Tips & Tricks

This is ***your*** column! Here is your opportunity to share with your fellow Delphi enthusiasts those hard-won hints and helps that make your life easier day by day. We have a similar column in our sister publication, The Pascal Magazine, and it is one which readers constantly comment on as being especially useful. Please do send in your Tips & Tricks to us (preferably by email), whether large or small, on any aspect of Delphi or related issues.

Editor Shortcuts

This one's only new to those of us who haven't scoured the Help pages with a fine tooth comb. I'm sure all those Delphi users who can't look at the screen at the same time as typing know that feeling of horror when they realise they've typed a large section of code with Caps Lock on. The case of every letter for the last few lines is wrong, and it all needs to be deleted and re-typed...

Well, no longer! Delphi to the rescue as usual. The Delphi editor is very flexible and has many keyboard combinations that do useful things. In this case, there is a keyboard shortcut for toggling the case of a marked block: `Ctrl-O, U` (or `Ctrl-O, Ctrl-U`). This, and many other shortcuts that can be used in the editor as well as in the form designer and the Object Inspector (although the latter aren't documented very accurately), can be found in the large number of topics that show up if you choose `Help | Topic Search`, type in "shortcuts" and press `Enter`. One or two other invaluable keyboard commands are column block marking (`Ctrl-O, C`) and insert compiler options at top of source file (`Ctrl-O, O`).

Contributed by Brian Long (whose email address is 76004.3437@compuserve.com)

Nick's Favourite Tips

Customize your toolbar. Expand the toolbar by placing the cursor over its right edge and dragging it open. Right click on the toolbar and select `Configure...` Drag and drop all the buttons you want onto the toolbar.

Quicker loading. If you don't plan on using OLE, save a lot of time loading up Delphi by removing the OLE 2.0 container from your component palette. When the `TOLEComponent` is installed and Delphi starts up, it must 'check in' with the OLE DLLs, slowing down start up. Go to `Options|Install Components` on the menu bar. Select `OLEReg` from the left listbox, and then `TOleContainer` from the listbox on the right. Press the `Remove` button and then `Ok`. Delphi will recompile the COMPLIB.DCL file and the OLE container will no longer be installed. Of course, you can reverse the procedure to reinstall it should you ever want the OLE component back on your palette.

Shrink the size of your executables! Open up `Options|Project` from the menu bar. Select the `Linker` page. Check the `Optimize for Size and Load Time` Checkbox. This will shrink your .EXE size by as much as 25%. If your system has a hard time with this feature, as some do, use the command line utility W8LOSS.EXE, found in the \DELPHI\BIN directory. I recommend only doing this for a final version of your applications, as it does add a few seconds to the compile time.

Use the cool routines in the FMXUTILS.PAS file in directory \DELPHI\DEMOS\DOC\FILMANEX. There are some very handy routines to copy, move and execute files, among others.

Add the demo program Image Viewer to your tools palette. Open the file IMAGVIEW.DPR in the directory \DELPHI\DEMOS\IMAGVIEW and compile it. Use the `Options|Menu` dialog box to add it tool your tool bar. Image View will allow you to quickly try out `BitButtons` and `SpeedButtons`, the large bitmap gallery that comes with Delphi in \DELPHI\IMAGES\BUTTONS.

Contributed by Nick Hodges (whose email address is: 71563.2250@compuserve.com)

Passing Variable Numbers Of Parameters

A couple of neat Delphi features now make it possible to write procedures and functions that pass a variable number of parameters (just like `WriteLn` has been able to do in Pascal for ages). The first thing which makes this possible is Delphi's support for passing open arrays – ie arrays where the number of elements in the array is not defined in the procedure or function declaration, eg:

```
Procedure TestMultiPar(
  const Args: array of const);
```

