MIKE'S Corner

by Mike Orriss

Mike welcomes your feedback and contributions at mjo@compuserve.com The best contribution each month wins £25

Welcome to my new column, a corner for technical gossip, tips, ideas and anything else that I think you might find useful. I hope that what you read will stimulate and/or provoke you into feedback, allowing me to present your views in subsequent issues. Following a bit of Editor arm-twisting, I am able to offer a £25 prize to the best contribution received each month.

As a start, I am interested in Delphi knowledge that took longer to grasp than you would have expected: see the item on indexed properties for one that tripped me up for a long while. Also, am I the only one to have abandoned the `with` statement in Delphi 3, and if I'm not, what do you use instead? I will discuss this in the next issue.

Indexed Properties

Since the advent of Delphi 3 interfaces, I have been making much more use of properties and, in particular, indexed properties. A few weeks ago, a light bulb went off in my head and I suddenly realised that the word 'indexed' had blinded me to the true nature of the beasts: they are better described as 'parameterised properties' as the index or indexes are actually just additional parameters for the `Get/Set` methods driven by the property, they don't have to be indexes in the classical sense at all. Consider the following statements:

```
Type
  TCaptionType =
    (ctNormal, ctAbbrev,
     ctUpper,ctCapital);
  property Caption[
    const value: string;
    opt: TCaptionType]: string
    read GetCaption;
```

Assuming that `Foo` is an instance of the class `TFoo` containing the above property, then you can issue statements like:

```
Panel1.Caption :=
  Foo.Caption[Edit1.Text,
    ctUpper];
item := Sender as TMenuItem;
Edit2.Text :=
  Foo.Caption[item.Caption,
  TCaptionType(item.Tag)];
```

In other words, you can use properties instead of standalone functions and procedures.

I find that bundling similar routines into a class, placed into its own unit, really promotes code re-use. If you instantiate `Foo` within the `Initialization` section (see the `clFoo` unit in Listing 1) then a `Foo` entry in the `Uses` statement is all that is needed in order to use the property. Note that the function `GetCaption` has two parameters that correspond to the two indexes of the property.

Clean Screen Updates

I have finally found a solution to a long-term simple requirement of clean screen updating of `TreeView` controls. I wanted to expand the branches of a `TreeView` as follows: the tree shown closed (using the default cursor), the tree shown closed (with the hourglass cursor) and the tree shown open (using the default cursor), but with no interim screen changes (flickering, full refresh or scrollbar creep)!

I looked at, but discarded, two methods. `LockWindowUpdate(TreeView.Handle)` works but is a sledgehammer to crack a nut: it is too resource intensive, can adversely affect other windows and only supports a single control. Using `TreeView.Items.BeginUpdate` and `EndUpdate` almost works, apart from the dancing vertical scrollbar and an annoying flash caused by an erase and redraw of the entire control...

➤ *Listing 1*

```
unit clFoo;
interface
Uses SysUtils;
Type
  TCaptionType = (ctNormal,ctAbbrev,ctUpper,ctCapital);
  TFoo = class
  protected
    function GetCaption(const value: string; opt: TCaptionType): string;
  public
    property Caption[const value: string; opt: TCaptionType]: string
      read GetCaption;
  end;
var Foo: TFoo;
implementation
function TFoo.GetCaption(const value: string;
  opt: TCaptionType): string;
begin
  if value='' then
    Result := ''
  else begin
    case opt of
       ctNormal: Result := value;
       ctAbbrev: Result := Copy(value,1,8);
        ctUpper: Result := Uppercase(value);
      ctCapital: Result := UpCase(value[1])+Copy(value,2,255);
    end;
  end;
end;
initialization
  Foo := TFoo.Create;
finalization
  Foo.Free;
end.
```

Searching in DTopics I found some code that worked (thanks are due to Jeff Johnson) and I modified it to build the TLockWindow class as a generic solution to manage all screen updating.

