Dynamic Arrays

by Brian Long

Delphi, Borland Pascal and Turbo Pascal before it have always supported arrays: convenient storage structures for holding many values of a specified data type. However, they have always been, rather irritatingly for some, fixed size entities. You always had to know how many elements you would need or, if you didn't know in advance, you were obliged to overestimate.

I'll grant you that these days the business of over-estimation of data requirements might not seem like too much of a problem, what with multi-megabytes of memory being the norm, but in the days of Delphi 1 and before memory was rather more scarce.

Of course, I'm aware that statements such as that just made are hardly future-proofed (yeah, we all hear the stories of the good old days when one kilobyte was a luxury, and 64Kb was deemed to be more than enough for anyone, don't we?), but you get the point.

Over the years, various techniques have evolved to counter this restriction of static arrays, to provide mechanisms for supporting dynamic arrays in one way or another. Some of these have involved using pointers and dynamic memory management. In this article, I am going to be looking at some of these dynamic array mechanisms, hopefully to give you some ideas that you can use in your own application developments.

The examples presented here, whilst not tremendously exciting in comparison to some of the inventive scenarios portrayed in some Delphi books, will hopefully be sufficient to show the syntax required to do the job in hand. In all cases, the dynamic arrays will have `Integer` elements. `Integer`, remember, is a *generic* type. That means it changes, depending upon the underlying operating system's native integer size.

```
type
  PIntegerArray = ^TIntegerArray;
  TIntegerArray = array[0..0] of Integer;
```

➤ Listing 1

```
var
  MyArray: PIntegerArray;
  Loop: Integer;
...
MyArray := AllocMem(DesiredSize * SizeOf(Integer));
{$R-} { Disable range-checking for now }
for Loop := 0 to Pred(DesiredSize) do
  MyArray^[Loop] := Loop;
...
FreeMem(MyArray, DesiredSize * SizeOf(Integer));
```

➤ Listing 2

So, in Delphi 1, `Integer` is a 16-bit signed integer, with a range of -32,768 (-32k or -2^{15}) to 32,767 (32k-1 or 2^{15}-1). Delphi 2, 3 and 4, being 32-bit development systems, define `Integer` to be the same as a `Longint`, -2,147,483,648 (-2G or -2^{31}) to 2,147,483,647 (2G-1 or 2^{31}-1). In the dim and distant future when we have a version of Delphi for 64-bit Windows systems, `Integer` will be defined in terms of Delphi 4's `Int64` type, for example from -9,223,372,036,854,775,808 (-8E or -2^{63}) to 9,223,372,036,854,775,807 (8E-1 or 2^{63}-1).

Heap-Based Array

The first approach involves using dynamic memory management routines to allocate the array on the heap. A normally declared array of a fixed size will reside either in the global data segment, or on the stack (if it is a local variable). This is true of all variables declared in `var` statements or declared as typed constants. When you manually allocate the storage for a variable with the dynamic allocation routines, the space comes from your program's heap. We can allocate as much space as we decide our array requires at runtime.

To get this idea off the ground, we need to realise that heap allocations tend to imply pointers, so we will be forced to use that arguably unpleasant pointer notation. Having got past that point (pun unintended), we can define some types. Since we don't know how many elements our array will require, the array type can be defined with as few elements as we can get away with (which turns out to be one). We also define a pointer type to point to such an array as per Listing 1, and we are nearly there. Notice that I have chosen to define array using element 0. This is an arbitrary choice: I can use element 1 if I wish to.

I still find it interesting that when defining some type (say `TIntegerArray`), and a pointer type that will point to it (say `PIntegerArray`), you can define the pointer type first. The compiler is happy to let you refer to the `TIntegerArray` even though it has not been defined at that stage. But then again, strange things interest me that everyone else typically finds very dull.

The next step is to realise that if we declare a variable of the array pointer type we can then allocate as much memory as we like for it to point to. If we explicitly turn off the compiler's range checking code, then we can access any valid element in the array with appropriate pointer de-referencing, so long as we don't use any constant expressions. If we were to do that the compiler would object at compile-time, spotting the conflict between the element number and the array type defined to only have one element. Listing 2 shows this. Notice that range-checking is turned off with a {$R-} compiler directive. In Delphi 2 onwards you can alternatively use the more descriptive {$RangeChecks Off}.

