stackClass.pbi

nxSoftware.

General.

stackClass.pbi is a Purebasic source code include file containing OOP classes for implementing various kinds of stacks in Purebasic programs.

This software is written in Purebasic 4.4 and has been tested on Win XP and Vista. 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 "stackClass.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 stack objects:

```
XincludeFile "stackClass.pbi"
```

That's it.

Types of stack object.

stackClass.pbi exports two kinds of stack object, namely a 'basic' stack object and a more versatile 'structured' stack 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.

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

Basic stacks.

These are simple stacks in which 32-bit values are pushed and popped. Of course such a stack 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 stack class :

```
MyStackObject.StackObject
MyStackObject = NewStack(100)
```

This creates a dynamic stack containing 100 elements.

You can push a value onto this stack using the Push() method, for example :

```
MyStackObject\Push(250)
```

etc.

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

Structured stacks.
These are more complex objects allowing the developer to push (and pop) entire structures, even those containing string fields.

The demo program "Structured stack class demo.pb" shows how to use such a stack to quickly (and effortlessly) swap two structured variables.

It is important to understand exactly how such a stack works.

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

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 stack class :

```
MyStackObject.StructuredStackObject
MyStackObject = NewStructuredStack(100, SizeOf(MyStructure), 2)
```

This creates a dynamic stack containing 100 elements and whose structured type contains two string fields (which must be the first two fields of the structure).

You can push a value onto this stack using the Push() method, for example :

```
MyStackObject\Push(MyVar)
```

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

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

Methods exposed by both stack classes.

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

Destroy()
NumberOfElementsPushed.i()
Peek.i()
Pop.i()
Push.i(value)
ResetStack()
ResizeStack.i()
SizeOfStack.i()

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

Destroy().
No parameters. No return value.

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

NumberOfElementsPushed().
No parameters. Returns the number of elements pushed onto the underlying stack.

Peek().
This method returns the contents of the specified stack element without adjusting the stack pointer etc. That is, it does not perform any kind of 'pop' action.

For basic stacks there is a single parameter, namely the (zero based) index of the element whose contents are to be retrieved. For structured stacks there is an additional parameter which holds the address of the structure to be filled from the structure found at the element specified by the first parameter.

For basic stacks the return value is that retrieved from the specified element of the stack, whilst for structured stacks a return of zero indicates an error (index out of bounds).

NOTE that the 'index' parameter, being zero-based, must be a value within the range 0 to NumberOfElementsPushed() – 1.

Pop().
For basic stacks there are no parameters. For structured stacks there is a single **optional** parameter which holds the address of the structure to be filled from the structure on the top of the stack. If no value is given for this parameter then the stack is popped and the resulting structure is discarded.

For basic stacks the return value is that taken from the top of the stack, whilst for structured stacks a return of zero indicates an error (stack empty).

Push().
For both types of stack there is a single parameter; the value to be pushed (for a basic stack) or an address of the structure to be pushed (for a structured stack).

Returns zero if an error.

ResetStack().
No parameters. No return value.

Purges all items from the stack but otherwise leaving the stack intact for continued use.

ResizeStack().
A single parameter specifying the new size for the underlying stack. Returns zero if an error.

This method attempts to resize the underlying stack whilst preserving the contents of the stack. Obviously if a stack is reduced in size then it is possible that (depending upon how many elements have been pushed onto the stack) some elements will be lost from the top of the stack.

SizeOfStack().
No parameters. Returns the size (in elements) of the underlying stack.