Getting started with Lazarus: Common controls

Michaël Van Canneyt

May 1, 2007

Abstract

In this next article on Getting started with Lazarus, more controls are investigated: editing controls, and (dropdown)list controls such as listboxes and comboboxes.

1 Introduction

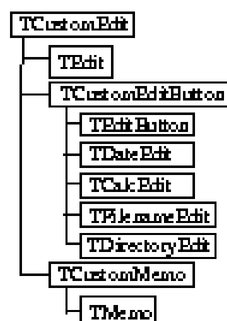
In the previous article on working with Lazarus, simple controls were presented: Passive layouting controls such as panels and bevels. Simple active controls such as checkboxes, radiobuttons and groups of these controls were also covered.

In this article, some more active controls are discussed: edit controls and list controls. The edit controls allow users to enter texts - either a single line of text (in a `TEdit` control) or multiple lines of text (in a `TMemo` control). Both controls are descendents of `TCustomEdit`, which introduces some common behaviour for both types of controls.

Some specialized forms of the `TEdit` control exist: The `TSpinEdit` and `TFloatSpinEdit` controls, which allow editing of numerical values. The `TEditButton` control displays a speedbutton next to the control, which can be used for popping up some dialog: The `TFileNameEdit`, `TDirectoryEdit`, `TDateEdit` and `TCalcEdit` are specialized descendents of this control, which introduce specialized actions for the button. (the complete dependency tree is in figure 1 on page 1.)

The list controls introduced by Lazarus are the well-known `TComboBox`, `TListbox` and `TCheckListBox` controls, with some specialized descendents such as `TColorBox`, `TColorListBox` and `TFileListBox`. They allow to select one or more items from a scrollable list of items.

Figure 1: Edit controls hierarchy



2 Basic edit controls

The most common input action for a user is probably typing some text in some field. This is done with the `TEdit` and `TMemo` controls. The difference between the 2 controls is that the former can only contain a single line of text, and the latter can handle more than one line of text. Both are descendents of `TCustomEdit`

The `TEdit` control has the following relevant properties:

AutoSelect if set to `True`, the whole text in the edit control will be selected when the control receives focus. This means that as soon as the user types some text in the edit control, any previous text will be completely overwritten with the newly typed text. If set to `False` (the default) then newly typed text will simply be inserted at the cursor position.

Charcase determines the case of typed text: `ecNormal` will leave any typed letters as-is. `ecLowercase` will translate any text to lowercase characters. `ecUppercase` will transform any typed letters to uppercase letters.

EchoMode Determines how typed characters are shown in the edit control: `emNormal` means they are shown exactly as they are typed by the user. A value of `emNone` means the user is typing blind: no characters are echoed to the screen. Lastly, `emPassword` means that the password character is substituted for all characters which the user types in the box.

MaxLength If set to a value larger than zero, this value will be used as the maximum length of the text the user can enter. If the maximum length is reached, any characters the user types will be discarded. (this includes copy and paste actions)

PasswordChar If `EchoMode` is set to `emPassword` then all letters in the `Text` property will be substituted by this character prior to displaying the text. Note that the `Text` property retains it's original value (i.e. what the user typed).

Read-Only If set to `True`, the edit control is read-only. This means the user cannot type anything in the control. In difference with setting the `Enabled` property to `False`, the control is still active, and the user can still give focus to the control, and can for example select the text in the control.

SelLength this run-time property returns the length of the selection in characters. It can also be set.

SelStart this run-time property returns the position of start of the the selection. It can also be set, and will cause the cursor to move. Note that the position is zero-based.

SelText this is the selected text in the edit control. It can be retrieved, but can also be set. Any previous selection will be replaced by the new value of this property, regardless of whether the control is in insert or overwrite mode.

Text This is the actual text displayed in the control.

The following events are important:

OnChange Fired whenever the text is changed in the edit box - it is e.g. fired for each typed character.

OnKeyPress is fired whenever a key is pressed: this event can be used to disable certain keys when typing in the control.

OnEditingDone is fired when the user has finished editing the contents in the edit control. Currently, 2 actions will trigger this event: the focus leaves the control, or the ENTER key is pressed. Using this event to check for changes is preferred if a time-consuming check must be done: the event is fired less often than the `OnChange` event.

The meaning of all these properties and events is demonstrated in the first application (editdemo). It has a simple main form, which allows to set most properties of a `TEdit` instance (called `ETest`) with the help of some specialized editors.

