Slimming The Fat Off Your Apps

by Hallvard Vassbotn

A few weeks ago I came across some unexpected initialisation code in one of my units while debugging. I was puzzled to find that the compiler had generated some code in the `initialization` section of the unit, even though I hadn't declared one. At the time I was busy tracking down a logical error in my code, so I didn't pursue the issue any further.

Later, I came across a discussion on how you could reduce the memory footprint of the Delphi IDE on Windows NT by simply minimising and restoring it. This applied to both Delphi 3 and 4. I tried it and it worked, but nobody could explain why. It turns out that, under NT, the operating system will page out the executable when you minimise an application. This is a very useful memory optimisation that is unfortunately not implemented in Win95 or Win98. But this still did not explain why the memory used in the Delphi IDE was so drastically reduced even *after* restoring it.

Examining The Patient

I suddenly remembered my rather puzzling debugging session and started to investigate, to see if the two issues were related. First I created a simple unit with `initialization` and `finalization` sections:

```
Unit TestInit;
interface
var UnitGlobal: longint;
implementation
initialization
  UnitGlobal := 0;
finalization
  UnitGlobal := 0;
end.
```

Then I set breakpoints on the two `UnitGlobal` assignment statements and ran the application. When the debugger stopped at the breakpoints, I opened up the CPU Window. To enable this window in Delphi 3, you need to set the following Registry key:

```
\HKCU\Software\Borland\Delphi\
  3.0\Debugging\EnableCPU="1"
```

By looking at the generated assembly code I could deduce what extra code the compiler had inserted.

This exercise was repeated with different versions of the `TestInit` unit to see how it would affect the generated code. I tested and debugged the code in Delphi 2, 3 and 4. The following is a report of what I found. For more details about the generated assembly code, look at the Turbo Debugger log files on this month's disk.

It turns out that Delphi 3 and 4 behave exactly the same, whilst Delphi 2 works a little differently. Lets look at how Delphi 2 compiles the `init.../final...` code first.

Delphi 2 Diagnosis

In Delphi 2, units get code in the `initialization` section if, and only if, it is explicitly declared by the programmer. The `initialization` sections are called directly from the startup code of the executable, so the code in the project file InitTest.Dpr:

```
begin
  Application.Initialize;
  Application.Run;
end.
```

is compiled into (pseudo-code):

```
begin
  System._InitExe;
  System.initialzation;
  SysUtils.initialzation;
  Classes.initialzation;
  Printers.initialzation;
  Menus.initialzation;
  Controls.initialzation;
  Forms.initialzation;
  TestInit.initialzation;
  Application.Initialize;
  Application.Run;
end.
```

If the unit does not contain a `finalization` section, no code is added by the compiler, otherwise a call to `AddExitProc` is inserted as the first statement in the `initialization` section, for instance the `initialization` section in InitTest.Pas:

```
initialization
  UnitGlobal := 0;
```

is compiled into (pseudo-code):

```
initialization
  System.AddExitProc(
    @InitTest.finalization);
  UnitGlobal := 0;
```

If the unit declares one or more global long string variables, the compiler adds a `try..finally` construct with an empty finally block. I don't know why this is done, as it seems totally redundant. If we add a global string variable to `InitTest`, our code now looks like this:

```
var LongstringRef: AnsiString;
initialization
  UnitGlobal := 0;
```

but the compiler generates the equivalent to:

```
initialization
  try
    System.AddExitProc(
      @InitTest.finalization);
    UnitGlobal := 0;
  finally
  end;
```

Under no circumstances does the compiler add any code to the `finalization` section of the unit. In fact, this means that all units which declare global long string variables will potentially cause a memory leak.

This is not as bad as it sounds, because it will not be a leak that accumulates over time and the operating system will make sure to reclaim the memory allocated by the application when it is closed down. But this is a point to be aware of, as some memory leak

detecting debugging tools might signal these leaks.

