Tips & Tricks

Cleaner, More Reliable Code

Many errors are due to programmers not properly creating or initialising objects. This is the reason many don't like global variables. Delphi, however, by default will auto-create your forms and these are global variables. When forms' behaviour needs to change, for example from modal to modeless, we have to change the calling unit(s) and the unit which defines the form. Another disadvantage of auto-create forms is that every form is created, even if not all of them are used during a particular session – if you have a lot of forms starting the application can take a long time and uses more memory than is strictly necessary. This is why I create forms dynamically, using class methods.

The advantage of class methods is that they reduce the amount of interaction between units. It's not necessary to create an instance in the calling unit. See the example in Listing 1. If we want to use another About form with a different name then we only have to change the call and the uses clause in the main form. All the other things which are specific to the About form are defined and used only in the About form's unit, thereby making the application less error prone.

At my company we often show forms as modeless, for example the About form. All we have to do is change the class procedure and add the `FormClose` method (`action := caFree`), the code in the main form remains the same. However, if the form was auto-created, the global variable in the About unit will become useless and all calls to the About form will need to be changed.

Contributed by Stephan Westen, SWesten@SIG.NL

Managing Stored Procedures

Steve Troxell has shown us how to use stored procedures in the client/server environment, but how do we install them with Delphi? I don't want to use ISQL or similar tools from the database server vendor to install the procedures on the server. Also I don't want to write the following code for each procedure:

```
CONNECT ...
DROP PROCEDURE ...
CREATE PROCEDURE ...
```

I solved this problem by creating a subdirectory called SQL underneath the directory where the application resides. In this subdirectory I place all the stored procedure scripts (with the extension .SQL).

Listing 2 shows the SQL for the `NextNoGet` procedure, placed in the file NEXTNOGET.SQL. In Listing 3 (over the page) you see the procedure `CheckProcedures`, which retrieves all the stored procedure names into the `aProcs` stringlist for the given database. After this it scans all the .SQL files in the SQL subdirectory with the `FindFirst` and `FindNext` functions. For every SQL script found, the procedure `UpdateSQL` is called.

After this the name of the file is checked against all the installed procedures, if the procedure already exists it is removed with the `DROP PROCEDURE` statement. The `TQuery` object we need for this is created on the fly.

Note that the `TQuery.ParamCheck` property is set to False. Without doing this we can't installed any stored procedure using the : character, as most do. The : char is a parameter and Delphi will try to replace the parameter with a value. If you try to install a stored procedure with the DataBase Explorer and get an invalid token ? value this indicates that Borland have not set the property correctly.

Finally, the SQL script is loaded into the `SQL` property and the query is executed. Now the stored procedure is installed or upgraded!

The call to `CheckProcedures` should be hidden for normal users and only visible for the system administrator who needs to update a procedure. All s/he has to do is edit the scripts in the SQL directory and do a call to the `CheckProcedures` method. No need for WISQL, DBEXPLORER or DataBase Desktop.

Contributed by Stefan Boether,
CompuServe 100023,275

➤ *Listing 1*

```
class procedure TfmAbout.ShowForm(pOwner: TComponent);
begin
  try
    with TfmAbout.Create(pOwner) do begin
      ShowModal;
      Free;
    end
  except
    ErrorMessage('Something is wrong');
  end;
end;
procedure TMainForm.AboutClick(Sender: TObject);
begin
  TfmAbout.ShowForm(self);
end;
```

➤ *Listing 2*

```
CREATE PROCEDURE NextNoGet(aTable CHAR(20))
RETURNS(aNextNo INTEGER)
AS
BEGIN
  SELECT Number FROM NEXTNO_BP
  WHERE Id = :aTable
  INTO :aNextNo;
  IF (aNextNo IS NULL) THEN
    BEGIN
      INSERT INTO NEXTNO_BP(Id, Number) VALUES(:aTable, 2);
      aNextNo = 1;
    END
  ELSE
    BEGIN
      UPDATE NEXTNO_BP SET Number = Number + 1
      WHERE Id = :aTable;
      aNextNo = aNextNo + 1;
   END
END
```

Re-Opening Closed Datasets

In some cases you may want to close one or all databases for a short time and then re-open them than with all the dependent tables and queries. That can be a problem because the `Active` property does not automatically remember its state. However, we can re-open the datasets with the code shown in Listing 4. It inserts all active datasets in a `TList` before closing the database and dropping the connections of the current session. After this you have nothing open and can do the stuff you want, then you can re-open the database. The dependent datasets are then re-activated.