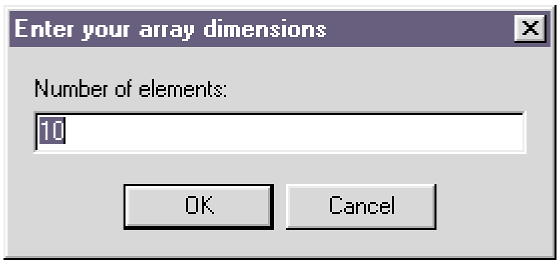
To allocate memory for the array you can use either GetMem or AllocMem. The only difference between them is that GetMem leaves the allocated memory block in an undefined state whereas AllocMem zero-fills it, leaving integers sensibly initialised to zero. Freeing the memory block is the responsibility of FreeMem. In Delphi 1, the second FreeMem parameter was mandatory, but in 32-bit Delphi it is optional (since we wish to free exactly the same amount as we allocated in the first place).

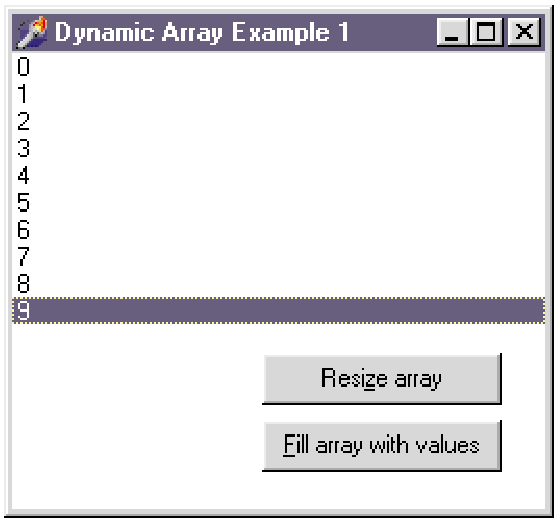
Whilst on the subject of 32-bit Delphi, I bent the truth a little earlier. When I said that because we were using dynamic memory management we were forced to use pointer notation, this is not strictly true from Delphi 2 onwards. The expression MyArray^[Loop] can in fact be written as MyArray[Loop] and the compiler will assume you wish to dereference the MyArray pointer. But since Delphi 1 doesn't support this convenience mechanism I shall continue to use the ^ symbol (called a caret, circumflex, or sometimes simply a hat).

If you wish to resize your array after it has been allocated, you can call ReallocMem. This routine changed quite a lot between Delphi 1 and 2 and so some conditional compilation may be necessary to keep your code version-independent. ReallocMem started life as a function that took three parameters, including the old memory block size and the new size. These days it is a procedure that doesn't need to be told how

➤ *Top: Figure 1*

➤ *Bottom: Figure 2*





big the memory block is, it works that out for itself.

As well as the simple syntax change (see Listing 3), the semantics are slightly different as well. If you reallocate your array and make it larger, Delphi 1's ReallocMem (from the SysUtils unit, where AllocMem lives) leaves all the new elements initialised with zero bytes. On the other hand, the 32-bit ReallocMem (from the System unit, where GetMem and FreeMem come from) leaves the extra memory with undefined values. If you want to leave all the new elements nicely initialised in 32-bit programs, you can call FillChar straight after the ReallocMem.

To test this idea out, a sample project Array1.Dpr is on this month's disk. When the project starts it asks you how many elements to create in the array (see Figure 1). The array is created, each element is filled with a simple value (the element number, as in Listing 2) and displayed in a listbox (Figure 2). There are two buttons on the form, one of which allows you to resize the array. It asks how many elements you would like to resize it to, and then calls ReallocMem to do the job. The listbox's view of the array contents is then refreshed. The second button refills the array with dummy values (remember if you make the array grow in size, some elements will simply have a value of zero) and redisplays the array in the listbox.

Nearly all the other sample projects, which implement dynamic arrays in various other ways, mimic this project's functionality and user interface.

