Uasm Object Oriented Language Extension

User Guide

Introduction

Uasm 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.

The OO library can be used as-is on Windows platforms and can be used on Linux / OSX with one caveat:

To provide memory allocation routines (and the implementation of MEMALLOC, MEMFREE macros) it is assumed that you have linked to libc and provided prototypes for malloc and free. As neither Linux nor OSX provide heap allocation routines as found on Windows libc's malloc/free are used instead.

- ***See LIN64_2 example in the Samples folder***.

Declaring a Class

```
IFNDEF _CLASS_PERSON_
_CLASS_PERSON_ EQU 1

CLASS Person
	CMETHOD GetName
	CMETHOD SetName
	CSTATIC IsHuman
	fname	! 1"# DUP $%&
	a'e	! (
	)uman	! (
ENDCLASS

*Person T+PEDEF 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, <USES rbx>, age:BYTE

        LOCAL ,sA-,.e/DOORD
        ; 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.
        mo. 1r2345)uman6 1                      ; Hence this is possible.
        mo. $Person PTR 1r234&5)uman6 1         ; Alternative forms of reference.
        mo. 1r2345Person5)uman6 1               ; " "

        mo. ,sA-,.e6(
        mo. a-6a'e
        mo. 1r2345a'e6a-

        5,f$ a'e 7 1(( &
                mo. ,sA-,.e61
        5en ,f

        ; Constructor MUST return thisPtr in rax (the implicit self reference
        ; passed in RCX).
        mo. ra36t),sPtr

        ret
ENDMETHOD

; Destructor -> Takes no arguments.
;-------------------------------------------------------
METHOD Person, Destroy, <>
        mo. 1r2345a'e6(
        ret
ENDMETHOD
```

```
; Return pointer to name.
;-------------------------------------------------------
METHOD Person, GetName, <>
       -ea ra361r2345fname
       ret
ENDMETHOD

; Set person name.
;-------------------------------------------------------
METHOD Person, SetName, <USES rbx>, pNameStr:QWORD

       -ea rs,6*NameStr
       -ea r ,61r2345fname

2o*8name/
       mo. a-61rs,4
       mo. 1r ,46a-
       5,f$ a- 99 ( &
              :m*  one
       5en ,f
       ,n2 rs,
       ,n2 r ,
       :m* s)ort 2o*8name
 one/

       ret
ENDMETHOD

; Static method to check if a person is a human.
;-------------------------------------------------------
STATICMETHOD Person, IsHuman, <>, somebody:PTR Person
       mo. ra36some!o 8
       mo. a-6$Person PTR 1ra34&5)uman
       ret
ENDMETHOD
```

Methods take an option USES clauses, which can be left empty with <>.
This maps directly to the generated backing PROC.

Be aware that under Windows 64bit calling convention, thisPtr is passed in RCX, while under SYSTEMV thisPtr will be supplied in RDI. This can be made platform-agnostic by using the built-in variable @Platform as described in the extended user guide.

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.

```
-o2a- m8Person/PTR Person
-o2a- a'e/;+TE
mo. a'e6<=

mo. r1(6_R;>NE0$Person6 1(&
_DELETE$r1(&

mo. m8Person6_NE0$Person6 a'e&
_DELETE$m8Person&
```

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 :

```
mo. r!36_ARRA+$Person6#&
_DELETEARRA+ r!3
```

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.

