Beating The System: Strings Unstrung

by Dave Jewell

Long-term denizens of the Borland conference on CIX will know that there's one subject which keeps cropping up with monotonous regularity, and that's the thorny issue of Delphi's support for long strings. Queries on how the `AnsiString` type is implemented internally, under what circumstances strings are reference counted, whether or not they are more efficient than short strings and so on, and so forth, are favourites.

As my humble contribution to world peace, this month's *Beating The System* is intended to cover the subject one last time for the benefit of those who pursue this particular topic with seemingly relentless enthusiasm. If I can put this issue to bed once and for all then I'll die a happy man.

But First

But first, in this necessarily rather technical discussion, it is important to get our terminology firmly bolted down. To begin with, whenever I say 'string' in this article, I am going to be talking about Delphi's default string type, `AnsiString`, unless otherwise stated. If I am talking about the older `ShortString` type (the classic Pascal string, limited to a maximum of 255 characters in length) then I will specifically say so.

It's also important to carefully distinguish between the *string data* and the corresponding *string variable*. The string data, or the actual text of a string, is *always* allocated on the heap, never on the stack. If we were talking about short strings, then this wouldn't be so, but it is always true for the `AnsiString` type. A string variable, on the other hand, can be allocated anywhere you like:

```
var Str: String;
```

Much of the time, you would place a declaration like this inside a method or procedure. In such cases, the variable is being allocated on the stack, because local variables are always stack-based.

Alternatively, you might place the above declaration outside of any procedure or method. In other words, you could place it in a `var` block following the `interface` or `implementation` part of a unit. In this case, the string variable will be allocated within the application's data segment.

Then again, you will very likely use declarations like the one above to create private string variables within a Delphi class. Because the storage space which is occupied by an object instance is always allocated on the heap, this means that those private string variables will also be allocated on the heap.

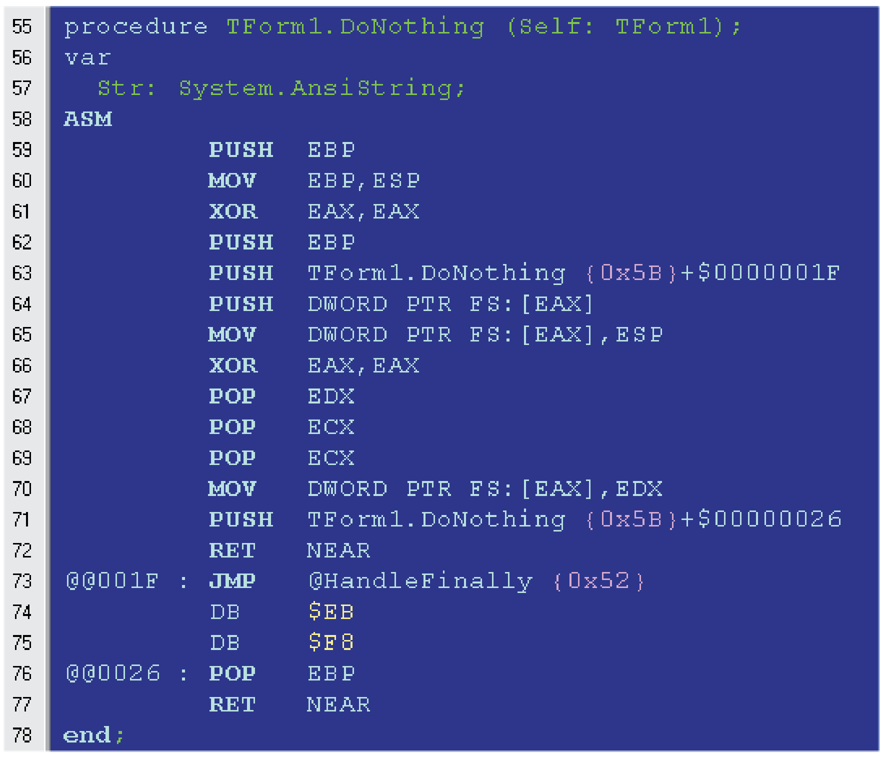
To summarise: string data is only ever allocated on the heap. A string variable itself, however, might be allocated on the stack, in the application data segment, or on the heap: it depends entirely on where it was declared.