➤ *Listing 3*

```
{$ifdef Win32}
  ReallocMem(MyArray, NewSize * SizeOf(Integer));
  if NewSize > DesiredSize then
    FillChar(MyArray^[DesiredSize], (NewSize-DesiredSize) * SizeOf(Integer), 0);
{$else}
  MyArray := ReallocMem(MyArray, DesiredSize * SizeOf(Integer),
    NewSize * SizeOf(Integer));
{$endif}
```

Heap-Based Array #2

A slight variation on the heap-based array allows you to leave range-checking on if you so wish. Instead of defining an array with one element, you can define an array with the maximum number of elements. Again, you only allocate memory for as many of them as are required at that time. The only difference in the coding approach as described by Listings 1, 2 and 3 is the type section. So with this method Listing 1 becomes Listing 4.

In 16-bit applications, the largest data structure that can be manipulated without added complications (such as selector tiling) is just under 64Kb. In 32-bit programs the limit is just under 2Gb. Listing 4 shows how to compute the largest number of elements you can have in your array on either platform. Again, the fact that my array starts at element 0 is my choice. I could start at element 1, or even 10 if I wanted.

So range-checking can now be left on, and you can also use constant expressions to access the array elements. If you allocate five elements, you can quite validly refer to MyArray^[0], MyArray^[1], ..., MyArray^[4] (remember this syntax would not compile with the previous approach). However, if you programmatically try to access the sixth element (MyArray^[5]), you will have a problem. The compiler will allow this invalid action because element six appears valid to the compiler based upon the type definition. It's down to you to make sure you don't try anything so silly.

Array2.Dpr on the disk acts exactly the same as Array1.Dpr, but uses this modified form of array declaration. Also, before any memory allocation or reallocation, the target size is checked to see if it is less than the calculated maximum size. If it is bigger, an exception is raised. Listing 5 shows the event handler that is triggered when the array resize button is pressed.

➤ Listing 4

```
const
{$ifdef Win32}
  MaxSize = 2147483647 div SizeOf(Integer);
{$else}
  MaxSize = 65535 div SizeOf(Integer);
{$endif}
type
  PIntegerArray = ^TIntegerArray;
  TIntegerArray = array[0..Pred(MaxSize)] of Integer;
```

➤ Listing 5

```
procedure TArray2MainForm.btnResizeArrayClick(Sender: TObject);
var Tmp: Integer;
begin
  Tmp := StrToInt(InputBox('Enter your new array dimensions',
    'Number of elements:', '20'));
  if Tmp > MaxSize then
    raise Exception.Create('Array too big');
 {$ifdef Win32}
  ReallocMem(MyArray, Tmp * SizeOf(Integer));
  if Tmp > SizeOfMyArray then
    FillChar(MyArray^[SizeOfMyArray], (Tmp-SizeOfMyArray) * SizeOf(Integer), 0);
 {$else}
  MyArray := ReallocMem(MyArray, SizeOfMyArray * SizeOf(Integer),
    Tmp * SizeOf(Integer));
 {$endif}
  SizeOfMyArray := Tmp;
  DisplayArray
end;
```

TList

The next possibility on my list is a..., well, a list. A TList to be precise. A TList is designed to represent a list of pointers. These pointers can refer to objects, allocated memory blocks or whatever you like. The pointers are accessed through the Items array property. However, since Items is the default array property, you can treat the list itself like an array. So these two statements would be identical:

```
MyList.Items[0] := nil;
MyList[0] := nil;
```

Items can be added to the list as you like, by using its Add method. This increments the Count property. But to treat a list more like a dynamic array (as opposed to a list), you can set the Count property directly, to dictate how many elements are accessible. Then you can simply use the Items array property to access them. Ignoring Add, Delete, Pack, Expand means that Count will only be changed explicitly by your code, and the values of the list elements won't be shuffled around behind the scenes.

