

FMS2 DOCUMENTATION

Douglas B. Hoffman October 19 2019 dhoffman888@gmail.com

Table of Contents

Introduction	2
What is New in FMS2?	2
Recommended File Load Order.....	3
Creating Objects.....	3
Creating Classes	4
Inheritance	4
Instance Variables	5
Name Scope	5
Mutability	5
Using Classes to Define Instance Variables	5
Duck Typing.....	6
Declaring/Binding/Overriding Messages and Methods	6
Using Self and Super	7
Function-based Methods	8
Make-selector	9
Early and Late Binding	10
Duality of Forth Data Types and Objects	10
Duality of Objects Allotted in the Dictionary and Objects Allocated in the Heap	10
?alloc	11
Normal Forth Definitions but with Scope Local to a Class.....	12
Restore.....	12
Using FMSCheck?	12
IS-A	13
High Order Functions (HOFs).....	13
Arrays	13
Strings.....	15
Files.....	16
FREEing Allocated Objects	18
Debugging Memory De-allocation	19
Class Reference	21
Class Creation.....	21
Class Related Words	21
Class Library Reference.....	21
Ptr Class (subclass of Object)	21

Array Class (subclass of Ptr)	21
Array HOFs.....	22
Farray Class (subclass of Array).....	22
Farray HOFs	22
String Class (subclass of Ptr)	22
String HOFs	23
File Class (subclass of Object)	23
Int Class (subclass of Object).....	24
Flt Class (subclass of Object)	24
Hash-Table Class (subclass of Object)	24
Hash-Table-m Class (subclass of Hash-Table)	25

Introduction

FMS is an object programming extension to (ANS) Forth. It is class based single inheritance using Duck Typing and explicit references to self.

In FMS everything is *not* an object, unlike Oforth and Factor. Oforth in particular is an excellent dialect of Forth that fully uses the object oriented programming (oop) paradigm and is not extremely different in syntax from standard Forth. Oforth achieves a pure oop environment by making all atomic data types objects. This includes integers, floating point numbers, big integers, strings, and arrays (and JSON objects with the Oforth JSON extension). These data types are recognized by the Oforth interpreter/compiler and the corresponding objects are created as needed. Also, garbage collection is included in the kernel. There is no need for double length integers. There is no floating point stack, or string stack.

In contrast, standard Forth has just the kernel data types of integers, floating point numbers, and doubles. None are objects and there is no garbage collection. The approach taken with FMS is to bridge Forth with an oop extension that makes sense given the environment of Forth kernel data types and dealing with the dictionary and heap.

The entire FMS object system and all classes reside only in the dictionary and so will be saved fully intact with a dictionary image save. All classes can create objects either in the dictionary or in the heap. Objects created in the dictionary will survive a dictionary image save. Objects created in the heap will not.

What is New in FMS2?

FMS2 is a direct derivative of the FMS-SI extension. But it is trimmed down significantly in code size and complexity without sacrificing important functionality.

The <indexed declaration and related support words are gone (arrays can be created without them). Function-methods and many High Order Functions (HOFs) have been introduced which are faster, simpler, and greatly reduce virtual method table sizes. HOFs encourage the use of quotations.

There is no mechanism provided for message-less access to instance variables. Though the code is simple enough that the motivated user could easily create such a utility. But I discourage this.

The provided class library is much improved and (I believe) is higher quality.

Documentation, what you are reading now, is improved and includes the class library.

Recommended File Load Order

Load file FMS2.f, and then the recommended class library files, but only if wanted. The library load order is provided at the end of the file FMS2.f but is also listed below.

```
\ optionally load file mem.f prior to FMS2.f for memory deallocation
\ debugging
```

```
include ptr.f
include array.f
include string.f
include farray.f
include file.f
include arrays.f
include objectArray.f
include hash-table.f
include hash-table-m.f
```

Note that none of the class library files are required to be loaded. FMS2 will work just fine without them.

Creating Objects

FMS objects can be created in the dictionary or in the heap.

The following will create a string object in the dictionary named s with a maximum character size of 10 and initialized with "Hello"

```
s" Hello" 10 string s
```

The following will create a nameless string object in the heap with no maximum character size and initialized with "Hello".

```
: make-string ( c-addr u -- str-obj ) heap> string ;
s" Hello" make-string \ will leave a str-obj on the stack
```

Note the different stack inputs required depending if the string is to be allotted in the dictionary or allocated in the heap. This is due to the way the string class is defined and can be controlled by the programmer for any class.

Nameless dictionary objects can also be created by using `dict>` in a definition as above in place of `heap>`.

Creating Classes

Creating a new class can have the following simple form:

```
:class point \ begin a new class definition, the new class is named point
  cell bytes x \ define a 1 cell sized instance variable named x
  cell bytes y
  :m :init ( x y -- ) y ! x ! ;m \ define the default constructor method
  :m :. x ? y ? ;m \ define a method to print the point
;class \ end the definition of the class

10 20 point p \ create a point object named p, created in the dictionary
p :. 10 20 ok \ the normal <object> <message> message send in FMS
```

The word `:class` begins a new class definition and is followed by the name of the new class. `x` and `y` are instance variables of size 1 cell.

