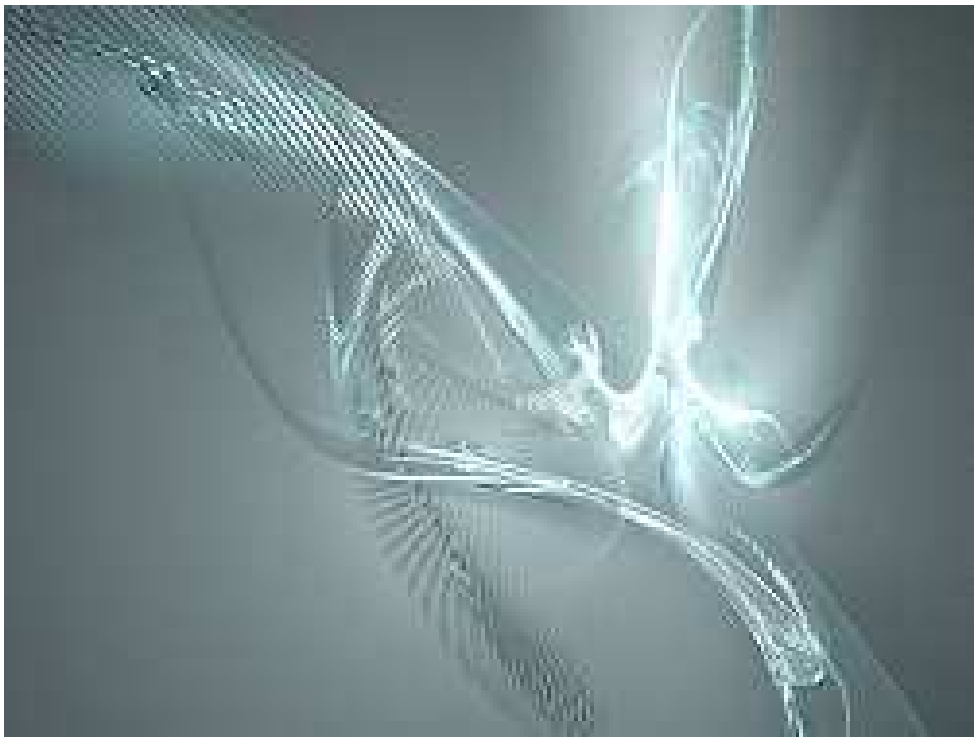

Windows programming with

Purebasic

OOP

In this tutorial we discuss how to use basic **Object Oriented Programming** (OOP) techniques in Purebasic, using nothing more than the basic tools provided by Purebasic itself.

It is assumed that the reader is familiar with basic OOP techniques as this tutorial does not seek to cover such ground. In particular, you will need some familiarity with classes, objects, methods and member variables (very simple really!)

This tutorial is for ***intermediate* to *advanced*** users of the Purebasic language for it requires, in particular, a detailed knowledge and understanding of Purebasic style pointers. Please make sure you know how to use pointers before attempting this tutorial.

What is OOP?

For more information regarding OOP theory, take a look at Wikipedia or some other detailed source.

Object Oriented Programming encapsulates an entire programming paradigm whereby applications and complex programming tasks are broken down into 'classes' and 'objects' designed (as far as is possible) to work independently of one another.

Each object contains all of the data central to its being and houses all of the programming 'logic' required to work with that data and its own internal structures etc. As far as the client application is concerned, an object is a completely self-contained little 'box' which can be called upon at any time to perform some task or other upon the data at its core. The client application need not care about how the 'object' performs its allotted tasks, or in what form it stores its data etc. but needs only know what the object can do and what information needs to be given to the object.

Contrast this with the more 'traditional' procedural style of programming in which the design of an application would typically be driven by the tasks required by the application rather than by the data it is charged with working upon. In OOP, 'tasks' and data are not completely separate entities, but are intertwined in that an object will carry both the code and the data required for it to fulfil its function.

This of course all adds nicely to code 'reuse' in that well designed objects (or more precisely, 'classes') can be used in many different applications (if appropriate).

Of course, this tutorial is not about to become bogged down in the general theory of OOP and so our quick overview stops before it even gets started! We are concerned purely with making use of OOP techniques in Purebasic with the minimum of fuss and the maximum gain!