A TList is implemented by way of a heap-based array as discussed previously. If you want to access that array directly, you can use the List property, but each element in the array is still a pointer. In many cases, using pointers in the list to point to other dynamically (or statically) allocated variables is fine. But the example used here wants to store simple integers. The *List Of Numbers* entry in last issue's *Delphi Clinic* explains how to use TLists to store integers by understanding that a pointer takes four bytes. An integer is either two or four bytes (depending which version of Delphi you are using). So the space taken by the pointer can be used to store an integer value (using an appropriate typecast). Additionally, the default TList item

➤ Listing 6

```
MyArray: TList;
...
procedure TArray3MainForm.FormCreate(Sender: TObject);
begin
  MyArray := TList.Create;
  MyArray.Count := StrToInt(InputBox('Enter your array
    dimensions', 'Number of elements:', '10'));
  btnFillArray.Click; {Pretend to push array filling button}
  DisplayArray
end;
procedure TArray3MainForm.FormDestroy(Sender: TObject);
begin
  MyArray.Free;
  MyArray := nil
end;
procedure TArray3MainForm.btnResizeArrayClick(
  Sender: TObject);
begin
  MyArray.Count := StrToInt(InputBox(
    'Enter your new array dimensions',
    'Number of elements:', '20'));
  DisplayArray
end;
```

```
procedure TArray3MainForm.btnFillArrayClick(
  Sender: TObject);
var Loop: Integer;
begin
  for Loop := 0 to Pred(MyArray.Count) do
    MyArray[Loop] := Pointer(Loop);
  DisplayArray
end;
procedure TArray1MainForm.DisplayArray;
var Loop: Integer;
begin
  with ListBox1, Items do begin
    BeginUpdate;
    try
      Clear;
      for Loop := 0 to Pred(SizeOfMyArray) do
        Add(IntToStr(MyArray^[Loop]));
      ItemIndex := Pred(SizeOfMyArray)
    finally
      EndUpdate
    end
  end
end;
```

```
type
  TIntegerArray = class
  private
    FElements: TList;
    function GetElement(Index: Integer): Integer;
    procedure SetElement(Index: Integer; const Value:
      Integer);
    function GetSize: Integer;
    procedure SetSize(const Value: Integer);
  public
    constructor Create(ArraySize: Integer);
    destructor Destroy; override;
    property Element[Index: Integer]: Integer
      read GetElement write SetElement; default;
    property Size: Integer read GetSize write SetSize;
  end; { TIntegerArray }
constructor TIntegerArray.Create(ArraySize: Integer);
begin
  inherited Create;
  FElements := TList.Create;
  FElements.Count := ArraySize
end;
destructor TIntegerArray.Destroy;
begin
  FElements.Free;
  FElements := nil;
  inherited Destroy
end;
function TIntegerArray.GetElement(Index: Integer): Integer;
begin
  Result := Integer(FElements[Index])
end;
procedure TIntegerArray.SetElement(Index: Integer;
  const Value: Integer);
begin
  FElements[Index] := Pointer(Value)
end;
function TIntegerArray.GetSize: Integer;
begin
  Result := FElements.Count
end;
procedure TIntegerArray.SetSize(const Value: Integer);
begin
  FElements.Count := Value
end;
```

➤ *Listing 7*

value of nil, when typecast to an integer, conveniently gives us a sensible default value of 0.

Array3.Dpr re-implements the same dynamic array example program using a TList. Listing 6 shows some bits of code from the project. You can see that resizing the array is simple; assign a new value to Count. Also you can see the typecasting of integer to and from pointer to get values in and out of the TList's elements.

Class With Default Array Property

The trouble with the TList approach is the constant need for typecasting. Since it was the inclusion of an array property marked with the default directive in the TList class that allowed us to treat it syntactically like an array, then there is nothing stopping us defining our own class with a default array property. We can use any internal storage mechanism we like and the class will act just like a dynamic array, with the exception of a call to the constructor to initialise it and to its Free method to destroy it.

The two projects Array4.Dpr and Array5.Dpr implement such classes. Array4's class uses a heap based array whereas Array5 makes it easy for itself by using a TList again. Effectively the class in this project simply hides the TList typecasting from the user. Listing 7 shows the extent of this latter integer array class. Quite simple I hope you agree. Listing 8 has some (frankly pretty senseless) code to show how the class works.

