The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Linking Data Files

Q I have a data file which in my Delphi 1 days I converted to an object file with that old BinObj.exe program we all had. Then I linked the object file into my program at compile-time, which worked great. But 32-bit Delphi won't let me link my object file, probably because it was compiled as a 16-bit object file. How do I get the desired results with 32-bit Delphi?

A BinObj.Exe is a simple command line utility that turns an arbitrary data file into an .OBJ (object) file, suitable for linking into a 16-bit executable. It used to come with Turbo Pascal and Borland Pascal With Objects, but has since slipped into obscurity. Its purpose is to avoid requiring external data files by linking the data directly into the executable (a bit like a Windows resource, but less accessible to the outside world).

BinObj.Exe works fine with Delphi 1 (a 16-bit development tool), but not with Delphi 2 and later, due to the differences between 16-bit and 32-bit object files. A quick search on the internet for the phrase BinObj32 yielded many references to BinObj32.Zip, containing BinObj32.Exe, which is a 32-bit freeware equivalent application. BinObj32.Exe is written by Jan Rekorajski (aka Mr Baggins) and is advertised as being primarily for Watcom C developers. It also works fine with 32-bit Delphi. *[We got it from http://ftp.xanet.edu.cn/pub/tools/converter/ and it's on the disk with this issue. Ed]*

Let's see how it works. Firstly we will need a data file of some description. What it contains is irrelevant, but it is important that the application knows how large the data block is. How this is achieved will be dependent on your application. You could either hard code it, or store the size as the first `DWord` in the data file. If the data file contains textual information, you could store the strings in null-terminated form and mark the end of the list of strings with a double null terminator (two ANSI characters, each having a value of 0).

As a simple example, the data file in this case will be a bitmap file. Fortunately, bitmap files have information on the size of data in the file header, so this can be read to work out the data size.

The following command line takes the Athena.Bmp bitmap file (supplied with Delphi) and makes an object file called Image.Obj:

```
BinObj32 Athena.Bmp Image.Obj
  AthenaImage
```

The object file contains the data from the original file and identifies it with the public symbol `_AthenaImage`. Notice the prefixed underscore which is always added by BinObj32 (due to the requirements of Watcom C).

Now we have the data file, we need to get it linked into the executable and then make the data accessible. This involves two steps.

Firstly, you link an .OBJ file into your application with the `$L` compiler directive. To make the data programmatically accessible, you then declare a dummy procedure with the same name as the public symbol in the .OBJ file, and mark it with the `external` directive. The linker then matches the 'procedure' up with the linked data.

It doesn't matter whether the data represents code or not; in this case we need to match a procedure declaration with image data. The point is that an external procedure declaration is the only available mechanism to access linked object file content. If the object file contains data, taking the address of the 'procedure' will give you the address that the data starts at.

Listing 1, from LinkFile.dpr, shows the principles, including the `$L` directive and the external procedure declaration. The form's `OnCreate` event handler sets up a `TMemoryStream` to point at the image data and then loads it into a bitmap managed by a `TImage` component.

➤ *Listing 1: Linking an image into an executable as raw data.*

```
{$L Image.obj} //Link in data file
//Declare symbol that marks start of data in linked data file
procedure _AthenaImage; external;
//Interposer class to access protected methods of TMemoryStream
type
  TMemoryStream = class(Classes.TMemoryStream);
procedure TForm1.FormCreate(Sender: TObject);
var
  MS: TMemoryStream;
  PBMF: PBitmapFileHeader;
begin
  MS := TMemoryStream.Create;
  try
    //Bitmap file header is at start of a bitmap file
    PBMF := @_AthenaImage;
    //Tell memory stream where the memory is, and how much there is
    MS.SetPointer(@_AthenaImage, PBMF^.bfSize);
    //Load image data into TImage component
    Image1.Picture.Bitmap.LoadFromStream(MS)
  finally
    MS.Free
  end
end;
```

However, as is often the case in programming, it is a little more complicated than the previous sentence suggests.