Any class can create objects either in the dictionary or in the heap. For example:

```
: make-point ( x y -- point-obj ) heap> point ;

40 50 make-point dup :. 40 50
<free ok \ free the memory allocated to the point
```

Note the use of `<free` to destroy the heap point. The word `<free` works for any heap-allocated object. See the section on Freeing Allocated Objects.

Inheritance

The more general form of defining a class includes declaring a superclass:

```
:class <newclassname> <super <superclassname>
...
;class
```

If `<super <superclassname>` is omitted then class object will be used as the superclass. Subclasses inherit all instance variables, methods, class variables, and helper definitions from its superclass.

Instance Variables

The word `bytes` is the instance variable definition primitive.

Subclasses should not re-use instance variable names from its superclass(s) because the name will then be shadowed. There is no checking for this.

Name Scope

The instance variable namespace scope is specific to the class so there can be no conflicts with existing or subsequently created names. When the name of the instance variable is used the action is to leave the address of the instance variable on the stack. Note that instance variable names are private to the class in which they are defined and any subclasses of that class. So, for example, in this case no global definition using the name 'x' will be shadowed and the name 'x' can be re-used as many times as desired in other classes that have no inheritance relationship.

Mutability

All instance variables are mutable, in this case x and y, but unless methods are explicitly defined to change the instance variables then they cannot be changed.

Using Classes to Define Instance Variables

When designing new classes it is often convenient to use already-defined classes to define instance variables, instead of using `bytes`. Consider a rectangle class.

```
:class rectangle
  point upper-left
  point lower-right

\ note how we can send messages to the point instance variables
\ exactly as if they are normal objects
:m :init ( x1 y1 x2 y2 -- ) lower-right :init upper-left :init
;m
:m .. upper-left .. lower-right .. ;m
;class

10 20 30 40 rectangle n \ create a rectangle object named n
n .. 10 20 30 40 ok
```

Again, we can use the same class to create an object in the heap. The point instance variables will automatically be created either in the dictionary or in the heap along with the owning object.

```
: make-rect ( -- obj ) 1 9 5 6 heap> rectangle;

make-rect value n2
n2 .. 1 9 5 6
n2 <free ok
```

Duck Typing

Duck typing (used by FMS) is the most flexible way to use objects and messages. There is no requirement that different classes using the same message(s) be related via inheritance. There is no requirement to define an interface.

Method names (messages) have global scope and so can be used anywhere and need not be prefaced with the name of an object during compilation. Messages can also be ticked to obtain their XT for normal use with EXECUTE.

Classes FOO and BAR below are unrelated, but both use the message PUT without resorting to special means such as declaring interfaces.

```
:class foo
  cell bytes x \ define an instance variable named x
  :m put ( n -- ) x ! ;m \ define a message named PUT,
                        \ define its method, and bind them together
                        \ if a superclass has a PUT method, then override it
;class

:class bar
  cell bytes x
  :m put ( n -- ) 10 * x ! ;m
;class
```

If a message is sent to an object that does not recognize the message (i.e., a programming error is made), then FMS will put up an error "Message Not Understood" and cleanly abort rather than crash. Note that this will only be true with the constant `fmsCheck?` set to true, which should be the case during development.

Declaring/Binding/Overriding Messages and Methods

Interface declarations are not required (or available) in FMS. Message names are created naturally and automatically as a class's methods/messages are defined, similar to colon definitions. Note that there is no requirement that message names begin with colon ':'. I have elected to use this convention because it results in few name collisions and is also a convenient indication when a message or object HOF is being used.

Methods are created with `:m <message-name> ... ;m .`
The word `;class` ends the definition of the class.

The `:init` method is special because it is automatically called whenever a new object of any class is created. The `:init` method is not required to be defined for any new class because it is defined in class object as a null method.

When defining a method the name of the message and the association of that message to the ensuing method definition are done at the same time. Further, message over riding is also performed at that same time and is also done automatically because the compiler knows if that message is used in a parent class. See the example below where subclass BAR' implicitly overrides the PUT method in class FOO. There is no need, or way, to declare 'override'.

```
:class bar' <super foo
  :m put ( n -- ) 10 * x ! ;m
;class
```

Using Self and Super

Self is a pseudo instance variable. Self can only be used in a method definition. Self will always cause a late bound message call. Open recursion is enabled due to the behavior of self.

Also, as long as the message name has been previously defined, either in some other class or using make-selector, then self <messagename> can be used in a method definition even if the <messagename> method has yet to be defined for that class or any of its superclasses. Ultimately it must be defined or the method call will fail at runtime "message not understood".

