

Keeping Hold of Your Things

by
Joanna Carter

Part 4

In this, the last in this series of articles on Custom Datasets, I will finish describing the remaining abstract methods of TDataset that must be implemented in order to give us a fully functioning component. As you may have noticed in the previous article, we are now starting to cover methods that have to be specialised for the kind of physical data that we wish to access.

Inner Parts

The majority of methods remaining are all concerned with bits of code that Delphi expects us to fill in, as TDataset will call these methods and expect a sensible response.

Although we have already looked at moving data to and from the buffers that we are provided with; along with getting the data for each TField to and from that buffer, we have not yet found how to get the dataset to create the TField objects that it will require. In order for the dataset to create TField objects, it has to use one TFieldDef for each TField in the dataset.

The internal method that creates these field definitions is InternalInitFieldDefs:

```
procedure TBinaryTable.InternalInitFieldDefs;
var
  i: Integer;
begin
  FieldDefs.Clear;

  for i := 0 to Pred(fHeader.FieldCount)
  do
    with fHeader.FieldDef[i] do
      FieldDefs.Add(FieldName, FieldType,
        FieldSize, False);
  end;
end;
```

As you can see, if you use a Schema class to hold details of each of the fields in the physical table, then the code becomes extremely simple. However I will point out that if you use the TSchemaFieldDef that we looked at in the first article, then you will also need a separate list to keep hold of the offsets of each of the fields in the physical table. Here is the interface for a better version that copes with almost everything you will need to keep track of:

```
ISchemaFieldDef = interface
[ '{YOUR GUID}' ]
  function ForeignFieldSize: Integer;
  function ForeignFieldType:
    ForeignFieldType;
  function DisplayName: string;
  function FieldName: string;
  function FieldOffset: Integer;
  function FieldSize: Integer;
  function FieldType: TFieldType;
  function TranslatorClass:
    TBufferTranslatorClass;
end;
```

You will see the TranslatorClass used later in this article, but for now we will move on to discuss the rest of the internal methods.

```
procedure TBinaryTable.InternalAddRecord(Buffer:
  Pointer; Append: Boolean);
begin
  InternalLast;
  fStream.Seek(0, soFromEnd);
  fStream.WriteBuffer(ActiveBuffer^, fRecordSize);
  Inc(fRecordCount);
end;
```

In the case of our basic binary file, InternalAddRecord can only append to the end of the file and therefore all that is required is to go to the end of the physical file stream and write the record out. InternalClose is responsible for destroying any field objects within the TDataset and then closing the physical file by freeing the associated stream:

```
procedure TBinaryTable.InternalClose;
begin
  BindFields(False);

  if DefaultFields then
    DestroyFields;

  fStream.Free;
  fCursorOpen := False;
end;
```

In the context of our basic binary file deleting records would cause all sorts of problems and therefore, all we can do is tell a user that this operation is not possible:

```
procedure TBinaryTable.InternalDelete;
begin
  ShowMessage('Delete: Operation not yet
  supported');
end;
```

Setting the dataset to the first or last record involves making the dataset believe that it is looking at a record before the first record or after the last record. See the code for GetRecord to understand how Delphi uses Bof and Eof flags. In the case of our basic system, the following code will suffice.

```
procedure TBinaryTable.InternalFirst;
begin
  fCurrentRecord := -1;
end;

procedure TBinaryTable.InternalLast;
begin
  fCurrentRecord := fRecordCount;
end;
```

Remember the article that covered bookmarks? Well here is the last little bit of code that Delphi requires to position the physical dataset on a record, the position of which has been stored in a bookmark:

```
procedure TBinaryTable.InternalGotoBookmark(Bookmark:
    Pointer);
begin
    with PBinaryBookmark(Bookmark) ^ do
    begin
        fCurrentRecord := RecPtr;
        // or use an API call to move to the
        // required record
    end;
end;
```

In our simple dataset, all we will do with exceptions, is to pass them to the application's exception handler:

```
procedure TBinaryTable.InternalHandleException;
begin
    Application.HandleException(Self);
end;
```