Finally, of course, we need to distinguish between the size of the string variable and the size of the string data! The former is always four bytes, corresponding to the 32-bit pointer that references the actual data. Thus `sizeof(String)` will always return `4`. The length of the string is, of course, returned by the `Length` function. Depending on the string in question, this can return anything you like, although negative numbers should perhaps be viewed with some element of suspicion... ☺

I'm sorry if this all seems to be labouring the point, but without a good grasp of the basics of the Delphi string implementation, the deeper issues will always remain a mystery.

Time to move on to more complex matters.

➤ *Figure 1: Even an apparently 'do-nothing' routine can generate a surprising amount of code once you start messing around with string variables. All of the stuff here is an empty try..finally block created by the code generator.*



```
procedure TForm1.DoNothing (Self: TForm1);
var
  Str: System.AnsiString;
ASM
             PUSH   EBP
             MOV    EBP,ESP
             XOR    EAX,EAX
             PUSH   EBP
             PUSH   TForm1.DoNothing {0x5B}+$0000001F
             PUSH   DWORD PTR FS:[EAX]
             MOV    DWORD PTR FS:[EAX],ESP
             XOR    EAX,EAX
             POP    EDX
             POP    ECX
             POP    ECX
             MOV    DWORD PTR FS:[EAX],EDX
             PUSH   TForm1.DoNothing {0x5B}+$00000026
             RET    NEAR
@@001F :     JMP    @HandleFinally {0x52}
             DB     $EB
             DB     $F8
@@0026 :     POP    EBP
             RET    NEAR
end;
```

Hidden String Initialisation And Finalisation

OK, so we understand the difference between a string variable and the string data. The next point to make is the way in which strings are automatically initialised by Delphi. The compiler actually goes to some trouble to ensure that strings are properly initialised before use. As I've already mentioned, a string variable is actually a 32-bit pointer to the actual string data. Borland designed the runtime library's string support so that, by convention, an empty string is represented by a `Nil` pointer. In other words, rather than having to allocate a pointer to a zero byte, as one might expect things to work, a `Nil` string pointer is always interpreted as an empty, zero-length, string.

There are some big advantages to this approach from a runtime perspective. As you probably know, whenever Delphi creates an instance of an object, all the instance data within that object is automatically initialised to zero by the runtime library. In other words, given some object called `TMyObject`, for which the standard `TObject.InstanceSize` function returns 230 bytes, the runtime library will pre-fill all 230 bytes of instance data with zeros. This means that any string variables defined inside a Delphi object are guaranteed to be initialised to empty strings, without having to place any explicit string initialisation code inside the object's constructor.

In the same way, any time you declare a string variable at global scope within a unit, no special code is generated by the Delphi compiler. This is because all global variables are implicitly initialised to zero by the Delphi runtime library. Therefore, globally declared string variables are always initialised to an empty string, unless of course you were to give them an initial value, like this:

```
var Str: String = 'Fred';
```

Similar things can be said about string variables which are declared inside a method. For example, here is an apparently do-nothing method:

```
procedure TForm1.DoNothing;
var
  Str: String;
begin
end;
```

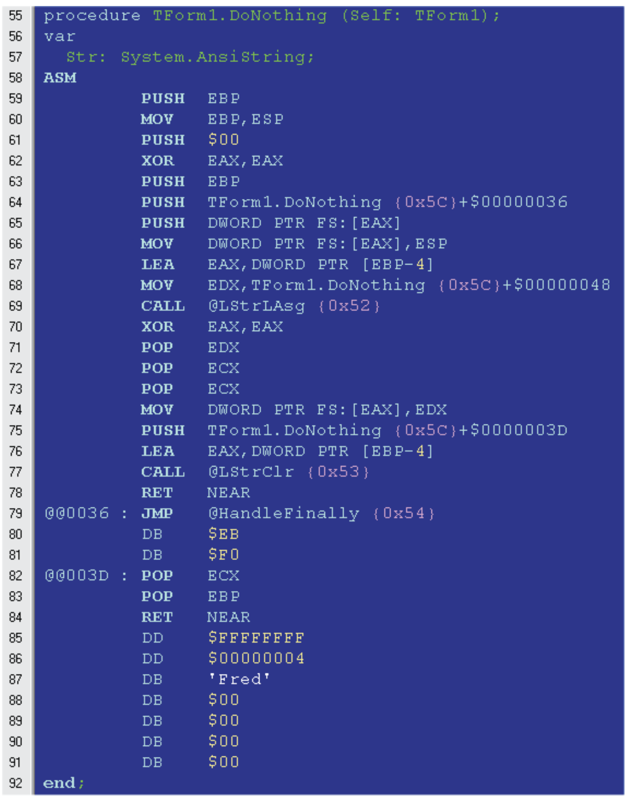
Looks like this doesn't generate any code, right? Surprisingly, `DoNothing` generates a fair amount of code, as you can see from the disassembly in Figure 1. Incidentally, I generated these disassemblies using a program called DCUExplorer, a relative newcomer to the DCU-disassembling scene. You can download DCUExplorer from www.puthoon.com/home.html. It's free, but sadly you don't get the source code. There's no such thing as a free lunch, I guess.

