HJWasm Object Oriented Language Extension

User Guide

Introduction

HJWasm 2.25 introduces a set of language extensions made available through the built-in Macro Library System. One of these extensions is the ability to implement Object Orientation in Assembler Code.

The approach is slightly different from traditional OO in that it doesn't make use of inheritance but provides the concept of an interface which can be mapped to homogenous or even heterogeneous classes as long as they conform to the interface's layout.

As with C++ it is good practice to keep each class definition in its own file and the implementation in another. This keeps code clean, modular and allows different modules to share the definition of the class and any related types.

Declaring a Class

```
IFNDEF _CLASS_PERSON_
_CLASS_PERSON_ EQU 1

CLASS Person
       CMETHOD GetName
       CMETHOD SetName
       CSTATIC IsHuman
       fname   db 128 DUP (?)
       age     db 0
       human   db 0
ENDCLASS

pPerson TYPEDEF PTR Person

ENDIF
```

As with C++ you should implement an inclusion guard in your class definition file through the use of IFNDEF.

A class is simply declared as CLASS <name> and ENDCLASS.

The class data shares a lot in common with a simple structure data type and thus allows member fields to be specified directly in the class definition.

Methods are purely named at this point using either CMETHOD (Instance method) or CSTATIC (Static method).

It is often useful to also define a pointer to object type such as pPerson in this case.

The class will automatically create 4 QWORD sized entries at the start of the structure for the constructor, destructor, release method and reference count.

In addition the class directive creates a static copy of the object structure. This is used to store static elements as well as provide a means to directly invoke methods without going through a vtable.

Each CMETHOD or CSTATIC entry creates not only the correct types, prototypes on the object but creates a relevant vtable entry for when the class is actually instantiated.

Implementing the Class

Implementation of Init (Constructor), Destroy (Destructor) methods is **mandatory**.

```
; Constructor -> Can take optional arguments.
;-------------------------------------------------------
METHOD Person, Init, age:BYTE

	LOCAL isAlive:DWORD
	; Internally the METHOD forms a traditional procedure, so anything that you can
	; do in a PROC you can do in a method.

	; On entry into any method RCX is a pointer to the instance
	; and the correct reference type is assumed.
	mov [rcx].human, 1                      ; Hence this is possible.
	mov (Person PTR [rcx]).human, 1         ; Alternative forms of reference.
	mov [rcx].Person.human, 1               ; " "

	mov isAlive,0
	mov al,age
	mov [rcx].age,al

	.if( age < 100 )
		mov isAlive,1
	.endif

	; Constructor MUST return thisPtr in rax (the implicit self reference
	; passed in RCX).
	mov rax,thisPtr

	ret
ENDMETHOD

; Destructor -> Takes no arguments.
;-------------------------------------------------------
METHOD Person, Destroy
	mov [rcx].age,0
	ret
ENDMETHOD
```

```
; Return pointer to name.
;-----------------------------------------------------
METHOD Person, GetName
       lea rax,[rcx].fname
       ret
ENDMETHOD

; Set person name.
;-----------------------------------------------------
METHOD Person, SetName, pNameStr:QWORD

       lea rsi,pNameStr
       lea rdi,[rcx].fname

copyname:
       mov al,[rsi]
       mov [rdi],al
       .if( al == 0 )
              jmp done
       .endif
       inc rsi
       inc rdi
       jmp short copyname
done:

       ret
ENDMETHOD

; Static method to check if a person is a human.
;-----------------------------------------------------
STATICMETHOD Person, IsHuman, somebody:PTR Person
       mov rax,somebody
       mov al,(Person PTR [rax]).human
       ret
ENDMETHOD
```

Declaring and Instantiating Objects

Objects are instantiated via the use of either the **$NEW** or **$RBXNEW** directives. Both can be in-lined into other expressions, statements and invokes.

```
       local myPerson:PTR Person
       local age:BYTE
       mov age,36

       mov r10,$RBXNEW(Person, 10)
       $DELETE(r10)

       mov myPerson,$NEW(Person, age)
       $DELETE(myPerson)
```

Instances can be delete via **$DELETE**. For most basic uses simple LOCAL or GLOBAL variables can be used to store pointers to object instances.

To declare an array of objects you can additionally use :

```
       mov rbx,$ARRAY(Person,8)
       $DELETEARRAY rbx
```

This will attempt to create an array of references (pointers) to the objects, or for primitive types and normal structures a fully sized array in memory.

