Under Construction: Customising Controls

by Bob Swart

In the previous columns, we have examined the process of writing our own visual and non-visual components. This time, we'll focus on some additional techniques that are helpful during component building. In addition, we'll examine the finishing touch of every component: a nice custom bitmap for the component palette.

We'll now develop a new edit box component: one that is single line and yet capable of right-aligned text. Normally, this combination is impossible (one of Windows' delicate blunders), but with a little hocus-pocus, we can make it work anyway. First of all, we need to derive from the existing TMemo (and not TEdit – the reason will become clear!). Just start Delphi, select File | New Component from the menu, and fill it in just like Figure 1.

As soon as we click OK, we're in a new source file with the component skeleton source code. Let's save this file as TREDIT.PAS somewhere in our Delphi search path. Now we can make our modifications to the new TRightEdit component. In order to find the right properties to modify, select the text TMemo in the source editor, and press Control F1 to get the on-line help for the standard TMemo component. The properties pop-up page shows the property Alignment of type TAlignment, with three possible values: taLeftJustify (the default), taCenter and taRightJustify. It's this property we're interested in here. Note that we could not use a TEdit, since this property is not available for a TEdit, but only for a TMemo, TLabel or TPanel.

Specialising

So, we know what we need to change for our TRightEdit, but now the question is how do we change it? In order to set the property Alignment to an initial value of taRightJustify, we need to override the constructor for the TMemo. Since almost every component has a constructor of the form

```
constructor X(AOwner:
  TComponent); virtual;
```

we just repeat this code in the public section of our TRightEdit component. The implementation of this constructor is easy (see Listing 1). First we need to call the inherited constructor to make sure the TMemo stuff is constructed all right, then we can go ahead and customise the TRightEdit component by setting the Alignment property to taRightJustify.

Will this be enough? Let's test it by installing the component onto the component palette. Select Options | Install Components from the menu and Add the TREDIT.PAS source file to the list. After compiling, we have a new component on the Dr.Bob palette page. If we drop this component onto a form, however, we see that not only is the text right-aligned, we also get a component with the same initial size as a TMemo, ie multi-line instead of single line. We need to set the initial size of the component!

Customisations

Let's go back to the source of the TRightEdit component. Apart from the initial size, we have to make sure the component cannot be resized to anything higher than a single line. How high can a single edit line become? Well, something just below twice the font size, I'd say. If we want to make sure the component size stays within that range, we can override the SetBounds method, which applies to just about all VCL controls. The declaration is as follows (again in the public section):

```
procedure SetBounds(ALeft,
  ATop, AWidth, AHeight :
  Integer); override
```

SetBounds sets the component's boundary properties (Left, Top, Width and Height) to the values passed in ALeft, ATop, AWidth and AHeight respectively. SetBounds enables us to control all of the component's boundary properties in one place, as it will be called for each re-size event! The implementation of SetBounds can be found in Listing 2. Note that we use the absolute value of the Height subproperty of the Font property, since the Height can become negative for some font types!

➤ *Listing 1 Constructor TRightEdit.Create*

```
constructor TRightEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Alignment := taRightJustify;
end;
```

➤ *Listing 2 Procedure SetBounds*

```
procedure TRightEdit.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  if AHeight > (2 * abs(Font.Height)) then AHeight := 2 * abs(Font.Height);
  inherited SetBounds(ALeft, ATop, AWidth, AHeight);
end;
```

Design Versus Run Time

If you've made the changes from Listing 2 in the source code of the `TRightEdit` but *did not* install the component on the palette again, you'll notice that the `TRightEdit` component will size correctly at run-time, but still has the old behaviour at design time.

In order to make sure the `SetBounds` procedure is used at design time too, we have to install the component again by using `Options | Install Component`. This time, just a click on the `OK` button will do, since TREDIT.PAS is already in the installed list of components.

Now, if we drop a `TRightEdit` component on a form, it immediately gets the correct height. We can also try to resize it by hand, or any other way, and it just keeps bouncing back (see Figure 2).