The use of the `OnKeyPress` event is demonstrated with a second edit control: if the text of this second control is not empty, the text is used to determine which characters can be entered in the test edit control. The `OnKeyPress` event is coded as follows:

```
procedure TMainForm.ETestKeyPress(Sender: TObject;
                                  var Key: char);

Var
  S : String;

begin
  // Only check alphanumerical characters.
  if Upcase(Key) in ['A'..'Z','0'..'9'] then
    begin
    S:=EAllowedChars.Text;
    If (S<>'') and (Pos(Key,S)=0) then
      Key:=#0;
    end;
end;
```

If the key is a alphanumerical characters, the event handler checks if it appears in the text entered in the edit control `EAllowedChars`. If it is not, the key is set to #0, causing it to be ignored.

The `SelStart` and `SelLength` properties are demonstrated using 2 spinedit controls and a button: As soon as the button is clicked, it reads the begin and end position for the selection from the 2 spinedit controls, and if these positions are valid, sets the `SelStart` and `SelLength` properties:

```
procedure TMainForm.BSelectClick(Sender: TObject);
begin
  If SEFrom.Value>=Length(ETest.Text) then
    Raise Exception.Create(SErrSelStartTooBig);
  If SETill.Value>=Length(ETest.Text) then
    Raise Exception.Create(SErrSelEndTooBig);
  If SETill.Value<SEFrom.Value then
    Raise Exception.Create(SErrTillSmallerThanFrom);
  ETest.SelStart:=SEFrom.Value;
  ETest.SelLength:=(SEtill.Value-SEFrom.Value);
end;
```

Similarly, the `SelText` property can be demonstrated with an edit control (`ESel`) and a button (`BSetSelection`). As soon as the button is clicked, the selection in the `ETest` edit control is replaced with the text in the `ESel` edit control:

```
procedure TMainForm.BSetSelectionClick(Sender: TObject);
```

Table 1: Scrollbars

Value	Visible scrollbars
ssNone	No scrollbars are ever shown
ssBoth	Both scrollbars are always shown
ssHorizontal	Always show a horizontal scrollbar
ssVertical	Always show a vertical scrollbar
ssAutoBoth	Show both scrollbars when needed
ssAutoHorizontal	Show a horizontal scrollbar when needed
ssAutoVertical	Show a vertical scrollbar when needed

```
begin
  ETest.SelText:=ESel.Text;
end;
```

The selection can be cleared with a single method call: `ClearSelection`. The whole text of the control can be cleared with the `Clear` method of `TEdit`. 2 buttons have been dropped on the form to demonstrate these methods.

The `TMemo` control has the same function as the `TEdit` control, but allows to enter multiple lines of text: the number of lines is not really limited. The control has the possibility to display scrollbars if the length of the lines is too long, or if the number of lines is bigger than the number of lines that can be displayed at once. Because of these features, the `TMemo` control has quite different properties than the `TEdit` control:

Lines this is the main property of the memo control: the lines of text that are displayed in the control: a `TStrings` instance. Any changes to the stringlist are immediatly displayed in the control.

ReadOnly As in `TEdit` - the contents of the control can be seen but not edited.

ScrollBars This property determines when horizontal or vertical scrollbars are shown. The allowed values are shown in table 1.

WantTabs if set to `True`, the `TAB` key is used by the memo control: it will display a tab character. The normal behaviour (when this value is `False`) is to switch focus to the next control.

WordWrap if set to `True`, when text is typed which is longer than the length of a visible line, the text will be wrapped to the next line.

The `SelText`, `SelStart` and `SelLength` properties function as in the `TEdit` control, which is not entirely convenient: the position origin is relative to the first character of the first line in the control.

Likewise, the `Clear` and `ClearSelection` methods of `TMemo` do the same as the methods in `TEdit`.