Basically, the class holds a TList of locked TWinControls and all Lock and Unlock calls are reference counted. The screen cursor is saved when the first lock is made and reset when the TList is emptied. Note that attempting to unlock a control not held in the TList does nothing (so that calling UnlockAll does not cause logic problems). TLockWindow has just three methods: see Listing 2. As the class manages the cursor, ctl=nil is permitted and this will just reference count changes between crHourGlass and the cursor value at the time of the first lock.

In order to expand a TreeView (assuming that lckwin is an instance of TLockWindow) you'd call:

```
lckwin.Lock(TreeView1.Parent);
try
  TreeView1.Items.GetFirstNode.Expand(
    True);
Finally
  lckwin.Unlock(TreeView1.Parent);
End;
```

Note that I call the TreeView's parent in order to immobilise that pesky scrollbar!

A Difference Engine

Check out the file DIFFM133.ZIP on this month's diskette (in directory DIFFM133) which contains DIFF.EXE and its Delphi source code. The program takes two versions of a file, let's call them FileA and FileB, and generates a difference file FileAB. Subsequently, you can regenerate FileB using FileA and FileAB. Data integrity is checked using a CRC32 algorithm, optimised for speed and designed for working with huge files (50Mb to100Mb isn't a problem).

This program intrigued me. The source code takes a bit of understanding (there are virtually no comments) but the method is quite elegant. The difference file

➤ *Listing 2*

```
unit clLockWindow;
interface
uses Windows,Classes,Forms,Controls,Messages;
type
  TLockedItem = Class
  private
    fControl:  TWinControl;
    fHandle:   THandle;
    fRefCount: integer;
  protected
    procedure SetLock; virtual;
    procedure ResetLock; virtual;
  public
    constructor Create(AControl: TWinControl);
    destructor  Destroy; override;
    property Control: TWinControl read fControl;
    property RefCount: integer
      read fRefCount write fRefCount;
  end;
  TLockWindow = Class
  private
    fList: TList;
    fCursor: TCursor;
  protected
    procedure DeleteLock(index: integer); virtual;
    procedure SetCursor; virtual;
    procedure ResetCursor; virtual;
    function UpdateCount(index,delta: integer): Integer;
      virtual;
  public
    constructor Create; virtual;
    destructor Destroy; override;
    function  IndexOf(Item: Pointer): Integer; virtual;
    procedure Lock(AControl: TWinControl); virtual;
    procedure Unlock(AControl: TWinControl); virtual;
    procedure UnlockAll; virtual;
  end;
implementation
constructor TLockedItem.Create(AControl: TWinControl);
begin
  inherited Create;
  fControl := AControl;
  if fControl=nil then fHandle := 0
  else fHandle := fControl.Handle;
  SetLock;
end;
destructor TLockedItem.Destroy;
begin
  ResetLock;
  inherited Destroy;
end;
procedure TLockedItem.SetLock;
begin
  if fHandle <> 0 then
    SendMessage(fHandle,WM_SETREDRAW,0,0);
  fRefCount := 1;
end;
procedure TLockedItem.ResetLock;
begin
  if fHandle <> 0 then begin
    SendMessage(fHandle,WM_SETREDRAW,1,0);
    RedrawWindow(fHandle,nil,0,RDW_FRAME+RDW_INVALIDATE
      +RDW_ALLCHILDREN+RDW_NOINTERNALPAINT);
  end;
end;
constructor TLockWindow.Create;
begin
  inherited Create;
  fList := TList.Create;
end;
destructor TLockWindow.Destroy;
begin
  UnlockAll;
  fList.Free;
  inherited Destroy;
end;
procedure TLockWindow.SetCursor;
begin
  fCursor := Screen.Cursor;
  Screen.Cursor := crHourGlass;
end;
procedure TLockWindow.ResetCursor;
begin
  Screen.Cursor := fCursor;
end;
function TLockWindow.UpdateCount(index,delta: integer):
  Integer;
var item: TLockedItem;
begin
  item := TLockedItem(fList[index]);
  item.RefCount := item.RefCount+delta;
  Result := item.RefCount;
end;
function TLockWindow.IndexOf(Item: Pointer): Integer;
begin
  for Result := 0 to fList.Count-1 do
    if TLockedItem(fList[Result]).Control=Item then exit;
  Result := -1;
end;
procedure TLockWindow.Lock(AControl: TWinControl);
var ix: integer;
begin
  if fList.Count=0 then SetCursor;
  ix := IndexOf(AControl);
  if ix < 0 then
    fList.Add(TLockedItem.Create(AControl))
  else
    UpdateCount(ix,1);
end;
procedure TLockWindow.DeleteLock(index: integer);
begin
  TLockedItem(fList[index]).Free;
  fList.Delete(index);
  if fList.Count=0 then ResetCursor;
end;
procedure TLockWindow.Unlock(AControl: TWinControl);
var ix: integer;
begin
  ix := IndexOf(AControl);
  if (0 <= ix) and (UpdateCount(ix,-1) < 1) then
    DeleteLock(ix);
end;
procedure TLockWindow.UnlockAll;
var ix: integer;
begin
  for ix := fList.Count-1 downto 0 do DeleteLock(ix);
end;
end.
```

