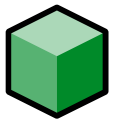

REALbasic 5

Compiler Update

REALbasic 5 introduces a brand-new compiler engine. This document describes the language features and implementation improvements in the new compiler.

Displaying multiple compiler errors at once

The new compiler has a preference to display multiple compile errors at once. This option is a check box in the "Build Process" pane of the REALbasic preferences. When the compiler has displayed an error, double-clicking on an item in the list will display the original item declaration or source code.

For Each loop

This version of the `For` loop allows you to iterate through each element of a one-dimensional array. Example:

```
Function SumArrayElements( values() As Integer ) As Integer
	Dim sum As Integer, element As Integer
	For Each element In values
		sum = sum + element
	Next
	Return sum
End Function
```

Array Pop method

Arrays can now be used as stacks. The `Append` method pushes a new item onto the end of the stack; a new `Pop` method removes the last element from the array and returns its value. Here's an example, a directory-crawling routine that performs a depth-first traversal without using recursion:

```
Sub ProcessAllFiles( directory As FolderItem )
	Dim itemsToInspect(0) As FolderItem
	Dim item As FolderItem, ctr As Integer
	itemsToInspect(0) = directory
	While Ubound( itemsToInspect ) >= 0
		item = itemsToInspect.Pop
		If item.Directory Then
			For ctr = 1 To item.Count
				itemsToInspect.Append item
			Next
		Else
			ProcessOneFile item
		End If
	Wend
End Sub
```

For...Next loops now accept singles and doubles as loop counters.

Operator overloading

You can now define comparison and arithmetic operators for your classes. For example, if you had a `QueryResult` object that stored a list of found records, you could join it with another `QueryResult` by defining a custom addition operator:

```
Function Operator_Add( rval As QueryResult ) As QueryResult
	Dim result As QueryResult
	Dim item As FoundRecord
	result = New QueryResult
	For Each item In FoundRecords
		result.FoundRecords.Append item
	Next
	For Each item In rval.FoundRecords
		result.FoundRecords.Append item
	Next
	Return result
End Function
```

When you add two `QueryResult` objects, this method will be invoked to generate the resulting value.

```
Operator_Add( opB As type ) As type
Operator_Subtract( opB As Type ) As type
Operator_Multiply( opB As Type ) As type
Operator_Divide( opB As Type ) As Type
Operator_IntegerDivide( opB As Type ) As Type
Operator_Modulo( opB As Type ) As Type
```

All of these operator methods define `Self` as the left operand and pass the right operand as a parameter. Sometimes this isn't enough; if you had a vector class and you wanted to allow multiplication by a scalar, you could define a `Multiply` method like this:

```
Function Operator_Multiply( opB As Double ) As Vector
```

This would let you write this expression:

```
vectorA = VectorB * 2.5
```

But you could not write this expression:

```
vectorA = 2.5 * VectorB
```

For these situations, there is an additional set of operators. They are identical to the ordinary arithmetic operators, but they reverse the order: `Self` is now the right operand and the left operand is passed as the parameter.

```
Operator_AddRight( opA As Type ) As Type
Operator_SubtractRight( opA As Type ) As Type
Operator_MultiplyRight( opA As Type ) As Type
Operator_DivideRight( opA As Type ) As Type
Operator_IntegerDivide( opA As Type ) As Type
Operator_ModuloRight( opA As Type ) As Type
```

The ordinary methods are always preferred; the compiler only falls back on the `Right` versions if there is no legal left version.

In addition to the binary operators, there's a unary negation operator:

```
Operator_Negate() As Type
```

Finally, there is a comparison method:

```
Operator_Compare( opB As Type ) As Integer
```

The comparison method must compare `Self` against the parameter in whatever way makes sense for the two values. If `Self` is greater, it should return a positive number; if the parameter is greater, it should return

a negative number. If the two values are equal, it should return zero. This is the same system the StrComp method uses. All of the comparison operators are based on this method, and it applies whether Self is on the left or right side of the expression.

If you use an object in an expression and it does not define an appropriate operator, the compiler will give you an undefined operator error.

There are two kinds of conversion operator. The "convert to" operator has no parameters and returns a value:

```
Operator_Convert() As Type
```

This allows you to use the object anywhere the return type is legal. You can overload this operator with as many different return types as you like, so long as it's always clear which conversion is intended. If the parameters match the parameter types of exactly one method, the compiler will invoke that method. If the parameters can be converted to the parameter types of exactly one method, the compiler will invoke that method. Otherwise, the call is declared ambiguous and the compiler reports an error.

The other conversion operator is a "convert from". This is like a fusion of the C++ copy constructor and assignment operator. It has one parameter and does not return a value:

```
Operator_Convert( val As Type )
```

In the "convert from" example, an existing object instance was converted into another type. With the From operator, a new instance is created and the appropriate conversion operator is called to initialize it. When an object instance is created by conversion, no constructor is called - the conversion operator takes its place.

If both types of conversion apply in a given situation, the language gives the "from" conversion priority.

Setter methods

You can assign a value to a method. The assigned value becomes the rightmost parameter. This allows you to create "virtual properties" out of a pair of methods:

```
Function Length () As Integer
      Return Ubound( internalArray )
End Function
Sub Length( newValue As Integer )
      Redim internalArray( newValue )
End Sub

myObject.Length = myObject.Length + 1
```

Virtual interfaces

When you override a superclass method, the new method is called even if a reference to the object has the superclass' type. In the old compiler, this did not work for interfaces - you would get the overridden method instead. With the new compiler, interface methods are just like regular methods.

ByRef improvements

You can now pass properties of the current object or module as ByRef parameters. In addition, you can pass any parameter by reference, even if the parameter itself was passed in to the current function by reference.