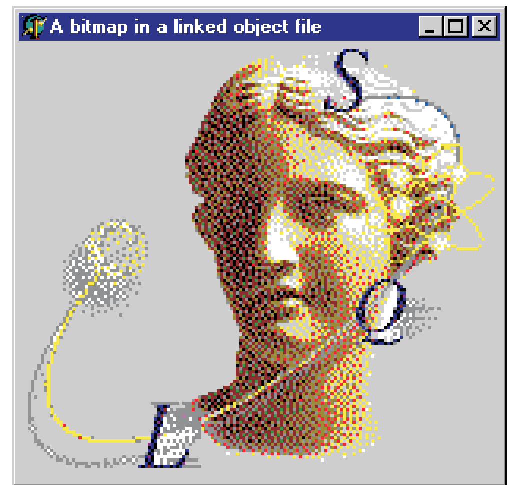
You can see the address of `_AthenaImage` being taken and assigned to a pointer to a `TBitmapFileHeader` record (which represents the beginning of a bitmap file). This is done so we can tell how large the bitmap file data is (the size is stored in the `bfSize` field of the header). The `TMemoryStream` needs to then be told to represent the memory starting at the address of `_AthenaImage` with the size just found.

`TMemoryStream` does have a `SetPointer` method which does just this, but unfortunately it is protected, so an access class is used to gain access to it. In fact the type of class used is sometimes referred to as an *interposer class* (see the article on the subject by Stephen Posey in Issue 33).

If you understand the `exact` definition of the `protected` keyword of a class, then the concept of an access class will be quite clear. Any code in the same unit as a class definition can access that class's protected members. So if you define a new class inherited from your target class in the unit you are working in, a simple typecast opens up the protected section of such an object to you. An interposer class is used in this case, where the memory stream is defined in terms of the new derived class.

The end result of the code is shown in Figure 1.

That would be the end of the matter, but earlier I mentioned you could do much the same with a custom resource linked into the executable. The downside of this is that various applications, such as Borland's Resource Workshop, and indeed the Delphi Resource Explorer demo application, can see custom resources and save them back to individual disk files. If this is not a problem, then the process is a bit simpler than linking in an object file.



➤ *Figure 1: The linked data shown in a TImage.*

The process was covered in detail in *The Delphi Clinic* in Issue 32 (*Playing Videos*), but here is a quick précis. First you need a resource script, which is a text file that identifies the data file, the data type (an arbitrary name) and a resource name (another arbitrary name). An example here would be a resource script called Image.RC that contains this single line:

```
AthenaImage BitmapData
  "Athena.bmp"
```

This says the Athena.bmp file will be a `BitmapData` resource (which means nothing really, as `BitmapData` is a made-up word in this context) called `AthenaImage`.

The resource script needs to be turned into a resource file with the resource compiler, a command line tool called BRCC32.EXE. Passing the Image.RC filename to BRCC32 yields the Image.Res resource file, which can be linked into the executable with the `$R` compiler directive.

So far we have a fairly similar set of steps (using a command-line tool to make a file which is linked with a directive); however, things are easier from now on. In the program you can either use Windows API calls or a `TResourceStream` to access the resource data. Listing 2 shows the new `OnCreate` handler (from LinkRes.dpr). The program running has much the same effect as in Figure 1.

➤ *Listing 2: Accessing a custom resource.*

```
{$R Image.res} //Link in resource file
procedure TForm1.FormCreate(Sender: TObject);
var
  RS: TResourceStream;
begin
  RS := TResourceStream.Create(HInstance, 'AthenaImage', 'BitmapData');
  try
    //Load image data into TImage component
    Image1.Picture.Bitmap.LoadFromStream(RS)
  finally
    RS.Free
  end
end;
```

CPU Window Bafflement

Q Occasionally when debugging an application I get faced with the CPU window. Like many of my fellow Delphi developers, I have yet to find out how this low-level display can be of any use to me. As someone who has never programmed in assembly language, should I simply ignore the window?

A On the contrary, you should grit your teeth and dive in, as the CPU window can be very helpful in many debugging tasks that can prove tricky without it. Obviously to get you familiarised with how you can use it would take a lot of space, which I don't have in this issue. So I would suggest reading my DCon 2000 paper, *Debugging With More Than Watches And Breakpoints (or How To Use The CPU Window)*. If you didn't attend DCon 2000, you can buy a copy of the conference CD by calling UK-BUG on +44 (0)1980 630032.

The paper gives a brief overview of assembly mnemonics and gives coverage of what the CPU window shows. It also looks into how to approach a number of debugging problems using the CPU window (and some knowledge that can be gleaned from browsing through the `System` unit source code).