All this is demonstrated in the second demo application: memodemo. The application is not that different from the edit application. An extra button exists (`BFill`), which fills the memo with a lot of data (standard 50.000 lines). Depending on the operating system, this operation can be very slow, since the addition of each line can cause the control to repaint itself. Obviously, this is a time-consuming operation, which should be avoided if possible. Luckily, there is a way to do this:

Under the `BFill` button is a `CBOptimize` checkbox. When checked, the filling routine calls the `BeginUpdate` at the start of the filling routine, and calls `EndUpdate` at the end of the filling routine. Both are methods of the `Lines` property of the memo. The effect

of this is that the memo will only update itself once, at the `EndUpdate` call. The routine `FillMemo` looks like this:

```
 procedure TMainForm.FillMemo(B : Boolean);

Const
  LineCount = 50000;

Var
  I : Integer;
  Start : TDateTime;
  S : String;

begin
  Start:=Now;
  With MTest.Lines do
    begin
    If B then
      BeginUpdate;
    try
      Clear;
      For I:=1 to LineCount do
        Add('This is line '+IntToStr(I));
    Finally
      If B then EndUpdate;
    end;
    S:=FormatDateTime('hh:nn:ss.zzz',Now-Start);
    S:=Format(SFillTime,[LineCount,S]);
    ShowMessage(S);
    end;
end;
```

At the end of the routine, the time it took to fill the memo with 50.000 lines is displayed. On some widget sets, the difference between the optimized routine and the 'normal' routine is not big, but on others it can be very big: it is therefore recommended always to surround heavy modifications of the lines property with calls to `BeginUpdate` and `EndUpdate`, best in a `Try...Finally` block to avoid errors (lockup) in case of an exception.

3 Specialized edit controls

The `TEdit` and `TMemo` controls are the basic editing controls. As shown in figure 1 on page 1, there are quite some more edit controls, descendent from the `TCustomEditButton` control. This control is an ordinary edit control, but has a speedbutton control attached to it, which can be clicked. In the controls delivered with Lazarus, the button is used to pop up a dialog. The result of the dialog (in some form) is then stored as the edit text.

The `TEditButton` control is a `TCustomEditButton` descendent which simply published the newly introduced properties and events. The following properties are published:

ButtonWidth Width of the speedbutton. The height is always equal to the height of the edit control.

ButtonOnlyWhenFocused If set to `True`, the speedbutton is only visible when the edit control has focus.

DirectInput if set to `True`, the user can type a value in the edit control. If it is set to `False`, the only way to enter something in the edit control is by clicking the speedbutton, thus allowing more control over the entered values.

Flat if set to `True`, the speedbutton has a flat look.

Glyph Contains the image shown in the button.

NumGlyphs Number of images in the glyph.

OnButtonClick Event triggered when the user clicks the speedbutton.

The effect of the various properties can be tested in the `editbuttondemo` application. The `OnButtonClick` event is coded as follows:

```
procedure TMainForm.ETestButtonClick(Sender: TObject);

Var
  S: String;

begin
  S:=ETest.Text;
  If InputQuery(SPromptCaption,SPrompt,S) then
    ETest.Text:=S;
end;
```

The `InputQuery` call is a standard call in the LCL, declared as follows:

```
Function InputQuery(Const ACaption, APrompt : String;
                    Var AValue : String) : Boolean;
```

It shows a dialog with caption `ACaption` and prompts the user with message `APrompt` to enter a string. The string is returnedin `AValue`. If `AValue` is not empty when the function is called, it is set as the default value. The function returns `True` if the user ended the dialog with the 'OK' button, it returns `False` otherwise.

Several controls are descended from `TCustomEditButton`:

TFileNameEdit the dialog popped up is one of the filename dialogs. When the dialog is closed, the selected filename is set as the text of the edit control. The control has some extra properties which control the kind of dialog that is shown, plus all the common properties of the file dialogs. It has an extra event `OnAcceptFileName`, which can be used to verify and possibly change the selected filename.

TDirectoryEdit the dialog popped up is the directory selection dialog. When the dialog is closed, the selected directory is set as the text of the edit control. The control has some extra properties which control the appearance of the directory dialog.

TDateEdit when the speedbutton is clicked, a small calendar pops op, and the user can select a date. The selected date is set as the text of the edit control.

TCalcEdit when the speedbutton is clicked, a small calculator pops op, and the user can perform some calculation. The calculated value is set as the text of the edit control.

All these controls are demonstrated in the various demo applications. In figure figure 2 on page 7, some of the demo applications are shown.

