arrayClass.pbi

by Stephen Rodriguez.

General.

arrayClass.pbi is a Purebasic source code include file containing OOP classes for implementing various kinds of single-dimensional dynamic arrays in Purebasic programs. It allows the programmer to embed arrays of complex structures within structures, within linked lists or indeed within other arrays etc. There is no real limit to how you can nest these objects.

This software is written in Purebasic 4.1 beta 4 and has been tested on Win XP. It should run on all platforms supported by Purebasic and is completely threadsafe.

Package, licence and terms of use.

This package contains all the relevant source files, two demo programs and this short instruction manual.

The software contained within this package is free to use in any project (commercial or otherwise) or as a learning tool. I do, however, assert my moral right to be identified as the creator of this software (except where acknowledgements are given) and thus ask that due acknowledgement is given within any product/creation in which my source code forms a part. Use the software for any purpose whatsoever.

This software is provided on an *as is* basis, with no warranty either given or implied, meaning that I am not liable for any damage caused by its use (or misuse!) nor by damage caused by other programs based on its source code.

To use "arrayClass.pbi" within a Purebasic program.

Simply ensure that the following command resides within your Purebasic source code file before any attempt is made to create any of the appropriate array objects:

```
XincludeFile "arrayClass.pbi"
```

That's it.

Types of array object.

arrayClass.pbi exports two kinds of array object, namely a 'basic' array object and a more versatile 'structured' array object. Such objects can be created dynamically and there are no limits (other than available memory) to how many such objects a program can create and utilise and indeed how they can be embedded and nested.

A brief outline of the two different kinds of array object follows.

Basic arrays.
These are simple arrays in which 32-bit values (64-bit if compiling with PB x64) are stored. Of course such an array could be used to hold pointers to more complex data structures, perhaps held in a linked list etc. and so they can be put to a whole host of uses.

The following is an example of how to create an instance of a basic array class :

```
MyArrayObject.ArrayObject
MyArrayObject = NewArray(100)
```

This creates a dynamic array containing 101 elements (indexed from 0 to 100).

You can write a value into this array using the SetValue() method, for example :

```
MyArrayObject\SetValue(0, 250)
```

etc.

See the demo program "Basic array class demo.pb".

Structured arrays.
These are more complex objects allowing the developer to store entire structures, even those containing string fields.

The demo program "Structured array class demo.pb" shows how to use such an array.

It is important to understand exactly how such an array works.

When the developer writes a structured variable to an array, the entire structure is copied (including string fields) and placed within the underlying array. This means that, having written such a variable onto a structured array, the developer is then free to modify the original variable, safe in the knowledge that the original fields are preserved within the array.

Also, string fields are handled in such a way that all associated heap memory is allocated and freed automatically.

The only proviso is that all string fields **must be placed before all other fields.**

The following is an example of how to create an instance of a structured array :

```
MyArrayObject.StructuredArrayObject
MyArrayObject = NewStructuredArray(100, SizeOf(MyStructure), 2)
```

This creates a dynamic array containing 101 elements and whose structured type contains two string fields (which must be the first two fields of the structure).

You can write a structure to this array using the SetValue() method, for example :

```
MyArrayObject\SetValue(0, MyVar)
```

where 'MyVar' is a variable of type 'MyStructure' etc.

See the demo program "Structured array class demo.pb".

Methods exposed by both array classes.

The following list of methods are exposed by both array classes :

Destroy()
GetArrayBase.i()
GetUpperBound.i()
GetValue.i()
ReDim.i()
SetValue.i()
ShiftElementsLeft()
ShiftElementsRight()
SwapElements()

although the parameters differ slightly for the different types of object.

The following additional method is exposed by the structured array class :

GetPointerToValue.l()

Destroy().
No parameters. No return value.

It is vital that you use this method when an array object is no longer required. Particularly important for arrays created locally within a procedure as Purebasic will not *garbage collect* such objects automatically.

GetArrayBase().
No parameters. Returns the base address of the memory buffer containing the individual array elements. In the case of a structured array this buffer will contain pointers to the individual structures.

GetUpperBound().
No parameters. Returns the upper bound of the underlying array. Remember that arrays are zero based and contain one more element than the upper bound.

GetValue().
For basic arrays there is a single parameter; namely the zero based index of the element to retrieve. For structured arrays there is an additional parameter which holds the address of the structure to be filled from the element specified by the first parameter.

For basic arrays the return value is that taken directly from the array.

NOTE that for structured arrays whose elements contain string fields, it is not safe to use this method across a dll boundary. Use the GetPointerToValue() method instead. See the section on '*A word on using this library within a dll*' for more details.

