The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?
Just email Brian Long, our Delphi Clinic Editor, on 76004.3437@compuserve.com or write/fax us at The Delphi Magazine

What Is The Image Base Option?

Q Should I bother specifying an image base for my Delphi 2 applications and DLLs?

A The `Project | Options | Linker | Image base:` option specifies where in memory Windows will attempt to load the DLL or EXE: it defaults to `$00400000`. Bearing in mind that each Windows process is individually mapped into the same address range, a program will always be safe to load at this address. However once the program has loaded there, any subsequent DLLs cannot, so the image base should be changed. If the default value is used, the DLL takes a bit longer to load as Windows must move it to a different address. The advised range for DLLs is `$40000000` to `$7FFFFFFF`, but the lower four digits are ignored and should be zero.

Each DLL should be given a unique image base address in the context of the application, but you should also take into account other DLLs that will be used, for example the BDE. If you look at all the DLLs supplied with the 32-bit BDE with Windows 95 Quick View, or with TDump, their Image Base values are all different. Borland strive to give unique values for every DLL they manufacture to stop possible performance penalties. Interestingly, the image base for the main BDE DLL, IDAPI32.DLL is `$4BDE0000`.

Floating Point Inaccuracy

Q I use an InterBase table with a `FLOAT` field. When I display the value of this field in a data-aware control the value is more precise than what was stored, but doesn't correspond to the value which should be displayed. For example, a value stored as 0.18 is displayed in my Delphi program as 0.180000007152557.

This seems to be a standard rounding issue. We use digital computers which operate in binary. We humans tend to work in decimal (to do with fingers and thumbs I gather). Given the number of bits a floating point type has (there are several types and their storage sizes vary), certain values can't be represented exactly in base 2. So the bits are set to store a value as close as possible to the value desired. That's why the field type `FLOAT` is described as having only up to seven digits of precision – the number of bits used can only accurately represent numbers with that number of digits in binary.

You can tell the database components to only display a certain precision using the `Precision` property, but if you programmatically compare floating point numbers, you tend to need to use an error factor. Something like:

```
const
  ErrorFactor = 0.000001;
  { varies depending on
    floating point type }
```

Then, instead of

```
if FloatVar = 0.18 then...
```

you use

```
if Abs(FloatVar - 0.18) <
  ErrorFactor then...
```

InterBase BLOb SubTypes And Segments

Q What are InterBase BLOb subtypes and segments and how are they used?

A Rather than launching into a discussion of these myself, I refer you to the *InterBase Programmer's Guide*. You get this with most versions of InterBase (except the local InterBase that comes with Delphi 1). Delphi 2 users can find an online version of this manual by bringing up the *Local InterBase Help* help file. From the *Contents* page, choose *Programmer's Guide*. For information on the topic in question, choose *Working with BLOb Data* and then either *BLOb subtypes* or *BLOb Segment Length*.

Raw Printer Access

Q I used to use the Win16 `Escape` routine with a `PassThrough` or `DeviceData` parameter to send raw data to the printer. The Delphi 2 online help says these parameters are obsolete in Win32. Additionally, the help and declaration for the `SpoolFile` API, used to send a binary printer file directly to a printer, are gone in Delphi 2. What do we use instead?

A The Win32 API provides support for directly writing to a printer which removes problems with these two older approaches. Listing 1 contains a translation of a routine from the MSDN (article Q138594) that sends raw data to a printer. The Win16 problem areas are highlighted and expanded upon in further MSDN articles: Q111010, Q35708, Q139011.

This routine can be called with any variable that holds some appropriate data. As a simple example, to send two form feeds (where a form feed is ASCII character 12) to an HP DeskJet printer, you can use this (remember, the names of all installed printers can be obtained from the

Printer.Printers property, a TStrings object):

```
const
  Data: array[1..2] of Char =
    #12#12;
RawDataToPrinter(
  'HP DeskJet 520 Printer',
  Data, SizeOf(Data));
```

if you wanted to write to the default printer on LPT1: you could use:

```
RawDataToPrinter('LPT1',
  Data, SizeOf(Data));
```

Virtual And Override Clarification

Q Why does Delphi have two keywords for polymorphism (virtual and override), where previous Borland Pascal products and C++ have just one (virtual)?

A Delphi actually has three keywords (virtual, dynamic and override) but let's not complicate matters just yet. Before looking at the question itself, let's have a brief recap on what polymorphism is and how it works. Polymorphism allows a situation where you can call a method of an object, but at run-time the object may be an instance of any of several different classes, all from one branch of your object hierarchy. Because the compiler can't know which class will be present at run-time, it can't calculate the address to branch execution off to. Instead, if told to, it needs to call code to perform some lookup at run-time, and get the appropriate class's code to run.