Super is also a pseudo instance variable and can only be used in a method definition. Super is used to direct a message call to the method of the superclass when that method has been overridden in a subclass.

```
:class ClassOne
  :m do1 ." do1 from ClassOne " ;m
  :m do2 ." do2 from ClassOne " self do1 ;m
;class

ClassOne obj1
obj1 do2
\ => do2 from ClassOne  do1 from ClassOne

:class ClassTwo <super ClassOne
  :m do1 ." do1 from ClassTwo " ;m
  :m do2 ." do2 from ClassTwo " self do1 ;m
  :m do2' ." do2' from ClassTwo " super do2 ;m
;class

ClassTwo obj2
obj2 do2
=> do2 from ClassTwo  do1 from ClassTwo
```



```
obj2 do2'
=> do2' from ClassTwo  do2 from ClassOne  do1 from ClassTwo
```

Note how calling `super do2` in method `do2'` calls the `do2` method in `ClassOne` (instead of the `do2` method in `ClassTwo`). Further, due to open recursion when method `do2` in `ClassOne`, called by `super` in `ClassTwo`, calls `self do1` the `do1` method from `ClassTwo` is invoked instead of the `do1` method from `ClassOne`. Calling `self do1` from a `ClassOne` object will invoke the `do1` method from `ClassOne` as expected.

Function-based Methods

Some have asserted that dispatch tables for duck typed oop must have a large number of cells to hold the method XTs and the tables must necessarily be sparse. The number of cells required has been said to be $\#ofclasses \times \#ofselectors$.

By relaxing a non-essential IMO requirement and using some other techniques I have found that far fewer cells can be used and sparseness thus reduced.

For example. Consider the case where we have 4 classes and 3 selectors. The classes are object, A, A1, and B. The selectors are nu (not-understood), foo, and bar. A full-matrix would then require $4 \times 3 = 12$ cells for the tables. See 1) below.

1) original full-matrix tables

object	A	A1	B
-----	--	--	--
nu	nu	nu	nu
ud	foo	foo	foo
ud	ud	bar	ud

I don't see a value in having an nu selector. If the method is not-understood then a call to it is an error. I agree that throwing up a "not understood" message can help debugging, but it can be used only during development.

So I replace the nu and ud (undefined) XTs with zeros. See 2) below.

2) place 0 in not-understood and un-defined cells

object	A	A1	B
-----	--	--	--
0	0	0	0
0	foo	foo	foo
0	0	bar	0

So all cells containing 0 in this example can simply be removed. The tables are trimmed (Vitek and Horspoolⁱ technique) to only be as large as necessary. Trailing cells with zeros are removed. We then have just 4 cells used. See 3) below.

3) table trimming

```
define nu only in class B
object    A      A1      B
-----   --      --      --
           foo    foo    foo
           bar
```

Lastly, especially considering that shallow inheritance is encouraged to reduce complexity and increase program understanding, the programmer should judge if any selectors might be used in other classes. It has been my experience that some message names, such as `UPPERCASE`, will only be used in a single class, in this case class `STRING`. So we can define those selectors as function-based (with no polymorphic capability) and so remove the need for a lookup table cell. Perhaps selector `BAR` can be made function-based. See 4) below. Note that a function-based selector still has full access to all instance variables and `SELF`. The only limitation is that locals cannot be used and an `EXITF` instead of `EXIT` must used (if used at all). Not very limiting IMO. The other advantage to function-based methods is they execute very quickly because they are early bound.

4) possibly make bar function-based

```
object    A      A1      B
-----   --      --      --
           foo    foo    foo
```

So in the above example the original full-matrix 12 cells have been reduced to 3 or 4 cells. A 75% to 66% reduction.

This means there will still be empty cells in practice. Just far fewer than the full-matrix number.

A more realistic example is an early version of the downsized class rewrite for. There were 6 classes and 54 selectors. The full-matrix criterion would dictate that 324 table cells were needed. But using the above techniques I only needed 81 with just 11 of those 81 being zero. A 75% reduction over full-matrix. YMMV.

Function-based methods are defined using the following syntax:

```
:f <methodname> ... ;f
```

Make-selector

Messages without methods can be declared anytime outside of a class definition using

`make-selector`. This will create a message that can be used with `SELF` in a class's method definition before the `make-selector` message is associated with a method. This is because `SELF` is late bound.

Early and Late Binding

Late binding, also known as dynamic binding, is the standard message sending mechanism used in FMS. There are two exceptions: An instance variable defined as a class object will have all message sends early bound. Also, public named objects created in the dictionary with the `<classname> <objectname>` syntax will also use early binding. The reason is that in both cases the FMS compiler knows the class of the object so automatic early binding for efficiency can be used. There is no extra or special syntax required by the programmer.

Duality of Forth Data Types and Objects

It may be possible to design a classic Forth object extension where all data are objects, but this does not seem practical or easy to do. For this behavior one should look to using `Oforth`.

An extension to classic Forth FMS makes some compromises. For example, it has been designed to co-exist with normal Forth data types. FMS makes no attempt to convert classic Forth operators such as `+` `-` `*` `/` to messages. This would be impractical and at best confusing.