Array Distinction

Q What is the difference between an open array and a dynamic array? To me, they both seem to be arrays without a fixed element count.

A Let's take a walk down memory lane and see how both these language elements came about. Picture yourself in 1991 with a copy of Turbo Pascal 6 (or earlier) and think about arrays. This historic compiler supported objects and many other useful features, but it was very limited in array support.

You could write a subroutine that took an array as a parameter, but only if you did it the right way and with various limitations imposed. The array type for the parameter had to be defined in advance, specifying the number of elements. Then the formal parameter declaration and the actual array parameter variable would be forced to be declared as that exact type, as shown in Listing 3. Notice that to avoid writing repetitive statements to initialise each array element, a typed constant is used (discussed in *The Delphi Clinic* in Issue 59).

Listing 4 shows some arrays that are ostensibly the same as the first array, but which are not acceptable as parameters due to their different type definitions. Only parameters of the argument's type (`TIntArray`) will be accepted by the routine.

This was inflexible. Being forced to use a specific array type was bad enough, but the killer was being forced to pre-determine how many elements would be passed.

To help with this restriction, Turbo Pascal 7 (released in 1992) relaxed the syntax rules and introduced *open parameter types*. These affected short string and array parameters. Since we normally use long strings these days, we can focus on *open array parameters*.

An open array parameter is one whose type is defined as

```
array of T
```

```
type
  TIntArray = array[1..4] of Integer;
function TP6Sum(var IntArray: TIntArray): Integer;
var
  I, Res: Integer;
begin
  Res := 0;
  for I := 1 to 4 do
    Inc(Res, IntArray[I]);
  TP6Sum := Res
end;
const
  IntArray1: TIntArray = (9, 3, 0, 10);
var
  I: Integer;
...
WriteLn(TP6Sum(IntArray1));
ReadLn
```

➤ *Listing 3: A Turbo Pascal 6 array parameter.*

```
type
  TIntArray = array[1..4] of Integer;
  TAnotherIntArray = array[1..4] of Integer;
...
const
  IntArray1: TIntArray = (9, 3, 0, 10);
  IntArray2: TAnotherIntArray = (9, 3, 0, 10);
  IntArray3: array[1..4] of Integer = (9, 3, 0, 10);
...

WriteLn(TP6Sum(IntArray1)); { Compiles fine, as before }
WriteLn(TP6Sum(IntArray2)); { Type mismatch }
WriteLn(TP6Sum(IntArray3)); { Type mismatch }
```

➤ *Listing 4: Illegal array parameters in Turbo Pascal 6.*

where `T` is a valid type. The actual parameter passed in must either be of type `T` or be an array variable whose element type is `T`. This allows many different types of arrays to be passed where an open array parameter is defined, rather than being restricted to a very specific type. This where the *open*ness in the term open array comes from.

Inside the routine, the formal parameter operates as if it was defined as:

```
array[0..N-1] of T
```

where `N` is the number of elements in the actual parameter. So the index range of the actual parameter, which could start at any value, is effectively mapped onto the integers `0` to `N-1`. If the actual parameter was just a variable of type `T`, the formal parameter acts as if it is defined as:

```
array[0..0] of T
```

The two standard functions `Low` and `High` can be applied to an open array parameter and `Low` will return 0 whilst `High` will return `N-1`. `SizeOf` returns the size of the actual array parameter in bytes.

The implementation of open arrays is quite simple. The compiler generates code to pass the address of the first element in the array as one parameter, followed by the `N` value as the next parameter. This gives the routine all it needs to know about the size of any open array parameter.

In short, an open array parameter allows many differently sized arrays of the same element type to be passed into a general purpose routine. Listing 4 compiles in Turbo Pascal 7 and in Delphi. However, Delphi added a couple of extra levels of help for open array parameters which were not available in Turbo Pascal 7.

Firstly, to pass an array to an open array parameter, Turbo Pascal 7 required you to have an array variable to start with. If you didn't have one, you had to declare and initialise one. Delphi allows you to declare an actual parameter that can be passed to an open array parameter on the fly, in situ. So another valid call to `TP6Sum` in Delphi would be:

```
TP6Sum([9, 3, 0, 10])
```

This uses an *open array constructor* (rather like a *set constructor*). This is a pair of square brackets surrounding a comma-separated list of values of the appropriate type and can be used as a shorthand open array parameter value, avoiding the declaration and initialisation of a variable. These open array constructors can only be passed by value or as `const` parameters, not as `var` parameters. Each element in the open array constructor is an expression of the array element type, so can be more interesting than just passing constant values.