GetPointerToValue(). **Structured array class only.**
There is a single parameter, the index of the element to whom a pointer is requested.

Unlike the GetValue() method, this method does not take a structure variable and fill it's fields with those retrieved from the specified array; instead it simply returns a pointer to the requested element etc.

This method is thus considerably faster to execute than GetValue() on a structured array object. **More importantly**, however, is the fact that this method can safely be used across a dll boundary for structured elements containing string fields. See the section on '*A word on using this library within a dll*' for more details.

A word of warning, under no circumstances use this pointer to directly overwrite string fields within the array itself. Use the SetValue() method for this. Other fields are okay to alter via the pointer.
Returns the requested pointer if the index parameter is within the correct range etc.

SetValue().
For both types of array there are two parameters (the second is optional in the case of a structured array).

For basic arrays, the first parameter is the zero-based index of the array element to be written to and the second parameter is the value to be written to this element of the underlying array.

For structured arrays, the first parameter is again the zero-based index of the array element to be written and the second parameter (**optional**) is the address of the structured variable to be written. If this parameter is omitted (or zero) then the underlying element of the array is 'zeroed', i.e. all it's fields are replaced by nulls. This element is still valid however and can be retrieved and written to as usual.

Returns zero if an error, non-zero otherwise. (For structured arrays, the return value is a pointer to the new structure added to the array – the same as returned by the GetPointerToValue() method.)

ReDim().
Attempts to redimension the underlying array whilst preserving the array contents. Takes a single parameter which specifies the new upper bound for the array.

Returns non-zero if the operation was successful.

ShiftElementsLeft().
Takes a single parameter detailing a zero-based index of an element within the array.

This method shifts all elements of the underlying array, starting from the last element and up to the element specified by the method parameter, along towards the beginning (top) of the array. This effectively deletes an element.
The newly vacated 'last' element is filled with a 'null' structure.
The array dimension is not altered etc.

No return value.

ShiftElementsRight().
Takes a single parameter detailing a zero-based index of an element within the array.

This method shifts all elements of the underlying array, starting from the element specified by the method parameter, along towards the end (bottom) of the array. This effectively inserts an element.
The final element is discarded and the newly vacated first element is filled with a 'null' structure.
The array dimension is not altered etc.

No return value.

SwapElements().
For both types of array there are two parameters; namely the indexes of the two elements to be swapped. This is a very fast function as even in the case of structured arrays, the only thingbeing swapped are two pointers.

No return value.

A word on using this library within a dll.

Using the array class within a dll (or placing the code for this array class within a dll and calling the methods from the main application) is, generally speaking, fine. There is however a potential problem involving the GetValue() method if used on a structured array whose elements contain string fields.

The problem will arise if the structured array object crosses the dll boundary!

Imagine, for example, that you've placed the code for the array class into the dll and exported the NewArray() and NewStructuredArray() commands. No problem. You can then call these functions from the host application without a hitch.

Now imagine your host application invokes the GetValue() method in order to fill a structure declared within the host application with the fields of a structure stored within the array by the dll. Again, not a problem,… unless…

Unless the structure contains strings!

Bang! Your computer just exploded!

Well, at best, this will result in a tangible crash and, at worst, result in horrendous memory leaks which may pass unnoticed.

The problem is that the dll and the host application are using different heaps upon which to store their strings etc. It's the same reason why, in order to return a string from a Purebasic dll, you should declare the string (in the dll) as being global.
Consequently, when using the GetValue() method to overwrite the fields of a structure in the host application, the dll functions will be attempting to free strings which belong to a different heap.

Bang!

There are two ways around this :

1) if the host application needs access to structured array objects (containing string fields), then place the code for this library within the host application. Ditto for the dll. Just do not try and mix them and cross the dll boundary etc.

or,

2) do not use the GetValue() method, use the GetPointerToValue() method instead and copy the fields (if required) yourself.

You've been warned!

A note on threadsafety.

This library is as threadsafe as any other Purebasic library meaning that, in itself, it can function fine within a multithreaded application. However, like all libraries exposing some kind of resource, there could be problems if an array object is being used as a shared resource, i.e. being used across multiple threads etc.

This means that if multiple threads are likely to attempt to access the same array object, chaos could ensue! Imagine two threads attempting to re-dimension the same array simultaneously!
Bang!

In such cases it is left to the user of this class library to use mutexes etc. to protect the shared array. No big deal. A future version of this library may well do this automatically.

Stephen Rodriguez.