I'm working on the assumption that you're not that familiar with the Delphi code generator. If you were, you'd see that Figure 1 corresponds to an empty `try..finally` block. In a nutshell, whenever you declare a string inside a procedure or method, a `try..finally` block is secretly inserted by the code generator. Ordinarily, the purpose of this would be to ensure that the string is de-allocated on exit from the declaring method, regardless of whether or not that method exits abnormally. However, in this particular case, the string is never used and, if you try compiling the above code, the compiler will rightly complain that you've declared a variable but not used it. The compiler is smart enough to spot this fact, but by the time it does (once it's parsed to the end of the function), the code generator has already been instructed to generate the `try..finally` block. The end result is that we get a spurious `try..finally` where we didn't really need one.

In a more typical case, you would of course make use of the string variable, or at least assign to it. Let's say that you add the following assignment between the `begin` and `end` statements of the `DoNothing` method:

```
Str := 'Fred';
```

The generated code will now look as shown in Figure 2. There are some subtle (in some cases, *very* subtle!) differences between this code and Figure 1. The most subtle change is that `PUSH` statement which you can see has appeared on line 61. This pushes a zero onto the stack. What's the explanation for this?



```
55  procedure TForm1.DoNothing (Self: TForm1);
56  var
57    Str: System.AnsiString;
58  ASM
59              PUSH  EBP
60              MOV   EBP,ESP
61              PUSH  $00
62              XOR   EAX,EAX
63              PUSH  EBP
64              PUSH  TForm1.DoNothing {0x5C}+$00000036
65              PUSH  DWORD PTR FS:[EAX]
66              MOV   DWORD PTR FS:[EAX],ESP
67              LEA   EAX,DWORD PTR [EBP-4]
68              MOV   EDX,TForm1.DoNothing {0x5C}+$00000048
69              CALL  @LStrLAsg {0x52}
70              XOR   EAX,EAX
71              POP   EDX
72              POP   ECX
73              POP   ECX
74              MOV   DWORD PTR FS:[EAX],EDX
75              PUSH  TForm1.DoNothing {0x5C}+$0000003D
76              LEA   EAX,DWORD PTR [EBP-4]
77              CALL  @LStrClr {0x53}
78              RET   NEAR
79  @@0036 : JMP   @HandleFinally {0x54}
80              DB    $EB
81              DB    $F0
82  @@003D : POP   ECX
83              POP   EBP
84              RET   NEAR
85              DD    $FFFFFFFF
86              DD    $00000004
87              DB    'Fred'
88              DB    $00
89              DB    $00
90              DB    $00
91              DB    $00
92  end;
```

➤ *Figure 2: The Delphi code generator uses a very elegant but subtle way of allocating space for and initialising string variables, all with a single machine code instruction. See the PUSH instruction in line 61? There is more there than meets the eye...*

Because we have made an assignment to the string, the compiler decides that the string variable really is being used, so it emits the necessary code to initialise the string. But remember I told you that all that's needed to initialise a string variable is to set it to `Nil`. The code generator tends to initialise the frame pointer (`EBP`) for a method and then to initialise any needed string variables. In this case, by simply assigning to `EBP` and *then* pushing a zero on the stack, it can then reference the new string variable as `[EBP-4]`. In essence, the `PUSH` statement allocates space for the string variable and initialises it to zero at the same time, all with a single machine-code instruction. Sneaky, or what?

