Object Oriented Programming in Assembly

Ethan J. Eldridge

December 15, 2011

1 Introduction

The Object Oriented Programming (OOP) Paradigm is an extremely important one in todays world. The ability to construct applications and programs using data structures is a powerful one. Encapsulation, abstraction, and modularity are just a few of the many features of OOP. As a curious computer scientist, one might wonder how the structures and dynamic memory needed for the paradigm exist in the low level of assembly. In this paper, we take a small sample of a C++ program and examine it in the context of it's native assembly language.

2 Neccesary Information

In order to understand the read outs, memory dumps, and other information presented, the following table is presented for clearity. It is not expected for people to memorize these, but to at least have this for reference.

Register	Full Name	Meaning
eax	Accumulator Register	All calculations occur here, holds data to read/write to a port
ecx	Counter Register	All counting related instructions use this register
edx	Data Register	holds significant bits from math operations as well as the address to a port to write to.
ebx	Base Register	A general purpose pointer register, useful for storing extra pointers or calculations
esp	Stack Pointer	Stores the most recently allocated address where registers should be spilled, when a function is called, all parameters and the return address are pushed onto the stack. The base pointer is then set to be equal to the stack pointer and all further addressing is done using the base pointer.
ebp	Base Pointer	In functions that store parameters or variables on the stack, the base pointer holds the location of the current stack frame. In other situations, however, EBP is a free data-storage register.
esi	Source Index	Used to read data from memory, or for extra storage space, holds location of an input data stream
edi	Destination Index	Holds the write address of all string operations, and used to write data out of the accumulator
eip	Instruction Pointer	Poits to the memory address which the processor willl try to execute next.
eflags	Flag Register	All comparisons and things of that nature are stored here

A program is not always stored in a continuous block of memory, rather it can be segmented into different locations, to deal with this there are registers that hold offset values for accessing certain data types.		
cs	Code Segment	Specifies in memory where the processor instructions are, and all instructions referenced by the eip.
ss	Stack Segment	Holds the offset to locations needing the stack
ds	Data Segment	Holds the neccesary offset to instructions requiring program data.
es	Extra Segment	The processor assumes the edi references this segment during string manipulation, and uses it to hold program data
fs	Extra Segment	Simply called F because F comes after E, does the same thing as the es
gs	Extra Segment	same as above
Memory models that use pages instead tell the segments to point to the same place with the exception of fs and gs, which are sometimes used to point at thread specific data.		

3 The Toy Program

Below is the program we will be examining. For each piece of the program we will look at analogous examples in the MIPS assembly language as well as the native assembly produced by the GCC[1] disassemlby code produced by the integrated development environment Code Blocks [2] The key data structures to be examined will be a simple structure, which is a crux of object oriented programming.[3]

```
#include <iostream>

struct Test{
     int16_t age;
     int16_t pay;
     int16_t getAge(){
         return age;
     }
};

int main()
{
     Test T;
     T.age = 3;
     T.pay = 4;

     std::cout << &T << std::endl;
     std::cout << (void*)Test::getAge << std::endl;

     return 0;
}
```

[1]GNU Compiler Collection - included as a compiler in CodeBlocks IDE.

[2]http://www.codeblocks.org/

[3]Some may argue that Classes are the crux, and I agree, but a Struct is simply a naturally public class, and is much easier to work with in my oppinion and thus we use that here.

4 Structs in C++ and in Assembly

Known also as records, structures are naturally public objects that store data fields into a single object. Their worth in object oriented programming is unstatably large. Before we tackle the Toy Program's structure we can look at a assembly level rendition of a structure, and with that knowledge under our belt the Toy Program will be just that, a Toy.