➤ *Listing 3*

```
function ExtractName(const Filename: String): String;
{ from my XProcs library }
var aExt : String;
    aPos : Integer;
begin
  aExt := ExtractFileExt(Filename);
  Result := ExtractFileName(Filename);
  if aExt <> '' then begin
    aPos:=Pos(aExt,Result);
    if aPos>0 then Delete(Result,aPos,Length(aExt));
  end;
end;
procedure CheckProcedures(aDb: TDataBase);
var
  aPath   : String;
  aSearch : TSearchRec;
  aResult : Integer;
  aProcs  : TStringList;
  procedure UpdateSql(const aFile: String);
  var aProc: String;
      i: Integer;
  begin
    if FileExists(aFile) then begin
      aProc:=ExtractName(aFile);
      with TQuery.Create(nil) do
      try
        DataBaseName := aDb.DataBaseName;
        if aProcs.Find(aProc,i) then begin
          SQL.Add(Format('DROP PROCEDURE %s',[aProc]));
          ExeSQL;
        end;
        ParamCheck   := False;
        SQL.LoadFromFile(aFile);
        ExecSQL;
      finally
        Free;
      end;
    end;
  end;
begin
  aPath := ExtractFilePath(ParamStr(0))+'sql\';
  aProcs := TStringList.Create;
  try
    aProcs.Sorted := True;
    Session.GetStoredProcNames(aDb.DataBaseName, aProcs);
    aResult := FindFirst(aPath+'*.sql',faAnyFile,aSearch);
    while aResult = 0 do begin
      UpdateSQL(aPath+aSearch.Name);
      aResult := FindNext(aSearch);
    end;
    FindClose(aSearch);
  finally
    aProcs.Free;
  end;
end;
```

➤ *Listing 4*

```
var
  aOpenList: TList;
  j: Integer;
begin
  aOpenList:=TList.Create;
  try
    with Session do for i:= 0 to DatabaseCount - 1 do begin
      for j:=0 to Databases[i].DataSetCount-1 do
        if DataBases[i].DataSet[i].Active then
          aOpenList.Add(Pointer(DataBases[i].DataSet[i]));
      Databases[I].close;
      DropConnections;
    end;
    ....
    for j:=0 to aOpenList.Count-1 do
      TDbDataSet(aOpenList[j]).Open;
  finally
    aOpenList.Free;
  end;
end;
```

Contributed by Stefan Boether

Table Cloning

Have you ever wanted a second table handle for accessing another record in the same table? For example to duplicate a record. I use the code shown in Listing 5 to do the job: it creates a table object on the fly using the properties from a source table. The table is then opened and positioned on the same record as the source table. Don't forget to dispose the returned table with `Free` after a call to `CloneTable`.

Contributed by Stefan Boether

BDE Paradox Driver Bug

In the work of converting an update program from the 16-bit protected mode Paradox Engine for DOS to the 32-bit Borland Database Engoine (BDE), I discovered a bug in the BDE Paradox driver. It seems that because of a bug in the implementation of the `DbiTranslateRecordStructure` function, you cannot create a Paradox table with a Blob field of size 0 using the VCL `TTable` component. I mananged to come up with a workaround by modifying the implementation of the `TTable.CreateTable` method.

The 16-bit version of the BDE also has this same problem, but the `TableType` property of `TTable` must be manually set to `ttParadox` for it to appear. It seems to be another bug that causes `GetDriverTypeName` to return `nil` instead of `PARADOX`. This makes `DbiTranslateRecordStructure` simply copy the logical record onto the physical record instead of actually converting it.

This bug affects the following released BDE versions: BDE32 3.12, BDE32 3.5 and BDE16 2.51, and Delphi versions 1 and 2.

To re-create the bug, create a new application, drop a button and a memo on the main form, then add the code in Listing 6 to the `Click` event of the button.

Now run the application and press the button. You will see in the memo that the size of the Blob field in the table that was created is 1, not 0 as we intended. This can also be verified by using the Database Explorer and seeing that the size and physical size for the blob field is 1 and 11, respectively, not 0 and 10 as we intended.