FMS arrays are designed to hold and work with classic Forth data types like integers and floats. Although since objects are represented by a one cell pointer then arrays serve the dual purpose of also working well with objects.

Strings are another class of FMS objects that are designed to interface seamlessly with classic Forth strings. Forth string operations tend to deal with the `(address length)` form. FMS string objects have many Forth interface methods that use that form. For example:

```
\ create a string object in the heap initialized to "Hello World"
s" Hello World" >string value s
s" lo" s" hi" s :sch&repl
s :@ type \ => Helhi World ok
```

Duality of Objects Allotted in the Dictionary and Objects Allocated in the Heap

Sometimes we want objects that are permanent and allotted in the dictionary. Sometimes we want temporary objects that are allocated in the heap such that all of the allocated memory can be `FREE`d when we are done using the object(s). FMS is designed to support both types of objects. We also want to be able to use the exact same methods on each kind of object, where possible, with minimum changes in method code. Certainly analogous messages must be

named exactly the same and used exactly the same way in either case, at least as much as possible.

`?alloc`

To enable this duality FMS2 uses a value, `?alloc`, that can be used to communicate to the `:init` method of an object whether any given object is allotted in the dictionary or allocated in the heap. Use of `?alloc` is optional when designing new classes. See the class definitions of `array` and `string` for examples of use.

Normal Forth Definitions but with Scope Local to a Class

Any normal Forth definition, such as a colon definition or variable etc., made inside a class definition (between `:CLASS` and `;CLASS`) will be private in scope to definitions of that class and any of its subclasses. See the definitions `?idx ^elem check (resize) qsort` etc. in the file `array.f`. These definitions have access to all instance variables of that class and any of that class's superclasses.

Restore

During the compilation of a class the wordlist search-order, current wordlist and `^class` are changed. Successful class compilation will restore these items to their standard state. But if a compilation error occurs before `;CLASS` is executed then these items will be left in an unknown state and so will likely cause problems if not restored. So FMS provides a word, `RESTORE`, that should be executed should a class compilation error occur. `RESTORE` will work for most Forths. But you may need a custom `RESTORE` for some.

It is convenient to have `RESTORE` execute automatically upon a compilation error. If your Forth has a compilation error hook then you may want to use it.

Using FMSCheck?

The constant `FMSCheck?` defined at the beginning of file `FMS2.f` is set to either true or false. When set to true several important error checks are made, including the following:

- Messages sent to objects that do not recognize the message will abort and display "message not understood" at the console.
- Array indices will be checked for validity, if not valid the program will abort with the message "array index out of range".
- String objects allotted in the dictionary will be checked when a resize operation is attempted. If the maximum size allotted for the string is exceeded the program will abort with the message "string max-size exceeded".
- Attempting to access a string character using an index that is out of range will cause the program to abort with the message "string index out of range".

Since making these checks uses extra time and code it is recommended that after a program is debugged it be recompiled with `fmsCheck?` set to false which will cause the checks to not be made.

IS-A

The class of any object can be tested using is-a. A compile time only word, is takes an object on the stack and the name of a class in the input stream as input. A true or false flag is returned.

High Order Functions (HOFs)

Since in FMS messages can be freely used outside of class definitions and compiled independently of any object reference, a HOF is simply created as a colon definition. Further, and just as importantly, sometimes HOFs can be used instead of defining a method.

See the FMS HOFs defined to be used on Arrays and Strings for typical examples.

Arrays

Class array is for 1-dimension arrays with elements 1 cell in size.

Array objects can be created in the dictionary or in the heap. An array in the dictionary will be initially empty and have a maximum size as provided on the stack when created. An abort and error message will be shown if the maximum size is attempted to be exceeded.

```
\ Create an array in the dictionary named a with a maximum size of 5 cells.  
\ So a can hold a maximum of 5 cell-sized elements.
```

```
5 array a
```

```
\ add elements to a
```

```
1 a :add
```

```
2 a :add
```

```
3 a :add
```

```
\ display the contents of a
```

```
a :.
```

```
0 1
```

```
1 2
```

```
2 3 ok
```

```
\ Create a nameless array in the heap.
```

```
\ The maximum number of elements are the size of an unsigned integer.
```

```
>array value b
```

```
: fill-b
```

```
10000 0 do i b :add loop ; ok
```

```
fill-b
```

```

b :.
0 0
1 1
2 2
3 3
4 4
5 5
6 6
...
9999 9999 ok
\ return the element at index 800
800 b :at .
800 ok

\ try to return an element at index 10000
10000 b :at
array index out of range \ aborted

```

\ Alternatively, there is a convenient syntax creating heap arrays
 \ with data.

```

i{ 2 3 4 5 } dup :.
0 2
1 3
2 4
3 5 ok \ 1
<free \ frees the allocated memory

```

There is a shorthand notation available for entering consecutive integers
 with the i{ ... } and o{ ... } syntax:

```

i{ 1 2 3 .. 15 20 100 } dup :.
0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13 14
14 15
15 20
16 100 ok

```

The class library provided also provided for arrays that hold Forth floating
 point elements.

```

3 farray f ok
1.5e f :add ok

```

```

9.99e f :add ok
0e f :add ok
f :.
0 1.5
1 9.99
2 0. ok

```

Also note that we have `>farray` and `f{ ... }` syntax, analogous to the Forth integer syntax.