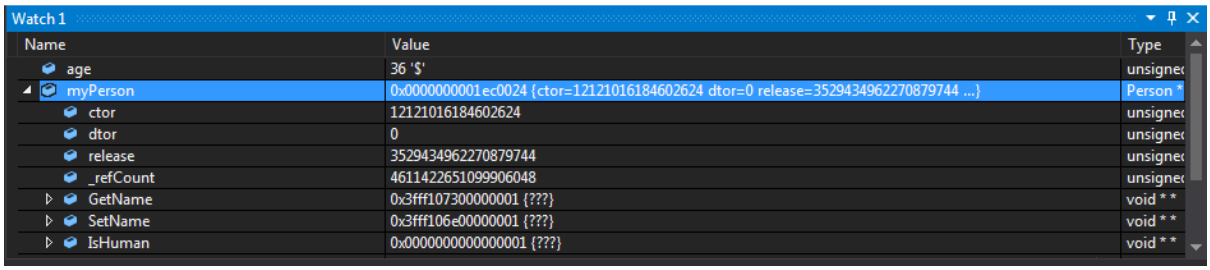
An alternative directive to LOCAL is supplied that supports < > in names. Technically this makes no difference in the code that is generated, however due to the assemblers parsing of quoted and literal text in macros it allows us to “simulate” higher level generic types in names, for example:

```
_DECLARE o!:Arra81#46 PTR Person
_DECLARE m8L,st6 PTR L,st7F-oat?

CLASS L,st7F-oat?
@ An8 2-ass A),2) )as t)e same met)o sB*arametersBor ,na- ,s sa,  to 2onform to an
,nterfa2e5 Met)o s 2an !e ,n.oCe  .,a t)e ,nterfa2e to ena!-e RTDPo-8mor*),sm5
       CMETHOD Items
       CMETHOD A  Item
       CMETHOD Remo.eItem
       CMETHOD C-ear
       CMETHOD Tr,m
       CMETHOD Sort
       CMETHOD InsertItem
       Count     E (
       Ca*a2,t8  E (
       Curl 3    E (
       ,temsPtr  E (
       ,temS,Fe  E (
       ,temT8*e  ! (
ENDCLASS

METHOD L,st7F-oat?6In,t6 2ount/QOORD6 ,temS,Fe/GARARG
       mo. !8te *tr 1rcx45,temT8*e6LIST_FLOAT
       mo. rax6H
       mo. 1rcx45,temS,Fe6rax
       5,f$rdx ? (&
              mo. 1rcx45Ca*a2,t86rdx
```

```
                ,mu-  rax62ount
                ,n.oCe Hea*A--o26_oo_)ea*6(6rax
                5,f$rax 99 (&
                       THROO OUT_OF_MEMOR+
                5en ,f
                mo. rcx6t),sPtr
                mo. 1rcx45,temsPtr6rax
        5en ,f
        mo. rax6t),sPtr
        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 a-so *ro., e to *ro., e ,nD-,ne .ta!-e ,n.o2at,ons t)at return a resu-t of t)e s*e2,f,e t8*e / G 99 QOORD6 GF 99 f-oatBrea-6 GD 99 DOORD6 GO 99 OORD6 G; 99 ;+TE5

Inter aces

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.

```
OINTERFACE IL,st            @ Common Conta,ner Proto2o-BInterfa2e5
       CGIRTUAL Items6 , 3/QOORD
       CGIRTUAL A  Item6 o!:Ptr/QOORD
       CGIRTUAL Remo.eItem6 , 3/QOORD6 re-ease/;OOL
       CGIRTUAL C-ear6 re-ease/;OOL
       CGIRTUAL Tr,m
       CGIRTUAL Sort
       CGIRTUAL InsertItem
ENDOINTERFACE
```

An interface definition begins with O**INTERFACE** <name> and ends with **ENDOINTERFACE**. 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
       mo. rax61rcx45,temsPtr
       mo. rbx61rcx45Count
       5,f$rdx 7 rbx&
              movss xmm061raxIrdxJH4
       5e-se
              THROO INDE>_OUT_OF_;OUNDS
       5en ,f
       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.
OINTERFACE Iterator         ; Common Container Protocol/Interface.
       CGIRTUAL Items6 , 3/QOORD
ENDOINTERFACE

; The direct specific way:

mo. ra36_V$m8L,st6L,st7F-oat?6Items6(&

@ T)e 'ener,2 ,terator ,nterfa2e Aa8/

Mo. ra3,_ITEM$m8L,st6(&
```