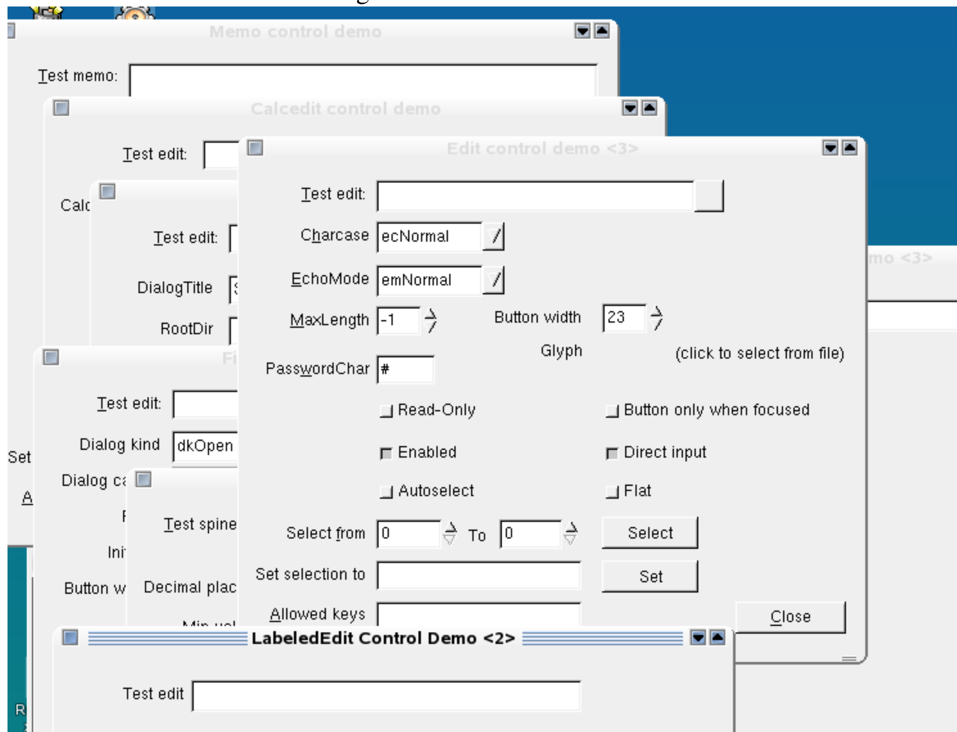


Figure 2: Edit control demos

2 edit controls do not appear on the class hierarchy chart: `TSpinEdit` and `TFloatSpinEdit`. These controls are useful for editing an integer or floating-point value: they have up and down arrows attached to them, which can be used to increment or decrement the value shown in the edit control. They are not on the list because they do not descend from `TCustomEdit`. Instead, they descend directly from `TWinControl`. Nevertheless, these controls have roughly the same properties as a `TCustomEdit` control, and have some extra properties which control their appearance.

Decimals The number of decimal places shown in the control - this property is only present in the `TFloatSpinEdit` control.

Increment The amount by which the value shown in the control is incremented or decremented when one of the up/down arrows is clicked.

Maxvalue The maximum value that can be shown in the control.

MinValue The minimum value that can be shown in the control.

These values and their impact can be tested in the provided test applications.

4 Listboxes

Listboxes can be used to select one or more items from a list. The list of is always visible in a `TListBox` control. The function of this control (to select one or more items from a list of items) is similar to that of the radiogroup or checkgroup. However, the radiogroup or checkgroup are not scrollable, which makes them unsuitable for large lists: the listbox shows as many items as possible, but allows to scroll in the complete list.

The list of items consists of a list of strings - a `TStrings` instance, but both controls have support for drawing the items, so it's possible to represent the items in a graphically attractive way.

The listbox is the more simple control of the 2. It simply shows the list of items, and the user can select one of the items by clicking on it. The behaviour of the listbox is controlled by the following properties:

ExtendedSelect if set to `True` (the default) then extended selection mechanisms become available when the control is in multiselect mode: CTRL-Clicking to add an item to a selection, or `Shift-Click` to add an item. If set to `False`, items must be selected (or deselected) by double-clicking them.

IntegralHeight If `True`, an item won't be drawn unless it fits completely in the listbox.

ItemIndex the index of the selected item in the list if the list is not in multiselect mode. This is a run-time property. The index is zero-based.

Itemheight The height of an item (in pixels).

Items the list of strings that is shown in the list.

Multiselect If set to `True`, multiple items can be selected from the list.

Selected a boolean array property: for each item in the list, the array indicates whether it is selected (`True`) or not (`False`). This property can be used only when the list is in multiselect mode.

Sorted if `True`, the items in the list are kept sorted.