Why use OOP in Purebasic?

There's no real answer to this other than to say that, like most things in life, it is a personal choice when deciding to use OOP or not?

Personally, I use what I would call 'simple OOP' (perfect for Purebasic!) as an aid for program design and readability. Maintaining my OOP code is one hell of a lot easier than maintaining my spaghettified non-OOP code!

When creating complex libraries for other developers to consume, then these days I will strive to offer up an OOP interface for the library (as opposed to a flat c-style interface) as I find using an OOP based library far easier than the alternative. Such libraries are generally self-contained entities and thus, in my opinion, are screaming out for a publicly accessible OOP interface to reinforce this fact. Again, personal preference as maybe, but it is serving me well.

With more general applications (without any kind of public interface for fellow developers) I will use OOP to wrap up data (and related

functionality) wherever possible. A 50000 line application is a damn sight easier to follow this way – at least for me!

What are Purebasic's capabilities in terms of OOP?

A 'class' is a 'template' for an object containing details on member functions (methods), and member variables etc. Kind of like a Purebasic structure which lists the member fields belonging to a user-defined data type etc.

The first thing to note here is that Purebasic is NOT inherently an OOP language; meaning that there is no way of simply defining a 'class' in the same way that we can in, say, Visual Basic (or c++ or Java etc.)

Defining a class in Purebasic requires that we undertake a little extra work ourselves, performing some of the tasks that an OOP language such as VB would perform for us behind the scenes (so to speak). Nothing too arduous mind!

When I say that there is '*…no way of simply defining a class…*' then I am being a little economical with the truth as there do exist a few '*pre-processors*' which do give this functionality to Purebasic. Such third-party 'tools' as they exist are beyond the scope of this tutorial, however, as they are not a part of the Purebasic product itself. We are concerned here with what can be achieved with the official Purebasic product as it is.

Rather than talking about what Purebasic cannot readily achieve in terms of OOP support, let me instead list those things which can be done without too much work; the most important things which give us a clean and simple OOP 'framework' within which to work.

- methods (member functions) with a *this (or *self) parameter
- private variables (which, when combined with suitable 'get' and 'set' methods can double as 'properties')
- simple inheritance
- a simple means of over-riding methods

Very simple, very efficient and, need I say it, very effective!

Okay, there are no automatic constructors or destructors or virtual methods and the developer is generally responsible for disposing of any memory used in creating an object etc. but none of these '*shortcomings*' need hinder the developer in any serious way. Constructors and destructors can easily be simulated. Virtual methods do not apply because we cannot cast derived objects as being an instance of a base class anyhow (Purebasic does not allow explicit casting of structured types) and as for garbage collection? Well, being responsible for disposing of memory ourselves not only gives us the greatest flexibility, but it does force a certain rigor upon our coding which is never a bad thing in my opinion! Nope, this is the way I likes it!

A developer coming from Visual Basic (say) might well scoff at the apparent limitations to the OOP capabilities of Purebasic, but let me just say that with simplicity comes clarity. Purebasic hides nothing and does nothing to obfuscate the true nature of the language and of OOP in particular. I repeat the fact that Purebasic is not an OOP language by

design; it's just that it does offer the means to easily add a basic yet very effective style of OOP to our programs (and the design of our programs) which for my mind is a nice compromise between procedural and 'true' OOP languages.

An outline of our first class...

Without further ado then; let us outline a basic class, it's methods and 'properties' before turning our attention to just how we implement the class in Purebasic.

I have decided upon a 'rectangle' class. One which **encapsulates** the business of storing a rectangle's length and width and allows us to calculate it's area and the length of a diagonal etc.

A simple enough class, but one for which the underlying implementation will carry through to far more complex classes without any noticeable increases in difficulty.

The following is **not** valid Purebasic code and is given for illustrative purposes only :

```
CLASS Rectangle:

  ;Methods
        Area.d()
        Destroy()
        IsSquare.i()
        LengthOfDiagonal.d()

  ;Properties (Get/Set)
        length.d
        width.d

  ;Private data
        length.d
        width.d

ENDCLASS
```

Figure 1 – our basic rectangle class

There you are, a class with two properties and four methods. Indeed the pseudo-code above could *almost* have come from a VB program (almost, but not quite of course!)