The second ingredient is the `TVarRec` type defined in the `System` unit (see Listing 1) and the fact that `array of const` is treated by the compiler just like `array of TVarRec`. When we use this in passing parameters to a procedure or function, eg:

```
Procedure TestMultiPar(
  const Args: array of const);
```

the compiler looks at the parameters and builds the array directly on the stack. For each item in the array it also sets the `VType` field to one of the pre-defined constants `vtXXXX` (see Listing 1). The actual value is always sent as four bytes of information. For the `Boolean` and `Char` types, only the first byte contains useful information.

```
const
  vtInteger  = 0;
  vtBoolean  = 1;
  vtChar     = 2;
  vtExtended = 3;
  vtString   = 4;
  vtPointer  = 5;
  vtPChar    = 6;
  vtObject   = 7;
  vtClass    = 8;
type
  TVarRec = record
    case Integer of
      vtInteger:  (VInteger: Longint; VType: Byte);
      vtBoolean:  (VBoolean: Boolean);
      vtChar:     (VChar: Char);
      vtExtended: (VExtended: PExtended);
      vtString:   (VString: PString);
      vtPointer:  (VPointer: Pointer);
      vtPChar:    (VPChar: PChar);
      vtObject:   (VObject: TObject);
      vtClass:    (VClass: TClass);
  end;
```

➤ *Listing 1*

```
program VarPar;
uses WinCrt, SysUtils;
const
  TypeNames : array [vtInteger..vtClass] of PChar  =
    ('Integer', 'Boolean', 'Char', 'Extended',
     'String', 'Pointer', 'PChar', 'Object', 'Class');
function PtrToHex(P: pointer): string;
begin
  Result :=
    IntToHex(Seg(P^), 4) + ':' + IntToHex(Ofs(P^), 4);
end;
procedure TestMultiPar(const Args: array of const);
var
  ArgsTyped : array [0..$fff0 div sizeof(TVarRec)] of
    TVarRec absolute Args;
  i : integer;
begin
  for i := Low(Args) to High(Args) do
    with ArgsTyped[i] do begin
      Write('Args[', i, '] : ',
        TypeNames[VType], ' = ');
      case VType of
        vtInteger:  writeln(VInteger);
        vtBoolean:  writeln(VBoolean);
        vtChar:     writeln(VChar);
        vtExtended: writeln(VExtended^:0:4);
        vtString:   writeln(VString^);
        vtPointer:  writeln(PtrToHex(VPointer));
        vtPChar:    writeln(VPChar);
        vtObject:
          writeln(PtrToHex(Pointer(VObject)));
        vtClass:
          writeln(PtrToHex(Pointer(VClass)));
      end;
    end;
end;
var
  MyObj : TObject;
begin
  MyObj := TObject.Create;
  TestMultiPar([123, 45.67, PChar('ASCIIZ'),
    'Hello, world!', true, 'X', @ShortDayNames,
    TObject, MyObj]);
  MyObj.Free;
  { To verify that the type-safety is used try this: }
  writeln(Format('%d', ['hi']));
  { The supplied parameter is not of the type expected.
    The '%d' format string signals that the parameter
    should be an integer value, but instead we send a
    string. At run-time this will generate an exception
    and if you have enabled IDE-trapping of exceptions,
    Delphi will show you the offending line. Using c-type
    sprintf functions like this will result in undefined
    behaviour (read: system crash, GP or whatever) }
end.
```

➤ *Listing 2*

So, go ahead, now you can write all those neat routines with variable numbers of parameters – and still retain type safety! Listing 2 contains a simple example program to demonstrate the technique.

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

Exporting Classes From DLLs

By fooling around with the compiler and some ideas from the Borland C++ manuals, I discovered a way to export classes from a DLL. This is, as far as I know, not documented by Borland anywhere.

The new Delphi classes are very similar to C++ classes. Microsoft's COM and OLE2 technology is very dependent on this class model; what MS calls an interface in OLE2 is really a pointer to a Delphi-style VMT!

I was also inspired by some interface code that can be found in DELPHI\DOC\OLE2.INT. Look at these funny declarations:

```
{ IUnknown Interface }
IUnknown = class
  public
    function QueryInterface(iid: REFIID;
      var Obj: Pointer): HResult; virtual;
      cdecl; export; abstract;
    function AddRef: Longint; virtual;
      cdecl; export; abstract;
    function Release: Longint; virtual;
      cdecl; export; abstract;
end;
```

Have you ever seen methods being declared as `virtual; cdecl; export; abstract;` before? Well no, neither had I. All the class declarations in OLE2.INT have a purely abstract interface. In addition the methods use the 'C' calling convention and include special prolog code to allow the methods to be called from another executable, ie from a running EXE program to a dynamically loaded DLL.

With this in mind and after some experimentation, I managed to export a class from a DLL and use it in a program.

