Linking Delphi to S: memory management and multi-dimensional arrays

Mark Bravington
CSIRO, PO Box 1538, Hobart, TAS 7001, Australia

19/1/2003

There is no convenient way to pass from S to Delphi a multi-dimensional array whose dimensions are unknown at Delphi compile time, in such a way that an arbitrary element can be accessed by its indices inside Delphi. Although Delphi provides facilities for static and dynamic multi-dimensional arrays, these do not interface easily with S; the static version requires that dimensions are known at compile time, while Delphi's "dynamic arrays" rely on memory structures specific to Delphi. The unit `USEFUL.PAS` provides a straightforward facility for handling "truly dynamic" arrays of any dimension in Delphi, by setting up arrays of pointers. It's useful in its own right, not just in linking with S.

The unit also provides a convenient way of doing memory allocations for vectors and multi-dimensional arrays, that avoids having to keep track of each individual allocation for later disposal. A handy feature is that numerous "identically-shaped" variables can be allocated with a single call. Disposal of *all* allocations is handled automatically in a single call to a destructor.

Usage

Include `USEFUL.PAS` in your project (so it appears in the `USES` statement). Your Delphi routine (i.e. the one that appears in the DLL export list, and is called from S) should start by instantiating an object of type `Teasy_alloc_object`, called `mymem` for example. Then all the operations of your routine should be performed inside a `WITH mymem DO BEGIN ... END` block. At the start of this block, multidimensional arrays passed in to Delphi from S (as vectors) should be mapped to "truly dynamic" arrays in Delphi with similar names. Your routine should end with a call to `mymem.destroy`, which frees any memory allocated during the routine. (Note that this refers to memory "internal" to Delphi, not to S memory that's allocated with calls to `S_alloc` etc.). For example, here's a routine to calculate the difference between the row and column indices of a two-dimensional integer array:

```
PROCEDURE row_col_diff(
 VAR nrow, ncol: LONGINT;
```

```
 _ar: PLONGINT_ARRAY); STDCALL;
VAR
 mymem: Teasy_alloc_object;
 ar: ARRAY_2L;
 pointless_variable: ARRAY_1D;
 i, j: INTEGER;
BEGIN
 mymem:= Teasy_alloc_object. create();
 WITH mymem DO
 BEGIN
   ar:= alloc_array( _ar, [ncol, nrow], [1,1], SizeOf( _ar[1]));
   pointless_variable:= alloc_array( [27], [5], SizeOf( pointless_variable[ 1]));
   FOR icol:= 1 TO ncol DO
     FOR irow:= 1 TO nrow DO
       ar[ icol][ irow]:= ABS( icol-irow);
   destroy;
 END;
END;
```

There are a few points to note:

1. There are some predefined types in `USEFUL.PAS` which make the job easier. `PLONGINT_ARRAY` just points to an array of `LONGINT` starting at index 1— a convenient way to refer to an S vector. `ARRAY_2L` is one of my "truly dynamic" arrays— specifically, a 2-D array of type `LONGINT`. There are analogues for `DOUBLE, LOGICAL` and (in S but not R) `SINGLE` vectors/arrays, too, and for dimensions of 1,2 and 3. Defining higher-dimensional array types is very easy— see below.

2. When accessing an element, each index comes within its own pair of square brackets (which I think is C-style), rather than separated by commas as for normal Delphi arrays: i.e. we have `ar[3][1]` rather than `ar[3,1]`.

3. The order of indices is REVERSED in Delphi compared to S; e.g. `ar[3][1]` in Delphi refers to `ar[1,3]` in S. (This matches static array declarations in Delphi, and is unavoidable anyway.)

4. The array variable passed into Delphi (`_ar`), needs to be "mapped" onto a new Delphi variable (`ar`) before it can be used; this is accomplished by the call to `alloc_array`. This is because some memory space is required to store pointers, which normally remain invisible to the user. However, the "mapped" variable `ar` refers to exactly the same memory locations as `_ar`.

5. In this example, the array indices start at 1 (that's the `[1,1]` in the call to `alloc_array`), but any value you like can be included here. If you put `[5,3]` into `alloc_array`, then `ar[5,3]` will thereafter refer to the first element in the vector passed from S.

6. You can deal with 1-dimensional arrays, too. The variable `pointless_variable` is a 1-D array starting at index 5 and of length 27— so the last valid index is 31.