Remember that Purebasic will not automatically destroy our objects when they are no longer in 'scope'; that is our job!

Note in particular the Destroy() method which is used to free all memory (as appropriate) whenever an instantiated class object is no longer needed. In Purebasic it is vital that we remember to provide some means of disposing of an object once it is no longer required etc.

Before discussing how to implement the above class in Purebasic, we first make a slight alteration to the above pseudo-code just to bring it more in line with the Purebasic way of thinking (and doing!)

You see in VB, for example, a class 'property' is treated rather specially in that 'GETting' and 'SETting' occur almost transparently. We unfortunately have no such luxury with Purebasic and instead have to convert our properties into appropriate GET and SET methods :


NOTE. Arranging methods alphabetically is just my personal preference!

```
CLASS Rectangle:

  ;Methods
        Area.d()
        Destroy()
        GetLength.d()
        GetWidth.d()
        IsSquare.i()
        LengthOfDiagonal.d()
        SetLength.i(newLength.d)
        SetWidth.i(newWidth.d)

  ;Private data
        length.d
        width.d

ENDCLASS
```

Figure 2 – our rectangle class modified for Purebasic

You see now why we have the two quantities listed above in the 'Private data' section?

So, how do we 'translate' the above pseudo-code into working Purebasic code?

The answer is; 'slowly, one step at a time!'

If the process as described in the sections below seem rather complex and convoluted then rest assure they are!

Uhm, what I mean is that things may seem complex at first, but with practice and some appreciation of the need for 'virtual tables' (which will come with experience) you will soon see how simple it all is in practice. Console yourself as well in the knowledge that the steps you must take are simply those which the VB compiler (for example) undertakes behind the scenes, hiding the programmer from the real work behind OOP. I rather like the idea of knowing what occurs 'under the hood' – makes me feel quite superior in a way! ☺

We begin with a false start!

How NOT to implement a class in any language!

Okay we have our class design shown above in figure 2. How do we translate our design into Purebasic code?

A first, and not unreasonable attempt, might be to wrap the class within a Purebasic structure, using pointers to functions (or prototypes) for our methods. This idea is shown in figure 3 below.

```
Structure RectangleObject
  *ptrArea
  *ptrDestroy
  *ptrGetLength
  *ptrGetWidth
  *ptrIsSquare
  *ptrLengthOfDiagonal
  *ptrSetLength
  *ptrSetWidth
  length.d
  width.d
EndStructure
```

Figure 3

Looks perfectly feasible and indeed this is perfectly valid Purebasic code (try it and see!) It also seems to satisfy the requirement for '*encapsulation*' laid down by the OOP paradigm in that the structure combines the underlying rectangle object's private data (length and width) with the methods (functions) required to process that data as appropriate.

With this code, we would 'instantiate' a rectangle class by simply defining a variable of type 'RectangleObject', setting it's function pointers to point at some suitable functions and voila; we have a rectangle object!

Indeed that is exactly what the following code does, try it!

NOTE how the pointers are used in this code.

```
Structure RectangleObject
  *ptrArea
EndStructure

Procedure.i MyAreaFunction()
  MessageRequester("Heyho!", "Hello from the Area() method!")
EndProcedure


MyObject.RectangleObject
  MyObject\ptrArea = @MyAreaFunction()

;Call the Area() method of the MyObject object.
  CallFunctionFast(MyObject\ptrArea)
```

Figure 4 – a rectangle 'object' with just a single method

So what is wrong with this – apart from the fact that anything using the CallFunctionFast() function is messy of course? Why is this an example of how not to implement an OOP class?

A good question indeed, and one with a mighty fine answer! Actually two mighty fine answers! The simplest being the fact that no programming language (at least no sensible one!) implements OOP in quite this manner.

The more precise answer has to do with another OOP 'buzz word', namely '*inheritance*'.

Suppose, for the sake of simplicity, we wish to create a new class based upon our rectangle class; say 'NewRectangle', and to this new class we wish to add an additional method, say MakeIntoSquare.i() which turns the underlying rectangle into the largest square able to reside within the original rectangle.