The unit `MyObj` (Listing 3) supports three different compilations. If `EXPORT` is defined, it declares and implements the class and function for export from a DLL. If `IMPORT` is defined it only declares the class and imports the function from the `ObjDll` DLL (shown in Listing 4). Otherwise, it compiles all code and links it directly into the executable.

Listing 5 shows a small test program. When compiling the DLL (Listing 4), define `EXPORT` and rebuild the project. When compiling the test program (Listing 5), define `IMPORT` and then rebuild the project.

There are some basic rules that must be followed when exporting classes from a DLL. All methods to be used *must* be virtual (or dynamic). When declaring variables of the class, use the declared type. When

```
unit MyObj;
interface
{$IFNDEF EXPORT}
  {$IFNDEF IMPORT}
    {$DEFINE NORMAL}
  {$ENDIF}
{$ENDIF}
type
  TMyObject = class(TObject)
  private
    FProp : integer;
    procedure SetProp(Value: integer); virtual;
      {$IFNDEF NORMAL} export;   {$ENDIF}
      {$IFDEF  IMPORT} abstract; {$ENDIF}
  public
    constructor Create; virtual;
      {$IFNDEF NORMAL} export;   {$ENDIF}
      {$IFDEF  IMPORT} abstract; {$ENDIF}
    destructor Destroy; virtual;
      {$IFNDEF NORMAL} export;   {$ENDIF}
      {$IFDEF  IMPORT} abstract; {$ENDIF}
    procedure  Free; virtual;
      {$IFNDEF NORMAL} export;   {$ENDIF}
      {$IFDEF  IMPORT} abstract; {$ENDIF}
    procedure TestIt; virtual;
      {$IFNDEF NORMAL} export;   {$ENDIF}
      {$IFDEF  IMPORT} abstract; {$ENDIF}
    property Prop: integer read FProp write SetProp;
  end;
  TMyObjectClass = class of TMyObject;
function _TMyObject: TMyObjectClass;
  {$IFDEF EXPORT} export; {$ENDIF}
implementation
uses WinProcs;
```

```
{$IFNDEF IMPORT}
constructor TMyObject.Create;
begin
  inherited Create; { Call non-virtual TObject.Create }
  FProp := 1;
end;
destructor TMyObject.Destroy;
begin
  inherited Destroy; {Call virtual destructor TObject.Destroy }
end;
procedure  TMyObject.Free;
begin
  { if Self = nil, then we wouldn't be here in the first place }
  Destroy;
end;
procedure TMyObject.TestIt;
var i : integer;
begin
  for i := 1 to FProp do WinProcs.MessageBeep(0);
end;
procedure TMyObject.SetProp(Value: integer);
begin
  if Value > 10 then FProp := 10
  else FProp := Value;
end;
{$ENDIF}
function _TMyObject: TMyObjectClass;
{$IFDEF IMPORT}
  external 'OBJDLL' index 1;
{$ELSE}
begin
  Result := TMyObject;
end;
{$ENDIF}
end.
```

➤ *Listing 3*

creating objects or otherwise using class references, use the function to get the type used in the DLL.

Contributed by Hallvard Vassbotn.

```
library ObjDll;
uses MyObj, WinProcs;
exports
  _TMyObject index 1;
begin
end.
```

➤ *Listing 4*

```
program TestObj;
uses
  MyObj, WinCrt;
var
  T : TMyObject;
  P : integer;
begin
  T := _TMyObject.Create;
  try
    repeat
      writeln('Enter a value for property: ');
      readln(P);
      T.Prop := P;
      T.TestIt;
    until P = 0;
  finally
    T.Free;
    DoneWinCrt;
  end;
end.
```

➤ *Listing 5*

```
unit DbGridFx;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Grids, DBGrids;
type
  TDBGridFix = class(TDBGrid)
  private
    { Private declarations }
    procedure SetFixedFixedColor(Value: TColor);
    function  GetFixedFixedColor: TColor;
  published
    { Published declarations }
    property FixedColor: TColor read GetFixedFixedColor
      write SetFixedFixedColor default clBtnFace;
  end;
procedure Register;
implementation
procedure TDBGridFix.SetFixedFixedColor(Value: TColor);
begin
  inherited TitleColor := Value;
  inherited FixedColor := Value;
    { Not really needed, just to be on the safe side }
end;
function TDBGridFix.GetFixedFixedColor: TColor;
begin
  Result := inherited TitleColor;
end;
procedure Register;
begin
  RegisterComponents('Data Access', [TDBGridFix]);
end;
end.
```