Note that all of the array data (elements) so far have been Forth data types, not objects. Since all objects are of size 1 cell, then any object of any type can be an array element. There is a convenient syntax for creating arrays objects when the objects are of type int, flt, or string.

```

o{ 2 4.5 'hello world' } value x
[: :. ;] x :apply 2 4.5 'hello world' ok

```

With the `o{ ... }` syntax we can easily nest arrays to any level (though array nesting can also be done without this syntax).

```

o{ { 0 1 2 } { 3 4 { 100 200 } 5 } { 6 7 8 } } value y
y :.
{
{ 0 1 2 }
{ 3 4
{ 100 200 } 5 }
{ 6 7 8 } } ok

```

All of the shorthand syntax described so far (`i{ ... }` `f{ ... }` and `o{ ... }`) has been used in interpret mode. It is also possible to use the same syntax in a compiled definition by using `s" ..."` and `evaluate`.

```

: test ( -- obj ) s" o{ { 0 1 2 } { 3 4 { 100 200 } 5 } { 6 7 8 } }" evaluate
;

```

Strings

Class string is for ASCII strings with chars 1 byte in size. String objects can be created in the dictionary or in the heap. A string object in the dictionary will initially contain the provided text and have a maximum size as provided on the stack when created. An abort and error message will be shown if the maximum size is attempted to be exceeded.

```

\ Create a string in the dictionary named s with a maximum size of 15 chars.
s" Hello" 15 string s \ the maximum size of s is 15 characters

```



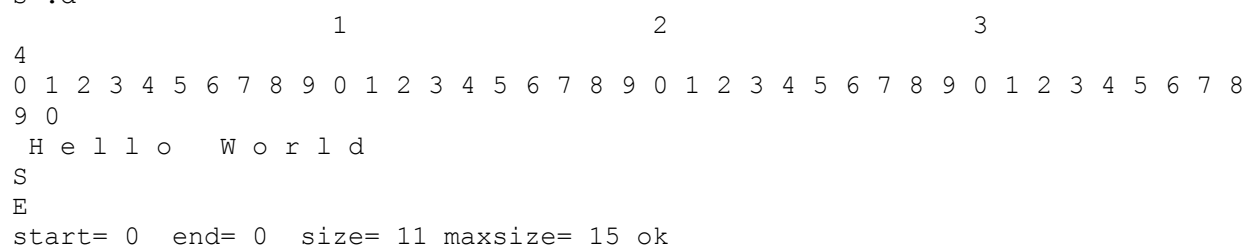
```

\ add text to s
s" World" s :add

\ print s
s :. \ => Hello World ok

\ do a string dump to inspect the string object state
s :d

```



```

          1                2                3
4
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
9 0
H e l l o   W o r l d
S
E
start= 0  end= 0  size= 11 maxsize= 15 ok

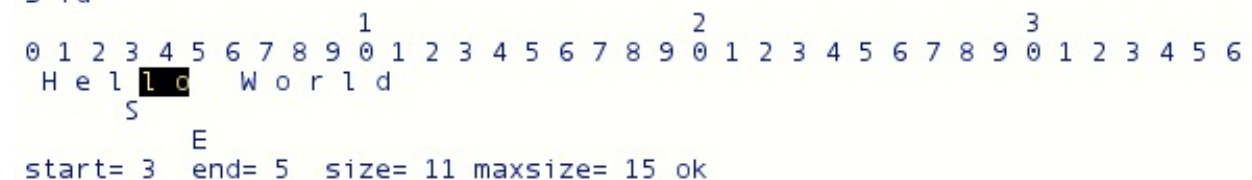
```

FMS strings use the familiar GUI string selection metaphor for substring searches, inserting, deleting, etc. String objects maintain internal pointers such as START and END for these purposes. Consider the following illustration in Figure 1:

```

s" lo" s :search . -1 ok
s :d

```



```

          1                2                3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
H e l l o   W o r l d
S
E
start= 3  end= 5  size= 11 maxsize= 15 ok

```

Figure 1

The `:search` method searches for the given text in the object. If found true is returned and the START and END pointers are set to mark the found substring. The advantage of this approach is that subsequent operations like `:insert` `:delete` and `:replace` will act on the "hilited" substring just as we would expect in any GUI text processor. The exact numerically indexed position and length of the found substring is not required (though it can be easily determined with the `:start` and `:end` methods).

The substring search engine for FMS strings uses the Boyer-Moore technique.ⁱⁱ This search engine is particularly fast when searching for strings in large pools of text such as 1 MB. I have measured a typical `:search` speedup ranging from 3 to 5 times faster than the fastest Forth `search` I could find. While it has a setup overhead and so must be slower for short pools of text the slowdown compared to other search methods will likely not be measurable.