Style determines whether the listbox is an ordinary listbox (`lbStandard`), an owner-drawn listbox with fixed height for the items (`lbOwnerDrawFixed`) or with variable height (`lbOwnerDrawVariable`). In both cases, the `OnDrawItem` event must be used to draw the items in the list.

TopIndex The (zero based) index of the item that is shown at the top of the list.

All these properties are demonstrated in the `listboxdemo` application. The `Delete` button shows how to use the `Selected` property to determine which items are selected:

```
procedure TMainForm.BDeleteClick(Sender: TObject);

Var
  I : Integer;

begin
  With LBTest do
    if MultiSelect then
      begin
      For I:=Items.Count-1 downto 0 do
        If Selected[i] then
          Items.Delete(I);
      end
    else
      Items.Delete(ItemIndex);
end;
```

Note that the list is traversed in reverse order.

The remark made for the memo control - about adding a lot of items to the list - is also valid for listboxes: when doing a long list of changes to the `Items` property, it's best to enclose this operation in a `BeginUpdate`/`EndUpdate` pair of statements. This is again demonstrated by the fill button in the demo application.

The fact that the `Items` property of a `TListbox` is of type `TStrings` is quite useful: the `Objects` property can be used to associate an object with each of the items in the list: this can be useful for OwnerDraw listboxes, or for small database objects. This can be easily demonstrated. Assume a small database application, which manages a collection of songs (e.g. for a playlist). Each song (`TSong`) is an item in a collection (TSongs), which forms the playlist. The classes could look as follows:

```
Type
TGenre = (gUnknown,gRock,gClassic,gSoul,gPop,gHouse,gFolk,gBlues);

{ TSong }
TSong = Class(TCollectionItem)
Published
  Property Author : String Read FAuthor Write FAuthor;
  Property Title : String Read FTitle Write FTitle;
  Property Duration : TDateTime Read FDuration Write FDuration;
  Property Genre : TGenre Read FGenre Write FGenre;
  Property Album : String Read FAlbum Write FAlBum;
end;

{ TSongs }

TSongs = Class(TCollection)
Public
  Procedure Populate;
  Property Songs[Index : Integer] : TSong Read GetSong
                                        Write SetSong; Default;
end;
```

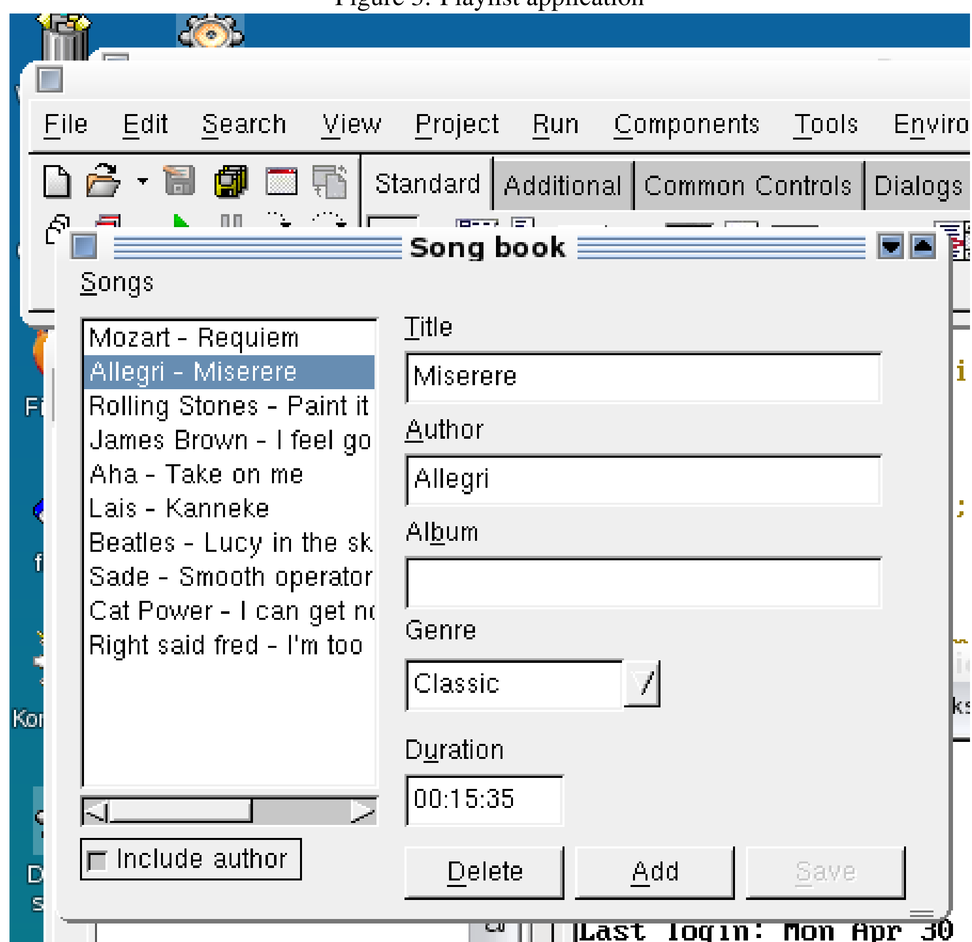
The playlist `TSongs` can be shown in a `TListBox` component on a form using the title and optionally the author. As soon as the user selects one of the items in the listbox, the rest of the data can be shown in edit controls, allowing the user to edit them. The application can look as in figure 3 on page 10. Assuming the form has an instance `FSongs` of type `TSongs`, the filling routine for the listbox on the left side can be coded as follows:

```
procedure TMainForm.ShowSongs;

Var
  I : Integer;

begin
  For i:=0 to FSongs.Count-1 do
    AddSong(FSongs[i]);
  If FSongs.Count>0 then
    begin
    LBSongs.ItemIndex:=0;
    ShowSong;
    end;
end;
```