➤ *Listing 6*

Delphi Compiler 'Features'

The Delphi compiler seems to be a bit picky about the order of units in the `uses` statement. There have been many people in the `comp.lang.pascal` newsgroup on the Internet reporting that they get compile errors such as:

```
WinCrt, MyObj;
              ^Error 85: ";" expected.
```

With some experimentation I've found that moving `WinCrt` so it becomes the last unit in the list solves the problem. The reason seems to be that `WinCrt` checks for the presence of VCL by checking some signature bytes and using a Windows handle if it finds it. The signature and data are stored at fixed addresses in the beginning of `DSeg`. I think that Borland hard-coded the compiler to force the `WinCrt` unit to be loaded after any other units to make sure the signature is written by the VCL (`Controls`, actually) prior to being checked by `WinCrt`.

Another 'bug' is that although the documentation (the *Delphi Object Language Guide* downloaded from Borland's FTP site) states that published properties cannot be of type `real` (Pascal-specific floating point type), the compiler doesn't complain and compiles it fine. However, if you try to use the Object Inspector on such a component, it will generally give you a GPF.

Contributed by Hallvard Vassbotn

TDBGrid Bug: FixedColor Is Ignored

There is a rather obvious bug in the `TDBGrid` data aware grid component: the `FixedColor` property does not work. In `TStringGrid` and `TDrawGrid` it works as documented, but in `TDBGrid` it is non-functional.

Looking through the VCL source we found the reason: `TCustomGrid` defines the `FixedColor` property and assumes that any code which draws the fixed areas will use this property. `TDrawGrid` and `TStringGrid` both do this.

However, `TCustomDBGrid` (from which `TDBGrid` is descended) defines another property called `TitleColor` and uses this when drawing. Unfortunately this property is defined as `protected` making it unavailable for component users at both design-time and run-time.

The solution is to derive your own component and use this instead of `TDBGrid` – this is shown in Listing 6. *[If you're going to make this change, see also Brian Long's fix for scroll bars in* `TDBGrid` *on page 35, which you may as well do at the same time! Looks like Borland finished this component in a bit of a hurry... Editor]*

Another bug is that the color for the lines in the grid is hard-coded to `clBtnHighlight` in `TDBGrid`, but `TDrawGrid` and `TStringGrid` uses `FixedColor` for these as well. There doesn't seem to be an easy way to solve this, other than by changing the VCL source code.

Contributed by Hallvard Vassbotn.

A Windows Desktop Canvas

Listing 7 shows the implementation of a canvas object for the Windows desktop area. This can be useful for screen-savers, splash-screens and programs that need access to the desktop such as magnifier tools. On the desktop canvas you will also see all the open windows, not just wallpaper bitmap! The best of all is that with these few lines of code you can use the complete set of Delphi graphic objects on the desktop (bitmaps, pens, brush...).

Custom Window Frames: If you want customized windows-frames (like the popup toolbars in Microsoft applications) you must draw to the Non-Client area of a window, usually known as the NC. The object shown in Listing 8 gives you a canvas to access the NC area.

Contributed by Stefan Boether, whose email address is: 100023.275@compuserve.com

Using RES Bitmaps

If you've previously used Borland Pascal Windows you will be used to handling bitmaps by creating a bitmap resource and using `LoadBitmap` in your program.

Delphi uses its own resource format for bitmaps, but the object shown in Listing 9, descended from `TBitmap`, will let you use your old `.RES` bitmap resource files but gives you access to all the great Delphi `TBitmap` functions (conversion, file storage etc) too1 It's very short but it works well.

Contributed by Stefan Boether.

```
type
  cDesktopCanvas = class(TCanvas)
  private
    DC           : hDC;
    function     GetWidth:Integer;
    function     GetHeight:Integer;
  public
    constructor  Create;
    destructor   Destroy; override;
  published
    property Width: Integer read GetWidth;
    property Height: Integer read GetHeight;
  end;
function cDesktopCanvas.GetWidth:Integer;
begin
  Result:=GetDeviceCaps(Handle,HORZRES);
end;
function cDesktopCanvas.GetHeight:Integer;
begin
  Result:=GetDeviceCaps(Handle,VERTRES);
end;
constructor cDesktopCanvas.Create;
begin
  inherited Create;
  DC:=GetDC(0);
  Handle:=DC;
end;
destructor cDesktopCanvas.Destroy;
begin
  Handle:=0;
  ReleaseDC(0, DC);
  inherited Destroy;
end;
```