Whenever you create an instance of a structure, the neccesary amount of memory for ALL the datafields inside of it is allocated. This is one of the many reasons why most programming languages require a size for arrays declared within an object. Once this chunk of memory is created, the instance pointer is placed at the top and all methods,variables, and what-have-you is found by an offset value from the beginning of the structure. In assembly, if you already know how big everything, and how its all ordered, then why bother with using OOP instead of a byte array and memorizing offset values for everyone of your structures? It may be the negative tone of the previous sentence, but I for one would prefer not to memorize protocols for access fior every single one of my data structures, even in a small program, and definately not in a large program, not too mention what happens if you were to add in another field in the middle of the others. In OOP assembly languages, the typical dot notation is supported. If we had a student structure with the second data field as Major and a first data field of a string type(which is 4bytes long), then some code with byte arrays might be:

```
var
     John: Student;
mov( ax, (type word John[4]) );
```

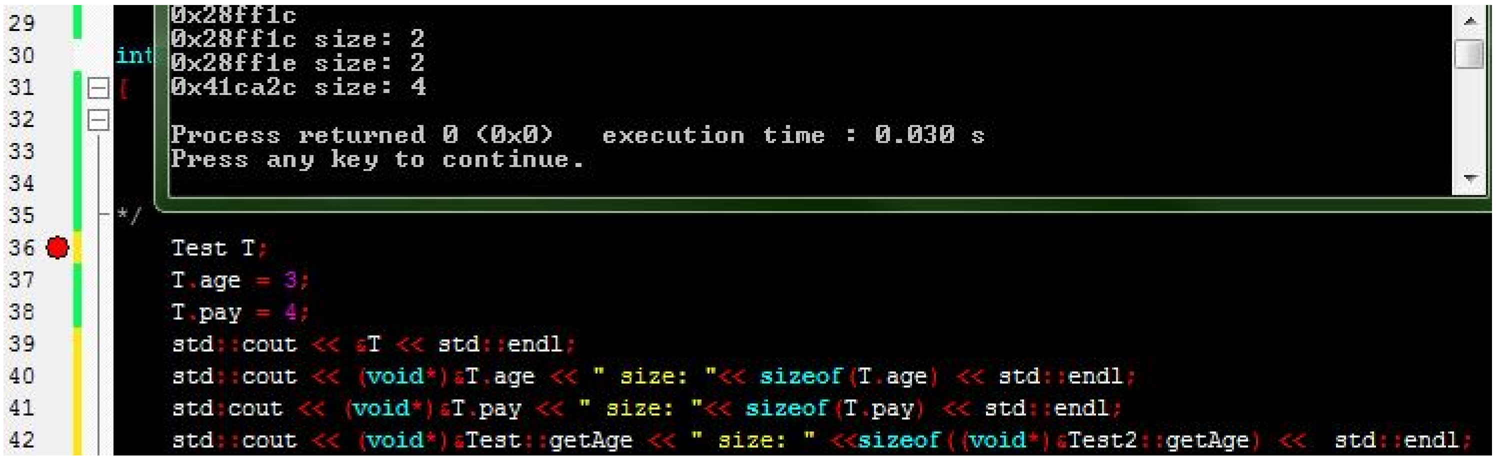
However, if were to use an OOP approach, and an assembly language such as HLA(High Level Assembly) that supports the dot operator, we could use the more easily readable and more robust form of

```
mox( ax, John.Major);
```

So then, lets move onto our toy example and look at what we actually need to consider. First, we must look at the first bit of code, the data structure itself.

```
struct Test{
     int16_t age;
     int16_t pay;
     int16_t getAge(){
         return age;
     }
};
```

As we've just seen, the fields, age and pay, can be accessed just the same as the variables in the John Example. And if we were to print out the address of these variables in relation to the instance address, we'd see what we expect for these two fields.Figure 1 If we take a closer look at the figure, we can see that the memory address of the structure is displayed first, offsetting by 0 gets us the first field, and then offsetting using the dot notation we can check out that the other variable is stored exactly where we predicted it should be! However, looking at the address of the getAge fucntion, we can see that it's stored somewhere far far away in memory. The reason this is, is because a method or procedure stored inside of a structure is not a contigious chunk of memory in relation to it's parent object. Rather, the parent stores the *address* of the instruction, and then calls that after placing any neccesary variables into their proper place in relation to the stack and base pointers. The assembly code for just the instructions for setting the age,pay, and calling the getAge functions are shown in the next code listing.



```
0x28ff1c
0x28ff1c size: 2
0x28ff1e size: 2
0x41ca2c size: 4

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