7. You need to pass in the size of the elements being remapped. The safe way to do this, is to call `SizeOf`; this keeps your program working even if you change the type of the variable.

8. Note that `pointless_variable` is de-allocated automatically by `destroy`. There are also some invisible memory allocations associated with `ar`, and these are de-allocated automatically too.

Referring to S vectors directly

S vectors (one-dimensional things passed from S) can be referred to directly, of course, without needing to remap them to `ARRAY_1x` (though the latter will work). But, to avoid confusion, always declare S vectors and arrays as `PLONGINT_ARRAY` or `PDOUBLE_ARRAY`, not as `ARRAY_1L` etc. This is because `Psomething_ARRAY` starts at 1, as per normal S usage, whereas `ARRAY_something` starts at 0 (by default). Again, to avoid confusion, I don't recommend using the `Psomething_ARRAY` types to declare Delphi variables that don't come from S. If you are creating new vectors and arrays, declare them with `ARRAY_something` and allocate them with `alloc_array`.

Creating and re-mapping a single variable: `alloc_array`

The routine `alloc_array` is overloaded; it can be called with four different syntaxes. The first distinction is between creating a completely new variable (as with `pointless_variable` above), versus mapping a Delphi variable onto an S variable (as with `ar` above). The second distinction is whether dimensions and starting indices are specified by array constants (as they are above), or by integer vectors (as below); array constants are usually more convenient, but integer vectors are more flexible.

When creating a new variable, you must specify the dimensions, lowest indices, and element size. In all cases, `alloc_array` returns a pointer, which can be treated as the array you want.

To create a new 2-D array with indices starting at zero, `nc` columns, and `nr` rows, use this:

```
newar:= alloc_array( [nc, nr], [0,0], SizeOf( newar[1][1]));
```

Remember that array constants don't have to be constant, exactly: you can have `[2*nc, nr*nc+1]` if you want.

The more cumbersome alternative via integer vectors, requires as a first parameter the number of dimensions :

```
VAR dimar, startar: ARRAY[ 1..2] OF INTEGER;
  dimar[1]:= nc; dimar[ 2]:= nr;
  startar[ 1]:= 0; startar[ 2]:= 0;
  newar:= alloc_array( 2, @dimar, @startar, SizeOf( newar[1][1]));
```

If you are mapping an S array to a Delphi variable, there is an additional first parameter to `alloc_array`: the (pointer to) the S array. The other parameters are as before, and the usage (for the array constant case) is as shown in the example.

Note that, in all cases, it is the *dimension* of the array that is required: i.e. the number of elements, not the highest values of the indices. If you want an array running from 5 to 6, then the call is

```
my_ar:= alloc_array( [2], [5], SizeOf( my_ar[ 1]));
```

Allocating multiple objects in a single call: `alloc_arrays`

If you have many different variables that all refer to arrays of the same shape and type, you can create them with one call to `alloc_arrays`. (As yet, you can't remap several variables in the same call, though.) The first parameter of `alloc_arrays` is always an array constant, containing the addresses of the variables to be created. There are again two versions of the syntax, one with array constants and the other with integer vectors for dimensions and lowest indices; these work as above. For example, to simultaneously set up the variables `myvar1`, `myvar2` and `myvar3` as 2D arrays:

```
alloc_arrays( [ @myvar1, @myvar2, @myvar3], [nc, nr], [1,1], SizeOf( myvar1[1][1]))
```

Note that `alloc_arrays` is a procedure, not a function; it doesn't return a value, but instead modifies things pointed to by its first argument.

Manual de-allocation: `dealloc`

This is rarely necessary because all allocated memory is freed automatically when `destroy` is called, but sometimes it's useful to reclaim large chunks of memory or to change the size of an already-allocated variable. The syntax for one variable is `destroy( myvar)` or, for several variables, `destroy( [@myvar1, @mvyar2])`.