Additionally, Delphi introduced a special type of open array parameter, called a variant open array. When the formal parameter is declared as `array of const`, each element can be any one of a number of fixed types: `Integer`, `Int64`, `Boolean`, `Char`, `WideChar`, `floating point`, `pointer`, `String`, `ShortString`, `WideString`, `PChar`, `PWideChar`, object reference, class reference, interface reference and `Variant`.

➤ *Listing 5: The TVarRec type responsible for variant open arrays.*

```
const
  vtInteger    = 0;
  vtBoolean    = 1;
  vtChar       = 2;
  vtExtended   = 3;
  vtString     = 4;
  vtPointer    = 5;
  vtPChar      = 6;
  vtObject     = 7;
  vtClass      = 8;
  vtWideChar   = 9;
  vtPWideChar  = 10;
  vtAnsiString = 11;
  vtCurrency   = 12;
  vtVariant    = 13;
  vtInterface  = 14;
  vtWideString = 15;
  vtInt64      = 16;
type
  PVarRec = ^TVarRec;
  TVarRec = record { do not pack this record; it is compiler-generated }
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);    //Added in Delphi 2
      vtPWideChar:  (VPWideChar: PWideChar);  //Added in Delphi 2
      vtAnsiString: (VAnsiString: Pointer);   //Added in Delphi 2
      vtCurrency:   (VCurrency: PCurrency);   //Added in Delphi 2
      vtVariant:    (VVariant: PVariant);     //Added in Delphi 2
      vtInterface:  (VInterface: Pointer);    //Added in Delphi 3
      vtWideString: (VWideString: Pointer);   //Added in Delphi 3
      vtInt64:      (VInt64: PInt64);         //Added in Delphi 4
  end;
```

The compiler translates an `array of const` into an `array of TVarRec`, where `TVarRec` is a variant record type defined in the `System` unit (see Listing 5). The subroutine code can take an element of the `array of const` parameter and typecast it to a `TVarRec` record. The code can then examine the `VType` byte field, comparing it with the `vtXXXX` constants to see which field of the variant record to access.

This facility is great. It offers the possibility of writing a type-safe routine that can take parameter values of effectively arbitrary types. Example routines that do this can be found in *The Delphi Clinic* all the way back to Issue 8, in the *Multiple Arguments* entry, and also in Issue 6, in *Setting Properties En Masse*.

Hopefully, it is now well understood that an open array is a mechanism to allow a subroutine formal parameter to be more flexible than it would be in traditional Pascal, taking arrays of any size, so long as they have the designated element type. So now we should turn our attention to *dynamic arrays*.

When you declare an array variable using traditional Pascal syntax, you typically specify the size in the type declaration. Then, the compiler can allocate sufficient memory for each variable that you use. These arrays can be manipulated and used as you like, possibly by being passed to subroutines with open array parameters.

But much like the restriction on Turbo Pascal array parameters being forced to be a fixed size, all Pascal compilers up to and including Delphi 3 forced your array variables to also be fixed size. There are ways around this, using pointers, `TList` objects and so on, and these have been covered in Issue 37, in my article *Dynamic Arrays*. However, Delphi 4 introduced formal support for dynamic arrays.

When you declare a dynamic array variable, you use syntax much like that for an open array parameter, specifying no element bounds, for example:

```
var
  IntArray: array of Integer;
```

This sets up an array of integers with zero elements. The programmer can specify how many elements they want to store in the array at any later point using the `SetLength` procedure, for example:

```
SetLength(IntArray, 10);
```

Dynamic arrays operate on the same basis as dynamic strings (normal 32-bit Delphi string variables). `SetLength` can be used to extend or shrink the size of the array, `Length` returns the number of elements in it, and the compiler ensures that the arrays storage space is tidied away as the array goes out of scope.

Beware, though, as `SizeOf` will return the size of the pointer pointing to the dynamically managed block of memory for the array, and so will always return 4 when used against a dynamic array. This point was explored in *Dynamic Array Question* in *The Delphi Clinic* in Issue 54

Another gotcha waiting in the wings occurs when you try and implement a subroutine with a parameter that you want to represent a dynamic array (as opposed to any fixed-size array): remember

to be careful! Since both open arrays and dynamic arrays use the *array of T* syntax, you must tread carefully.