If you write a method that declares a large number of string variables, you shouldn't be surprised to find code that starts off something like this:

```
PUSH  EBP
MOV   EBP,ESP
XOR   ECX,ECX
PUSH  ECX
PUSH  ECX
PUSH  ECX
PUSH  ECX
PUSH  ECX
PUSH  ECX
PUSH  ECX
```

This particular routine declares no less than seven strings and, sure enough, we see seven zeros pushed onto the stack, each of which allocates space for and initialises a string variable. In this case, the sneaky code generator has realised that pushing seven immediate values would needlessly waste space, so it stores zero in a single register (`ECX` in this case) and then uses this register as the source for the `PUSH` instructions. With even more strings declared, the code generator uses another space-saving tactic: implementing a hidden loop using `ECX` as the loop variable to push the required number of zeros.

The remainder of Figure 2 is relatively easy to explain. The code at lines 67 to 69 assigns the literal string `Fred` to our local variable, while the code at lines 76 and 77 corresponds to the `finally` clause of the `try..finally` block and is responsible for de-allocating the string.

This brings me neatly onto the subject of finalisation or, as our American cousins would say, finalization. (As I recently moved to North Wales, just think yourselves lucky I'm not using the Welsh translation.) Strings are a special type of Delphi variable because they need to be finalised. Interfaces are another example of finalisable variables, about which more later. Whenever you use a finalisable variable in your code, Delphi transparently takes care of finalising the variable when it goes out of scope.

In essence, the pseudo-code for our do-nothing routine looks something like Listing 1.

```
procedure TForm1.DoNothing;
var
  Str: String;
begin
  Str := '';         ← initialise the string (sets string variable to zero)
  Str := 'Fred';
  try
  finally
    Str := '';       ← finalise the string (calls @LStrClr)
  end;
end;
```

➤ *Listing 1*

Notice that in pseudo-code, string initialisation looks identical to string finalisation: a simple assignment to an empty string. But at the grass roots level, things are somewhat different. As we have already seen, initialising a string is just a question of storing zero into the string variable. Finalising, however, requires that the address of the string variable is passed to an internal routine called `@LStrClr`. You can clearly see this in Figure 2. When de-allocating a string, we can't just set the string variable to zero because this would 'cast adrift' any dynamically allocated string data already on the heap. Instead, `@LStrClr` automatically takes care of de-allocating the string data as well as setting the string variable to zero.

Well, OK, that covers string finalisation in a procedure or method, but what about string finalisation at global scope: how does the Delphi runtime system guarantee finalisation of strings declared at global scope in a unit? By definition, such strings never go out of scope during the execution of a program, but the code generator auto-magically inserts code into the `finalization` clause of any unit which has global strings. Even if you don't declare a `finalization` block yourself, a hidden `finalization` block will take care of any needed string cleanup, calling `@LStrClr` on any global strings. Again, this is very sneaky stuff.

But what about strings that are declared within a Delphi object? How do they get cleaned up? Earlier, I said that because class members are automatically initialised to zero, no special initialisation is needed for strings in a Delphi object. Although this is strictly true, it's only giving you part of the story. What actually happens is that when the Delphi compiler parses a class declaration, it makes a special note of any finalisable items (remember: these are strings and interfaces) that are declared within the class. Using this information, the compiler builds a special table called the init-table, which lists all the finalisable items within the class. For each finalisable item, an init-table entry contains a pointer to the item's type information together with a byte offset into the instance data for the enclosing class.

So, to take a simple example: if you create a Delphi class containing three strings, `Tom`, `Dick` and `Harry`, these strings might be located at offsets `$20`, `$24` and `$28` within the instance data for that class. The init-table will contain three entries, with the type information for all three entries pointing to the `String` type, and the byte offsets being set at `$20`, `$24` and `$28`. Somewhat inappropriately named, the init-table really comes into play when a Delphi object is destroyed, not initialised. At that time, the runtime code works its way through the init-table, finalising any items found there. For strings, of course, this means that our old friend `@LStrClr` will be called.