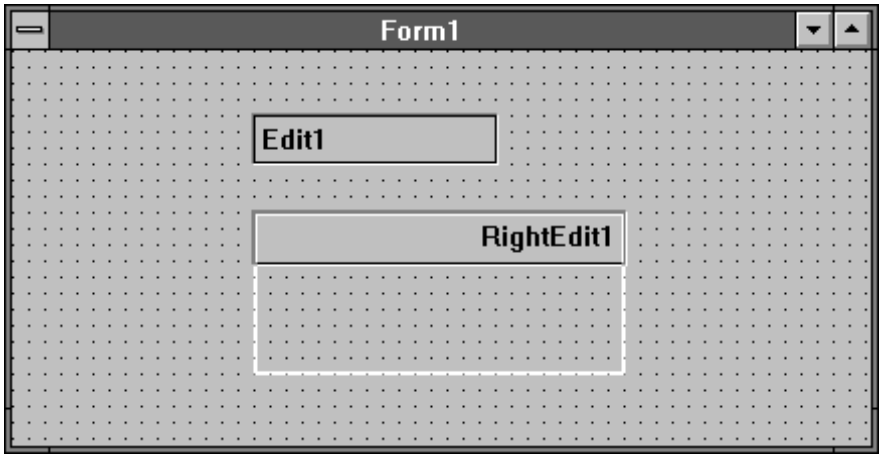
So, What Have We Learned?

It seems we have learned that we can actually have two "behaviours" of a component: one at design time (from the component that's inside the COMPLIB.DCL) and one at run-time (here, a possible source or .DCU file overrides the code in the COMPLIB.DCL).

Generally, we can simply use the Component Expert to derive our new component and immediately install the new (empty) component skeleton code into the component palette. This will give us the initial design time component behaviour (from the base class). Any additional code can be added and tested at run-time (and not at design time, yet).

This has two benefits: first of all, it won't be necessary to rebuild the entire COMPLIB.DCL every five minutes, and second, we don't have to be afraid of accidentally "breaking" COMPLIB.DCL (I once wrote a component that generated a stack fault inside COMPLIB.DCL; in such an event, all you can do is re-install the backup copy of your COMPLIB.DCL and *don't* try installing the component again, which will generate a new backup of your already corrupt COMPLIB.DCL – again, as I did once). *[The answer is to make your own backup copy of a working COMPLIB.DCL, eg to COMPLIB.BKK, before installing any new components. Editor]*

➤ *Figure 1 Component Expert*



➤ *Figure 2 SetBounds in Action*

Are We Done Yet?

Not by a long shot! Just have a look at the Object Inspector when our new `TRightEdit` is selected. A whole bunch of properties that are actually `TMemo` specific and have nothing to with a single one-line right-aligned edit box. You can compare the properties from a `TEdit` and `TRightEdit` in Figure 3.

Specifically, I would like to hide the following properties: `Align`, `Alignment`, `Lines`, `ScrollBars`, `WantReturns`, `WantTabs` and `WordWrap`. Of course, we also have to make sure these properties are initialised to a sensible value.

We also seem to be missing the following `TEdit` specific properties: `AutoSelect`, `AutoSize`, `CharCase`, `PasswordChar` and `Text`. Let's start with these first.

Adding New Properties

Adding new properties to our `TRightEdit` component is not hard to do. First of all, the five missing properties are already defined in the `TCustomEdit` base class. This class is the base class of `TEdit`, but also of the `TCustomMemo`, `TMemo` and our `TRightEdit` class. The `TCustomXXX` classes are just the same as their `TXXX` counterparts, except for the fact that the properties of a `TCustomXXX` class are all public instead of published. So, they're here, but you can't see them in the Object Inspector.

In order to visualise them again, we can re-declare them in the published section of our `TRightEdit` component, as shown in Listing 3.