```
int
{
*/
    Test T;
    T.age = 3;
    T.pay = 4;
    std::cout << &T << std::endl;
    std::cout << (void*)&T.age << " size: "<< sizeof(T.age) << std::endl;
    std:cout << (void*)&T.pay << " size: "<< sizeof(T.pay) << std::endl;
    std::cout << (void*)&Test::getAge << " size: " <<sizeof((void*)&Test2::getAge) <<  std::endl;
```

Figure 1: The address of our Test structure and it's fields

```
void SimpleShow(){
    Test T;
    T.age = 3;
    T.pay = 4;
    T.getAge();
}
Becomes:

00401318 push   %ebp                        Save the base pointer
00401319 mov    %esp,%ebp                   Overwrite base with stack pointer
0040131B sub    $0x14,%esp                  Allocate redzone space for function
0040131E movw   $0x3,-0x4(%ebp)             Assign 3 to age
00401324 movw   $0x4,-0x2(%ebp)             Assign 4 to pay
0040132A lea    -0x4(%ebp),%eax             Load the address of the first local variable (T)
0040132D mov    %eax,(%esp)                 Move that address to the stack pointer
00401330 call   0x41cadc <Test::getAge()>   call the function
00401335 leave                              leave SimpleShow
00401336 ret
```

Now some things from the code listing may not be clear, the first 2 lines are primarily logistic code that allows us to return back to our main function out of the SimpleShow. The third line is a result of our compiler creating space on the stack for the function call that it knows is coming. This is efficiency of our compiler at work and for the purpose of this example is not neccesary to know. The next two lines are fairly easy to understand, the variables age and pay are set using their offset value from the base pointer. Finally, the command *lea* or load effective address, loads the first parameter (in this case the first local argument of our function SimpleShow) into the second argument, the accumulator. We then move the accumulator to the stack pointer so that we can use the variables we need once we've moved to our function. And then we go to our function itself, this is shown in the next listing.

This listing is of the getAge function itself. The assembly code is fairly short, we begin with the logistics of saving our stack pointer so that we can return to where we need to as well as access all the variables we need for this function. We then grab the first parameter of our function using $mov 0x8(\%ebp), \%eax$ and we store this in the accumulator. Those familiar with programming with objects will have no trouble understanding where this parameter came from, but for those of us who dont, the first parameter to any function in a method stored in a class is a pointer[4] to the instance calling the method. We then move

[4]Normally stated as this in C and Java, and self in Python

the extended accumulator register into the ax register, the ax register. according to my sources, it the least significant bits of the eax register, and is only 16 bits. You might wonder, how can we possibly place a 32 bit register into the 16 bit one? In all honestly, I'm really not sure how it works either. But I stepped through the memory dump myself and the correct value is placed into the ax register and this does update everything. My closest guess, is that for the outdated 16 bit registers there may be a special syntax, from wikipedias page on the x86 mov instruction, the command movzx eax, ax, or in our syntax, movzx ax eax, would place the value in ax into eax with zeros padding it all. However, because this syntax is not what we have going on in the code listing, and we're using mov and not movzx[5] I doubt this is the case.