Back in Issue 4, in the *Tips & Tricks* section, I provided a Delphi 1 class that mimicked an array in a not too dissimilar way. Its purpose was to allow you to get an array that could occupy more than 64Kb of memory (the largest single structure size in a Delphi 1 app). The implementation of that class was somewhat more involved as it required Windows APIs to allocate larger blocks of memory, and then the use of selector tiling to jump from one 64Kb segment to another. That particular example did not support resizing.

Dynamic Arrays For Paradox People

If you use the term 'dynamic array' to a Paradox for Windows person, it tends to conjure up specific ideas in their mind. In Paradox, a DynArray is a dynamic array but with a distinct difference to those that we have seen so far.

The elements of a DynArray are indexed not by integers, as with fixed-size Paradox arrays and all the Pascal arrays we have seen so far are (see box-out for more details). Instead they are indexed by strings. In Delphi syntax, this means that instead of writing:

```
MyArray[0] := 100;
```

you could say:

```
MyArray['Hello World'] := 100;
```

Our next implementation is going to try and emulate this behaviour.

```
var
  MyArray: TIntegerArray;
...
MyArray := TIntegerArray.Create(1);
MyArray[0] := 100;
MyArray.Size := 2;
MyArray[1] := 101;
MyArray.Free
```

➤ *Listing 8*

TStringList

The TStrings class, as used by TListBox, TMemo, TQuery, TComboBox, TRadioGroup etc, is an abstract class with various descendant classes implemented in different ways by these components. The generally useful class inherited from TStrings is TStringList. This class manages a collection of strings in memory, where each string can have a pointer associated with it.

In many ways, a TStringList is quite like a TList in that it has a default array property although it is called Strings, and gives access to the contained strings. The pointers are stored in another array property called Objects. However there is a prime difference between it and a TList, which is that you cannot write to a TStringList's Count property to resize it. Generally, the string list only grows when you explicitly add a new string to it.

There is yet another array property of a string list that gives us an alternative way of handling the beast. The Values array property was added to help when working with the contents of INI files or the parameter settings of TDatabase objects. In those scenarios you often deal with strings of the format:

```
Name=Value
```

for example:

```
SERVER NAME=D:\IBData\Employee.GDB
```

In the case of this string, it is considered to represent a string with a name part of SERVER NAME and a value part of D:\IBData\Employee.GDB. The Values array property is a convenient mechanism for either reading or writing a value associated with a given name. When you use this array property, you pass in a string as the array index. If you are reading, Values returns the value part if a string with the specified name part exists, otherwise it returns an empty string. If you are writing, it either updates a string that has the specified name part or creates a new string. So if a string list called List is empty, this line will ensure a string as listed above will exist in the string list:

```
List.Values['SERVER NAME'] :=
  'D:\IBData\Employee.GDB';
```

To work, the string list has to do a lot of string searches and so this approach is never going to be the most efficient storage mechanism in the world, but for small data collections, where indexing by strings is required, it may be a convenient approach.

Array6.Dpr implements another dynamic array class that works by way of a TStringList. Since the array needs to store integers, the class converts to and from strings to fit the requirements of a string list. Since this is a slightly different example, the form this time lets you enter a string index and either give it an integer value or obtain its current value. As you set new values for various array elements, the contents of the string list are displayed in a listbox so you can see the internal data representation (see Figure 3).

Again, just to emphasise, if the string list represented in Figure 3 is called List, then the value of List.Values['This Issue'] is the string value '37'. However Array6 implements a dynamic array class called TIntegerArray (as they all are in these examples). An object called MyArray is declared of that type, and so MyArray['This Issue'] has an integer value of 37.

Variant Array

And now onto the next possible implementation of a dynamic array. We will leave these classes with default array properties behind now. We will also leave Delphi 1 programmers behind. Everything shown so far works in all versions of Delphi. This section only works in 32-bit Delphi (and the next section only works with the new Delphi 4!).