This, in OOP speak, is an example of 'polymorphism'!

To do this we would need to adjust the above structure by adding an extra function pointer (e.g. *ptrMakeIntoSquare) and this will cause the structure to physically change. In particular, the offsets of the private data fields (length and width) will invariably change and this is a big no no for OOP! The problem being that by physically altering the above structure, we can no longer claim that a NewRectangle object is also an instance of a Rectangle object because it is using a different class structure! This is undesirable because we wish all instances of NewRectangle objects to still remain Rectangle objects as well!

So, how do we proceed? How **should** we translate our class design into Purebasic code?

The answer to this follows in the next section.

How to implement a class in Purebasic – Virtual Tables!

To avoid beating around the bush, take a look at what is the correct way to declare our rectangle class in Purebasic (correct in the sense that this IS the method which underpins most OOP implementations – such as COM).

```
Structure _RectangleClassTemplate
  *vTable
  length.d
  width.d
EndStructure
```

Figure 5 – a class template for the 'Rectangle' class

Now doesn't that look one hell of a lot better?

The reason we have renamed this structure will become clear when we introduce 'interfaces' later on in this tutorial.

Aside from the 'fishy' looking field named *vTable, we can clearly see our private data members 'length' and 'width'. We therefore wonder what has happened to the function pointers?

Well, that is the purpose of the new *vTable field, which is simply a pointer to a table whose entries point to our member functions, allowing us to remove the function pointers themselves from the class structure and which in turn, of course, solves the problem discussed above with our first attempt at a class structure (figure 3).

Our new class structure, along with the *vTable field, is represented quite nicely in the following diagram.

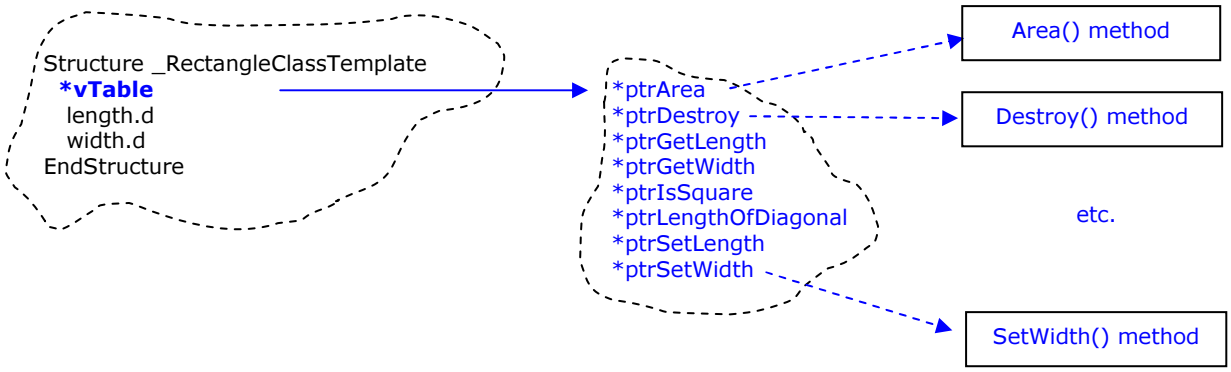

*Figure 6 – showing how *vTable points to our '**virtual table**' of function pointers*

I will say right now that, when we have all of this in place, calling our method functions is made very very simple (and painless) through the use of what is called an '***interface***', something which Purebasic offers natively. The point is that we will not need to use the mucky CallFunctionFast() and things are actually much simpler than they probably look right now! ☺

Right, let us get this down in code.

Open a new Purebasic source file for holding the rectangle class description and methods. Call it "rectangleClass.pbi".

First type in our class template structure shown in figure 5 above.

Now for each of the eight methods listed in figure 2, type an empty function. We will fill in the code later on after sorting out the virtual table.

NOTE THAT each method function takes, as a first parameter, a *this parameter. More precisely, it takes a pointer to the underlying _RectangleClassTemplate structure.