An alternative directive to LOCAL is supplied that supports < > in names. Technically this makes no difference in the code that is generated, however due to the assemblers parsing of quoted and literal text in macros it allows us to “simulate” higher level generic types in names, for example:

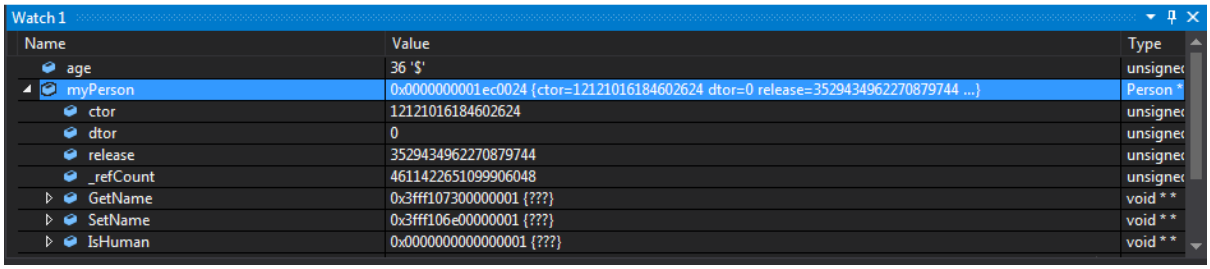
```
$DECLARE objArray[8], PTR Person
$DECLARE myList, PTR List<Float>

CLASS List<Float>
; Any class which has the same methods/parameters/ordinal is said to conform to an
interface. Methods can be invoked via the interface to enable RT-Polymorphism.
       CMETHOD Items
       CMETHOD AddItem
       CMETHOD RemoveItem
       CMETHOD Clear
       CMETHOD Trim
       CMETHOD Sort
       CMETHOD InsertItem
       Count    dq 0
       Capacity dq 0
       CurIdx   dq 0
       itemsPtr dq 0
       itemSize dq 0
       itemType db 0
ENDCLASS

METHOD List<Float>,Init, count:QWORD, itemSize:VARARG
       mov byte ptr [rcx].itemType,LIST_FLOAT
       mov rax,4
       mov [rcx].itemSize,rax
       .if(rdx > 0)
              mov [rcx].Capacity,rdx
              imul rax,count
              invoke HeapAlloc,_oo_heap,0,rax
              .if(rax == 0)
                     THROW OUT_OF_MEMORY
              .endif
              mov rcx,thisPtr
              mov [rcx].itemsPtr,rax
       .endif
       mov rax,thisPtr
       ret
ENDMETHOD
```

Debugging

Debugging support is implicit due to all methods and members being fully typed, arguments are visible inside methods and entire object instances can be examined:



Invoking Methods

A number of accelerator macros are provide to call methods either directly, indirectly via their vtable entry or inline in other invokes including specified return types:

```
; Direct invoke (via the generated structure type):
$INVOKE List<String>,AddItem,myList,myString

; Indirect invoke via the object instance vtable:
$VINVOKE myString,String,Trim, FALSE

; Direct, with in-line direct $I call
$INVOKE String,ToLower,$I(List<String>,Items,myList,0),FALSE
```

$V, $VF, $VD, $VW, $VB are also provided to provide in-line vtable invocations that return a result of the specified type : V == QWORD, VF == float/real, VD == DWORD, VW == WORD, VB == BYTE.

Interfaces

An interface is effectively a generic contract, which can be applied to invoke methods and access members of unrelated object instances as long as they conform.

An example of this in action is combined with < > support to implement a range generic container class types for List<int>, List<float>, List<double>.

By creating an IList interface we can access any of these types in a consistent manner.

```
INTERFACE IList             ; Common Container Protocol/Interface.
       CVIRTUAL Items, idx:QWORD
       CVIRTUAL AddItem, objPtr:QWORD
       CVIRTUAL RemoveItem, idx:QWORD, release:BOOL
       CVIRTUAL Clear, release:BOOL
       CVIRTUAL Trim
       CVIRTUAL Sort
       CVIRTUAL InsertItem
ENDINTERFACE
```

An interface definition begins with **INTERFACE** <name> and ends with **ENDINTERFACE**. Common methods are declared with the **CVIRTUAL** specifier. Note however that virtual methods do specify their arguments as these generate only prototypes and no actual code. This is to ensure type-conformance.

The first entry on this particular interface and any classes which want to share this type specify an **Items** method.

This is a special method, which allows for accelerator macros to be used to access any object which implements some form of iterator (IE: a container).

For example on the specialisation class List<float> we have:

```
METHOD List<Float>,Items, idx:QWORD
	mov rax,[rcx].itemsPtr
	mov rbx,[rcx].Count
	.if(rdx < rbx)
		movss xmm0,[rax+rdx*4]
	.else
		THROW INDEX_OUT_OF_BOUNDS
	.endif
	ret
ENDMETHOD
```

Which will then allow other code to access its internal collection with:

```
; This protocol interface is purely to allow for acceleration of typeless calls to get
; items out of classes that need an iterator / [] access.
INTERFACE Iterator		; Common Container Protocol/Interface.
	CVIRTUAL Items, idx:QWORD
ENDINTERFACE

; The direct specific way:

mov rax,$V(myList,List<Float>,Items,0)

; The generic iterator interface way:

Mov rax,$ITEM(myList,0)
```