Of course, when Delphi puts data into a record buffer it would be good to ensure that there is no garbage left over from its last use:

```
procedure TBinaryTable.InternalInitRecord(Buffer:
    PChar);
begin
    FillChar(Buffer^, fRecordBufferSize, 0);
end;
```

After setting the Database and TableName properties in the object inspector, the next thing that can happen is the table may be opened. Once again Delphi knows about opening TDatasets, but it does not know how to open your special dataset:

```
procedure TBinaryTable.InternalOpen;
begin
    try
        if fTableName = '' then
            DatabaseError(SNoTableName);
    end;
```

DatabaseError(SNoTableName); is a pre-defined error routine and constant. Next we create a stream associated with the physical data file:

```
fStream := TFileStream.Create(fFileName,
    fmOpenReadWrite or fmShareDenyNone);

InternalInitFieldDefs;

if DefaultFields then
    CreateFields;
```

If persistent fields have not been created in the Fields Editor then Delphi needs to create fields dynamically; this is done in CreateFields. BindFields then sets up all sorts of internal things like field offsets, as well as creating any calculated or lookup fields. It is within BindFields that CalcFieldsSize is actually set for later use in InitBufferPointers:

```
BindFields(True);

fRecordCount := fStream.Size div
    FRecordSize;
fCurrentRecord := -1;
```

As you will know by now, our simple dataset relies on records being sequential in a binary file and being of fixed length; therefore the number of records can be calculated by dividing the file size by the record size. Don't forget to set the positioning pointer to the 'crack' before the first record.

```
BookmarkSize := SizeOf(TBinaryBookmark);
InitBufferPointers;

// everything OK: table is now open
fCursorOpen := True;
except
    fCursorOpen := False;
    raise;
end;
end;
```

To finish opening the file, we set the BookmarkSize property, set up all of our buffer offset pointers and (hopefully) set the fCursorOpen flag ready for the IsCursorOpen method:

```
function TBinaryTable.IsCursorOpen:
    Boolean;
begin
    Result := fCursorOpen;
end;
```

Should anything have failed, then the flag will be false and everything will close down gracefully??

```
procedure
    TBinaryTable.InternalSetToRecord(Buffer:
    PChar);
begin
    InternalGotoBookmark(Buffer +
        fBookmarkOfs);
end;
```

Using a simple binary table, we are very limited in the operations that we can perform; but we can, at least, write back any changes in the buffer to the appropriate record. Not only do we have to implement InternalGotoBookmark, but also InternalSetToRecord, as this is called from other places. Once the physical table is pointing at the correct record, then we can tell the dataset to write the data back to the file:

```

procedure TBinaryTable.InternalPost;
begin
  CheckActive;
  if State = dsEdit then
  begin
    fStream.Position := fRecordSize *
      fCurrentRecord;
    fStream.WriteBuffer(ActiveBuffer^,
      fRecordSize);
  end
  else
  begin // State = dsInsert
    InternalLast;
    fStream.Seek(0, soFromEnd);
    fStream.WriteBuffer(ActiveBuffer^,
      fRecordSize);
    Inc(fRecordCount);
  end;
end;
end;

```

Well that just about sums up what is needed to interact successfully with Delphi in order to connect to a non-Delphi format data file. Now I will go on to describe the TBufferTranslator that I touched on at the end of last month's article.

How to Understand a Drug Addicted Parrot

In other words how to construct a polymorphic translator class hierarchy. Still don't understand, eh?

Let me remind you of the GetFieldData method that we looked at last time; especially the bit where we had to determine which type of field we were getting the data for:

```

function TForeignTable.GetFieldData(Field:
  TField; Buffer: Pointer):
  Boolean;
var
  Ptr: PChar;
  Offset: Integer;
  Translator: TBufferTranslator;
  FieldClass: TBufferTranslatorClass;
begin
  ...
  FieldClass :=
    fHeader.FieldDef[Field.FieldNo -
    1].TranslatorClass;
  if FieldClass = TOurString then
    Translator :=
      FieldClass.Create(fHeader.FieldDef[Field.FieldNo
    - 1].FieldSize)
  else
    Translator := FieldClass.Create;

  with Translator do
  try
    ReadForeignBuffer(Ptr);
    WriteDelphiBuffer(Buffer);
  finally
    Free;
  end;
  ...