In C++ and Delphi the way we can introduce a new polymorphic routine into a base class is by adding the word virtual onto its declaration. If we make a new class inherited from the base class and wish to define new functionality for this virtual method, C++ lets us redeclare the method and again use the virtual keyword. In Delphi we must use the override keyword.

There are two distinct operations here: adding a new polymorphic routine and re-declaring an existing polymorphic routine. C++ does both operations with one keyword, Delphi distinguishes between the two with different keywords. Look at Listing 2, from the CARS.DPR project, for a Delphi example of a virtual and overridden routine. This allows code written like this to work:

```
var Cars: array[1..2] of TCar;
...
Cars[1] := TCar.Create;
Cars[2] := TRacingCar.Create;
Cars[1].Drive;
Cars[2].Drive;
Cars[1].Free;
Cars[2].Free;
```

Even though the array is defined to be of two TCar objects, it is quite acceptable to place objects derived from TCar in it. Any TCar or TCar-descendent could be placed in the array at any time during the execution of the program. The compiler has no chance of predicting which object will be there and so has to generate code which will find that out at run-time and act accordingly.

The two calls to the Drive method work because the virtual and override keywords have been used to enable polymorphism to work, but how do they function?

This business of waiting until run-time to find out what to execute is called *late binding*, where the normal compiler job of deciding what needs to execute at compile time is called *early binding*. Late binding is implemented in Delphi and C++ by the use of lookup tables of addresses called virtual method tables (VMTs). Each class has a VMT made by the compiler which is stored on the heap (in the old object model used by Borland Pascal it was stored in the data segment, rather restricting the total number of classes). The VMT includes the addresses of all virtual methods for that class and, when you call a virtual method, the generated code effectively does an indirect jump by jumping to the address stored at the relevant offset in the VMT.

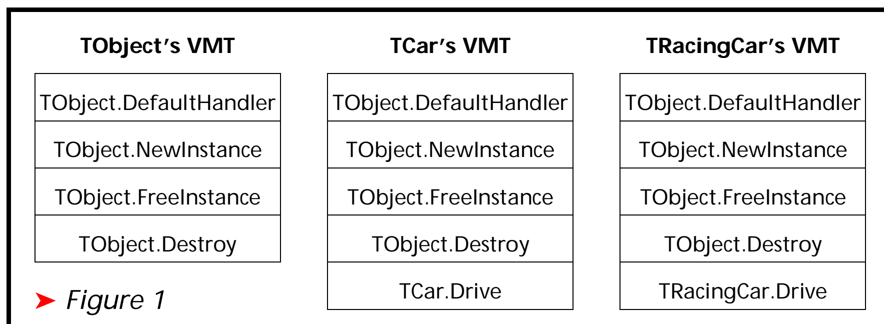
Using the word virtual in Delphi adds a new entry onto the VMT, whereas override changes an entry in the VMT. This differs from C++ and the old Pascal object model where virtual will add a new VMT entry if there is not already one for a method of that name, otherwise virtual changes the relevant VMT entry. Pictorially, the TCar and TRacingCar VMT are shown in Figure 1.

Notice that each class's VMT has all virtual methods for that class and its ancestors. It's worth bearing in mind that had we tried to use virtual in both the ancestor and descendent (as we would in C++) we would not get the desired result. Instead, the VMT would look as in Figure 2. The use of virtual twice has added two new entries, despite the two methods having the same name.

I said at the start that there was a third keyword: dynamic. This can be used instead of virtual and adds an entry to a dynamic method table (DMT). A DMT is much the same as a VMT but for one difference. Where a VMT stores entries for all virtual methods including those inherited from ancestors, a DMT only includes entries for dynamic methods defined in that class. The implication of this is that DMTs are much smaller than VMTs. Delphi message handlers are implemented as dynamic methods. There could be many message handlers in all the ancestors of any given class, but only that class's own message handlers are stored in the DMT to stop it swelling excessively. However, there is another implication for DMTs. In order to work, the code for dynamic methods might need to search the DMT for the given class, and also for many of its ancestors, before finding the relevant entry. It is therefore potentially much slower.

For methods where there is no efficiency requirement, you can make methods dynamic instead of virtual. The override keyword is used when redefining both virtual and dynamic methods.