Figure 3: Playlist application



The actual work is done by the `AddSong` procedure. The second part of the `ShowSongs` procedure is simply making sure that a song is selected, and showing the details of this song. The `AddSong` routine is coded as follows:

```
function TMainForm.AddSong(S : TSong) : Integer;

begin
  Result:=LBSongs.Items.AddObject(SongCaption(S),S);
end;
```

As can be seen, the `AddObject` procedure is used to add both a description of the song as well as the song object itself to the `Items` property of the listbox. The `SongCaption` function constructs a display text from a song object:

```
function TMainForm.SongCaption(S : TSong) : string;

begin
  Result:=S.Title;
  If CBAuthor.Checked then
    Result:=S.Author+' - '+Result;
end;
```

It takes into account the setting of the 'Authors' checkbox.

The `AddSong` function above returns also the `Index` property of the newly added item (as returned by AddObject) This is used to set the current item when a new song is added:

```
procedure TMainForm.BaddClick(Sender: TObject);

Var
  S : TSong;

begin
  S:=FSongs.Add as TSong;
  S.Author:='New author';
  S.Title:='New song';
  LBSongs.ItemIndex:=AddSong(S);
  ShowSong;
end;
```

More methods deal with the listbox objects, but the interested reader is referred to the sources of the demo application (listboxobjects).

The `MultiSelect` property of a `TListBox` determines whether a user can select more than one item in the listbox. From a user's point of view, it may be preferrable to have a checkbox next to the items he wants selected, such as in a `TCheckGroup` control. The `TCheckListBox` shows all it's items with a checkbox in front of it. The state of each checkbox can be verified using the `Checked` array property. The demo application checklistboxdemo has a button which, when clicked, executes the following code:

```
procedure TMainForm.BShowClick(Sender: TObject);

Var
  I : Integer;
  S : String;
```

```
begin
  S:=SCheckedItems;
  For I:=0 to LBTest.Items.Count-1 do
    If LBTest.Checked[i] then
      S:=S+sLinebreak+LBTest.Items[i];
  ShowMessage(S);
end;
```

It simply loops over all items, and saves the ones that are checked. Note that the `TCheckListBox` still has the `MultiSelect` property, and the `Selected` array such as they exist in `TListbox`. The `Checked` and `Selected` properties are independent of each other and can both be used.

Another descendent of the `TListBox` control is the `TFileListBox`. It can be used to show a list of files from a directory, allowing the user can select some files. There are 2 properties which determine which list of files is shown:

Directory The directory from which files should be shown. This property currently is not published, and should be set run-time.

FileName is the name of the currently selected file.

FileType The attributes which a file should have to be shown in the listbox. The values are equivalent to the values used in the `FindFirst`/`FindNext` calls found in the sysutils unit.

Mask A file mask specification, as used in `FindFirst` call of the sysutils unit. Do not specify a path here.

The `UpdateFileList` method can be used to update the list of files. It's called automatically when one of the above properties is changed. The filelistboxdemo application shows how to use this component.