Assignment to ByVal parameters
Parameter variables passed by value are now treated like ordinary local variables. You can pass them as reference parameters to another function or assign values to them, though of course you are only changing your local copy and not the original as with `ByRef`.

Better overloading
All methods can be overloaded. It doesn't matter whether you use parentheses when calling the method or not. If the parameters match the parameter types of exactly one method, the compiler will invoke that method. If the parameters can be converted to the parameter types of exactly one method, the compiler will invoke that method. Otherwise, the call is declared ambiguous and the compiler reports an error.

New menu system
The `MenuItem` class has been expanded. It has the following new methods:

```
Function Item( index As Integer ) As MenuItem
Function Count( ) As Integer
Function Child( name As String ) As MenuItem
Sub Append( newChild As MenuItem )
Sub Insert( index As Integer, newChild As MenuItem )
Sub Remove( index As Integer )
Sub Remove( child As MenuItem )
```

These methods allow you to inspect and modify menus at runtime. For a menu, `Count` returns the number of menu items; for a menu item, this is the number of items on the submenu or zero if it does not have a submenu. `Item` returns the n'th item (zero-based) and throws an `OutOfBoundsException` if the supplied index is illegal. `Child` looks up menu items by name or by text.

In addition, the `Application` class and `Window` class have a `MenuBar` property of type `MenuItem`. This represents the entire menu bar; its children are the menus. This property is readable and writable, so you can replace the entire menu bar if you like. See the "New Menu System" read me for more information.

String functions are now methods
All of the global methods that operate on strings can be called as methods of a string variable. `text.Left(9)` means the same thing as `Left(text, 9)`; `text.CountFields(" ")` is the same operation as `CountFields(text, " ")`.

Super keyword refers to the immediate superclass
When you've overridden a method, you can call the overridden method with the superclass' name. This causes problems if you reorganize your class hierarchy later or move the code to a different class. You can now use the `Super` keyword instead; it refers to the immediate superclass, whatever that happens to be.

Dim foo() As type means the same thing as `Dim foo(-1) As type`

More flexible variable positioning
`Dim` and `Const` statements can appear anywhere on the top level of a method. They can't be placed inside conditionals or looping structures, and they must still be declared before they are used.

Smart linker
The linker automatically eliminates unused classes, global methods, and module properties. At present, this only applies to REALbasic code, not to intrinsic classes or methods. This means you can throw all the code

libraries you might ever need into your project and your built app will only contain the pieces you actually use.

Better arrays
Array access involves less overhead, especially for multidimensional arrays, and code which does a lot of array manipulation should speed up.

Default parameters
You can supply default values for parameters, thus making them optional. If a caller omits an optional parameter, the default value will be supplied automatically. This is the syntax:

```
Sub ResetGlobal( newValue As Integer = 0 )
    gSomeGlobal = newValue
End Sub
```

You can now call `ResetGlobal` with or without a parameter:

```
ResetGlobal
ResetGlobal 47
```

If a method has more than one optional parameter, the default values are used from right to left. Here's a method with two default parameters:

```
FancyMethod(flag As Boolean = True, index As Integer, val As Double = 3.14)
```

If you were to call it, supplying two parameters, they would be the left two:

```
FancyMethod True, 42
```

This would not be legal:

```
FancyMethod 42, 19.22
```

New pragmas
Four new pragmas let you turn compiler options on and off. In addition to the option name, you specify `True` or `False`:

`BoundsChecking` - enables or disables bounds checking (setting this false is the same as using `disableBoundsChecking`)

`BackgroundTasks` - enables or disables auto-yield to background threads (setting this false is the same as using `disableBackgroundTasks`)

`StackOverflowChecking` - lets you control whether to check for stack overflows when the function is entered

`NilObjectChecking` - lets you control whether to automatically check objects for `Nil` before accessing properties and calling methods

Since these options can be enabled or disabled, you can now do things like disable background tasks for an inner loop, but leave them enabled for the outer loop:

```
For y = 0 To Height
    #pragma BackgroundTasks false
    For x = 0 To Width
          ...process some pixels...
    Next
```

```
    #pragma BackgroundTasks true
Next
```

New Finally clause in exception handling
Sometimes a function needs to do some cleanup work whether it is finishing normally or aborting early because of an exception. RB now offers an optional "Finally" block at the end of the function, after the exception handlers if it has any. Code in this block will be executed even if an exception has occurred, whether the exception was handled or not.

Single-line If statements
RB now supports Basic's traditional single-line `If-Then` statement. You can execute one statement if some condition is true and optionally a different statement if the condition is false:

```
If x < 0 Then x = 0
If myObj.IsLocked Then MsgBox "It's locked!" Else myObj.Reset
```

Call statement
You can now ignore a method or function's return value by calling it with the `Call` statement:

```
Call GetFolderItem("my file.txt").CreateTextFile
```

Changes in the REALbasic debugger

Safer debugging
When you run a project in the debugger, it now starts up as a separate application instead of sharing REALbasic's memory. This protects REALbasic in case your application leaks memory, corrupts data, or crashes, and the result should be a more stable working environment. In addition, your programs will act more consistently whether they are running in the debugger or as standalone apps. Their memory usage patterns and event orders will be the same in either mode.

Better performance
The new debugger imposes less overhead than the old one, so your code should run faster.

Centralized display
All debugger information shows up in one window.

Better handling of exceptions
The new debugger breaks anytime an exception occurs, instead of only breaking when an unhandled exception occurs. Additionally, it stops before the exception is handled, so you can see what the application state was that created the exception.

Breakpoints work differently in the new debugger than they did in the old one. You can no longer set a breakpoint on a comment, blank line, constant, `Dim`, or `Declare` statement. If you have a breakpoint set on such a line, it will be drawn in grey to show that it is disabled.