A routine declared as:

```
procedure Sum(
  Nums: array of Integer);
```

defines an open array parameter, not a parameter corresponding to a dynamic array of integers. To get a routine that can only take a dynamic array as a parameter, use the approach shown in Listing 6.

In summary, an open array is a flexible formal parameter type, whilst a dynamic array is a flexible variable type.

Corrupt Component Palette

Q When I start Delphi on my PC, the Component Palette icons appear corrupted (or sometimes black). How can I fix this?

A This issue has been prevalent over the last couple of years with various video cards and has been looked at in *The Delphi Clinic* in Issues 31 and 42. The question is being covered again as more information has recently come to light as to how to remedy the problem.

Firstly, though, an overview of information printed previously:

- Get the most recent version of COMCTL32.DLL from Microsoft's website and install it.
- Verify the problem is video card or video mode dependent by testing Delphi after restarting Windows in safe mode, which uses standard VGA mode.
- Get the latest version of the video driver from your video card vendor's website.
- Reduce the graphics hardware acceleration in Windows 9x in the *System properties* dialog.
- Ensure you are not trying to run in Windows 98 on 16 colours (16 colours is not supported, although 16-bit colours is).
- Remove any unnecessary component packages, as the more component images being managed, the more likely the problem is to occur.

```
type
  //Dynamic array type
  TDynamicIntegerArray = array of Integer;
//Parameter must be a dynamic array
function Sum(Nums: TDynamicIntegerArray): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Nums) to High(Nums) do
    Inc(Result, Nums[I])
end;
...
var
  //Variable uses same dynamic array type
  Array1: TDynamicIntegerArray;
  Array2: array[0..1] of Integer;
...
SetLength(Array1, 2);
Array1[0] := 1;
Array1[1] := 10;
ShowMessageFmt('%d', [Sum(Array1)]); //Displays 11
Array2[0] := 1;
Array2[1] := 10;
//ShowMessageFmt('%d', [Sum(Array2)]); //Will not compile
```

➤ *Listing 6: A routine that takes a dynamic array.*

The new piece of advice relates to Delphi running on Windows 98 and can be found in MSDN article Q218171. It suggests you add a new entry in SYSTEM.INI. This can be done in a normal text editor, with SYSEDIT.EXE, or MSCONFIG.EXE.

In MSCONFIG.EXE, click the `System.Ini` tab, click the `Display` node, click the + sign next to `Display`, click `New`, type `BusThrottle=1` and click `OK`. Thanks are due to Tony Hardman for pointing this new setting out to me at DCon 2000.

Amendments

I have been notified of a number of things which warrant a new batch of updates and amendments.

Following on from my articles on Paradox table corruption in Issues 17 and 42 (also available at www.TheDelphiMagazine.com on the *Sample Articles* page), Bruno Caprez notified me of something new. Traditional advice includes calling the BDE `dbiSaveChanges` API in a `TTable` object's `AfterPost` event handler. Bruno kindly introduced me to the `FlushBuffers` method of `TBDEDataSet` and descendants which calls `dbiSaveChanges` on your behalf, saving you having to make a BDE API call.

Terry Jepson wrote to say that the PageCtl3.pas code from my Issue 26 (October 1997) article on customised `TPageControl` components no longer compiles in Delphi 5. This is because Delphi 5 moved the `TOwnerDrawState` set type from the `StdCtrls` unit to the `Windows` unit. It also added several more possible values to the type, causing it to now require two bytes rather than one.

To fix the code, change this expression from the `CNDrawItem` message handling method, which accesses the low byte:

```
TOwnerDrawState(
  WordRec(Word(ItemState)).Lo)
```

to this, which accesses the low word:

```
TOwnerDrawState(
  LongRec(ItemState).Lo)
```

Thomas Mueller emailed me after reading the *Dual Processor Woes* entry on SMP machine issues in *The Delphi Clinic* column back in Issue 52. He mentions that the BDE is not thread-safe on multiprocessor machines and was forced to make a call to `SetProcessAffinityMask` to keep all the database threads on one processor. Thanks for that tip, Thomas.