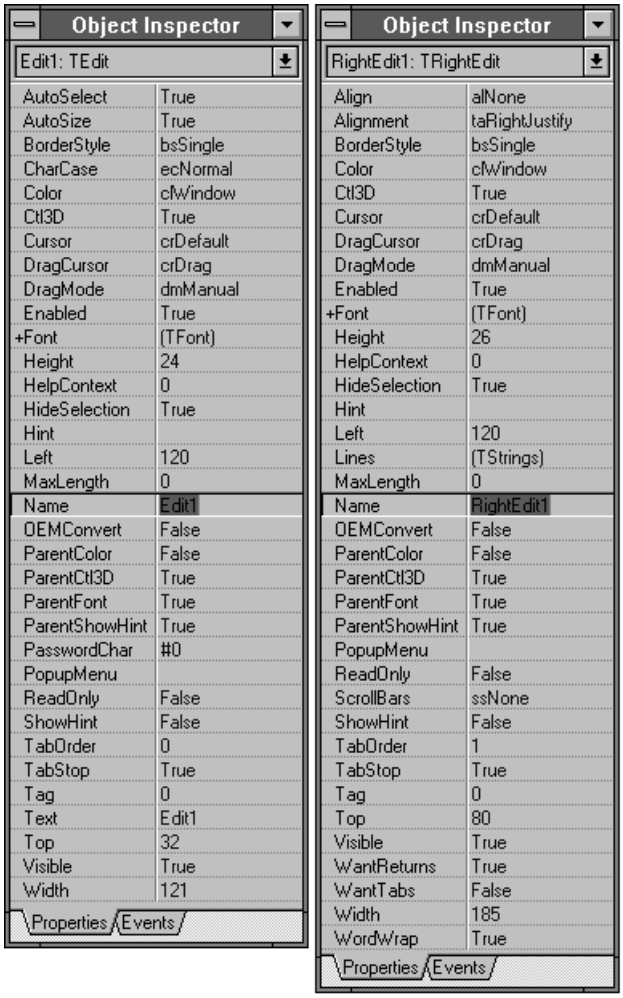
If we compile the new component and fire up the Object Browser (in the `Views` menu), we can see that the five new properties appear for our `TRightEdit` component.

But will this be enough? If we set the `Text` property, do the contents of the `TRightEdit` change also? Surprisingly enough, yes. We can even use both the `Text` and the `Lines` properties at the same time! It seems that both properties are actually only "wrappers" around the standard Windows edit control, and both get their value (ie their contents) from this edit

```
type
  TRightEdit = class(TMemo)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner:
      TComponent); override;
    procedure SetBounds(ALeft,
      ATop, AWidth, AHeight:
      Integer); override;
  published
    { Published declarations }
    property AutoSelect;
    property AutoSize;
    property CharCase;
    property PasswordChar;
    property Text;
  end;
```

➤ *Listing 3*
Published properties

➤ *Figure 3*
Properties of TEdit and TRightEdit



control. This is also the reason, by the way, why a `TMemo` can never hold more than about 32K characters: another nice limitation of 16-bit Windows! Also note that the `Text` property is actually the sum of the `Lines`, with a CR-LF pair added after each line. These two non-printable characters cannot be shown by Windows and are represented by two little black boxes in the value of the `Text` property. In order to solve this problem, we'll have to make sure the `Lines` property disappears, so the only access method to the internal data of the Windows control is the `Text` property.

For `TRightEdit`, this will make the `Text` property work the same way as for a normal `TEdit`. I consider the other four properties to be less important, so I leave them to you to implement (if you feel you need them – have a look at the original VCL source code for `TEdit` if you need hints how to do it). They are published, but not all are actually working (the `PasswordChar` property, for example, doesn't have any effect for our `TRightEdit` – but I wouldn't want to use a right aligned field to type a password anyway). The `CharCase` property is the only one that also works by default.

Property Censorship

It seems that in order to let the `Text` property work without problems, we have to somehow disable or hide the `Lines` property. But how can we hide properties? The manual clearly states that it should be considered impossible to "de-publish" properties. Well, this doesn't seem to be the case. A simple re-declaration of the properties, only this time in the `protected` section (or any section other than the `published` section) will *seem* to have exactly the effect we want, as shown in Listing 4. The properties will still be accessible at run-time and should be no longer be visible at design-time. Let's check this to be sure...