To avoid the leaks it would have been better for the compiler to automatically insert code into the finalization section of each unit that declares global long string variables. We will see later that this is handled better by Delphi 3 and 4. If you are still using Delphi 2 and want to get rid of these memory leaks, you can simply add the code to the finalization section manually. In our case, we would add:

```
finalization
  UnitGlobal := 0;
  LongStringRef := '';
```

to ensure that the reference count of the long string is decremented when the application closes down and get rid of the memory leak.

The conclusion is that Delphi 2 is pretty well-behaved in generating code for unit initialization and finalization sections. It misses out on cleaning up global long string variables, but at least it doesn't add any dummy code behind our backs when there are no explicit initialization or finalization sections in the unit.

Delphi 3 And 4 Diagnosis

With Delphi 3 and 4 things are a little different. The initialization sections are no longer called directly from the startup code of the executable. Instead, the linker builds a structure with pointers to all the initialization and finalization sections that should be called during startup and shutdown. Look at the InitUnits and FInitUnits procedures in the System unit for details.

The startup code in the project file InitTest.Dpr is compiled into:

```
begin
  System._InitExe;
  Application.Initialize;
  Application.Run;
end.
```

System._InitExe calls System.InitUnits which loops through the structure generated by the linker, calling each initialization section.

It seems like Inprise was determined to get away from the long string memory leaks experienced with Delphi 2. At the same time the new Variant and Interface types were introduced. Just as with long strings, global variables of these types are reference counted and need to be cleaned up at shutdown. Delphi 4 also added dynamic arrays into the pot.

Another new feature added with Delphi 3 was package support. With this, Inprise needed a mechanism to ensure that finalization sections were only run if the corresponding initialization section had been executed. To implement this, the compiler automatically adds a global longint variable to all units. This variable is used as a flag to indicate if the initialization code has run or not. This variable is decremented in the initialization section and incremented in the finalization section.

Finally, for some reason, the compiler now always adds both initialization and finalization sections to *all* units, even if none are explicitly declared by the programmer. With this in mind, there are three possible variations of automatically generated code in Delphi 3 and 4.

If the unit does not declare any global reference counted variables (long strings, variants, interfaces or dynamic arrays), for instance for a completely empty unit

```
Unit TestInit;
interface
implementation
end.
```

the compiler still generates the equivalent to this code:

```
Unit TestInit;
interface
implementation
var
  AutoGlobal: longint;
initialization
  Dec(AutoGlobal);
finalization
  try
    Inc(AutoGlobal);
  finally
  end;
```

As you can see all this is redundant: 54 bytes of code and 4 bytes of global data for every unit in the application and any runtime packages it might be using. This is not so bad, but knowing that this code is run at application's startup, a lot of 4Kb blocks of code are paged in and left in physical memory.

We will see later how we can solve this problem. The best solution would be for Inprise to fix the compiler so that it does not generate this unnecessary code for units with no explicit initialization or finalization sections and no reference counted global variables. This behaviour is also the main reason why the IDE takes up so much memory and why this can be drastically reduced by minimising and restoring (at least on NT).

Let's continue with looking at the code which the compiler generates for other situations. If the unit has explicit initialization and finalization sections with user code in them, the compiler adds checks to see if the AutoGlobal variable is valid. So for our test unit:

```
Unit TestInit;
var UnitGlobal: longint;
interface
implementation
initialization
  UnitGlobal := 0;
finalization
  UnitGlobal := 0;
End.
```

the compiler adds this code:

```
Unit TestInit;
interface
var UnitGlobal: longint;
implementation
var AutoGlobal: longint;
initialization
  Dec(AutoGlobal);
  if AutoGlobal >= 0 then Exit;
  UnitGlobal := 0;
finalization
  try
    Inc(AutoGlobal);
    if AutoGlobal <> 0 then
      Exit;
    UnitGlobal := 0;
  finally
  end;
end.
```

Except for the dummy `try..finally` block, this is fine.