A Variant is a type-unsafe variable, a variable that can be assigned values of many different types. A Variant can also hold a reference to an Automation server and be used to call that server's exposed methods and access its properties in a late-bound fashion. It was added to Delphi 2 primarily to support Automation because

Non-Integer Array Index Types

Many people seem to be under the impression that static arrays need to have their elements indexed by integer numbers. This is a false impression. The syntax in the Delphi help file defines a non-dynamic array type as:

```
array[indexType1, ..., indexTypen] of baseType
```

where each indexType is an ordinal type whose range does not exceed 2Gb in 32-bit or 64Kb in 16-bit programs. So the index type must be ordinal. This means the index elements can be Integer, Byte, Word, Cardinal, Smallint, and Shortint. Okay, so array index numbers 1 to 127 fit into any of those types. But also valid are types Char, Boolean and any enumerated type. Also any subrange type is valid. In fact, an array declared like this:

```
MyArray: array[1..10] of Byte;
```

is using a subrange type as the index specifier. Because of the syntax, if you wanted to declare a 256 element array, given the definitions of Shortint, Byte and Char, you could use any of these definitions:

```
Array1: array[0..255] of Byte;
Array2: array[Byte] of Byte; {Byte is a range of values from 0 to 255 }
Array3: array[-128..127] of Byte;
Array4: array[Shortint] of Byte;
  {Shortint is a range of values from -128 to 127 }
Array5: array[#0..#255] of Byte;
Array6: array[Char] of Byte;
  {Char is a range of values from #0 to #255 }
```

The elements of the last two arrays can be accessed not by numbers, but by single characters, eg Array6['A']. The final example here, to get you into the idea involves a two-element array whose elements are accessed by a Boolean value. Examples of such an array can be declared in either of these two ways:

```
Array1: array[False..True] of String;
Array2: array[Boolean] of String; {Boolean is a range of values
from False to True}
```

These arrays' elements can be accessed using any Boolean expressions, so for example:

```
const
  Captions: array[Boolean] of String = ('Disabled', 'Enabled');
...
Label1.Caption := Captions[Edit1.Enabled];
```

```
MyArray: Variant;
...
procedure TArray7MainForm.FormCreate(Sender: TObject);
var Tmp: Integer;
begin
  Tmp := StrToInt(InputBox('Enter your array dimensions',
    'Number of elements:', '10'));
  MyArray := VarArrayCreate([0, Pred(Tmp)], varInteger);
  btnFillArray.Click; { Pretend to push array filling button }
  DisplayArray
end;
procedure TArray7MainForm.btnResizeArrayClick(
  Sender: TObject);
var Tmp: Integer;
begin
  Tmp := StrToInt(InputBox(
    'Enter your new array dimensions',
    'Number of elements:', '20'));
  VarArrayRedim(MyArray, Pred(Tmp));
  DisplayArray
end;
procedure TArray7MainForm.btnFillArrayClick(
  Sender: TObject);
var Loop: Integer;
begin
  for Loop := VarArrayLowBound(MyArray, 1) to
    VarArrayHighBound(MyArray, 1) do
    MyArray[Loop] := Loop;
  DisplayArray
end;
procedure TArray7MainForm.DisplayArray;
var Loop: Integer;
begin
  with ListBox1, Items do begin
    BeginUpdate;
    try
      Clear;
      for Loop := VarArrayLowBound(MyArray, 1) to
        VarArrayHighBound(MyArray, 1) do
        Add(IntToStr(MyArray[Loop]));
      ItemIndex := VarArrayHighBound(MyArray, 1)
    finally
      EndUpdate
    end
  end
end;
```

➤ *Listing 9*

Automation in Windows makes good use of variants.

It is common to see variants being assigned simple values, such as an integer or a `TDateTime`, but variants can also contain arrays. A variant array (not to be confused with an array of variants) can be set up where each element is of a specified type (which can include the `Variant` type, thereby allowing heterogeneous arrays: arrays whose elements can be of differing types).

Array7.Dpr is the same example as Array1.Dpr to Array5.Dpr but implemented via variant arrays. Listing 9 is much the same as Listing 6, but contains variant array code instead.