So, back to the original question. Why the change from the established way of doing things? It was realised that adding the override keyword would provide greater version resiliency in classes. Let's



➤ *Figure 1*

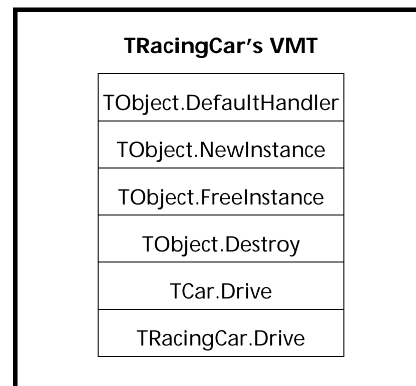
consider a few possible scenarios. Firstly, a user of Delphi 1's VCL creates a descendant of `TEdit` and adds a new method, `Foo`, as `virtual`. In Delphi 2's VCL, Borland adds a `virtual` method to `TEdit`, called `Foo`. Were it not for the `override` syntax, the user's code would break when they changed version, since they would now be redefining a VCL method.

The second, but rather less likely, scenario involves Delphi 1's VCL having a method called `Bar` that is `virtual` and a user `overrides` this method in a descendant class. In Delphi 2's VCL, Borland decides to make that method `dynamic` since it isn't a performance critical section. As in the first scenario, the user's code would break if the `override` syntax weren't there, since they would have to use `virtual` for a re-declared virtual method and `dynamic` for a re-declared dynamic method.

The first scenario is principally the reason `override` was added to the language. Introducing new virtual methods in base classes happens fairly regularly between major product versions. The second scenario is almost incidental as methods are rarely changed from virtual to dynamic. The important point is that the use of `override` takes away the problem of brittle classes, and makes your classes more tolerant of base class changes. This means your component classes are less likely to require modification to work in subsequent versions of Delphi. This also gives Borland, and other authors of widely used base classes, more scope to enhance the base classes over time.

In short, `virtual` or `dynamic` and `override` provide a superior solution to just `virtual` or `dynamic` alone.

Note that the term resiliency is being used here to mean source code resiliency across versions, not DCU binaries. If a change is made to the interface of a base class, all descendent source code must be re-compiled. However, because of the existence of `override`, the descendent source should not necessarily need modifying first.



➤ *Figure 2*

```
TCar = class
public
  {Declare new polymorphic routine}
  procedure Drive; virtual;
end;
TRacingCar = class(TCar)
public
  { Redefine existing polymorphic
    routine }
  procedure Drive; override;
end;
procedure TCar.Drive;
begin
  ShowMessage('Chug, chug, chug');
end;
procedure TRacingCar.Drive;
begin
  ShowMessage('Vroomm, vroommm');
end;
```

➤ *Listing 2*

Acknowledgements

Thanks to Allen Bauer and Danny Thorpe at Borland for some of the details in the `virtual` and `override` answer.

➤ *Listing 1*

```
// RawDataToPrinter - sends binary data directly to a printer
// Params: PrinterName - printer name
//         Data        - raw data bytes, any variable will do
//         Count       - length of Data in bytes
// Gives an exception upon failure
// (with Break on Exception on it may give several)
procedure RawDataToPrinter(PrinterName: String;
   const Data; Count: Integer);
implementation
uses WinSpool, Windows, Consts, SysUtils;
type EPrinterError = class(Exception);
procedure Error;
begin
  raise EPrinterError.CreateRes(SInvalidPrinterOp);
end;
procedure RawDataToPrinter(PrinterName: String;
 const Data; Count: Integer);
type
  TDoc_Info_1 = record
    DocName,
    OutputFile,
    Datatype: PChar;
  end;
var
  hPrinter: THandle;
  DocInfo: TDoc_Info_1;
  BytesWritten: Integer;
begin
  // Need a handle to the printer
  if not OpenPrinter(PChar(PrinterName), hPrinter, nil) then
    Error;
  // Fill in the structure with info about this "document"
  DocInfo.DocName := 'Document';
  DocInfo.OutputFile := nil;
  DocInfo.Datatype := 'RAW';
  try
    // Inform the spooler the document is beginning
    if StartDocPrinter(hPrinter, 1, @DocInfo) = 0 then Error;
    try
      // Start a page
      if not StartPagePrinter(hPrinter) then Error;
      try
        // Send the data to the printer
        if not WritePrinter(hPrinter, @Data, Count,
          BytesWritten) then Error;
      finally
        // End the page
        if not EndPagePrinter(hPrinter) then Error;
      end;
    finally
      // Inform the spooler that the document is ending
      if not EndDocPrinter(hPrinter) then Error;
    end;
  finally
    // Tidy up the printer handle
    ClosePrinter(hPrinter);
  end;
  // Check to see if correct number of bytes written
  if BytesWritten <> Count then Error;
end;
```