If we install the new `TRightEdit` component from Listing 4 and check the Object Browser again, we see that the seven to-be-unpublished properties are indeed marked as protected, just as we wished. If, however, we drop a `TRightEdit` component on a form and check the Object Inspector, we see in horror that the un-wanted properties are still available and hence remain published.

Could this perhaps be a bug in the Object Browser or in the Object Inspector? A closer look at the internal working of Delphi's run time type information (RTTI) provides us with the answer: each component's RTTI lists the properties it publishes. When searching for a property name, RTTI scans the object, then each of its ancestors in turn. No matter what you do, you can't change the ancestors' RTTI info, and the property will be found. So it seems that the RTTI published-list works like a black list: once you're on it there's no way to get off it!

Methinks the Object Browser should also use RTTI better to show the properties as published instead of protected, by the way (Borland, are you reading this?).

TCustomXXX

Of course, there are usually more ways to solve a problem in Delphi than one! In this case, we could have derived our `TRightEdit` from `TCustomMemo` instead of from `TMemo`. Then, all our properties would have been hidden, including the

```
unit Tredit;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TRightEdit = class(TMemo)
  private     { Private declarations }
  protected   { Protected declarations }
    property Align;
    property Alignment;
    property Lines;
    property ScrollBars;
    property WantReturns;
    property WantTabs;
    property WordWrap;
  public      { Public declarations }
    constructor Create(AOwner: TComponent); override;
    procedure SetBounds(ALeft, ATop, AWidth, AHeight :
      Integer); override;
  published   { Published declarations }
    property CharCase;
    property Text;
  end;
procedure Register;
```

```
implementation
constructor TRightEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Align := alNone;
  Alignment := taRightJustify;
  ScrollBars := ssNone;
  WantReturns := False;
  WantTabs := False;
  WordWrap := False;
end;
procedure TRightEdit.SetBounds(ALeft, ATop, AWidth,
AHeight: Integer);
begin
  if AHeight > (2 * abs(Font.Height)) then
    AHeight := 2 * abs(Font.Height);
  inherited SetBounds(ALeft, ATop, AWidth, AHeight);
end;
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TRightEdit]);
end;
end.
```

➤ *Listing 4 Almost-final TRightEdit (without the three non-working properties)*

```
unit TRedit;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Menus, Dialogs, StdCtrls;
type
  TRightEdit = class(TCustomMemo)
  private   { Private declarations }
  protected { Protected declarations }
  public    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    procedure SetBounds(ALeft, ATop, AWidth, AHeight:
      Integer); override;
  published { Published declarations }
    property BorderStyle;
    property CharCase;
    property Color;
    property Ctl3D;
    property Cursor;
    property DragCursor;
    property DragMode;
    property Enabled;
    property Font;
    property Height;
    property HelpContext;
    property HideSelection;
    property Hint;
    property Left;
    property MaxLength;
    property Name;
    property OEMConvert;
    property ParentColor;
    property ParentCtl3D;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ReadOnly;
    property ShowHint;
    property TabOrder;
    property Tag;
    property Text;
    property Top;
    property Visible;
    property Width;
  end;
```

➤ *Listing 5 Final class definition for TRightEdit (implementation is the same as Listing 4)*

ones we would like to hide now. We just have to re-publish all the properties we need in order to get the same end result. That's the main reason why the `TCustomXXX` classes exist: to allow Component Builders to easily customise the basic VCL components.