For example, our Area() method, which takes no additional parameters, will look like :

```
Procedure.d RectangleClass_Area(*this._RectangleClassTemplate)

EndProcedure
```

Figure 7 – an empty Area() method implementation

NOTE that you can name the function anything you wish because we will rarely call this method function explicitly. I have chosen a nicely descriptive name of 'RectangleClass_Area' but you can use whatever you like.

The full list of empty method functions is shown in figure 8.

```
Procedure.d RectangleClass_Area(*this._RectangleClassTemplate)

EndProcedure


Procedure RectangleClass_Destroy(*this._RectangleClassTemplate)

EndProcedure


Procedure.d RectangleClass_GetLength(*this._RectangleClassTemplate)

EndProcedure


Procedure.d RectangleClass_GetWidth(*this._RectangleClassTemplate)

EndProcedure


Procedure.i RectangleClass_IsSquare(*this._RectangleClassTemplate)

EndProcedure


Procedure.d
RectangleClass_LengthOfDiagonal(*this._RectangleClassTemplate)

EndProcedure


Procedure.i RectangleClass_SetLength(*this._RectangleClassTemplate,
newLength)

EndProcedure


Procedure.i RectangleClass_SetWidth(*this._RectangleClassTemplate,
newWidth)

EndProcedure
```

Figure 8 – empty method function implementations

Now for the virtual table itself.

The virtual table (as indicated in figure 6) is simply a table of pointers to our method functions and if, as is the case with our rectangle class, the method functions are to be shared by all instances of our class, then we can simply embed this table within a datasection.

Add the following code to the bottom of "rectangleClass.pbi".

```
DataSection
  VTable_RectangleClass:
    Data.i @RectangleClass_Area()
    Data.i @RectangleClass_Destroy()
    Data.i @RectangleClass_GetLength()
    Data.i @RectangleClass_GetWidth()
    Data.i @RectangleClass_IsSquare()
    Data.i @RectangleClass_LengthOfDiagonal()
    Data.i @RectangleClass_SetLength()
    Data.i @RectangleClass_SetWidth()
EndDataSection
```

Figure 9 – our virtual table housed within a simple datasection

We now have our rectangle class in a 'bare bones' form!

Simple really. Sure more work than would be the case with VB, for example, but then with VB you are not allowed to see these virtual tables for yourself; and that is something which I find somewhat disconcerting at times!

Aside from filling in the code for the as yet empty method functions, all that remains now is the business of creating new instances of our rectangle class and that of invoking our method functions etc. which are the subjects of the next two sections.

Creating new instances of our rectangle class

It goes without saying that Purebasic has no NEW command in that we cannot simply instantiate a new instance of our rectangle class with something like :

```
MyObject = NEW(RectangleObject)
```

Instead we have to create such a function ourselves which we shall add to our "rectangleClass.pbi" source file.

The job of this function is simply to allocate memory for a _RectangleClassTemplate structure and set the *vTable pointer to point to our virtual table (within the datasection shown in figure 9).

This function also doubles as a 'class constructor' in that it is free to set any properties/member variables as appropriate.

Add the following code to your source file so that it sits just beneath our structure definition.

```
Procedure.i NewRectangleObject(length=0, width=0)
  Protected *object._RectangleClassTemplate
  ;Attempt to allocate memory for a new class template.
    *object = AllocateMemory(SizeOf(_RectangleClassTemplate))
    If *object
      ;Make sure the *vTable field points to our virtual table.
        *object\vTable = ?VTable_RectangleClass
      ;Initialise the length and width.
        *object\length = length
        *object\width = width
    EndIf
  ;Return a pointer to our object.
    ProcedureReturn *object
EndProcedure
```

Figure 10 – our object 'constructor' function

The above function is the only function from the "rectangleClass.pbi" source file which we would call directly. All of the (as yet empty) method functions will be called through the newly created object itself.

After allocating the required memory and setting the virtual table etc. the above function, if successful, returns a pointer to our newly created object.

The only question now is exactly how do we call our object's methods?

The pointer returned by the above function is a pointer to a structure variable (which is just a chunk of memory) and offers us no way, as yet, of invoking the object's method functions.

The answer is the subject of the next section.

It is always sensible to add code to your Destroy() method as you code the NewObject() function etc.