```
int16_t getAge(){
         return age;
}


0041CADC push   %ebp
0041CADD mov    %esp,%ebp
0041CADF mov    0x8(%ebp),%eax
0041CAE2 mov    (%eax),%ax
0041CAE5 leave
0041CAE6 ret
```

5 Linked Lists

Moving on from our previous example, there are other data structures to consider. Specifically, the linked list. We can implement this easily in MIPS, but first, lets go over what a linked list is. A linked list is a data structure that can be anysize, all traversals must occur in sequential access. The first node in the list holds a pointer to the next, and the second to the third, and on and on until the last node which contains holds a null pointer in the address where the others contained the next node. The benefits of a linked list are that they can grow to any size unlike arrays, also, because they use pointers to memory addresses themselves, they do not need to be stored in a contiguous block of memory. The first node is commonly called the head of the list, and the last is called the tail. In C++, this structure can be created in the following way:

```
struct node{
    node * prev;
    node * next;
    int val;
};
void traversal(){
    node * head = new node();
    node * node2 = new node();
    node * node3 = new node();
    node * tail = new node();
    head->val = 1;
    node2->val =2;
    node3->val=3;
    tail->val=4;
    head->next=node2;
    node2->next=node3;
```

[5]Although this could be a pseudo instruction

```
    node3->next=tail;
    tail->prev=node3;
    node3->prev=node2;
    node2->prev=head;
    node *temp;
    temp = head;
    while(temp->next != 0){
        cout << temp->val << endl;
        temp = temp->next;
    }
};
int main()
{
    cout << "Hello world!" << endl;
    traversal();
    return 0;
}
```

This is a very quick and dirty implementation of a hard-coded linked list, and in a real application this would be far more robust and rely on many other ideas such as templates, functions, and other things. However, we do it this way in order to draw a clear analogy in how this can be done in MIPS. We'll need a head, and a way to point to the next node. This can be done rather simply with the following bit of code:

```
.data
head:
elmnt01:  .word  1
          .word  elmnt02
elmnt02:  .word  2
          .word elmnt03

elmnt03:  .word  3
          .word elmnt04

elmnt04:  .word  5
          .word  elmnt05

elmnt05:  .word  7
          .word  0
sep:      .asciiz "  "
linef:    .asciiz "\n"
endmess:  .asciiz "done\n"
```

We can use the pointer to the head to start out at elmnt01 and then we can use the .word values for the next node to move throughout the list. In the final node, we use a null pointer to signify the end of our list. In orderal to traverse this, we can create a simple loop that will print out our value and move throughout the list

```
main:
          la     $s0,head       # get pointer to head

loop:     beqz   $s0,done       # while not null
```

```
          lw     $a0,0($s0)     #   get the data
          li     $v0,1          #   print it
          syscall               #
          la     $a0,sep        #   print separator
          li     $v0,4
          syscall
          lw     $s0,4($s0)     #   get next
          b      loop

done:     la     $a0,linef      # print end of line
          li     $v0,4
          syscall               # print ending message
          la     $a0,endmess
          li     $v0,4
          syscall

          li     $v0,10         # return to OS
          syscall
```

This code is straightforward, but we'll walk throughit anyway. The main starts out with us loading the address of the head node, then we begin our loop that continues until we reach an address that is null. Inside the loop we load and print teh data from each node and also output the seperator so we can see the values easily. Once the loop has fnished we go down to the done label. All that happens after that, is that we just load in our ending message and then finish up and return to the operating system.

6 The Corner Stone - Objects

While we went over structures, when you create a class a little bit more of sophistication occurs. The variables are found by offsetting as well, the methods are stored in a jump table. This structure is generally created in a dynamic link library and loaded in at runtime *as needed*. This dynamic loading allows for multiple programs to reference the functions and share system resources. The use of a jump table to reference functions of a class is a powerful method, but conceptually, not very different from what the structure we saw before do. It used the address of it's function, which was stored somewhere else in memory, to access the function. So why use a class versus a structure?

A class allows *inheritance* , and this is a big deal! In C++ this can be done easily with say the following code:

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
```

```
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

In the listing we can see that the CRectablge and CTriangle classes don't define any parameters on their own, but rather they inherit them all from their parent class CPolygon. While this in itself is powerful, we can take another step forward with another OOP paradigm. Polymorphism! This is the ability of a pointer to a derived class to it's base class. Using this ability we can then create what are commonly known as virtual classes. These are very similar to interfaces in Java, and a pointer to a base class to point to it's child classes and in implementing various applications, such as those that use finite state machines, this is a wonderful use. The following code does this:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
```

```
      { return (width * height / 2); }
  };

int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Now this is all well and good, but what about in our low level world of assembly? Well, turns out using an assembly language like HLA (High Level Assembly), we're able to do the same things! Let's take an example you might find in some type of math program.

