# Message Passing Virtual Machine

Author, Alan Israel M

***Abstract*— To create an object-based virtual machine. To extend an object's capability by adding methods. To use the virtual machine to subclass from objects, add methods, create instances and use them in expressions.**

## I. Introduction

In a object-based virtual machine, we build a higher level of abstraction by modeling memory as an Array of Objects and referred using pointers known as Object-oriented pointer or OOP for short. Each object itself is an variable length array of machine words each of which is big enough to store a pointer. The first few words hold a header and is always present in every object. This is followed by words holding an object's structure variables. When reporting the size of an object in memory we exclude the space taken up by header words. This is a convention. In our simple virtual machine, we reserve two words for the header.

## II. Problem statement

To create **ovmPoint**.c and to create a subclass of Object called Point with two instance variables named x and y. To populate the Point's dispatch table with methods #x:y: that will create a new instance Point with the two arguments corresponding to x and y values. To create accessor methods #x and #y to return the receiver's x and y values respectively. Another method #basicSize that will return the number of bytes in memory to hold each instance. To create a method #offset: that will take another Point instance as argument and then add its x and y values to the corresponding values of the receiver

## III. Research

In the message passing virtual machine, the memory is modeled as an array of objects. Anytime memory is allocated to object, it is always referred to by it's internal Object Oriented Pointer(**OOP**). The internal members are always accessed only through methods associated with the Object, never directly.

A **dispatch table** holds the list of valid methods for object of the same kind. A dispatch table can be modified or extended at run time by adding new methods, changing the method bound to a selector or even deleting a binding. Dispatch table is also known as a **virtual table** or **vtable** because it is part of an object's header and not part of Slots. It is inaccessible to methods which refer to Slots. The collection of functions are called virtual function table. The functions are stored in a dictionary which the object memory can dispatch.

Therefore, the virtual table defines behavior and object defines computation. To allocate memory for an object, a primitive method is required. This method is looked up in a vtable. But this vtable is also an object that needs to be allocated in **ObjectMemory**. This circular dependency is broken by creating a special vtable called a Proto vtable.

With this loop, **Classes** can be created from Object vtable but not Proto vtable. A class object contains members like names of instance variables, constants, shared dictionaries and so on. Class is typically the first object for a particular type.

With the above information, the machines is ready for **Bootstrapping**. First the vtables for atoms, proto, objects, symbols and closures are created. The vtable entries for each of these are manually setup and methods are added directly to the **Proto vtable**. Then singleton objects for **Proto**, **Object**, **Symbol** and **Closure** can be allocated and tied to their vtables.

There are four methods to lookup method selectors in a vtable and to add new method entries to a vtable - **lookup**, **add_method**, **allocate** and **delegate**.

Now the Virtual Machine is ready for creating Class objects and creating instances from these classes.

## IV. Resources and methods

With the machine Bootstrapped, up and running, we start adding methods to Symbol and Closure vtables.

The subclass Point is created. But before creating this subclass, Point vtable must be created and made to point to the Object vtable. This is the dispatch table that holds the valid operations on the Point instances. After creating the Point vtable, the subclass Point is created and made to point to its vtable i.e, Point vtable.

Now the Point vtable has to be populated with the accessor methods that is required to be operated on the instances of Point. The methods are:

**Point>>#x:y**: "method to create a new Point instance with the given x and y values and return that instance"
**Point>>#x** "method to return receiver's x value"
**Point >>#y** "method to return receiver's y value"
**Point>>#offset**: "method to add argument's x to receiver's x and argument's y to receiver's y and return the receiver. ignore overflow for now"
**Point>>basicSize**: "method to return the size of memory allocated to hold for an instance "

Once these methods are created and added to the vtable with the selector and the closure, the instances can be created.

## V. CODE FLOW

A structure is created for the object Point with the instance variables and a refernce to the Object's vtable. The Point vtable and the subclass Point is created. The atomic symbols are created and these selectors are bound to their methods in the Point vtable.

The methods are defined as simple C functions. Selectors for these functions are defined by passing message to Symbol class and creating symbol instances. These selectors are bound to their method definitions in the Point vtable by message passing with the add_method function.

The instances are created by sending messages to the Point subclass. A start point and a delta point is created. These points are added and an end point is created using the offset accessor method.

As Proto vtable is the first object that is created, it is delegated to the Object vtable which is created after the creation of Proto vtable. This means that the Proto vtable can now inherit the behavior of the Object vtable. After this dependency has been created, the subclasses for creating new instances can be created.

For every subclass, a dispatch table is required. This dispatch table acts as the lookup table for the methods that needs to operated on the instance variables of the subclasses.