Declaring higher-dimensional array types

The declarations for 2-D and 3-D arrays of `LONGINT` are:

```
ARRAY_2L = ^ZAR_ARRAY_1L;
ZAR_ARRAY_2L = ARRAY[ 0..99] OF ARRAY_2L;
ARRAY_3L = ^ZAR_ARRAY_2L;
```

To define a 4-D array, just copy this two-stage process:

```
ZAR_ARRAY_3L = ARRAY[ 0..99] OF ARRAY_3L;
ARRAY_4L = ^ZAR_ARRAY_3L;
```

and so on for higher dimensions. The number 0 is important, and mustn't be changed. The choice of 99 is arbitrary, and doesn't affect the real size of the array, which can be larger or smaller. However, a larger number makes life easier when debugging; if I'd put `[0..1]` instead, then the debugger complains when I try to evaluate `ar[2][1]`.

Note that there are some other more cumbersome ways of extending to higher dimensions, and some duplicate definitions, in the header of `USEFUL.PAS`. Just ignore these; they're there only for back-compatibility with various other bits of my code.

Deriving a new class from `Teasy_alloc_object`

It's straightforward, and sometimes useful, to derive a new class from `Teasy_alloc_object`, that actually includes all the arrays and dimensions and other stuff that's of interest, encapsulated into a single object. Usually, all you need to do is add the fields, and any methods you want. You might also want to define a special creator that initializes the data structures, though you can also do this "by hand" in the Delphi routine that's called from S.

Persistent objects

Derived classes are particularly handy if you want to preserve a data structure at the end of the call to the Delphi routine. You might want to make several calls to Delphi from S that refer to the same data structure, and you might not want to have to re-create it every time. To enable this, I usually pass an extra `LONGINT` parameter, to be filled with the Delphi pointer to the object:

```
PROCEDURE called_first_from_s(
  VAR persistent_object: LONGINT;
<<other stuff>>);
VAR holder: Tderived_from_easy_alloc_object;
BEGIN
  holder:= Tderived_from_easy_alloc_object. create();
  WITH holder DO <<...>>
  POINTER( persistent_object):= holder;
END;
PROCEDURE called_second_from_s(
  VAR persistent_object: LONGINT;
<<other stuff>>);
VAR holder: Tderived_from_easy_alloc_object;
BEGIN
  holder:= POINTER( persistent_object);
  WITH holder DO <<...>>
```

```
END;
PROCEDURE cleanup_called_by_s(
  VAR persistent_object: LONGINT);
BEGIN
  Tderived_from_easy_alloc_object( persistent_object). destroy;
END;
```

If you do this, you must be careful to ensure that the cleanup routine really does get called from S.

I'm not sure whether the `LONGINT` method is really safe— there might be specific values of `LONGINT` that signal something weird to S or R— but it has worked well for me. A completely safe alternative, is to use an 8-byte character string containing the hexadecimal address of the Delphi object.

Advanced array indexing ("ragged arrays") and internal details

How does an `ARRAY_2L` actually work? It sets up an array of pointers, one for each column of the array; each points to a vector of `LONGINT` that constitutes one column of the array. In higher dimensions, extra sets of pointers to pointers are declared, as needed.

This is a very flexible structure (it's very fast to access, too, though potentially slightly wasteful of memory). It is actually possible for each column pointer to refer to a column of different length, i.e. in a "ragged array". I have actually done this for real, once, to set up an indexing structure for cohorts and years on data defined by age and year. The way to set this up, is to do the allocation in two stages, along these lines:

```
ar:= alloc_array( [nc], [1], SizeOf( ar[1]));
FOR index2:= 1 TO nc DO
  ar[ index2]:= alloc_array( [ nind1[ index2]], [1], SizeOf( ar[1][1]));
```

The real work of `USEFUL.PAS` is done by the heavily-overloaded routine `alloc`, which the preternaturally curious are welcome to investigate. For 1D arrays, this can be used to do memory allocations more succinctly, but I encourage the use of `alloc_array` instead for clarity. Each call to `alloc` creates a record of the allocation, and this chain of records is used by `destroy`, which successively calls `dealloc`.

Errors

AFAIK there are now no bugs in `USEFUL.PAS`— gulp— though many have been weeded out along the way. Sometimes there *seems* to be an error, in that the Delphi debugger gets upset about something happening in `Teasy_alloc_object.destroy`. Invariably (in recent years) I have traced this down to overwriting a pointer somewhere— a user error (me being the user). Bugs like this are notoriously hard to track down; the only strategy that seems to work for me,

is to start by to commenting out almost all the code between `Teasy_alloc_object.create` and `Teasy_alloc_object.destroy`, and then to progressively allow bits of code back in until the error occurs.