➤ *Listing 5*

```
function CloneTable(aTable: TTable): TTable;
begin
  Result:=TTable.Create(nil);
  with Result do begin
    DataBaseName:=aTable.DataBaseName;
    TableName:=aTable.TableName;
    IndexFieldNames:=aTable.IndexFieldNames;
    Open;
    GotoCurrent(aTable);
  end;
end;
```

For the 16-bit BDE, you can get around the bug by relying on another bug... Simply make sure `TableType` is `ttDefault` (the default). Then use `.DB` as the extension of the `TableName` property to signal that this should be a Paradox database. To solve the problem for all cases, follow the guidelines below.

For the 32-bit BDE, the only way I have found to fix this (other than calling the DBI functions directly) is to modify the implementation of the `TTable.CreateTable` method found in the SOURCE\VCL directory. At line 1790, insert the code shown in Listing 7 (line 1338 for Delphi 1). The first and the last statements are taken from the original source so that you know where to insert the code.

➤ *Listing 6*

```
procedure TForm1.Button1Click(Sender: TObject);
var Table: TTable;
begin
  Table := TTable.Create(Self);
  try
    with Table do begin
      Table.DatabaseName := 'C:\';
      Table.TableType := ttParadox;
      Table.TableName := 'TestIt.DB';
      Table.FieldDefs.Add('Blob', ftBlob, 0, false);
      Table.CreateTable;
      Table.Open;
      Memo1.Lines.Add(Format('Blob.Size = %d',
        [Table.FieldByName('Blob').Size]));
    end;
  finally
    Table.Free;
  end;
end;
```

This is a quick-and-dirty fix, but it does work. The code first checks to see if the we are in fact using the Paradox driver. Then it loops through all the logical fields in the `FieldDescs` array. If it finds a Blob field that has a logical size of 0, it changes the correspondig physical field in the `TableDesc.pFLDDesc` array so that the `iLen` and `iUnits1` fields are set correctly.

If you do not want to modify the VCL source, you can create your own `TTable` descendant component that implements this corrected `CreateTable`. Note that this would be a static replacement rather than a virtual override: this means that any code that calls the old

➤ *Listing 7*

```
Check(DbiTranslateRecordStructure(nil, iFldCount,
  FieldDescs, GetDriverTypeName(DriverTypeName),
  PSQLLName, pFLDDesc, False));
{$DEFINE Fix_Paradox_ZeroSizeBlob_Bug}
{$IFDEF Fix_Paradox_ZeroSizeBlob_Bug}
  { Correct a bug in the Paradox driver. This enables us
    to have blob fields that use zero extra bytes in the
    record structure. }
  if StrComp(DriverTypeName, 'PARADOX') = 0 then
    for I := 0 to iFldCount-1 do
      with PFieldDescList(FieldDescs)^[I] do
        if (iFldType = fldBLOB) and (iUnits1 = 0) then
          with PFieldDescList(TableDesc.pFLDDesc)^[I]
            do begin
            { The Paradox driver has a bug in that it sets
              all Blob fields of size 0 to size 1 }
            Dec(iLen, iUnits1);
            iUnits1 := 0;
          end;
{$ENDIF}
iIdxCount := IndexDefs.Count;
```

➤ *Listing 8*

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
  private
    procedure Test;
  public
  end;
var
  Form1: TForm1;
  x : Integer = 10;  // <<<<
  y : Integer = 99;  // <<<<
implementation
{$R *.DFM}
procedure TForm1.Test(); // <<<<
begin
  label1.caption := 'x= ' + inttostr(x);
  label2.caption := 'y= ' + inttostr(y);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  test();  // <<<<
end;
end.
```

`TTable.CreateTable` (like `TBatchMove.Execute`) would still call the old code.

Contributed by Hallvard Vassbotn,
email: hallvard@falcon.no

Initialized Variables

In Listing 8 you can see some Delphi 2 language features you may not have known about. The first new feature is initialized variables. You can do the same thing with variables that previously we could only do using typed constantsL: set an initial value for a variable. In fact it is probably best to use the new `$J-` compiler directive to turn writable constants off and use the new initialized variables instead. You are excused if you did not know about this since it is mentioned only in the language syntax diagrams and under the `$J` compiler directive, with no code examples anywhere.

The second feature may be more of a curiousity than a feature you wish to use. In C and C++ you always have to add the parentheses to a procedure or function call, even when there are no parameters, like `test();`. In Pascal you used to get a syntax error if you wrote code like that. It seems that Delphi 2 allows this calling syntax. This may be a bug or the result of the shared compiler backend with C++.

Contributed by Rick Hansen,
email rickh@artemisalliance.com

Thanks for the Tips, everyone – keep them coming in! Just drop an email to the Editor on DelphiMagazine@compuserve.com