The `TColorListbox` control is a descendent of `TListbox` which allows to select a set of colors in a list of pre-defined system colors (the number of items in the list is determined by the `Palette` property). Each color is shown as a name with a small square in front of it, colored in the appropriate color. The shown colors are available in the `Colors` array property - this is useful when determining which colors are selected when the listbox allows to select multiple colors. If only a single color can be selected, the selected color is available in the `Selection` property - it can be both read and set.

5 Combobox

The `TComboBox` control can be used for the same purposes as a listbox. In difference with a `TListbox` control, the list of items to select from is always hidden and can be shown using a button, which will cause the list to drop down. As a result, the `TCombobox` control uses less room than a listbox or a radiogroup.

Additionally, an item which is not present in the list of items can be entered in the combobox by simply typing the text in the editable part of the control: in this case the combobox acts as a normal edit control, with a list of predefined values from which the text can quickly be set, much like a histroy list. This behaviour is customizable.

The `TCombobox` control has a lot of properties controlling it's behaviour:

ArrowKeysTraverseList if set to `True`, using the `Up` and `Down` arrows while the control has focus, will select the previous or next item in the list. If `False`, the mouse must be used to select an item.

AutoComplete If set to `True`, when typing text in the control, the text will be completed with the first matching text in the items. A match is found when the item starts with the same characters as the already typed text. Much like an 'incremental search' kind of behaviour.

AutoCompleteText can be used to fine-tune the autocomplete option. It is a set with various values, which are self-explaining. Including the first item in the set (`cbActEnabled`) is equivalent to setting the `AutoComplete` property.

AutoDropDown if set to `True`, then the list will be made visible as soon as a character is typed in the combobox.

AutoSelect is the same as for the `TEdit` control: if set to `True`, the whole text is selected when the combobox gets control.

CharCase is also the same as for the `TEdit` control: it determines the case (upper/lowercase) of the typed characters.

Dropdowncount Determines how many items are shown when the dropdown list is shown. If more items are in the list than the value set here, scrollbars will be shown to allow scrolling in the list.

ItemHeight The height of the items in the list.

ItemIndex The index of the currently selected item. It can also be set: it has only a meaning if the style is one of `csDropDownList` or one of the ownerdraw styles

ItemWidth The width of the dropdown list - in pixels

MaxLength The maximum length of text that can be shown. Note that if the text of an item in the list is longer than the value specified here, the text will be clipped.

Sorted Determines whether the items in the list appear sorted.

Style can be used to specify the style of the combobox.

The `Style` property can have several values:

csDropDown the user can enter text freely, and can use the list to set the text to a predefined value. Newly entered texts are not automatically added to the list.

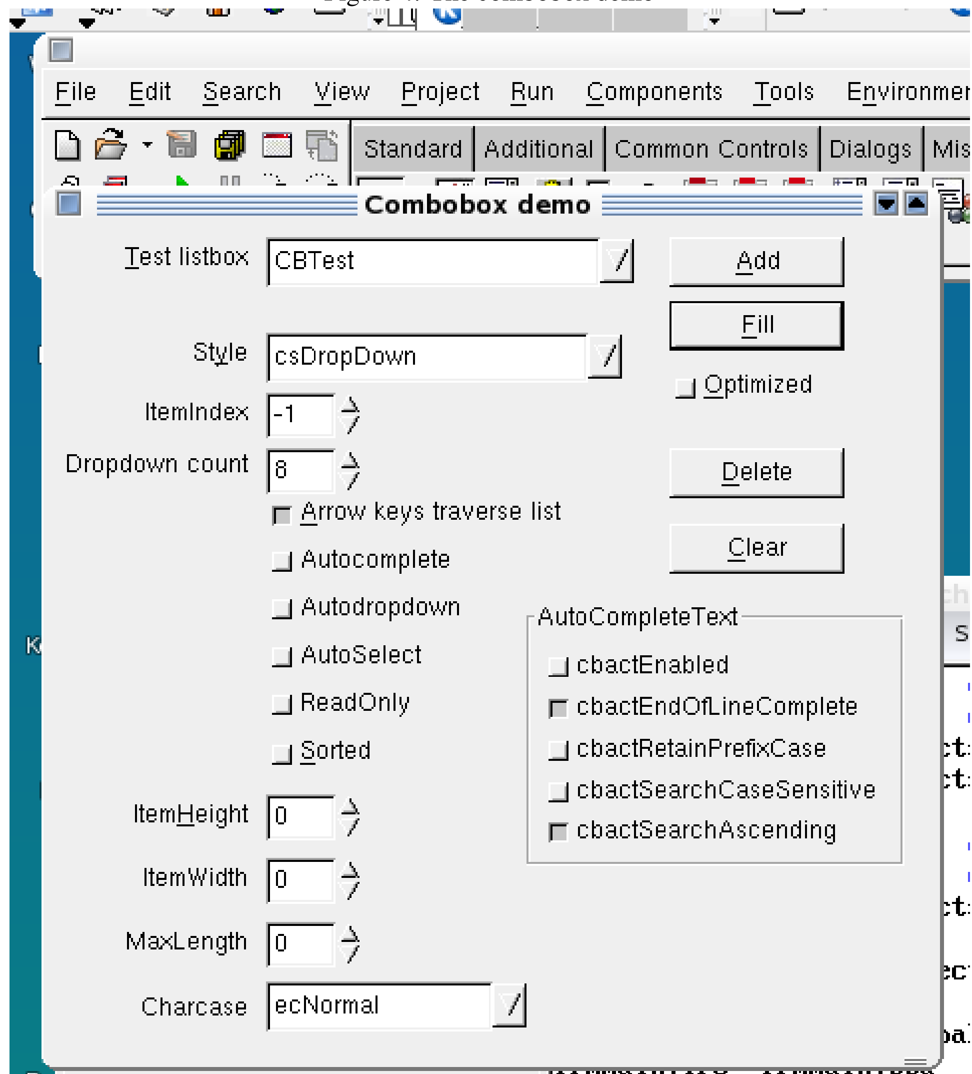
csDropDownList The user can only select a value which appears in the list.

