HashTableClass.pbi

by Stephen Rodriguez.

General.

HashTableClass.pbi is a Purebasic source code include file containing a single OOP class for implementing what is a **very powerful** hash table engine; far far more powerful than my earlier version which is detailed within the Purebasic forums :

http://www.purebasic.fr/english/viewtopic.php?t=24683

This version not only allows you to effortlessly store complete structures within a hash table, but has far less overhead than my previous version in terms of memory requirements and is thus suitable for very large hash tables. Indeed, this was the motivation behind the creation of this new engine.

For those new to the subject, a brief description of hash tables is given below.

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.

What is a hash table?

The $1 000 000 question!

Basically, a hash table acts like an in-memory array; but rather than indexing each entry of the array with an integer, a hash table allows you to use a string value.

E.g. I might wish to store peoples telephone numbers in an array. In order to then lookup Bob Smiths phone number, a hash table would allow something along the lines of

Code:

```
bobstel = TelArray("Bob Smith")
```

The great thing about hash tables, however, is that the speed of data access is far far greater than simply searching an array of strings etc. Indeed, if you've set aside 'sufficient' memory and the 'hashing algorithm' is a 'good' one, then access can be almost immediate and requires a single 'lookup'. (Contrast this with a search through a string array etc!)

Of course there can be 'collisions', but a good library will have contingency plans for such cases where more than one item of data attempt to share the same 'index'. In the case of this new library, if, for example, 3 'keys' generate the same hash, then I shove all the accompanying data into an ordered array which can be (and is!) subsequently searched using a 'binary chop', making access very fast indeed.

Anyhow, allocating sufficient memory and a good '*hashing*' algorithm are the key.

This library currently uses the 'djb2' hashing algorithm which is very simple and rumoured to be very effective at generating hash indexes which are quite uniformly spread out (which is the ideal situation).

I've set the library up, however, so that you can very easily switch this algorithm for another one if you so wish.

The ideal situation is in knowing the maximum number of 'rows' of data which you wish to store and then allocating the corresponding amount of memory. However, if you do not know the max, then simply allocate as much memory as you feel comfortable. The library will automatically handle all 'collisions'.

Where might I use a hash table?

Hash tables can be used in any situation in which data is to be indexed on a string field. Indeed, many commercial grade database management systems (such as MySQL for example) use hash tables to index data in memory which has been retrieved by an SQL query etc.

Imagine an address book application, for example, in which you wish to store data (addresses, phone numbers etc.) on named individuals, but are wondering how best to arrange this data in memory so as to quickly access any individual?
Well, in this case you could use a hash table object and index each item using a combination of surname, middle name and Christian name etc. This would allow you to look up the data for a given individual very quickly indeed.

Imagine writing a language compiler in which you need to build a symbol table containing information on all the source variables and procedures etc. A hash table could be very effective here indeed.

The point is that there are a million and one applications for which hash tables can be utilised.

How does this engine differ from your previous – how does this new engine work?

Skip this section if you wish.

My previous hash table engine used a global linked list to store all the data from ALL the hash tables in an application. That is, if you had 10 hash tables, then all the data for all of these tables was stored in the same linked list! This was convenient because of the need to handle collisions etc. This engine could also only accept a single 32-bit value to store alongside each key.

Now whilst this arrangement did not suffer a loss of performance as you increased the number of hash tables being used, it did however mean that a lot of additional information needed to be stored alongside each item of data added to a hash table. There were the internal linked list pointers (8 bytes per item), the need to store the table ID with each item (since all tables used the same linked list), the need to store the hash itself (again because of the reason given previously) and so on. This was a lot of overhead which hindered the scalability of such hash tables.

And this issue of scalability led me to develop this new engine. I had need of a hash table capable of storing up to a million items!

This new engine no longer uses linked lists of any kind. It also allows you to store any kind of data within a hash table, so you are no longer restricted to a single 32-bit value.

Indeed, such is the sophistication of this engine that you can store strings, structures, structures with string fields etc. and all of this is handled seamlessly by the engine itself, meaning, for example, that all strings are freed automatically etc.

More importantly, however, is the reduction in overhead per item stored within a hash table. Byte for byte, there is a saving of roughly 12 bytes per item (if storing just single 32-bit values etc), which is a lot when you scale this up!

Just as importantly, the performance should be just as good (or bad!) as my previous engine and should out perform my original engine when looking up items sharing the same hash as other items etc.

Also, this new engine will automatically protect individual hash tables from multi-thread access if you request that it does so when you create the hash table object.

Requirements.