Files

The file class objects make it convenient to deal with files. Once created the name of the file (full path if that is necessary or desired), and file ID are maintained in the file object's internal data. An example of FMS2 file usage is shown below.

```
\ The test large text file can be downloaded from the following website:  
\ https://www.gutenberg.org/files/135/135-0.txt
```

```
s" /Users/doughoffman/Desktop/FMS2/135-0.txt" file f  
f :read value s  
f :size . \ => 3369772
```

```
: test-searchCI ( case insensitive! )  
  s :reset  
  timer-reset  
  s" Facility: www.Gutenberg.org" s :searchCI .  
  .elapsed ;  
: test10 10 0 do cr test-searchCI loop ;
```

```
test10  
-1 1.81 msec  
-1 1.70 msec  
-1 1.58 msec  
-1 1.73 msec  
-1 1.68 msec  
-1 1.56 msec  
-1 1.78 msec  
-1 2.27 msec  
-1 1.75 msec  
-1 1.72 msec ok
```

```
\ now test VFX search ( case sensitive )  
: test-search  
  s :reset  
  timer-reset  
  s :@ s" facility: www.gutenberg.org" search . . .  
  .elapsed ;  
: test10 10 0 do cr test-search loop ;
```

```
test10  
-1 297 218225155 8.20 msec  
-1 297 218225155 8.34 msec  
-1 297 218225155 8.30 msec  
-1 297 218225155 8.37 msec  
-1 297 218225155 8.44 msec  
-1 297 218225155 9.00 msec  
-1 297 218225155 8.28 msec  
-1 297 218225155 8.28 msec  
-1 297 218225155 8.32 msec  
-1 297 218225155 8.26 msec ok
```

```
f :close \ will close the file and free the memory in the string
```

FREEing Allocated Objects

FMS does not add automatic garbage collection, so any objects that either allocate memory or are themselves allocated in the heap must have that memory manually FREEd. The problems here are there are several different ways in which one can allocate memory in FMS and each must be recognized and handled appropriately.

1. The entire object is allocated in the heap.
2. The object is created in the dictionary but some instance variables are allocated in the heap.
3. An array is allocated in the heap and the array contains a variable number of objects, each object also allocated in the heap and some of those objects are themselves heap arrays so the arrays can be nested to any level.

Case 1. is simple. One need only call free throw with the object on the stack.

```
:class foo
  cell bytes data
;class

: make-foo ( -- obj ) heap> foo ;
make-foo \ create the object
free throw \ free the object's memory
```

Case 2. is also not difficult.

```
:class foo2
  cell bytes data
  :m :init 10 allocate throw data ! ;m
  :m :free data @ free throw ;m
;class

: make-foo2 ( -- obj ) heap> foo2 ;
make-foo2 value f2 \ create the object
f2 :free \ free the object's instance variable memory
f2 free throw \ free the object's memory
```

FMS provides a HOF that will handle both Cases 1. and 2., <FREE .

```
: <free ( heap-obj --) dup :free free throw ;
```

Since class object has the method :free defined (though it does nothing) we can use <free on a foo object created in the heap.

Lastly, consider Case 3.

```
o{ { 0 1 2 } { 3 4 { 100 200 } 5 } { 6 7 8 } } value y
```

Here we have an array that contains all heap objects with some of the heap objects being nested arrays themselves. For this we have the HOF <freeAll.

```
: <freeAll \ { obj -- }
locals| obj |
begin
  obj :each
  while
    dup is-a array
    if recurse else <free then
  repeat obj <free ;
```

Debugging Memory De-allocation

Loading the file mem.f prior to the FMS2 code will provide a development-only. It should assist in debugging allocated object/memory leaks.

File mem.f is provided to assist assuring that a program that allocates, resizes, and frees memory has no memory leaks. It is a very simple routine that maintains a list of allocated pointers. There are just three user words:

```
n constant mem-size
.mem
clr-mem
```

n will set the maximum number of pointers to be tracked in the list.

.mem is used to query the number of un-freed pointers pointers.

clr-mem will free all of the un[freed pointers in the list.

File mem.f redefines the words ALLOCATE RESIZE and FREE such that their behavior is exactly the same but the unfreed pointers are tracked as described above. Mem.f is loaded prior to any other file. Mem.f is not written to be efficient. It will slow the execution time of any program that uses ALLOCATE RESIZE and FREE.

Example use:

```
\ A list that can track 50000 pointers is the default.
\ If the list size is exceeded your program will abort
\ with a " no room left in mem-list" error message.
\ Change the mem-size constant to suit your needs.
```

```
50000 constant mem-size \ choose a large enough size
```

```
100 allocate drop value x
```

```
100 allocate drop value y
```

```
.mem
```

```
0 9916368
```

```
1 9892576
```

```
2 unFREEd pointers
```

```
x 200 resize drop to x
```

```
.mem
```

```
0 9862976
```

```
1 9892576
```

```
2 unFREEd pointers
```

```
x free drop
```

```
.mem
```

```
0 9892576
```

```
1 unFREEd pointers
```

```
clr-mem
```

```
.mem
```

```
0 unFREEd pointers
```

Note that .mem will list the value of the unfreed pointer(s).

