Tips & Tricks

Name That Enumeration!

I once almost quit the programming profession out of frustration with translation tables. Those ancient woes are now history, I'm glad to report, as Delphi has made programming most compelling again. The Delphi compiler allows you to tap into what it knew about your interface when it compiled your app. This is known as Run Time Type Information (RTTI) and, for one, is how Borland was able to build Delphi using Delphi. RTTI is also what makes it possible for the Object Inspector and Browser to work with the same information and methods. That's quite a feat and why we're here today looking at how the Object Inspector would know about the strings used to define `Enumerated` and `Set` types. Say you're translating from `Boolean` to text by implementing a constant:

```
array BoolString[false..true] of string[5]
```

Whoops, keep your array. `Boolean` is a built in simple type and can not be decoded via RTTI. Any other `Enumerated` or `Set` name can be obtained as a string from RTTI by calling `TypInfo.GetEnumName` as in this example:

```
String := GetEnumName(TypeInfo(TWindowState),
                      ord(aState))^;
```

where this is what it all means:

```
String:=          {text for current WindowState}
GetEnumName(      {returns a Pointer to a String}
TypeInfo          {returns pointer to RTTI info
                   for type}
(TWindowState)    {this is the type we want to
                   know about}
Ord               {get the sequential number of
                   the value}
(aState)          {typed value we're decoding}
)^;               {de-reference pointer,
                   get the string}
```

Listing 1 shows a sample unit to populate a memo with possible form-states. That's it. Try it on the colors. Use a string to set a color. Perhaps next time I'll give you some pointers on borrowing the VCL's writer procs to create a simple run-time object inspector...

Contributed by Michael Ax of Ax-Systems.
© 1995 Michael/Ax-Systems.

Components Going PString!

You can change component fields from `String` to `PString` types and make big savings! You can also use the 'third' string type to save memory.

Pascal and even C strings are of fixed length (at least in Delphi 1.x). You write and read them from the same pre-allocated memory location. Enter `PString`, the *pointer to a string* string type. It's a bit more work but a `PString` will never occupy more than 4 bytes if empty.

To use them in a component you will have to have at least two methods, as you must prepare and destroy the `PStrings` along with the component. To publish them, every `PString` needs to have two additional procedures. To access them, you must de-reference the pointer variable to get the string, thus you need a get procedure. To write to them, you need to use the `AssignStr` procedure, which first frees any space used previously before reserving memory for the new string.

Listings 2 and 3 show excerpts from a sample component, before and after converting to `PStrings`, respectively.

Contributed by Michael Ax of Ax-Systems.
© 1995 Michael/Ax-Systems.

➤ *Listing 1*

```
unit SetNames;
interface
uses
  TypInfo, SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Memo1 : TMemo;
    procedure FormCreate(Sender: TObject);
  private
  public
  end;
var
  Form1 : TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender : TObject);
var
  aState : TWindowState;
begin
  Memo1.Lines.Clear;
  with Memo1.Lines do
    for aState := low(TWindowState) to
high(TWindowState) do
      Add(GetEnumName(TypeInfo(TWindowState),
ord(aState))^);
end;
end.
```

➤ *Listing 2*

```
type
  TStringBin = class(TComponent)
  private
    fString : String;
  published
    aString : String read fString write fString;
    end;
```

OOTable

Would you like to know how to make a database field a property? Sometimes you simply want to control a few fields while making them read-only to the user.

In this adventure we're going to descend a table from TTable that is going to have a published property representing a hypothetical UniqueID field that can be manipulated directly via code. Take a look at the code in Listing 4.

Contributed by Michael Ax of Ax-Systems.
© 1995 Michael/Ax-Systems.

Debugging

Have problems using the integrated debugger to see what's going on inside a TListbox or a TCombobox? When I tried to look at the contents of the items, I found each and every one gave me a string of length 0. When I did a character dump, I found the title of the form I was currently working on, preceded by a zero. I knew there were other values there, since previous lines used text from the items array, but I wasn't getting what I expected and I wanted to check and see what MyListBox.items[2] actually contained.