consists of records containing a mixture of new data from FileB and references to position/length of data held in FileA.

In order to create FileAB, FileA and FileB are processed via 2Mb buffers. For each buffer, a hash table is first built from FileA. For each byte, a hash value is calculated from it and the two following bytes. If the hash value differs from the previous byte, an entry is stored in the hash table, chaining any previous values via a hash list. So, for any set of three bytes in FileB all possible matching positions in FileA are available via the hash value of the first byte and the hash list. The program has a compression factor parameter that varies from 10 to 1000 and this governs how many entries are checked for each byte in the hash list. For a given byte in FileB, its hash value is calculated, and for each hash list entry checked an assembler compare function returns the number of matching bytes. The position matching the largest number of bytes is used to write an output reference, unless the match is less than 20 bytes when the FileB data is written as is.

The regeneration of FileB from FileA and FileAB is very simple in comparison, just a question of rebuilding via the FileAB records.

I was interested in the compression capability and speed, particularly when I realised the internal methods were stream-based and thus very suitable for using within applications on the fly: a nice way to hold previous versions of files. Table 1 shows the results of my tests. PKZIP will compress difference files and so my percent saving is based on the comparison of the *zipped* sizes of FileAB and FileB.

E1 and E2 are two versions of a Delphi application, with a suitable number of changes to represent a Version 0.01 upgrade. E-10 through E-1000 show the difference made by using different compression factors. Ship out E1 to your paid customers and subsequently put E-25 on the web: it's no use unless you possess E1 and the CRC checking makes it absolutely safe. Using an installation utility (like Wise) that supports post file processing, you could hide the use of DIFF.EXE and make it all user-transparent.

P1 and P2 are two versions of a cumulative pricing list in mainframe 132 byte per line format, hence the incredible Zip compression. DIFF.EXE does not perform well on this type of file. S1 and S2 are two Delphi .PAS files, quite a few small changes apart. Here DIFF really comes into its own and I was really surprised by the speed.

I also tried DIFF on the NTR file created by my forthcoming `NoteTree` application. The file uses OLE structured storage and stores RTF files compressed with the `zlib` routines (in the info\extras\zlib folder on your Delphi 3 CD-ROM). I was very surprised to find a 90% saving, compared with less than 10% using zip: there is no longer any excuse and I must support saving multiple versions of files!

Mike Orriss runs 3K Computer Consultancy in Staffordshire, UK.

➤ *Table 1*

	Size	Zipped Size	Time taken (seconds)	Size Saved %
E1	648K	230K		
E2	656K	234K		
E-10	281K	138K	3	41
E-25 (default)	269K	135K	4	42
E-100	258K	131K	8	44
E-1000	251K	129K	33	45
P1	15.2M	151K		
P2	15.6M	285K		
P-10	6.1M	269K	12	6
P-25 (default)	6.1M	263K	16	8
P-100	6.0M	262K	20	8
P-1000	6.0M	255K	75	10
S1	52622	9068		
S2	53004	9157		
S-10	537	476	< 1	95
S-25	518	466	< 1	95
S-100	487	456	< 1	95
S-1000	483	455	< 1	95