Therefore using the above analogy, the Symbol vtable and the subclass Symbol was created and the dispatch table was populated with 3 basic functions. This Symbol subclass is used to create new atomic symbols that are the selectors.

Then the Point vtable and Point object is created. The Point vtable is delegated to the Object vtable. The Point subclass points to Point vtable.

The instances for the Point class can now be created. The 5 accessor methods are:

1) **Point_newp** - creates new Point instance
2) **Point_x** - returns x
3) **Point_y** - returns y
4) **Point_offset** - returns new Point by adding 2 points
5) **Point_basicSize** - returns the size of the object

Code is attached as part of Appendix.

## VI. OBSERVATIONS AND ANALYSIS

The atomic symbols are created using the Symbol subclass. We need 5 atomic symbols as we have 5 accessor methods and have to assign selectors for each. The symbol for these selectors are allocated by creating new Symbol objects. The Symbol object points to the Symbol vtable as can be seen below.

The **Proto vtable**'s memory dump:
vt **0x804d068** (Proto_vt) parent **0x804d0c0** (Object_vt) size 4 tally 4

| no key | value |
|---|---|
| 0 0x804d1c8 (#lookup | ) 0x804d3b8() |
| 1 0x804d1f8 (#add_method | ) 0x804d3d0() |
| 2 0x804d230 (#allocate | ) 0x804d418() |
| 3 0x804d298 (#delegate | ) 0x804d430() |

The **Object vtable**'s memory dump:
vt **0x804d0c0** (Object_vt) parent (nil) (_vt) size 2 tally 1

| no key | value |
|---|---|
| 0 0x804d380 (#basicSize | ) 0x804d048() |

The **Symbol vtable**'s memory dump:
vt **0x804d118** (Symbol_vt) parent **0x804d0c0** (Object_vt) size 4 tally 3

| no key | value |
|---|---|
| 0 0x804d2d0 (#new: | ) 0x804d860() |
| 1 0x804d250 (#print | ) 0x804d878() |
| 2 0x804d350 (#length | ) 0x804d8c0() |

The **Symbol** subclass memory dump:

obj **0x804d850** (Symbol) vt **0x804d118** (Symbol_vt)

The **Point vtable**'s memory dump before adding methods:

vt **0x804d310** (Point_vt) parent **0x804d0c0** (Object_vt) size 2 tally 0

The **Point** subclass memory dump:

obj **0x804da78** (Point) vt **0x804d310** (Point_vt)

The memory dump for instances are shown below. since these instances are allocated using the allocate function of the Proto vtable, the first hexcode suggests the size, the second suggests the vtable reference and this is followed by the instance variables. But while dumping the memory, it should be noted that from the allocate method in our Bootstrap program, 2 locations are allocated for size and vtable pointer respectively. So the 2 locations behind the actual pointer must be noted. Hence the gdb function - **x/10x (void *)pointer-0x8** is used. This will show the size, vtable pointer, x value and y value.

The **Point vtable**'s memory dump after adding methods:

vt **0x804d310** (Point_vt) parent 0x804d0c0 (Object_vt) size 8 tally 5

| no key | value |
|---|---|
| 0 0x804d9a8 (#x:y: | ) 0x804da88() |
| 1 0x804d9d8 (#x: | ) 0x804daa0() |
| 2 0x804da08 (#y: | ) 0x804dae8() |
| 3 0x804da38 (#offset: | ) 0x804db00() |
| 4 0x804d380 (#basicSize | ) 0x804dad0() |

**start** object memory dump (10,20):

obj **0x804dab8** (start) vt **0x804d310** (Point_vt)

**0x804dab0**: 0x00000010 **0x0804d310**

0x0000000a 0x00000014

From the above we can see that the first field specifies the size and the secod field the vtable pointer. The remaining two arguments are the values of x and y - start(10,20).

**delta** object memory dump (5,8) :

obj **0x804db40** (delta) vt **0x804d310** (Point_vt)

**0x804db38**: 0x00000010 **0x0804d310**

0x00000005 0x00000008

**end** object memory dump (15,28) :

obj **0x804db58** (end) vt 0x804d310 (Point_vt)

**0x804db50**: 0x00000010 0x0804d310

0x0000000f 0x0000001c

The size of the object start was noted to be 16 bytes as returned from the basicSize function. This is because, when the new point object is created, it calls the allocate method which alloctes a size of 2*sizeof(vtable *)+sizeof(mypoint).

The sizeof vtable is 4 bytes and size of mypoint is 8 bytes. It is 8 bytes because it has two object pointers of size 4 bytes each.