Having said all that, I should mention plain-vanilla records for the sake of completeness. When you declare a record, the compiler will once again check the fields of the record for finalisable types. If any are present then, as for classes, it builds an init-table. Whenever you use `New` or `Dispose` to create and destroy records that are allocated on the heap, the compiler passes a pointer to the record's RTTI (runtime type information) to the `New` or `Dispose` routine. This allows the runtime library to correctly initialise and finalise strings contained within a record declaration.

Surprise, Surprise

At this point, you should have a pretty solid understanding of the way that Delphi implements the `AnsiString` type, but this is not the whole story by any means. Listing 2 is a surprising little example that shows things aren't always what they seem. Note that thanks are due to Matthew Jones of CIX for posting up this sample code.

What would you expect to be displayed by the `ShowMessage` call? Matthew reckoned that the result should be `Hello`, but what you'll actually get is `HelloHello`.

What exactly is happening here? First and foremost, `MyFunc` is an example of bad code, because it uses the `Result` variable on the right-hand side of an assignment statement before `Result` has been given an initial value, or so it would seem. Based on what we've learned so far, you might think that this is not a problem. After all, `Result` is a `String`, so we know that it will be initialised to an empty string, right? Wrong.

```
function MyFunc: String;
begin
  Result := Result + 'Hello';
end;
procedure MyProc;
var
  szText: String;
begin
  szText := MyFunc;
  szText := MyFunc;
  ShowMessage(szText);
end;
```

➤ *Listing 2*

If `Result` was indeed initialised to zero on entry to the function, then `MyFunc` would always return `Hello` in line with most people's expectations. In practice, this isn't what happens. Why not? The real issue here is one of ownership. Is the `Result` variable owned by the calling routine, `MyProc`, or is it owned by the called routine, `MyFunc`? Again, common sense would suggest that a function's result belongs to the function, but this isn't the case. Here's another way of looking at it: we've seen how a string is initialised, used, and eventually finalized, or disposed of. The corollary to this is that a locally allocated string variable (that is, a local string declared within a routine) *will always be finalised in the same routine that created it.*

This is a very important point. And now the penny should be starting to drop. If the string Result of a function were initialised within the function, then who gets to finalise the string? It has to be done in the function, because the function created it. But it *can't* be done in the function, because the function result is required once the function itself has gone out of scope. Dilemma!

The solution that Borland devised was to treat string function results as var parameters from an implementation perspective. In other words, it looks like a function result at the source code level, but it's implemented as a var parameter 'under the hood'. Thus, the equivalent pseudo-code for Listing 2 is really doing what is shown in Listing 3.

You can now see exactly why the final result of these deliberations is HelloHello. The szText variable starts off as an empty string, and then has Hello concatenated onto it each time MyFunc is called. By inadvertently accessing the Result variable on the right-hand side of an assignment, we've exposed details of the underlying implementation that were best left covered up!

In Passing

Things become even more interesting when we start passing strings from one routine to another. As I'm sure you'll have realised from previous articles, passing a string by value (that is, without the const or var prefix) is rarely a good idea. Back in the old days of 'classic' Pascal-style short strings, passing a string by value forced the compiler to immediately create a new, local, copy of the string, which was allocated within the stack frame of the called procedure, and this is still the default behaviour whenever you use ShortString parameters. Because a ShortString always occupies a fixed 256 bytes of memory, this means that you have immediately consumed 256 bytes of stack space. This behaviour occurs whether or not the passed string is actually modified within the routine.

When Borland came up with the AnsiString implementation, it obviously had to change this strategy. An AnsiString can potentially be very big, in fact, enormous, and it would hardly be a sensible implementation to gratuitously copy megabytes of string data every time some novice developer forgets to put var or const in front of his string argument. Clearly, a different approach was required.

What Borland came up with was the elegant idea of reference-counting. When you pass a non-var, non-const string to a routine, the compiler automatically generates (yes, congratulations, you've guessed it!) another hidden try..finally block within the called routine. This routine increments the reference count of the passed string on entry to the procedure and then decrements it on exit. In fact, the routine used on exit is @LStrClr.

Previously, I have touted this routine as simply de-allocating a string, but it really does more than that. It decrements the string's reference count, only de-allocating the string if the reference count falls to zero.