Before that, however, let us add code to the Destroy() and GetLength(), GetWidth() methods.

```
Procedure RectangleClass_Destroy(*this._RectangleClassTemplate)
  ;All we need do here is free the memory previously allocated for this
object.
    FreeMemory(*this)
EndProcedure


Procedure.d RectangleClass_GetLength(*this._RectangleClassTemplate)
  ProcedureReturn *this\length
EndProcedure


Procedure.d RectangleClass_GetWidth(*this._RectangleClassTemplate)
  ProcedureReturn *this\width
EndProcedure
```

Figure 11 – code for three of our method functions

Note how we use the *this pointer to access the private member data for the underlying object; e.g. *this\length etc.

Invoking method functions through an object

With our newly instantiated rectangle object to hand (as returned by our NewRectangleObject() function) we simply need to know how to invoke the method functions : Area(), Destroy(), … etc.

Well, this is the job of an ***interface*** which lies at the heart of most OOP implementations.


An interface is actually a pointer leading (indirectly) to our virtual table.

An interface is, to all intents and purposes, a list of methods lying at the heart of a virtual table. In the case of our rectangle class, it is a list of the methods shown in figure 2.

In many ways an interface is simply the public face of our OOP class. It is what our client applications refer to when creating new instances of our class and it is what Purebasic itself uses when ploughing through our object's virtual table looking for the address of some method function or other.

Indeed, an interface is the usual starting place when beginning the code for a new class (as opposed to the approach forced on me by this tutorial!)

So, to complete our class implementation, add the following code at the top of your "rectangleClass.pbi" source file.

```
Interface RectangleObject
  Area.d()
  Destroy()
  GetLength.d()
  GetWidth.d()
  IsSquare.i()
  LengthOfDiagonal.d()
  SetLength.i(newLength)
  SetWidth.i(newWidth)
EndInterface
```

Figure 12 – our interface definition for the rectangle class

Note how each method prototype includes the return type, excludes the *this parameter (which Purebasic kindly fills in automatically behind the scenes) but includes all other parameters (such as newLength).

And we are done!!!

Let us test this by creating a couple of rectangle objects; setting their dimensions and retrieving the dimensions.

Testing our rectangle class

Create a new Purebasic source file and call it "test.pb".

Add the following code which simply creates two rectangle objects; sets the first to have a length of 10 units and the second with a length of 20 units. We then use the GetLength() method to retrieve these different lengths.

Finally, we use the Destroy() method to dispose of both objects when we no longer require their services.

```
XIncludeFile "rectangleClass.pbi"

;Define two rectangle objects.
;Note that we declare each variable to match the interface type :
'RectangleObject' in this case.
  rect1.RectangleObject
  rect2.RectangleObject

;Create the two objects. First with length and width = 10, and the
second with length and width = 20.
  rect1 = NewRectangleObject(10, 10)
  rect2 = NewRectangleObject(20, 20)

;Retrieve the lengths.
  length.d = rect1\GetLength()
    Debug "Rectangle 1 has length " + StrD(length, 2) + " units."
  length.d = rect2\GetLength()
    Debug "Rectangle 2 has length " + StrD(length, 2) + " units."

 ;Destroy the objects when finished.
  rect1\Destroy()
  rect2\Destroy()
```

Figure 13 – "test.pb"

Very nice if I say so myself! ☺

Note how this example illustrates just how the two aforementioned objects each have their own member data in that each has it's own length and width values etc.

Accompanying this document is a completed "rectangleClass.pbi" file (with all method functions completed).

The final section of this document gives a few additional pointers and notes etc.

Final thoughts...

We have obviously only scratched the surface here of what is possible using OOP with Purebasic. Granted, we do have to roll up our sleeves and involve ourselves with the business of setting up our virtual tables etc. but these are very simple constructs and easily maintained.

Of course, since Purebasic is not specifically an OOP language, it offers little assistance with things like garbage collection of objects or, for example, reference counting (where appropriate) or even in copying objects.

For example, consider the following code where we instantiate two rectangle objects and then attempt to place a copy of rect2 in rect1.

```
XIncludeFile "rectangleClass.pbi"

;Define two rectangle objects.
;Note that we declare each variable to match the interface type :
'RectangleObject' in this case.
  rect1.RectangleObject
  rect2.RectangleObject

;Create the two objects. First with length and width = 10, and the
second with length and width = 20.
  rect1 = NewRectangleObject(10, 10)
  rect2 = NewRectangleObject(20, 20)

;Attempt to copy rect2
  rect1 = rect2
```

Figure 14

Problems problems!