This can be useful in pinpointing the source of a memory leak your program. For example if you suspect that the pointer is an unfreed object then you can try sending a class-specific message to that pointer to verify.

Class Reference

Class Creation

:class	"className" --	begin new class definition with class name directly following
<super	"superclassName" --	declare superclass, default is class object if <super is omitted
bytes	n "name" --	create an instance variable definition of n bytes
:m	"messageName" --	begin normal method definition bound to given name
;m	--	end method definition
:f	"messageName" --	begin function-method definition with given name
;f	--	end function-method definition
exitf	--	must use instead of exit inside a :f ... ;f method definition
self	-- addr	pseudo instance variable
super	-- addr	pseudo instance variable for calling method of superclass
;class	--	end new class definition

Class Related Words

heap>	"className" -- obj	allocate a new nameless object in the heap of the given class
dict>	"className" -- obj	allot a new nameless object in the dictionary of the given class
is-a	obj "classname" -- f	return true if object is of the given class
make-selector	"messageName" --	define a new message outside of class definition, not yet bound to any method
restore	--	used to restore the search order and other state parameters in the case of a class compilation error
fmsCheck?	-- f	a conditional compilation constant, if true then error checks will be in place such as "Message Not Understood"
?alloc	-- f	a value used to communicate to the :init method if an object is allocated in the heap

Class Library Reference

Ptr Class (subclass of Object)

:init	n ptr --	set initial allocated size in bytes
:resize	newsize ptr --	
:size	ptr -- n	get the size in bytes
:free	ptr --	free allocated memory

Array Class (subclass of Ptr)

:to	elem idx arr --	stores element
:at	idx arr -- elem	gets element
:add	elem arr --	increase size of array by one and add elem to end
:size	arr -- n	return the number of elements in the array
:.	arr --	print the array to the console
:insert	elem idx arr --	insert the elem at idx position
:delete	idx arr --	delete the elem at idx position
:remove	idx arr -- elem	return the elem at idx, then :delete it from the array
:+	array1 arr --	concatenate all elems of array1 to the end of this array
:each	arr -- elem t or f	return next elem under true, or false if no more elems
:uneach	arr --	reset :each parameters to beginning

:first	arr -- elem	returns first element (same as 0 arr :at)
:second	arr -- elem	returns second element (same as 1 arr :at)
:last	arr -- elem	returns last element
:sort	arr --	only if this is an array of Forth integers, performs ascend sort
:sortWith	xt arr --	sorts array using xt for comparisons
:search	elem arr -- idx t f	searches for elem

Array HOFs

>array	-- arr	creates an empty resizable array in the heap
>oArray	-- oArr	creates an empty resizable obj-array in the heap
:apply	xt arr -- <varies>	apply xt to every item in the array, array is unchanged
:applyIf	xt1 xt2 arr -- <varies>	apply xt2 only if xt1 returns true
:map	xt arr --	apply xt to every item in the array, array is changed
:filter	xt arr -- arr'	return new arr of elems that respond true to xt
:count	xt arr -- cnt	count the number of times xt returns non-zero
<freeAll	arr --	free memory for heap array filled with heap objects as elements

Farray Class (subclass of Array)

Farray objects have all of the same methods as Array objects except the elem is a fp number and so always resides on the floating point stack when storing or retrieving. Both dictionary and heap objects are supported just as with class Array.

Farray HOFs

>farray	-- farr	creates an empty resizable farray in the heap
---------	---------	---

String Class (subclass of Ptr)

>string	adr len -- str	create a string object in the heap initialized with text adr len
:!	adr len str --	replace entire string with given text
:@	str -- adr len	return adr len of entire string
:to	char idx str --	stores char
:at	idx str -- char	gets char
:add	adr len str --	append adr len to the end of the string
:size	str -- n	return the number of chars in the string
:.	str --	print the entire string to the console
:insert	adr len str --	insert the given text starting at END, START and END are moved to end of str
:delete	str --	delete the substring bounded by START and END
:replace	adr len str --	replace the substring bounded by START and END with text adr len
:+	str1 str --	concatenate all of str1 to the end of str
:each	str -- char t or f	return next char under true, or false if no more chars
:uneach	str --	reset :each parameters to beginning
:first	str -- char	returns first char (same as 0 str :at)
:second	str -- char	returns second char (same as 1 str :at)
:last	str -- char	returns last char