So let's quickly review the overall scenario. You create a routine declaring some string variable, and use it within the routine. When first used (remember that the string variable is just a Nil pointer initially) the string gets assigned a reference count of one. Within the finally clause of the hidden try..finally block, the @LStrClr routine decrements the reference count and, if it falls to zero, disposes of the string. When you pass a string (reference count of 1) to another string without using the const or var prefixes, the reference count of the string gets bumped up to 2 within the called routine. This means that the corresponding @LStrClr routine simply decrements the reference count back to 1 and exits *without destroying the string*, which belongs to the calling routine.

The final part of the picture is another library routine called UniqueString. This routine is called automatically whenever you change characters in a string, take a pointer to characters within the string, or do anything that implies a write operation. Internally, UniqueString examines the reference count of the string and, if it is greater than one, creates a new, unique copy of the string with a reference count of one, but leaves the old string untouched. This makes it possible to pass an AnsiString by value, mess around with it in the called routine, and expect the original string to remain the same. Technically, this is called a copy-on-write strategy.

```
procedure MyFunc(
  var Result: String);
begin
  Result := Result + 'Hello';
end;
procedure MyProc;
var
  szText: String;
begin
  MyFunc (szText);
  MyFunc (szText);
  ShowMessage (szText);
end;
```

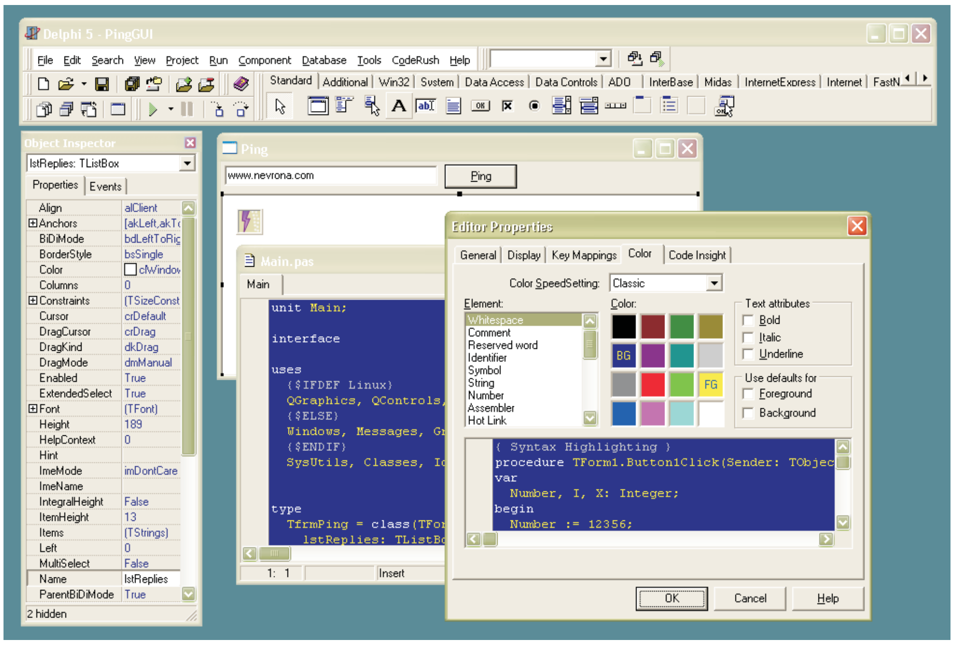
➤ *Listing 3*

```
var
  bmp: TBitmap;
  jpeg: TJPEGImage;
begin
  bmp := TBitmap.Create;
  jpeg := TJPEGImage.Create;
  try
    bmp.LoadFromFile('C:test.bmp');
    jpeg.Assign(bmp);
    jpeg.SaveToFile('C:test.jpg');
  finally
    bmp.free;
    jpeg.free;
end;
```

➤ *Listing 4*

➤ *Listing 5*

```
var
  bmp: TBitmap;
  jpeg: TJPEGImage;
  is1, is2 : ISafeGuard;
begin
  bmp := TBitmap(
    Guard(TBitmap.Create,is1));
  jpeg := TJPEGImage(
    Guard(TJPEGImage.Create,is2));
  bmp.LoadFromFile('C:test.bmp');
  jpeg.Assign(bmp);
  jpeg.SaveToFile('C:test.jpg');
end;
```