First, note that we did not destroy the original rectangle object pointed to by the rect1 variable. This of course results in an unreported memory leak.

Second, we might wonder what the actual result of the assignment `rect1 = rect2` is? Does it perform a 'deep copy' of the underlying object memory or does it simply make rect1 point at the rect2 object (a shallow copy)?

Ah, you guessed it; it is indeed a shallow copy and without any kind of reference counting!

A 'deep copy' requires the developer to take steps to copy the underlying physical memory etc.

Before the assignment, each variable contains a pointer to it's underlying object memory and so, from Purebasic's point of view, each variable is simply a 32-bit (or 64-bit) integer. The assignment then results in the 32-bits from variable rect2 being copied into rect1. It therefore does not perform a 'deep copy' of the underlying object memory.

This does mean, for example, that issuing the command rect1\Destroy() will effectively invalidate both rect1 and rect2 in one foul swoop!

The point is that we do need to take a little care when thinking about an object's lifetime etc.

Of course, with the ability to synthesise and replicate many OOP techniques with Purebasic, only a complete idiot would claim that we have access to the full OOP toolkit. For example, abstract classes have little meaning for Purebasic and an attempt to implement multiple inheritance would likely send one completely bananas! ☺

No, what we have is a very 'clean' and simple means of applying the OOP methodology to our projects, but that is very much in keeping with the 'Purebasic way'!

And I say again that if OOP is your thing, but you are new to virtual tables and interfaces, then do not despair, for by the time you have created a couple of classes of your own, you will appreciate just how easy it all is to implement. That or your money back by George! ☺

Interfaces.

As noted above, the public face of any class we create is the **interface**.

In terms of using ready made classes, the interface really is the 'be all and end all'.

For example, any application wishing to use our rectangle class will need just two things. Firstly, it needs access to our NewRectangleObject() function (in order to create new instances of the rectangle class), and secondly it needs a copy of the appropriate interface definition so that the Purebasic compiler can locate each method function in the underlying virtual table etc.

And that's all the application would need.

You see how this can lead to the creation of OOP dll's (for example) in Purebasic; something which I have done on more than one occasion. (That way, the dll is secure from prying eyes since no amount of poking and peeking will ever reveal the method names and/or details etc. because the methods are not exported!)

Even Visual Basic will represent it's classes through an interface type mechanism.

One thing to note with interfaces is that the order of the method functions as listed between the Interface / EndInterface statements must match exactly the order of the function pointers in the virtual table. In the case of our rectangle class, our interface listing must match the order of the function addresses listed in the datasection.

Private member variables.

With our basic rectangle class example, each object had it's own copies of the length and width member variables. No rectangle object can access the member data belonging to another instance of the rectangle class (unless it has access to the object variable itself).

Now, it goes without saying that you are of course free to use as many different types of member fields as you see fit and of any data type going. Integers, floats, pointers, other objects, static arrays, strings, … Anything!

The only problem (generally affecting dynamic string fields) is one of garbage collection. If, as with the rectangle class, you create new instances of your classes by using AllocateMemory() (as opposed to using a linked list for example), then you are left with the problem of not only freeing this block memory when disposing of individual objects, but of also disposing of the individual member fields in cases when Purebasic will not do this for you!

In the case of our rectangle class, all of our member fields were simple integers (length and width) and were freed automatically when we freed the object memory itself because they formed part of that memory.

However, in the case of pointers and strings (which are pointers anyhow) Purebasic has no way of freeing the additional memory pointed to by the pointers! (This is because Purebasic pointers are not 'strongly typed' and can point to any chunk of memory and for any purpose!) The upshot is that we must take steps to free any additional memory ourselves. In the case of pointers, no problem; we just issue additional FreeMemory() commands (assuming that we allocated the underlying memory ourselves that is). In the case of dynamic strings, however, we have to cheat!!! See the Purebasic forums for details on how to free dynamic structure strings.