csOwnerDrawFixed Like `csDropDownList`, but the items are drawn in the `OnDrawItem` event, with fixed heights.

csOwnerDrawVariable Like `csDropDownList`, but the items are drawn in the `OnDrawItem` event, with variable heights.

csSimple On Windows, this gives the combobox the appearance of an edit control with a listbox under it (i.e. the list is always visible). Other than that the control acts like a `csDropDown` combobox. On other platforms, this value equals the `csDropDown` style.

Figure 4: The combobox demo



The combobox is demonstrated in the comboboxdemo application, as in figure 4 on page 14.

As for the Listbox, the `Items` property is of type `TStrings`, and objects can again be associated with it. The songs application demonstrated above, can be adapted to use this: Instead of making the author a string in the `TSong` class, a separate `TAuthor` class is introduced which introduces some new properties:

```
TAuthor = Class(TCollectionItem)
Published
  Property Name : String Read FName write FName;
  Property FirstName : String Read FFirstName Write FFirstName;
  property IsGroup : Boolean Read FIsGroup Write FIsGroup;
end;
```

The simple edit control to edit the author can now be replaced with a combobox (`CBAuthor`), which shows the name of the author, but keeps a reference to the `TAuthor` instances of the list of authors in the `Objects` property. This instance can then be used to set the `Author` property of the `TSong` instance when the user selects an author. Likewise, when showing the details of an author, the `ItemIndex` property of the combobox can be set to the index of the `TAuthor` instance in the items:

```
procedure TMainForm.SetAuthor(const AValue: TAuthor);

begin
  With CBAuthor do
    if (AValue=Nil) then
      ItemIndex:=-1
    else
      Itemindex:=Items.IndexOfObject(AValue);
end;
```

The rest of the code can be found in the comboboxobjects demo application.

The `TColorBox` control is a descendent of `TComboBox` which allows to select a color in a list of pre-defined system colors (the number of items in the list is determined by the `Palette` property). Each color is shown as a name with a small square in front of it, colored in the appropriate color. The selected color is available in the `Selection` property - it can be both read and set; Like the `TColorListbox`, the shown colors are available in the `Colors` array property.

6 Conclusion

In this article, 16 controls available by default in the LCL were discussed. While these are considered to be more simple controls, they offer a wealth of possibilities: all of them are very customizable, simply by setting a few properties. Their versatility is demonstrated further by the fact that they in fact descend from 4 basic controls: the Edit control, the Memo control, the Listbox and Combobox. By slightly changing the behaviour of one out of these basic controls, a wealth of new controls can be created, making common programming tasks easy and freeing the time of the programmer for the real challenge: creating a good-looking user application.