Delphi 5 - PingGUI
File Edit Search View Project Run Component Database Tools CodeRush Help
Standard Additional Win32 System Data Access Data Controls ADO InterBase Midas InternetExpress Internet FastN
Object Inspector
Properties Events
Ping
www.nevrona.com
Ping
Main.pas
Main
1: 1 Insert
Editor Properties
General Display Key Mappings Color Code Insight
OK Cancel Help

lstReplies: TListBox

Align	alClient
⊞Anchors	[akLeft,akT
BiDiMode	bdLeftToRig
BorderStyle	bsSingle
Color	☐ clWindov
Columns	0
⊞Constraints	(TSizeConst
Cursor	crDefault
DragCursor	crDrag
DragKind	dkDrag
DragMode	dmManual
Enabled	True
ExtendedSelect	True
⊞Font	(TFont)
Height	189
HelpContext	0
Hint	
ImeMode	imDontCare
ImeName	
IntegralHeight	False
ItemHeight	13
Items	(TStrings)
Left	0
MultiSelect	False
Name	lstReplies
ParentBiDiMode	True

2 hidden

```
unit Main;

interface

uses
  {$IFDEF Linux}
  QGraphics, QControls,
  {$ELSE}
  Windows, Messages, Gr
  {$ENDIF}
  SysUtils, Classes, Id


type
  TfrmPing = class(TFor
    lstReplies: TListBo
```

Color SpeedSetting: Classic

Element: Whitespace, Comment, Reserved word, Identifier, Symbol, String, Number, Assembler, Hot Link

Color: BG, FG

Text attributes: ☐ Bold ☐ Italic ☐ Underline

Use defaults for: ☐ Foreground ☐ Background

```
{ Syntax Highlighting }
procedure TForm1.Button1Click(Sender: TObjec
var
  Number, I, X: Integer;
begin
  Number := 12356;
```

➤ *Figure 3: It's Delphi, Jim, but not as we know it. Next month we'll be getting stuck into some Delphi code that exploits the new visual themes support in Windows XP.*

And Finally

A lot of what I've said here about strings is also true of interfaces. As with strings, an interface is a finalisable object and, as with strings, the code generator goes to some lengths to ensure that interfaces are correctly disposed of at the end of the scope block where they're declared, using hidden `try..finally` statements to work its magic.

And that would be the end of the story if it weren't for Will Watts and the JEDI Code Library. Will recently brought to the attention of assembled 'cixen' an interesting new feature of this library called SafeGuards. Using SafeGuards, it's possible to bind an interface to some arbitrary variable which needs to be de-allocated at the end of the current block, such as a bitmap, memory handle, form or whatever. Because the Delphi runtime library automatically takes care of destroying an interface object at the end of the declaring scope, these other objects effectively 'piggy back' onto the interface object and are automatically destroyed by the JEDI library code when the interface object is destroyed. To make this a little clearer, Listing 4 is a simple example taken from an article on the Borland community website at

```
http://community.borland.com/
  article/0,1410,27465,00.html
```

Here, an image is loaded from a bitmap file and copied out to a .JPG file. In the usual way, the `bmp` and `jpeg` objects are protected by a `try..finally` block such that they are automatically destroyed even if an exception occurs. Listing 5 is the same code using SafeGuards.

This time, it's necessary to declare two `ISafeGuard` variables, one to 'guard' each of our two resource variables. The `Guard` routine sets up the association between the interfaces and the `bmp` and `jpeg` variables. As you can see, it's no longer necessary to explicitly destroy these variables because it happens implicitly when `is1` and `is2` are themselves destroyed. The advantage of this technique is that you don't need to mess around with `try..finally` blocks, which, although not a major benefit in this trivial example, can make a big difference to code readability in a more complex example.

I'm neither advocating this technique nor advising against it, but it seems to be an interesting idea worth further investigation. My philosophy is, and always has been, that the less code one has to write, the less opportunity there is to create bugs, and that's why I prefer developing with Delphi!

As you'll see from my final screenshot, I now have Delphi running under Release Candidate 1 of Windows XP! Next month, we'll be developing some Delphi code which makes use of XP's new Theme Manager API.

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level Windows programming and cross-platform issues. He is the Technical Editor of *The Delphi Magazine*. You can contact Dave at TechEditor@itecuk.com