➤ *Listing 7*

```
type
  cNCCanvas = class(TCanvas)
  private
    FDeviceContext: HDC;
    FWindowHandle : HWnd;
    function     GetWindowRect:TRect;
  protected
    procedure    CreateHandle; override;
    procedure    FreeHandle;
  public
    constructor  Create(aWindow: hWnd);
    destructor   Destroy; override;
    property     WindowRect: TRect read GetWindowRect;
  end;
constructor cNCCanvas.Create(aWindow: hWnd);
begin
  inherited Create;
  FWindowHandle:=aWindow;
end;
destructor  cNCCanvas.Destroy;
begin
  FreeHandle;
  inherited Destroy;
end;
procedure cNCCanvas.CreateHandle;
begin
  if FWindowHandle=0 then
   inherited CreateHandle
  else begin
    if FDeviceContext = 0 then
      FDeviceContext := GetWindowDC(FWindowHandle);
    Handle := FDeviceContext;
  end;
end;
procedure cNCCanvas.FreeHandle;
begin
  Handle := 0;
  if FDeviceContext <> 0 then begin
    ReleaseDC(FWindowHandle, FDeviceContext);
    FDeviceContext:=0;
  end;
end;
function cNCCanvas.GetWindowRect:TRect;
begin
  winProcs.GetWindowRect(FWindowHandle,Result);
  with Result do begin
    Right:=Pred(Right-Left);
    Bottom:=Pred(Bottom-Top);
    Left:=0; Top:=0;
  end;
end;
```

➤ *Listing 8*

```
type
  cResBitmap = class(TBitmap)
  public
    constructor Create(aId:PChar);
  end;
constructor cResBitmap.Create(aId: PChar);
begin
  inherited Create;
  Handle:= LoadBitmap(hInstance,aId);
end;
```

➤ *Listing 9*

Smart IDE

As well as my office PC I also use a notebook for Delphi development. I needed a smaller, faster Delphi installation. Often I only test small projects like streams, lists, new components etc. For this I don't need the OLE, DDE, database and VBX stuff.

To save a lot of memory and resources, you can create several different COMPLIB.DCL versions: perhaps one for each topic you address. For example, for testing basic stuff I only need the Standard and Additional pages of the Component Palette, which are registered using LIB\STDREG.PAS – now Delphi uses less than 1Mb of memory and 8% of system resources after startup! With this configuration I can work well with Delphi even on a 486/33 notebook with 8Mb of memory.

Another way to increase compilation speed is to enable Smartdrive's write cache. Before everyone flames that it's not a good idea for development platforms let me say how exactly I do this. I only need this feature for compilation and linking, so before running the program the cache should be committed to the disk (in case the program crashes, in which case and data remaining in the write cache would not be written to disk). Most disk caches act on the disk-reset interrupt, which you can trigger with a smart program or by pressing Ctrl-C in a DOS box. I would encourage Borland to put this call directly into the IDE, because it makes the compiler even faster.

Contributed by Stefan Boether.

Clipboard Object

Devotees of The Pascal Magazine may remember my Windows Clipboard access object from Issue 4. Well, Delphi's clipboard support is quite good, however, it only supports one 255 character string with the `CF_TEXT` format. To overcome this, let me introduce you to the `TMemoClipboard` object, which supports `TStrings` objects for longer text. The object declaration is shown below (a full example is in the MEMCLIP.LZH archive on this issue's free disk, in the TIPTRIX directory):

```
type
  TMemoClipboard = class(TClipBoard)
  private
    function  GetLines: TStrings;
    procedure SetLines(Value: TStrings);
  public
    property  AsLines: TStrings read GetLines
      write SetLines;
  end;
```

To use it you must include the `MemoClip` unit (which is on the disk) in your uses list. It frees the standard global Clipboard variable and replaces it with an instance of `TMemoClipboard`. If you want to access the `AsLines` property you must do a typecast of the clipboard:

```
Memos:= TMemoClipboard(Clipboard).AsLines
```

or

```
with Clipboard as TMemoClipoard do
  Memos:= AsLines;
```

Contributed by Stefan Boether.