The hash table class, if used as a source code include file, requires my OOP class on structured arrays which is detailed in the following Purebasic forum post :

http://www.purebasic.fr/english/viewtopic.php?t=29506

and can be downloaded here :

http://www.purecoder.net/array.zip.

I have included the structured array source files within the download package (zip file) for this library so everything should run fine if you do not alter the default file locations etc.

If you do alter the default location of the files, then make sure you alter the following lines in the HashTableClass.pbi source file in order to point the hash table library at the structured array source files :

Code:

```
IncludePath "..\array\"
  XincludeFile "arrayClass.pbi"
IncludePath ""
```

To use "HashTableClass.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 *"HashTableClass.pbi"*

That's it.

Creating a hash table object.

To create a new instance of the hash table class, you first declare an object of type 'HashTableObject' and use the 'NewHashTable()' function.

Before doing this, however, you need to give some consideration to the data you wish to store within the hash table and to then create a structure to contain this data.

For example, the demo program included within the hash table class package defines the following structure :

Code:

```
Structure MyHashTableItemData
  key$
  SomeText$
  SomeValue.l
EndStructure
```

The name of the structure is not important, you can call it whatever you like, as indeed you can name the fields as you see fit.

There are, however, two provisos, two rules if you like which **MUST** be adhered to :

i) the first field is reserved for the key index of the items.
In the demo code above, I have named this field 'key$' but you can call it whatever you wish. Just ensure to declare it as a string.

ii) All remaining string fields must follow immediately after the key field discussed above.

The demo program included within the hash table class package uses the following code to create a hash table object containing 100 hashes (or elements) :

Code:

```
MyHashTableObject.HashTableObject
MyHashTableObject = NewHashTable(100, SizeOf(MyHashTableItemData), 2, #True)
```

NOTE that this particular hash table is not confined to 100 elements, far from it. You could happily store 10000 objects within this table. The 100 simply limits the number of unique hashes available and thus a maximum of 100 items can be stored in this table before 'collisions' occur.

The first of these two lines of code given above, namely :

Code:

```
MyHashTableObject.HashTableObject
```

declares our hash table object which we have named '*MyHashTableObject*'.

The second line :

Code:

```
MyHashTableObject = NewHashTable(100, SizeOf(MyHashTableItemData), 2, #True)
```

actually instantiates the object (allocates all necessary memory etc.) after which your hash table object is ready to use and abuse as you see fit!

Before moving on, let us examine the function 'NewHashTable()' in a little more detail.

The prototype for this function is as follows :

Code:

```
Declare.i NewHashTable(numberOfHashes, SizeOfEachElement, NumberOfStringFields
[, threadsafeSwitch])
```

The first parameter is self explanatory, as is the second which gives the size (in bytes) of the underlying structure comprising individual items to be stored in the hash table. You can see in my example code above how to use the 'SizeOf()' compiler function to assist here.

The third parameter specifies how many string fields there are in the aforementioned data structure. This number must include the prerequisite key index field.

The fourth and final parameter is optional and is an important consideration if developing a multi-threaded application.

In such cases, you must of course ensure that the Purebasic 'threadsafe' compiler switch is used as per usual.

However, whilst this will ensure that 'threadsafe' versions of all the Purebasic libraries are used, it still does not guarantee that a resulting multithreaded application will itself be threadsafe.

No, this will depend on how the various threads are allowed to access any 'shared resources' etc. By this I mean global variables, global linked lists etc. If multiple threads have access to such resources then chaos could ensue if steps are not taken to synchronise such access.

This is where the fourth parameter is used. Set the value of the '*threadsafeSwitch*' parameter to #True if you wish the library to synchronise threaded access to the hash table object being created, thus protecting the object from simultaneous access from multiple threads etc.

One thing to note, however, is the fact that you do not have to request such protection for a particular hash table in a multithreaded application. Only do so if the application is such that more than one process or thread may require access to the hash table in question etc. **This is an important point.** A multithreaded application in which only the main process requires access to a particular hash table, should not set this parameter to #True as there will be a slight hit in performance if you do so.

The '*NewHashTable()*' function returns a value of non-zero if the hash table object was successfully created. A failure to check this could result in your application wrapping itself around a lamppost in the middle of Trafalgar square!

Methods exposed by the hash table class.

The following list of methods are exposed by the hash table class :

AddItem.i()
ClearItems()
Destroy()
EnumKeys()
GetItem.i()
GetPointerToItem.i()
IsKeyPresent.i()
RemoveItem()

AddItem().
Adds a new item to the underlying hash table.

This method takes a single parameter, namely the address of the structure variable which you wish to add to the hash table. (Strictly speaking, a copy of the structure is made and added to the hash table.)