```

As you can see, there is no convoluted case statement; the only choice we have is whether we call the parameterised version of the constructor needed for determining the length of a string field. The lack of complexity relies on the polymorphic nature of the class hierarchy designed around the base TBufferTranslator class:

```

TBufferTranslator = class
private
  fDelphiBuffer: PChar;
  fDelphiBufLen: Integer;
  fForeignBuffer: PChar;
  fForeignBufLen: Integer;

```

These private variables are kept hidden, even from inheriting classes. As well as providing a utility method for reversing the bytes in a field buffer, properties in the protected section ensure that these variables are only accessed through approved means; also, although the protected properties only read and write directly from the private fields, the fact that they are in the protected section means that we can override them at some later stage, if we want to change the method of access:

```

protected
  procedure ReverseBytes(Source, Dest:
    PChar); virtual;
  property DelphiBuffer: PChar
    read fDelphiBuffer;
  property DelphiBufLen: Integer
    read fDelphiBufLen
    write fDelphiBufLen;
  property ForeignBuffer: PChar
    read fForeignBuffer;
  property ForeignBufLen: Integer
    read fForeignBufLen
    write fForeignBufLen;

```

Next we move on to the public section of the class. These methods are the only way we will have of using objects of this class, whatever their eventual derived type; this will ensure the simplicity of code demonstrated in the GetFieldData method:

```

public
  constructor Create; overload; virtual;
  constructor Create(ByteCount: Integer);
    overload; virtual;
  destructor Destroy; override;
  procedure ReadDelphiBuffer(Buffer:
    PChar); virtual;
  procedure ReadForeignBuffer(Buffer:
    PChar); virtual;
  procedure WriteDelphiBuffer(Buffer:
    PChar); virtual;
  procedure WriteForeignBuffer(Buffer:
    PChar); virtual;
end;

TBufferTranslatorClass = class of
  TBufferTranslator;

```

After defining the class TBufferTranslator, we must also define a type which is capable of holding the type of the object instantiated from a class derived from the TBufferTranslator class. Now we move on to implementing this class, not forgetting that several different classes will be inheriting from this base class:

```

constructor TBufferTranslator.Create;
begin
  inherited Create;
  fDelphiBuffer :=
    AllocMem(fDelphiBufLen);
  fForeignBuffer :=
    AllocMem(fForeignBufLen);
end;

constructor TBufferTranslator.Create(ByteCount:
  Integer);
begin
  Inc(fDelphiBufLen, ByteCount);
  Inc(fForeignBufLen, ByteCount);
  Create;
end;

destructor TBufferTranslator.Destroy;
begin
  FreeMem(fDelphiBuffer);
  FreeMem(fForeignBuffer);
  inherited Destroy;
end;

```

Look carefully at the second constructor; you will notice that the standard constructor is actually called after the first two statements. This is to allow the internal storage allocated in Create to be set to an appropriate value for such field types as string or blob. Of course we must not forget to free any memory allocated.

```

procedure TBufferTranslator.ReverseBytes(Source,
  Dest: PChar);
var
  i: Integer;
begin
  if fForeignBufLen = fDelphiBufLen then
    for i := 0 to Pred(fForeignBufLen) do
      Dest[Pred(fForeignBufLen) - i] :=
        Source[i];
end;

```

ReverseBytes is a simple utility routine that reverses the bytes from one internal buffer into the other. (who said this OO stuff was complicated?)

```

procedure TBufferTranslator.ReadDelphiBuffer(Buffer:
  PChar);
begin
  Move(Buffer^, fDelphiBuffer^, fDelphiBufLen);
  Move(fDelphiBuffer^, fForeignBuffer^,
    fForeignBufLen);
end;

```

```

procedure TBufferTranslator.WriteDelphiBuffer(Buffer:
  PChar);
begin
  Move(fDelphiBuffer^, Buffer^,
    fDelphiBufLen);
end;

procedure
  TBufferTranslator.ReadForeignBuffer(Buffer:
  PChar);
begin
  Move(Buffer^, fForeignBuffer^,
    fForeignBufLen);
  Move(fForeignBuffer^, fDelphiBuffer^,
    fDelphiBufLen);
end;

procedure
  TBufferTranslator.WriteForeignBuffer(Buffer:
  PChar);
begin
  Move(fForeignBuffer^, Buffer^,
    fForeignBufLen);
end;

```

These read/write methods are used for copying data from a buffer into or out of the internal storage; you will notice that as the data is read in, it is subsequently also copied to the internal buffer of the opposite data format. This is a default behaviour and may well be overridden in derived classes, the like of which we will discuss next.