Finally we have the situation where there are one or more global reference counted variables in the unit, for instance Listing 1 is compiled into Listing 2. This code is also fine. Now the previously useless `try..finally` block in the `finalization` section is used to clean up the global variables. This is very good: we get rid of those nasty memory leaks (and with variants and interfaces potentially close down external OLE servers). Note, however, that the compiler has introduced yet another useless `try..finally` block, this time in the `initialization` section of the unit. Oh, well...

The conclusion is that Delphi 3 and 4 have fixed the potential memory leaks that Delphi 2 missed. However, they have also introduced the rather irritating and memory consuming practice of adding `initialization` and `finalization` code to every unit in the EXE file and all DPL files. Let's see how we can get around this.

➤ *Listing 1*

```
Unit TestInit;
interface
var
  UnitGlobal: longint;
  LongstringRef: AnsiString;
  InterfaceRef : IUnknown;
  VariantRef   : Variant;
 {$IFDEF VER120}
  ArrayRef: array of integer;
 {$ENDIF}
implementation
initialization
  UnitGlobal := 0;
finalization
  UnitGlobal := 0;
end.
```

➤ *Listing 2*

```
Unit TestInit;
interface
var
  UnitGlobal: longint;
  LongstringRef: AnsiString;
  InterfaceRef : IUnknown;
  VariantRef   : Variant;
  {$IFDEF VER120}
  ArrayRef: array of integer;
  {$ENDIF}
implementation
var AutoGlobal: longint;
initialization
  try
    Dec(AutoGlobal);
    if AutoGlobal >= 0 then Exit;
    UnitGlobal := 0;
  finally
  end;
finalization
  try
    Inc(AutoGlobal);
    if AutoGlobal <> 0 then Exit;
    UnitGlobal := 0;
  finally
    LongstringRef: = '';
    InterfaceRef := nil;
    VariantRef   := Unassigned;
    {$IFDEF VER120}
    ArrayRef := nil;
    {$ENDIF}
  end;
```

Sweetening The Pill

The best cure for the memory problems we've seen in Delphi 3 and 4 would of course be to fix the compiler. But what else can we do?

A method of working round the problem has been developed by Roy Nelson (rnelson@inprise.com) of Inprise European Professional Services. This code was originally created to get around the extra memory usage of Delphi applications. Roy's code makes the application's footprint smaller, as it forces the operating system to only keep the required code in memory. If this code is called from a unit's `initialization` section in a package added to the IDE, the IDE's memory footprint also reduces. Roy's routine is shown in Listing 3.

This little gem calls the Windows NT API `SetProcessWorkingSetSize` (check Delphi's online help for the definition of the routine) to page out all the code for the currently running process. If both `dwMinimumWorkingSetSize` and `dwMaximumWorkingSetSize` have the value `0xffffffff`, the function temporarily trims the working set of the specified process to zero. This essentially swaps the process out of physical memory.

Proof Of The Pudding

In the article, we see how the compiler generates different versions of the initialization and finalization code. I said the compiler adds code to units with no explicit `init/final` sections in them, but I didn't explain how I came to this conclusion. It is true that I stumbled upon one of these magic `init` sections when debugging one day, but how can we prove that this is the general case for all units? And how do I know that the memory reference that gets incremented and decremented refers to a global variable declared locally in that unit and not some other global structure?

For the more adventurous of you, you can try the following steps. Include this unit in a project:

```
Unit TestInit2;
interface
var
  AutoGlobalP : pointer;
  UnitGlobal: longint;
implementation
//var AutoGlobal: longint;
initialization
//Dec(AutoGlobal);
//if AutoGlobal >= 0 then Exit;
  AutoGlobalP := pointer(longint(@UnitGlobal) + 4);
  UnitGlobal := 0;
finalization
//try
//  Inc(AutoGlobal);
//  if AutoGlobal <> 0 then Exit;
    AutoGlobalP := pointer(longint(@UnitGlobal) + 4);
    UnitGlobal := 0;
//finally
//end;
end.
```

Notice that I have included commented out code that corresponds to the assembly added by the compiler.