Conditional defines to the rescue. I set up a test stringlist and assigned the values in my Tlistbox.items to it; now I could see them. I have no idea why Borland set up TListboxes and descendants so that you can't see their contents in the watch window, but at least the TStringlist can be seen. How to do that? In the implementation or interface section, put the following code fragment:

```
{$IFDEF buggy}
var tstlst : tstringlist;
{$ENDIF}
```

Then, in the FormCreate procedure, activate tstlst:

```
{$IFDEF buggy}
tstlst := tstringlist.create;
{$ENDIF}
```

➤ *Listing 3*

```
type
  TPStringBin = class(TComponent)
  private
    fString : PString;
  protected
    procedure SetString(const Value : String);
    function GetString : String;
  public
    constructor Create(aOwner : TComponent); override;
    destructor Destroy; override;
  published
    aString: String read GetString write SetString;
  end;
implementation

constructor TPStringBin.Create(aOwner : TComponent);
begin
  inherited Create;
  fString := NullStr; {prepare the PString}
end;

destructor TPStringBin.Destroy;
begin
  DisposeStr(fString); {free the PString}
  inherited Destroy;
end;

procedure SetString(const Value : String);
begin
  AssignStr(fString,Value); {free old and store new}
end;

function GetString:String;
begin
  Result := fString^;
  {fString is a pointer. '^' points to the string}
end;
```

➤ *Listing 4*

```
unit OoTable;
{ We're going to create a property that represents
  the current record's unique id }
interface
uses db, dbtables;
type
  TTableUniqueIDField = class(TTable)
  private
    fUniqueID: TField;
  protected
    function GetUniqueID:LongInt;
    procedure SetUniqueID(aValue:LongInt);
  public
    procedure DoAfterOpen; Override;
  published
    property UniqueID: LongInt
     read GetUniqueID write SetUniqueID stored false;
  end;

{ Note: while you could make this a component you'd
  probably want to define individual components for
  every major table in the system via inheritance...
  but that gets tricky if you want tables to know
  about each other. On the other hand you can create
  new instances of tables easily if they have a type.
  You can set table names in the Create procedures so
  that you can instantiate and open them without
  knowing a table name or alias! For example, at
  least do:}
  TCustomerTable = class(TTableUniqueIDField);

{ that's enough to get an alias type for this }

const
  cUniqueID = 'UniqueID'; { reference to field name }

implementation

procedure TTableUniqueIDField.DoAfterOpen;
begin
  inherited DoAfterOpen;
  fUniqueID := FieldByName(cUniqueId);
    {raises exception if not there}
  fUniqueID.ReadOnly := True;
  { Now the table is open and the field has been
    hooked up }
end;

procedure TTableUniqueIDField.SetUniqueID(
  aValue:LongInt);
begin
  if State <> dsInactive then
    {while the table is open}
    with fUniqueID do
      {change the property/field if needed}
      if AsInteger <> Value then begin
        ReadOnly := False;     {gain access}
        AsInteger := aValue;   {set}
        ReadOnly := True;      {restrict access}
      end;
end;

function TTableUniqueIDField.GetUniqueID:LongInt;
begin
  if State <> dsInactive then
    {while the table is open}
    Result:=fUniqueID.AsInteger {return the property}
  else
    Result:=0;
end;
end.
```

Then, where you want to use the debugger to see what's happening, include:

```
{$IFDEF buggy}
tstlst.assign(MyListBox.items);
{$ENDIF}
```

And, especially during long sessions, be sure to release the space in the FormDestroy procedure:

```
{$IFDEF buggy} tstlst.free; {$ENDIF}
```

To make it all work, the final step is to click on Options | Project | Conditionals and enter buggy (or whatever term you are using) in the conditional box.