The first field (string) of this structure must be the item's key value and all remaining string fields must immediately follow this one. See the discussion above and the demo program for more details etc.

If an item with the specified key already exists within the hash table, then the value of the constant :

#HashTableClass_OVERWRITEDUPLICATES

determines whether or not the existing item is overwritten or not. This constant resides towards the top of the main source file : "HashTableClass.pbi" and can be altered to suit your own purposes.

Returns non-zero if the item was successfully added to the hash table. Zero is returned if the item couldn't be added, either because of a memory allocation problem or because, in the case of a duplicate key, the operation is prevented by the value of the *#HashTableClass_OVERWRITEDUPLICATES* constant.

ClearItems().
No parameters. No return value.

Leaves the hash table intact but with all items removed.

Destroy().
No parameters. No return value.

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

EnumKeys().
Allows the host application to iterate through all the keys of the underlying hash table. This is useful if some of the fields being stored contain pointers to memory which needs to be freed by the host application etc.

This method takes two parameters (the second is optional). The first parameter gives the address of a user-defined callback function within the host application. This function is called repeatedly; once for each key and is passed the key and a pointer to the related structure stored within the hash table.
The second parameter is optional and specifies a 32-bit (64-bit under PB x64) application defined value (lParam) which is passed to the callback function.

The EnumKeys callback function should have the following prototype :

Declare.i EnumKeysCallback(key$, *ptr.MyStructure, lParam)

where *ptr will hold the address of a structure of whatever type you are using to store data within the hash table and lParam is the 32-bit application defined value specified in the call to the EnumKeys() method.

On exit, the EnumKeys should return a value of #True if the enumeration is to continue. Returning #False will halt the enumeration.

See the second of the two demo programs for an example of how to create your EnumKeys callback.

NOTES.

i) Under no circumstances use the '*ptr' pointer to directly overwrite string fields within the hash table itself. Use the AddItem() method for this. Other fields are okay to alter via the pointer.

ii) It is safe to delete items being enumerated if you wish (this is because the enumeration is performed in reverse). I wouldn't really advise doing this however!
If removing items within the EnumKeys callback function, then you are advised to only remove the current item whose details have been passed to the callback. **Removing other items may cause some elements to be included more than once in the enumeration!**

iii) Adding new items whilst enumerating all the keys in the hash table is not really advisable as this may cause some elements to be missed out of the enumeration.

iv) One way of calling the object's methods from within the EnumKeys callback (for example to remove the *current* item) is to use the lParam parameter to pass the address of the object itself. Your callback will then need to contain code similar to :

Code:

```
Procedure.i EnumKeysCallback(key$, *ptr.MyHashTableItemData, lParam)
  Protected *MyHashTableObject.HashTableObject
  *MyHashTableObject = lParam

etc.
```

and the underlying object (and thus it's methods) can now be accessed through the *MyHashTableObject pointer.

GetItem().
Retrieve an item (given it's key) from the underlying hash table.

This method takes a single parameter, namely the address of a structure variable whose fields will be filled from the element retrieved from the hash table.

Before calling this method make sure that the 'key' field of the structure contains the key value (string) of the required item.

Returns non-zero if the item specified by the key written to the aforementioned structure was successfully retrieved. If the key does not exist within the hash table then zero is returned.

NOTE that it is not safe to use this method across a dll boundary. Use the GetPointerToItem() method instead. See the section on '*A word on using this library within a dll*' for more details.

GetPointerToItem().
There is a single parameter, namely the key (string) to search for.

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

This method is thus considerably faster to execute than GetItem(). **More importantly**, however, is the fact that this method can safely be used across a dll boundary. 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 hash table itself. Use the AddItem() method for this. Other fields are okay to alter via the pointer.

Returns the requested pointer if the key can be found etc.

IsKeyPresent().
This method tests to see if an item exists within the underlying hash table containing the specified key. It takes a single parameter, namely the key (string) to search for.

Returns non-zero if such an item was located.

RemoveItem().
This method searches for and (if found) removes an item from the underlying hash table containing the specified key. It takes a single parameter, namely the key (string) to search for.

No return value.

A word on using this library within a dll.

Using the hash table class within a dll (or placing the code for this class within a dll and calling the methods from the main application) is, generally speaking, fine. There is however a potential problem involving the GetItem() method.

The problem will arise if the hash class object 'crosses' the dll boundary because the dll uses a separate heap upon which it stores it's string variables. Situations could arise, therefore, in which the dll is 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 hash table objects then simply 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 GetItem() method, use the GetPointerToItem() method instead and copy the fields (if required) yourself.

You've been warned!

Stephen Rodriguez.