Son of Incomprehensible Drug Addicted Parrot

This section will not be a comprehensive description of a complete hierarchy of translator classes, but rather, I will cover one or two data conversions that are worthy of special note.

Let's look at a simple example of a data type that needs virtually no additional code to convert it to Delphi format; we will assume that the bytes in the buffer are in the same order as Delphi would require them:

```

TOurInteger2 = class(TBufferTranslator)
public
  constructor Create; override;
end;

```

The only extra code required in a case like this is to define the size of the internal storage buffers:

```

constructor TOurInteger2.Create;
begin
  DelphiBufLen := 2;
  ForeignBufLen := 2;
  inherited Create;
end;

```

However, in the case of a string, we have to take into account that strings are not of a standard length and also that Delphi requires a null terminator, whereas many other data files do not account for a terminator byte in their storage:

```
TOurString = class(TBufferTranslator)
public
    constructor Create(ByteCount: Integer);
        override;
    procedure ReadDelphiBuffer(Buffer:
        PChar); override;
    procedure ReadForeignBuffer(Buffer:
        PChar); override;
end;
```

In this case we make use of the parameterised version of the constructor to pass in the length of the string field that is to be translated. The constructor increments the initial size of the Delphi buffer by the one byte necessary for the null terminator before calling the constructor in TBufferTranslator:

```
constructor TOurString.Create(ByteCount:
    Integer);
begin
    DelphiBufLen := 1;
    inherited Create(ByteCount);
end;

procedure TOurString.ReadDelphiBuffer(Buffer:
    PChar);
begin
    Move(Buffer^, DelphiBuffer^,
        DelphiBufLen);
    Move(DelphiBuffer^, ForeignBuffer^,
        StrLen(DelphiBuffer));
end;

procedure TOurString.ReadForeignBuffer(Buffer:
    PChar);
begin
    Move(Buffer^, ForeignBuffer^,
        ForeignBufLen);
    Move(ForeignBuffer^, DelphiBuffer^,
        ForeignBufLen);
    DelphiBuffer[ForeignBufLen] := #0;
end;
```

When reading in the Delphi buffer, StrLen ensures that we do not copy the null terminator over to the shorter foreign buffer. When reading in the foreign buffer we copy the buffer over to the Delphi buffer and then append a null byte.

Boolean values are often stored in only one byte and therefore require special handling; see the following code:

```
TOurBoolean = class(TBufferTranslator)
public
    constructor Create; override;
    procedure ReadDelphiBuffer(Buffer:
        PChar); override;
    procedure ReadForeignBuffer(Buffer:
        PChar); override;
end;
```

```
constructor TOurBoolean.Create;
begin
    DelphiBufLen := 2;
    ForeignBufLen := 1;
    inherited Create;
end;

procedure TOurBoolean.ReadDelphiBuffer(Buffer:
    PChar);
begin
    Move(Buffer^, DelphiBuffer^,
        DelphiBufLen);
    if WordBool(DelphiBuffer^) then
        ForeignBuffer^ := #1
    else
        ForeignBuffer^ := #0;
end;

procedure TOurBoolean.ReadForeignBuffer(Buffer:
    PChar);
begin
    Move(Buffer^, ForeignBuffer^,
        ForeignBufLen);
    if Boolean(ForeignBuffer^) then
        FillChar(DelphiBuffer^, DelphiBufLen, $FF)
    else
        FillChar(DelphiBuffer^, DelphiBufLen, $0);
end;
```

Real numbers can cause major headaches, especially if stored in BCD format; I will not go into detail here, but I found it was possible to define a TOurReal class to perform the common translation, whilst using the constructors of TOurReal4 and TOurReal8 to initialise the buffer sizes.