Now set breakpoints on the two `UnitGlobal` assignments, then run. When you hit the breakpoints, bring up the CPU View. Notice the address of the compiler generated global variable from the assembly code. Now evaluate the value of the `AutoGlobalP` variable. The address and the value should match. This proves that Delphi adds a global variable just after the last declared global variable.

Next (assuming you have TASM32) copy the RTL to a new folder and MAKE it with full debug info. Add the debug LIB path to a test project with some units without any explicit initialization or finalization sections. Compile the project. Open System.Pas and go to the procedure `InitUnits`. Set a breakpoint on the `try` statement. Now run the project. When you hit the breakpoint, evaluate the `Table` variable (de-reference it). See the list of `init...` and `final...` code that will run. Notice that all units are included even if they have no explicit `init/final` sections.

```
procedure TrimWorkingSet;
var MainHandle : THandle;
begin
 MainHandle := OpenProcess(PROCESS_ALL_ACCESS, false, GetCurrentProcessID);
 SetProcessWorkingSetSize(MainHandle,-1,-1);
 CloseHandle(MainHandle);
end;
```

➤ *Above: Listing 3*

Unfortunately this API is only implemented on NT platforms, but it is stubbed out in Win95 and should do no harm. It seems that when NT minimises an application, it calls this function, paging out all the inactive memory pages.

As an experiment I added a call to `TrimWorkingSet` to a sample project. Testing (on NT 4.0 sp3) showed that the initial memory footprint (Mem Usage) reported by the NT Task Manager dropped from 8,728Kb to 7,528Kb, a reduction of about 14%! This project has about 260 units, so your mileage may vary. I saw no performance degradation after adding this call.

It would also be very useful to automatically trim the memory usage of the Delphi IDE without having to minimise and restore it each time. As Roy pointed out, this can be easily done by adding a design-time package that calls `TrimWorkingSet`. On the disk you will find both a `TrimMem` unit to include with your own applications and a `TrimMemP` package to install into Delphi (there is also a `TrimMem4` package compatible with Delphi 4).

You should try to ensure that the `TrimMemP` package is loaded after all other packages to get maximum effect. My testing showed that the initial memory footprint was reduced from 12Mb to 2.5Mb for Delphi 3 and from 20.8Mb to 4.4Mb for Delphi 4. Very nice!

In your applications, add the `TrimMem` unit to the `uses` clause and call `TrimWorkingSet` directly from the project file (after any splash windows or startup code) like this:

```
program TrimTest;
uses
  TrimMem,
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  TrimMem.TrimWorkingSet;
  Application.CreateForm(
    TForm1, Form1);
  Application.Run;
end.
```

We could simplify things by including a call to `TrimWorkingSet` in the `initialization` section of the `TrimMem` unit, and include this as the last unit in the project file. However, this would have been less practical as the unit name would have to be moved down each time you add another form or unit to the project. To get code to run when the `TrimMemP` package is loaded we include the `RegTrim` unit that provides a dummy `Register` function to call `TrimWorkingSet`.

Conclusion

For Delphi 2 the overhead of the 'secretly' added code is negligible, but if you have more than a few units with `init.../final...` code you could benefit from using the `TrimMem` unit.

For Delphi 3 and 4, the overhead is substantial, especially for projects with many units. Adding `TrimMem` to your application could lose 10% to 15% of the initial memory footprint.

Finally, for the Delphi IDEs, using the `TrimMemP` package may mean you can delay buying that extra 128Mb of RAM...

Currently this cure only works on Windows NT platforms. It would be nice if we could find a similar solution for Windows 95 and 98, but my research has shown that there are no easy ways of paging out processes in these OSs. The only solution seems to be writing a VxD driver to do the job and that is outside the scope of this article...

Hallvard Vassbotn is a Senior Software Developer at Reuters Norge AS, Falcon R&D. You can reach him at hallvard@falcon.no