Contributed by Brandon Smith
Email: synature@aol.com

Customising Forms

In Issue 4, Claus Ziegler showed some code snippets to show and hide title bars at run-time. If you know that you want a particular look to your form all the time, you can take a different approach and override the form's CreateParams method.

CreateParams is used to set up the window style bits used by Windows when building our form. We can take the default bits and modify them with the window style flags defined in the WinTypes unit. You can find the definitions of them towards the bottom of the online help for the CreateWindow API. Values are added in by "or"ing them, and taken away by "and not"ing them.

As an example, Listing 5 shows code to make a form with a border but no caption.

Based on a submission by Soenke Pries (Email: CompuServe 100347,2040) with further illumination provided by Brian Long

TNoteBook Page Backgrounds

I write computer assisted learning (CAL) software and sometimes this involves using an electronic book to get over a few pages of information to students (to prevent death threats from the CAL community I hasten to add this book is *not* the whole lesson). It has become common practice for the pages of such books to have a graphical background, as opposed to a flat wash of colour.

This brings me on to the Delphi issue. The TNoteBook is the ideal component for creating these books since it only takes the addition of a few buttons to allow the user to browse through the pages. Unfortunately it is not possible to place an image onto a form and use it as the background to a TNoteBook (perhaps Borland could look at adding a transparent property to the notebooks controls?).

A first attempt involved using a separate TImage component on each page of the notebook. I assumed that Delphi would only store the bitmap once in the EXE. This is not so: every TImage is stored as a unique occurrence and, as I was using a 300Kb BMP file over five pages, this added an unnecessary 1200Kb to my EXE file.

Several days browsing in the CompuServe Delphi forum did not help. I found one other person with a similar problem and the suggestions made to him did not work. However, some of the information from CompuServe did stick and I had one of those rare inspirational flashes...

The solution is to place one TImage component onto the notebook, set its alignment as required (alClient in my case) then load in the picture. Now create a handler for the notebook OnPageChanged event and insert the code shown in Listing 6.

Now whenever the notebook's page is changed the background graphic remains constant. This method is fast enough to go almost unnoticed on a 386/40.

Contributed by Stewart McSporran
Email: CompuServe 100753,1703

Clean Up Your Code

In most forms there are a lot of TLabel or TBevel controls. In the unit source code these controls are also declared, although in most cases they don't need to be, because you wouldn't usually assign event handlers for a TLabel, for example.

To avoid the declaration in your source code of a lot of controls you don't need, simply empty the Name property in the Object Inspector and *voila* the control is deleted in the corresponding source code but available on the screen!

➤ *Listing 5*

```
TForm1 = class(TForm)
public
  procedure CreateParams(var Params: TCreateParams);
    override;
end;
procedure TForm1.CreateParams(
  var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  with Params do
    Style := Style and not
      (ws_Border or ws_ThickFrame) or ws_Popup;
end;
```

➤ *Listing 6*

```
procedure TForm1.NoteBook1PageChanged(Sender: TObject);
begin
  { Make the current page the parent
    of the TImage control}
  TImage1.Parent :=
   TWinControl(
     NoteBook.Pages.Objects[NoteBook.PageIndex]);
  { Ensure the image is at the back of the z order
    so that the text can be seen}
  TImage1.SendToBack;
end;
```

If you delete all the `TLabel` controls from the source you may find that you get the problem that Delphi says *"TLabel isn't registered"* so in this case you must do the registration yourself, before the form is created. To do this, include the line:

```
RegisterClasses([TLabel,TBevel]);
```

(for example) at the start (just after `begin`) of the .DPR project file will register the required components.

You can do the samething with other design-only components such as `TGroupbox` – it will make your source more readable, because only the components you need for programming appear.

Contributed by Stefan Boether
Email: CompuServe 100023,275

Send In Those Tips Please!

If you have some useful tips accumulated from your long hours of development (all highly enjoyable of course!) why not share them with your fellow Delphi developers? Just drop an email to the Editor, Chris Frizelle, on 70630.717@compuserve.com or send us a disk, letter or fax. Who knows, you could even become famous!