Therefore the total size was found to be 16 bytes.

## VII. ISSUES FACED

The size of the allocated object Point instance was observed to be 16 bytes. But the structure in C for object Point has a pointer to vtable, and two instance variables of type object. The size when calculated by hand was turning out to be 32+2(4) = 40 bytes. But as noticed in the GNU debugger it was 16 bytes since the allocate method was allocating 8 bytes for size and vtable pointer. This was then added to the size of the two object pointers which was 8 bytes and the total size of the Point instance was found to be 16 bytes.

## VIII. RESULTS

The ovmpoint.c successfully created Point subclass and delegated it to the Object vtable. The instances start point (10,20) was created and delta point (5,8) was created. These were then added using message passing and a new end point was created (15,28). The basicSize function returned the size of the instance point to be 16 bytes.

## IX. REFERENCES


*1) Objectmodel.txt*

*2) Ovm.c*

*3) LMS MPVM.pdf*

*4) Ovmhello.c*


## X. APPENDIX

```
#include "ovm.c"

// Object subclass: #Point instanceVariableNames: 'x y' "create a new class Point(x, y) by subclassing from Object"
struct mypoint { _VTABLE_REF; struct object *x; struct object *y; };
typedef struct mypoint *HPoint;

static struct vtable *Point_vt;
```

```
static struct object *Point;

static struct symbol *xpyp;
static struct symbol *get_x;
static struct symbol *get_y;
static struct symbol *offset;

static struct object *Point_newp(struct closure *cls, struct object *self,int x,int y)
    {
        HPoint clone = (HPoint)send(vtof(self), s_vtallocate, sizeof(struct mypoint));

        clone->x = i2oop(x);
        clone->y = i2oop(y);
        return (struct object *)clone;
    }

static struct object *Point_x(struct closure *cls, struct mypoint *self)
    {

        return self->x;
    }

static struct object *Point_y(struct closure *cls, struct mypoint *self)
    {
        return self->y;
    }

static struct object *Point_offset(struct closure *cls, struct mypoint *receiver, struct mypoint *delta)
    {
        HPoint offset_point = (HPoint)send(vtof(receiver), s_vtallocate, sizeof(struct mypoint));

        int offset_x = oop2i(send(receiver,get_x))+oop2i(send(delta,get_x));
        int offset_y = oop2i(send(receiver,get_y))+oop2i(send(delta,get_y));

        offset_point->x = i2oop(offset_x);
        offset_point->y = i2oop(offset_y);
        return (struct object *)offset_point;
    }

static struct object *Point_basicSize(struct closure *cls, struct object *self)
    {
        assert(self);
        assert(szof(self) < 4096); // catch bad pointers
        return i2oop(szof(self));
    }

int main(int argc,char *argv[])
    {
        init_ovm();

        xpyp = (typeof(xpyp))send(Symbol,s_newp,"x:y:");
        get_x = (typeof(get_x))send(Symbol,s_newp,"x:");
        get_y = (typeof(get_y))send(Symbol,s_newp,"y:");
        offset = (typeof(offset))send(Symbol,s_newp,"offset:");
        s_basicSize = (typeof(s_basicSize)) send(Symbol, s_newp, "basicSize");

        Point_vt = (typeof(Point_vt))send(vtof(Object),s_vtdelegate,"Point");
        Point = (typeof(Point))send((struct object *)Point_vt,s_vtallocate,0);

        dump_vt(Proto_vt);
        dump_vt(Object_vt);
        dump_vt(Symbol_vt);
        dump_obj(Symbol);
        dump_vt(Point_vt);
        dump_obj(Point);

        assert(vtof(Point) == Point_vt);

        send(Point_vt, s_vtadd_method, xpyp, (method_t)Point_newp);
        send(Point_vt, s_vtadd_method, get_x, (method_t)Point_x);
        send(Point_vt, s_vtadd_method, get_y, (method_t)Point_y);
        send(Point_vt, s_vtadd_method, offset, (method_t)Point_offset);
        send(Point_vt, s_vtadd_method, s_basicSize, (method_t)Point_basicSize);

        struct object *start = send(Point, xpyp, 10, 20);
        struct object *delta = send(Point, xpyp, 5, 8);

        struct object *end = send(start, offset, delta);
        struct object *size_start = send(start, s_basicSize);
        struct object *proto_vt = send(Proto_vt, s_basicSize);

        dump_vt(Point_vt);
        dump_obj(start);
        dump_obj(delta);
        dump_obj(end);

        printf("Size of start: %d\n", oop2i(send(start,s_basicSize)));

    }
```