:=	adr len str -- b	returns true if entire string is equal to adr len (case sensitive)
:=CI	adr len str -- b	Case Insensitive version of :=
:search	adr len str -- b	searches for adr len case sensitive, if found START and END delimit the found text. The next use of :search will begin just after END unless :reset is used.
:searchCI	adr len str -- b	Case Insensitive version of :search
:=sub	adr len str -- b	returns true if substr bounded by START and END is equal to adr len (case sensitive)
:=subCI	adr len str -- b	Case Insensitive version of :=sub
:reset	str --	sets START and END to zero
:selectAll	str --	sets START to zero and END to end of str
:start	str -- adr	returns address of START instance variable
:end	str -- adr	returns address of END instance variable
:copy	str -- str2	return string object copy in dict or heap, same as original
:upper	str --	converts all chars in string to uppercase
:lower	str --	converts all chars in string to lowercase
:ch+	char str --	append char to end of string
:chsearch	char str -- b	if char is found return true and set START and END to enclose the char
:chsearchCI	char str -- b	Case Insensitive version of :chsearch
:chinsert	char str --	inserts char at END, START and END are moved to just past inserted char

String HOFs

>string	adr len -- str	create a heap string object with initial text adr len
:sch&repl	ad1 n1 adr2 n2 str -- b	Search for text1 starting at END. replace just one occurrence
:sch&replCI	ad1 n1 adr2 n2 str -- b	Case Insensitive version of :sch&repl
:replall	ad1 n1 adr2 n2 str --	replace all occurrences of ad1 n1 with ad2 n2
:replallCI	ad1 n1 adr2 n2 str --	Case Insensitive version of :replall
:split	char str -- array	find all substrings delimited by: 1) start of string and char 2) and char and char 3) and char and end of string return all of them as an array of string objects allocated in the heap, the array object is also allocated in the heap
:remove-extra-chars	char str -- str2	Remove leading and trailing chars, removes more than one char(s) occurring consecutively between words. Original string is untouched, new heap string is returned.
:apply	xt str -- <varies>	apply xt to every item in the str, str is unchanged
:applyIf	xt1 xt2 str -- <varies>	apply xt2 only if xt1 returns true
:map	xt str --	apply xt to every item in the str, str is changed
:filter	xt str -- arr'	return new str of chars that respond true to xt
:count	xt str -- cnt	count the number of times xt returns non-zero
:dequote	str --	remove the leading and trailing 's from strings created using the o{ 'a string' ... } technique

File Class (subclass of Object)

:init	adr len fileObj --	adr len must be the name of the file
:fam	fam fileObj --	set the file access method
:open	fileObj --	open the file

:delete	fileObj --	delete the file
:create	fileObj --	create the file
:flush	fileObj --	flush the file
:size	fileObj -- n	return the size of the file in bytes
:read	fileObj -- strObj	read the entire file and return the contents in a string object
:read-line	max fileObj -- strObj true false	read one line from the file up to a maximum of MAX bytes, return the line as a string object under true. If there are no more lines then just return false
:write	adr len fileObj --	overwrite the entire file with adr len
:write-line	adr len fileObj --	write one line to the file at the current position
:!pos	n fileObj --	reposition the file to n bytes from the start
:@pos	fileObj -- n	return the current position of the file
:close	fileObj --	close the file
:free	fileObj --	FREE the memory allocated for the file name and the one-line and read buffers

Int Class (subclass of Object)

Int objects are simple objects for classic Forth one cell integers.

>int	n -- obj	create an int object in the heap initialized to n
:!	n obj --	store n in the obj
:@	obj -- n	place the value of the obj on the stack
:.	obj --	print the contents to the console
:=	obj1 obj2 -- f	f is true if the two int objects contain the same data
:>	obj1 obj2 -- f	f is true if the obj1's data is greater than obj2's

Flt Class (subclass of Object)

Flt objects are simple objects for classic Forth float sized numbers.

>flt	(-- obj) (F: r --)	create a flt object in the heap initialized to r
:!	(obj --) (F: r --)	store r in the obj
:@	(obj --) (F: -- r)	place the value of the obj on the FP stack
:.	obj --	print the contents to the console
:=	obj1 obj2 -- f	f is true if the two flt objects contain the same data
:>	obj1 obj2 -- f	f is true if the obj1's data is greater than obj2's

Hash-Table Class (subclass of Object)

An :insert with an existing key will simply over-write the value.

:insert	val key-addr key-len ht --	insert a new value-key pair
:@	key-addr key-len ht -- val true false	return the value associated with the key under flag
:delete	key-addr key-len ht -- flag	delete the node under success flag
:d	ht --	print the entire contents to the console

Hash-Table-m Class (subclass of Hash-Table)

An :insert with an existing key will add another key/value pair to the table, not over write the value.

The same key may have multiple values. Use :next to inspect for additional values.

:next	ht -- val true false	use after :@ to return additional values for same key, if any
-------	------------------------	---

ⁱ Compact Dispatch Tables for Dynamically Typed Object Oriented Languages, Jan Vitek Univ. of Geneva Object Systems Group CUI, R. Nigel Horspool Univ. of Virginia Dept of Computer Science

ⁱⁱ A Fast String Searching Algorithm, Robert S. Boyer Stanford Research Institute, J Strother Moore Xerox Palo Alto Research Center