I also had an email from Inprise's Roy Nelson regarding Issue 58's *Optimised Working Set* entry. He mentioned that minimising the working set at arbitrary points in an application's run can lead to page faults and recommends only calling the appropriate code if the application is in the background (as is done by the code in Listing 7).

An updated version of Issue 58's TrimWorkingSet.pas is included on the disk. Note that you can take this idea much further, as Roy has himself, to implement this code in a background thread rather than in a timer. You can also add in checks so that the code only kicks in once the working set has risen by, say, 25%. Thanks are due to Roy.

Thanks also to Bob Swart who has some additional news on the *Strings To Numbers* entry from Issue 59. He points out that there is a SysUtils routine called `StrToIntDef` which does much the same job as `StrToInt`. However, erroneous input makes `StrToIntDef` return a supplied default value.

There is no equivalent for floating point numbers, but you could write one easily, as shown in Listing 8. Delphi 1 would need the routine written slightly differently, as `TextToFloat` took only two parameters with that version.

Another point on Issue 59 came from David Markie, who pointed out that the code shown in the *Delphi Grammar Problem* entry was lacking a little. The code sets up a call to `CreateProcess` and was based on some code sent into *The Delphi Clinic*. Unfortunately, the code does nothing with the `TStartupInfo` record, `SI`, before passing it to `CreateProcess`. This record should be set up before the call, possibly by passing it to the `GetStartupInfo` API or possibly by setting the data fields individually (as can be seen in Issue 51's *Clinic* in the *CreateProcess Alert* entry).

Gerry Haynaly also pointed out that my commented piece of code in Listing 10 from the same entry in Issue 59 was missing a pair of parentheses. In other words, what was printed as:

```
while not WaitForSingleObject(
  PI.hThread, 40000) in
  [WAIT_OBJECT_0,WAIT_TIMEOUT]
  do
```

should have looked like:

```
while not (WaitForSingleObject(
  PI.hThread, 40000) in
  [WAIT_OBJECT_0,WAIT_TIMEOUT])
  do
```

(note the extra parentheses). Thanks David and Gerry for cleaning up after me there ☺.

Also in Issue 59, I mentioned the location of a beta update to the InterBase Express components. Thanks are due to Wyatt Wong who tells me that the finished IBX update 4.2 for Delphi 5 and C++Builder 5 can now be downloaded from

```
www.interbase.com/open/
  downloads/IBX_updates.html
```

One final update is to something I said in one of my talks (*IDE/RTL/VCL/ObjectPascal Tips*, at this year's DCon 2000 conference) . As the talk drew to a close, someone asked for a recommended way to initialise an array of data entities, for example strings, an operation that may be needed several times in a given subroutine.

My impulse reaction was to answer the question of initialising an array and focused on that. I mentioned that strings are represented by pointers, so an array of strings would be an array of 32-bit entities. You could use

```
FillChar(TheArray,
  SizeOf(TheArray), 0)
```

to set each string to a `nil` pointer, representing an empty string. By the way, `SizeOf` will work for a normal array, but not a dynamic array (see *Dynamic Array Question* in *The Delphi Clinic*, Issue 54).

Of course at the time I completely forgot that long string variables are auto-managed by compiler-generated code which endeavours to free up any memory that has been allocated for a string. Splatting a `nil` over the real pointer value loses the original string causing heap leaks. The correct way to re-initialise any array of strings is:

```
for I := Low(TheArray) to
  High(TheArray) do
  TheArray[I] := '';
```

or:

```
for I := Low(TheArray) to
  High(TheArray) do
  SetLength(TheArray[I], 0);
```

Don't forget that local long strings, be they in an array or not, will automatically be initialised as empty strings, just as dynamic arrays are initialised as zero-sized arrays.

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  CurrentPID, FocusedPID: THandle;
begin
  CurrentPID := GetCurrentProcessId;
  GetWindowThreadProcessId(GetForegroundWindow, @FocusedPID);
  if (Win32Platform = VER_PLATFORM_WIN32_NT) and (CurrentPID <> FocusedPID) then
    SetProcessWorkingSetSize(CurrentPID, Cardinal(-1), Cardinal(-1) );
end;
```

➤ *Listing 7: Trimming an application's working set.*

```
function StrToFloatDef(const S: String; Default: Extended): Extended;
begin
  if not TextToFloat(PChar(S), Result, fvExtended) then
    Result := Default
end;
```

➤ *Listing 8: An alternative error-free string to number translator.*