabListbox = class(TListbox)
  private
    FTabStopSupport: Boolean;
    FScrollWidth: Word;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure SetTabStopSupport(Value: Boolean);
  public
    function SetTabStops(Tabs: array of Integer): Boolean;
  published
    property TabStopSupport: Boolean
      read FTabStopSupport write SetTabStopSupport default False;
    { Note this property has been given a default of False but that value
      is not being assigned to FTabStopSupport since it will get that value
      anyway. All class data fields are initialised with zero memory byte
      values }
  end;
procedure Register;
implementation
uses
  WinTypes, Messages;
procedure TTabListbox.CreateParams(var Params: TCreateParams);
const
  TabStopSupport: array[Boolean] of Longint = (0, lbs_UseTabStops);
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style or TabStopSupport[FTabStopSupport];
end;
procedure TTabListbox.SetTabStopSupport(Value: Boolean);
begin
  if Value <> FTabStopSupport then begin
    FTabStopSupport := Value;
    RecreateWnd;
  end;
end;
function TTabListbox.SetTabStops(Tabs: array of Integer): Boolean;
begin
  Result := Perform(lb_SetTabStops, High(Tabs) - Low(Tabs) + 1,
    Longint(@Tabs)) <> 0;
  if Result then Repaint;
end;
procedure Register;
begin
  RegisterComponents('Clinic', [TTabListBox]);
end;
end.
```

➤ *Listing 7*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Set some arbitrary tab stops. Note the multiplication by four to turn
    the approximate character positions into horizontal dialog box units.
    The default tab stops are every 8 * 4 }
  if not TabListbox1.SetTabStops([8 * 4, 12 * 4, 24 * 4, 40 * 4]) then
    ShowMessage('Can''t set tab stops')
end;
```

To set up custom tab stops, there is a method in TTabListBox called SetTabStops (Listing 6 again). This takes an open array of integers and sends the passed in values to the listbox as parameters of a lb_SetTabStops message. The specified message takes the address of an array of integers, so the @ operator helps out here. The tab stops must be sorted in ascending order: back-tabs are not allowed. After custom tab stops are set up, the listbox must be redrawn to have its current visible entries reflect the new settings.

One point to note is that these tab stops can only be set if tab stop support is currently enabled. If they are successfully set and then TabStopSupport is set to False, they are abandoned and forgotten. An exercise for you, gentle reader, is to employ DestroyWnd and CreateWnd to remember them if TabStops is set back to True, possibly using a fixed size integer array data field.

There are two projects on the disk which use this component. LBOXTABS.DPR creates one on the fly, so that you do not have to add the component to your component palette, whereas LBOXTBS2.DPR relies on the component having been installed. Listing 7 shows how to pass in an open array of integers to a call to the TabListBox's SetTabStops method, as done in the button's OnClick event handler in LBOXTBS2.DPR (see Figure 4).

Adding A Horizontal Scrollbar

Even though TListBox components are created with the ws_HScroll window style, if the text in the listbox goes past the right hand border, no horizontal scrollbar appears. Normally scrollbars appear in listboxes only if you have multiple columns of entries (see the Columns property).

To get a scrollbar for a one column listbox we need to send it a lb_SetHorizontalExtent message. The horizontal extent of a listbox defaults to 0, but you can set it to a higher value. If the width of the listbox is less than the horizontal extent, a scrollbar is shown, otherwise it is hidden.

Rather than just sending the message when a particular scroll width is chosen, we have to be a bit more wily. We need to also send the message if the listbox gets re-created with `RecreateWnd` as described earlier. We can do this in an overridden version of the `CreateWnd` method. Note that since we are in the throes of component writing, we will send the message to the listbox using the Delphi `TControl` method `Perform`, rather than calling the Windows APIs `SendMessage` or `PostMessage`. A second version of the tabbed listbox component, which I've called `TTabScrollListbox` and is in LBOXTAB2.PAS, includes code to do this (see Listing 8 for the appropriate snippets). The sample project LBOXTBS3.DPR has a spin edit to allow you to test the list box horizontal extent (see Figure 5).

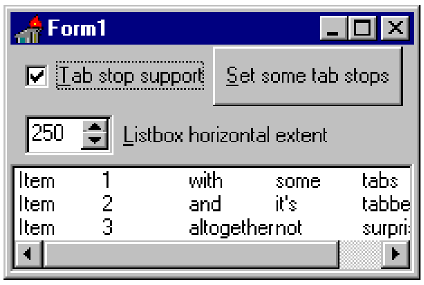
➤ *Listing 8*

```
TTabScrollListbox = class(TListbox)
private
  FTabStopSupport: Boolean;
  FScrollWidth: Word;
protected
  ....
  procedure CreateWnd; override;
  procedure SetScrollWidth(Val: Word);
  ...
published
  property ScrollWidth: Word read FScrollWidth write SetScrollWidth;
...
procedure TTabScrollListbox.CreateWnd;
begin
  inherited CreateWnd;
  { After reconstructing window, reset scrollbar }
  Perform(lb_SetHorizontalExtent, FScrollWidth, 0);
end;

procedure TTabScrollListbox.SetScrollWidth(Val: Word);
begin
  if Val <> FScrollWidth then begin
    FScrollWidth := Val;
    Perform(lb_SetHorizontalExtent, FScrollWidth, 0);
  end;
end;
```

Brian Long is a UK-based freelance Delphi and C++ Builder consultant and trainer. He is available for bookings and can be contacted by email at blong@compuserve.com

➤ *Figure 5*



```
Item    1     with      some    tabs
Item    2     and       it's    tabbe
Item    3     altogethernot     surpri
```