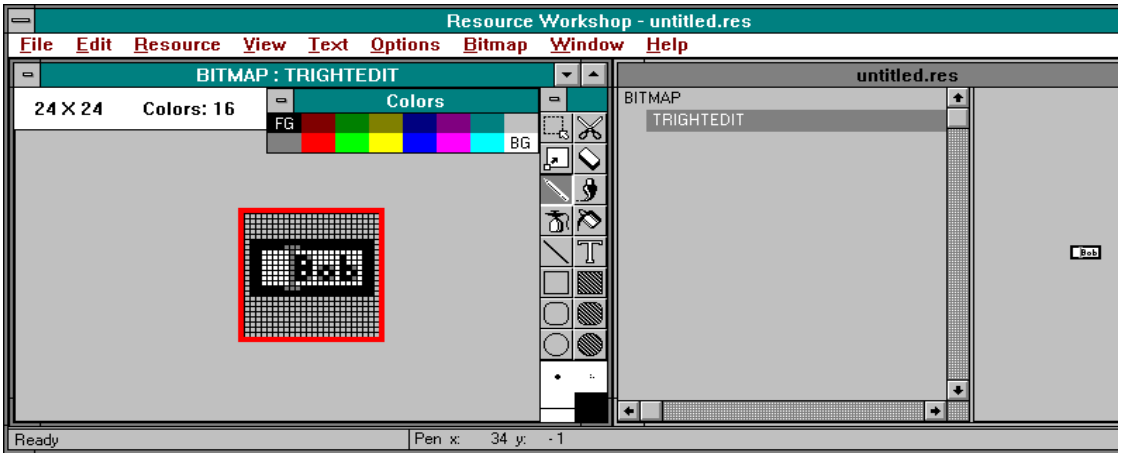
The final class definition for our `TRightEdit` component, now based on a `TCustomMemo`, is seen in Listing 5. Note that we now derive from a `TCustomMemo` and only need to re-publish the properties we want, using our earlier technique.

If you've read the source code of Listing 4, you will notice that it already contains the last thing we must not forget if we un-publish properties: giving them an initial value. Since the designer cannot give these properties an initial value, we (the programmer of the component) need to make sure the hidden properties have a sensible value assigned at creation time.

Component Palette Bitmaps

Now that we've designed and implemented a new component, one more thing we need to do is give it a unique Component Palette bitmap. Every component needs a bitmap to represent itself on the Component Palette. If a component does not have a bitmap specified, Delphi uses the bitmap of the ancestor (back to `TComponent` itself, if necessary). That's why the components from the first issue all had the same bitmap on the palette. If it wasn't for hints, we would never have been able to distinguish between them.

In our case, the `TRightEdit` has the same bitmap as its parent, `TMemo`. Since our component is not a memo but a right aligned edit, we really need to give it a more fitting bitmap. A palette bitmap for a component is in fact very easy to create: all we need to do is to make a Windows (compiled) resource file with a 24x24 bitmap. My preference is to use Resource Workshop, since I already have a lot of experience with this tool (it



➤ *Figure 4 Creating the palette bitmap in Resource Workshop*

is not part of Delphi itself, but comes with Borland Pascal, Borland C++ and is also part of the RAD Pack for Delphi).

We start a new project for a .RES file and create a new BITMAP resource (24x24, 16 colour) for our `TRightEdit` bitmap (see Figure 4). There's only one more thing we have to make sure of: the name of the BITMAP resource itself must be the name of the component: `TRIGHTEDIT` in our case.

Resource Workshop by default saves this file as a .RES file, which we must rename to a .DCR file. The .DCR file needs to have the same prefix as the unit file: TREDIT. The TREDIT.DCR file needs to be present with the TREDIT.PAS or TREDIT.DCU file when installing the component in order to be used.

Next Time

Well this sure was a whole lotta work for one simple `TRightEdit` component, wasn't it!

But we learned a lot in the process: how to test the run-time behaviour of components without having to install them each time, how to re-publish or de-publish properties and how to give a component its own palette bitmap. The complete source code of the `TRightEdit` component is of course included on the free disk with this issue.

In the next issue, we'll build some more components and take a look at making a Component Help File, including merging keywords with the Delphi help file, so that component users will be able to access on-line help for our components' properties and events easily whilst in design mode.

Bob Swart (you can email him on 100434.2072@compuserve.com) is a professional software developer using Borland Pascal, C++ and Delphi. In his spare time, he likes to watch video tapes of *Star Trek The Next Generation* with his 1.5 year old son Erik Mark Pascal.