```
type
point: class
var
x:int32;
y:int32;
method distance;
endclass;

type
point3D: class inherits( point );
var
z:int32;
override method distance;
endclass;
```

Since this is the first bit of HLA we've seen so far, I'll explain each bit a little bit even though it's rather clear. We've created a new type which is called a point, and this point is a class by its *point* : *class* decleration. To create it's data fields, we state that we're declaring variables with the keyword *var* and then create two integers x and y. The last bit creates a method called distance, and then ends the first class. Note that this code doesn't actually work, not yet anyway, the method distance isn't actually created, nor have we created constructors. The second type inherits from the previous class and also has the values for x,y, and distance. Because we've added a new point, the distance equation has in fact changed, and in order to call the correct method, we use the overide keyword to overide the parents class method when dealing with a point3D one.

Moving on from this simple example, lets actually start placing in the methods and constructors. The following code segment allows for this.

```
type
point2D:class
const UnitDistance: real32 := 1.0;
var
x: real32;
y: real32;
```

```
static
LastDistance: real32;
method distance( fromX: real32; from:real32); returns ("st0");
begin distance;
fld( this.x );
fld( fromX ); // Compute (x-fromX)
fsub();
fld( st0 ); // Duplicate value on TOS.
fmul(); // Compute square of difference.
fld( this.y );
fld( fromY ); // Compute (y-fromY)
fsub();
fld( st0 ); // Compute the square of the difference.
fmul();
fsqrt();
fst( point2D.LastDistance ); // Update shared (STATIC) field.
end distance;
procedure Create(); returns("esi");
endclass;
procedure point2D.Create; @nodisplay;
if(esi = 0) then
push(eax);
mov(malloc(@size(point2D)),esi);
pop(eax);
endif
mov( &point2D._VMT_,this.pVMT_);
mov(0,this.x);
mov(0,this.y);
mov(0,this.LastDistance);
end Create;
...
```

For brevity I've left out the creation of the 3 dimensional point, however before I go into the code, it's important to note that the line $mov(\&point2D.\_VMT\_, this.\_pVMT\_)$;, works only for the 2D point. For the 3D constructor, you must specify the class name as point3D where point2D is. Moving on, it's easy to see the code we had before is, but we've also introduced a constant for units, and actually implemented the distance method as well as the constructor for the class. The distance method uses some floating point operations in HLA and computes the standard equation of $\sqrt{x^2 + y^2}$, and updates the static field of last distance. The creation procedure makes room for the instance via the malloc command and the esi register. After creating the room, we actually instantiate our variables and then move on. HLA is a little long winded on this point, and I hope that the general ideas are expressed through this, but lets turn to a different example, a smaller compact example using MIPS.

The following code segment creates two objects that share a procedure and outputs their data to the console. Its a rather simple and toy example, but despite it's simplicity illustrates a small amount of what we've talked about in the previous examples. This example is static memory, we'll see another bit with dymanic memory after this one. Each object is a simple ascii string and an address that allows the object to use the function print. The main program itself loads the address of each object, throws the right variables into the right registers as the protocol of the procedure requires, and then jumps and links to the procedure. After calling both functions the program ends.