The help stresses that variant arrays are slower and bulkier than standard Object Pascal arrays and should only be used in special circumstances. A good reason for using variant arrays is if you wish to exchange non-standard information between two applications, particularly between an Automation client and server application. If you make a variant array of bytes and plug your data in, you can send it across the process boundary with no problem.

If you are looking to use variant arrays, make sure you read up on the `VarArrayLock` and `VarArrayUnlock` routines that allow more efficient access to the contents of certain variant arrays (those whose elements are defined to be of type `Integer`, `Bool`, `String`, `Byte` or `Float`).

Incidentally, the MIDAS technology available in Delphi 3 and 4 Client/Server Suite uses variant arrays to send database data between the client and server applications in exactly the manner described.

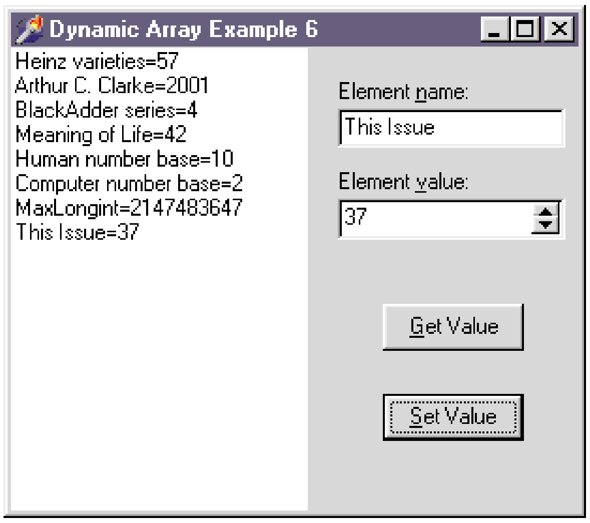
Delphi 4 Dynamic Arrays

The approaches to implementing dynamic arrays so far are all reasonably well known in various circles and many Delphi developers have employed these general techniques over the years. Delphi 4 introduces another option as now dynamic arrays are part of the language. Of course, this is of no use to those who are stubbornly refusing to upgrade because they feel that the cost of the upgrade, only one year after Delphi 3's release, is not worthwhile. But for those who have taken the plunge (or who are thinking of jumping in), we will press on.

To declare a dynamic array, you use similar syntax to an open array subroutine parameter. For example to declare a dynamic integer array you could use:

```
MyArray: array of Integer;
```

The underlying array implementation is modelled on that of huge strings. The arrays themselves are allocated on the heap, and also resized by a call to `SetLength` (which was originally designed for



```
Heinz varieties=57
Arthur C. Clarke=2001
BlackAdder series=4
Meaning of Life=42
Human number base=10
Computer number base=2
MaxLongint=2147483647
This Issue=37
```

Element name: This Issue

Element value: 37

➤ *Figure 3*

strings) and sections of an array can be copied using `Copy` (also applicable to strings). `Length` tells you how many elements are in the array, and you can use the `High` and `Low` functions to return the highest and lowest element index respectively. So to set up this array you could use:

```
SetLength(MyArray, 1);
MyArray[0] := 100;
SetLength(MyArray, 2);
MyArray[1] := 101;
```

To declare a multidimensional dynamic arrays, you just extend the syntax. So a two-dimensional dynamic array can be declared like:

```
MyArray: array of array of
  Integer;
```

`SetLength` is still used to allocate space for the array, but you can use additional parameters for the extra dimensions, eg:

```
SetLength(MyArray, 2, 2);
```

for a 2 by 2 array. You can even allocate the size of each dimension individually to get uneven multi-dimensional arrays. For example:

```
SetLength(MyArray, 2);
SetLength(MyArray[0], 3);
SetLength(MyArray[1], 4);
```

This generates an array with two rows where the first row has three columns, but the second has four.

The sample Array8.Dpr uses Delphi 4 dynamic arrays to reproduce most of the previous examples.

Final word

There is clearly a whole number of choices for implementing dynamic arrays. Some are less efficient than others, but offer certain benefits like ease of use, or string indexing. Also, if you wish to get information out of COM objects then variant arrays should be given special consideration.

I'll leave you to play with the eight examples on the disk.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com