However due to an e-mail I received asking about Date conversion, I will cover how I cope with one particular conversion here:

```
TOurDate = class(TBufferTranslator)
public
    constructor Create; override;
    procedure ReadDelphiBuffer(Buffer:
        PChar); override;
    procedure ReadForeignBuffer(Buffer:
        PChar); override;
end;

constructor TOurDate.Create;
begin
    DelphiBufLen := 4;
    ForeignBufLen := 2;
    inherited Create;
end;

procedure TOurDate.ReadDelphiBuffer(Buffer:
    PChar);
var
    iDate: SmallInt;
    aDate: array[0..1] of Char absolute iDate;
begin
    Move(Buffer^, DelphiBuffer^,
        DelphiBufLen);
    Move(DelphiBuffer^, aDate,
        ForeignBufLen);
    Dec(iDate, 693595);
    Move(aDate, ForeignBuffer^,
        ForeignBufLen);
end;
```

```

procedure
    TOurDate.ReadForeignBuffer(Buffer:
        PChar);
var
    iDate: LongInt;
    aDate: array[0..3] of Char absolute
        iDate;
begin
    FillChar(aDate, SizeOf(aDate), 0);
    Move(Buffer^, ForeignBuffer^,
        ForeignBufLen);
    Move(ForeignBuffer^, aDate,
        ForeignBufLen);
    Inc(iDate, 693595);
    FillChar(DelphiBuffer^, DelphiBufLen,
        0);
    Move(aDate, DelphiBuffer^,
        SizeOf(aDate));
end;

```

I use the absolute directive to ensure that iDate and aDate both point to the same address in memory; this allows the same functionality that you would get from a union in C++; I can read the variable as either an integer or an array of bytes, assuming that the bytes are in the same order in both formats. The key to successful date conversion is the use of a 'magic' number to bring the date into the correct epoch.

Just In Case

Well all good things must come to an end, and this series is no exception. See how I managed to create easy, readable, polymorphic code with a wave of the hand! And not a case statement to be seen anywhere!... Well, almost; unfortunately you will have to use a case statement in the SetDataType method that is part of the DataType property of the TSchemaFieldDef class.

We should start by defining an enumerated type that is suitable to the needs of the data stored in our foreign data file:

```

type
    TOurFieldType = (ourUnknown, ourString,
        ourInteger2, ourInteger4,
        ourBoolean, ourReal4, ourReal8,
        ourByte, ourDate, ourYear,
        ourHours, ourSecs);

procedure TSchemaFieldDef.SetFieldType(const
    Value: string);
begin
    fFieldSize := 0;

    fOurFieldType := TOurFieldType(<value in
        data header>);
    case Ord(fOurFieldType) of
    ourString:
    begin
        fTranslatorClass := TOurString;
        fFieldType := ftString;
        fFieldSize := StrToInt(<value in data
            header>);
    end;

```

```

ourInteger2:
begin
    fTranslatorClass := TOurInteger2;
    fFieldType := ftSmallInt;
end;
...
ourBoolean:
begin
    fTranslatorClass := TOurBoolean;
    fFieldType := ftBoolean;
end;
...
ourDate:
begin
    fTranslatorClass := TOurDate;
    fFieldType := ftDate;
end;
...

```

Conclusion

In this article we finished implementing the remaining internal methods of TDataset and in so doing looked at the ease of coding the InternalInitFieldDefs method that polymorphism allowed us. We then went on to look at the basis of a polymorphic class hierarchy to facilitate conversion of data from one format to another in the GetFieldData method. Finally we discussed some of the difficulties encountered in converting some types of data from one format to another.

I hope you have found this series of articles informative and useful, I have only touched the surface of what a TDataset can do; maybe one day I will go into how to manage indexes, ranges, filters, etc. If you want to know more, maybe I will see you at DCon 99. Bob Swart, the gauntlet is down! Can you build a better dataset?

Copyright © 1999 Joanna Carter

First published in [UK-BUG](#) News September/October 1999.

Not to be reproduced without consent; please contact [Joanna Carter](#)