```
.globl   main
          .text
main:                                   # object1.print();
          la       $a0,object1          #   get address of first object
          lw       $t0,0($a0)           #   get address of object's method
          jalr     $t0                  #   call the object's method

                                        # object2.print();
          la       $a0,object2          #   get address of second object
          lw       $t0,0($a0)           #   get address of object's method
          jalr     $t0                  #   call the object's method

          li      $v0,10                # return to OS
          syscall
# Objects constructed in static memory.  An object consists of the data
# for each object and a jump table of entry points common to all objects
# of its type.
          .data
object1:
          .word    print                      # Jump Table
          .asciiz  "Hello World\n"            # This object's data

object2:
          .word    print                      # Jump Table
          .asciiz  "Silly Example\n"          # This object's data

# Single copy of the print method
# Parameter: $a0 == address of the object
          .text
print:
          li       $v0,4                     # print string service
          addu     $a0,$a0,4                 # address of object's string
          syscall                            #
          jr       $ra
```

While static objects and memory fields are nice, dynamic memory is an even better idea. Using dynamic memory, as long as we keep track of where variables are storerd then we can create objects as big or small as neccesary. The next code listing shows a small example of dynamic memory allocation during runtime. First, we allocate the memory for our object, and then load it into memory, once thats done we initialize it's fields to be the right addresses for the functions it has (print and read). We then call the objects functions and observe the results. The read method prompts the user for an input and the print outputs it to the screen.

```
.globl   main
          .text
main:                                   # object1 = new object();
          li       $v0,9                #   allocate 32 bytes
          li       $a0,32               #
          syscall                       #   $v0 = address
          sw       $v0,object1          #
```

```
         la      $t0,print          #   initialize jump table
         sw      $t0,0($v0)         #
         la      $t0,read           #
         sw      $t0,4($v0)         #

                                    # object1.read();
         lw      $a0,object1        #   get address of object1
         lw      $t0,4($a0)         #   get address of read method
         jalr    $t0                #   call the method

                                    # object1.print();
         lw      $a0,object1        #   get address of first object
         lw      $t0,0($a0)         #   get address of print method
         jalr    $t0                #   call the method

         li     $v0,10              # return to OS
         syscall

         .data
object1: .word   0
object2: .word   0

# print() method
# Parameter: $a0 == address of the object
         .text
print:
         li      $v0,4              # print string service
         addiu   $a0,$a0,8          # address of object's string
         syscall                    #
         jr      $ra

# read() method
# Parameter: $a0 == address of the object
#
         .text
read:
         move    $s1,$a0            # save object's address
         li      $v0,4              # print string service
         la      $a0,prompt         # address of object's string
         syscall                    #

         addiu   $a0,$s1,8          # $a0 = address of buffer
                                    #       in object
         li      $a1,24             # $a1 = size of buffer
         li      $v0,8              # read string service
         syscall

         jr      $ra                # return to caller

         .data
```

```
prompt:   .asciiz  "enter data > "
```

7 Final Comments

Object oriented programming in Assembly level languages opens up a lot of solid coding techniques and powerful tools for a programmer willing to use them. Most of the useful OOP assembly level code in HLA seems to be ale to implemented rather easily in higher level languages without the added burden of thinking on such a low level. However, understanding how data structures work at the lowest level is very helpful and can result in a better understannding of data structures in general. It's definately a beneficial piece of knowledge for any programmer to have some grasp of assembly level commands and structuring, but not neccesarilly required for many jobs.

8 References

Numerous Thanks to the following website for having such great tutorials and helpful information

```
http://chortle.ccsu.edu/assemblytutorial/Chapter-33/ass33_7.html
http://www.swansontec.com/sregisters.html
http://www.cpu-world.com/Arch/8088.html
http://webster.cs.ucr.edu/AoA/Windows/HTML/ClassesAndObjects.html#998258
http://webster.cs.ucr.edu/Page_TechDocs/Structures.pdf
http://www.cplusplus.com/doc/tutorial/polymorphism/
```