Derek Johnson, dej3tc

11/6/2019

Post-Lab 8 Report

Parameter Passing:

1) Passing by value is accomplished using the mov command while passing by reference is done using the lea command. In both cases, the value is moved from its spot on the stack into a register and then from that register to a new spot on the stack. In the case of pass by reference, the register used is rax(return register) and in pass by value the register used is eax. The code snippet on the right shows pass by value and the left shows pass by reference. As you can see, pass by reference is using the lea command to copy a location in memory into the register without using the DWORD PTR.

```
77          mov     eax, DWORD PTR [rbp-40]
78          mov     DWORD PTR [rbp-44], eax
```

```
86          lea     rax, [rbp-40]
87          mov     QWORD PTR [rbp-24], rax
```

2) In order to show the difference between passing objects by value and passing objects by reference, I have created a Dog object. I have also created a pass by value method *getHeight(Dog * d)* and a pass by reference method *getHeight2(Dog *& d)*. Below left is the assembly call of the first and below right is the assembly call of the second. As you can see they are very similar in their compiled state. The difference is just in the command used to copy the value into a register. The reference call uses lea command while the value call used the mov command. This means that the rax will contain the address of the object in the reference call and the value of the object in the value call.

```
128         mov     rax, QWORD PTR [rbp-56]
129         mov     rdi, rax
130         call    getHeight(Dog*)
131         mov     DWORD PTR [rbp-28], eax
```

```
132         lea     rax, [rbp-56]
133         mov     rdi, rax
134         call    getHeight2(Dog*&)
135         mov     DWORD PTR [rbp-32], eax
```

3) When an array is passed into a function, this is accomplished by moving the address of the start of the array into the register rax. The value in the rax register is then moved into the appropriate register to store the parameter. In my example below, since the array is the first parameter, the address of the start of the array is moved into rdi. The method that takes the command then uses the array start as a pointer and converts the method parameter type. This can be seen below. In my example, the start of my array is stored at address [rbp-16] as it is the first of 4 ints used in my main method.

```
20      int big = biggest(arr);
```

```
39          lea     rax, [rbp-16]
40          mov     rdi, rax
41          call    biggest(int*)
42          mov     DWORD PTR [rbp-4], eax
```

4) At the C++ level, passing by reference and passing by pointer are different. Hacker.io summarizes the differences by explaining the differences between pointers and references. When passing by pointer , the pointer can be null, something that can not be done with pass by reference. Additionally, in pass by reference, the reference can not be reassigned to another memory address in the method, while pointers can be.

At the assembly level there are differences but in the actual implementation they are small. I defined two methods, the first is pass by reference(left) and the second is pass by value(right). The assembly generated for these two can be seen below. In both cases, they use the same register, rdx to temporarily store the variable and in both cases they move this value into rdi and rsi. The difference is the way they get the value initially. Pass by reference pulls the address of the

value using lea while pass by pointer uses the mov command to get the value of the pointer.

```
124         lea     rdx, [rbp-44]
125         mov     rax, QWORD PTR [rbp-32]
126         mov     rsi, rdx
127         mov     rdi, rax
128         call    Dog::isOlder2(int&)
129         mov     BYTE PTR [rbp-34], al
```

```
130         mov     rdx, QWORD PTR [rbp-24]
131         mov     rax, QWORD PTR [rbp-32]
132         mov     rsi, rdx
133         mov     rdi, rax
134         call    Dog::isOlder3(int*)
135         mov     BYTE PTR [rbp-35], al
```

Objects:

1) In order to learn about how objects are laid out in memory I created an object Dog() with two data members, int age and int height. In order to store different data members associated with an object together, the compiler first determines how much space will be needed to store them. It then uses this to create a space in memory similar to an array. All the data about the object is stored next to one another. The call of the constructor assembly is below left and the assembly for the constructor itself is below right. Since there are two ints stored, the array space will be 8 bytes.

```
101         mov     edi, 8
102         call    operator new(unsigned long)
103         mov     rbx, rax
104         mov     rdi, rbx
105         call    Dog::Dog() [complete object constructor
106         mov     QWORD PTR [rbp-32], rbx
```

```
1   Dog::Dog() [base object constructor]:
2           push    rbp
3           mov     rbp, rsp
4           mov     QWORD PTR [rbp-8], rdi
5           mov     rax, QWORD PTR [rbp-8]
6           mov     DWORD PTR [rax], 0
7           mov     rax, QWORD PTR [rbp-8]
8           mov     DWORD PTR [rax+4], 16
9           nop
10          pop     rbp
11          ret
```

2) In order to access the data members of an object, the compiler has to keep track of where the “array” of data is stored. It does this by creating a “pointer” to the array and storing this pointer in memory. In our case, the Dog object we created in the main method is stored in space [rbp-32]. Thus when we call a method of that Dog object, the address for the data is moved into the rax variable. Then, once in the method, the assembler can traverse the array to find the data member it needs to access. Below left is the assembly for a call of *setAge(4)* on the dog object. Below right is the assembly for the setAge method which takes an integer and sets it equal to the age member of the dog object.

```
107         mov     rax, QWORD PTR [rbp-32]
108         mov     esi, 4
109         mov     rdi, rax
110         call    Dog::setAge(int)
```

```
12  Dog::setAge(int):
13          push    rbp
14          mov     rbp, rsp
15          mov     QWORD PTR [rbp-8], rdi
16          mov     DWORD PTR [rbp-12], esi
17          mov     rax, QWORD PTR [rbp-8]
18          mov     edx, DWORD PTR [rbp-12]
19          mov     DWORD PTR [rax], edx
20          nop
21          pop     rbp
22          ret
```

3) As explained above, in assembly, objects are really just pointers to arrays. As can be seen in the above right example. When calling a method on our *Dog d*, the pointer to the space in memory where the array with d’s attributes is located ([rbp-32]) is moved into the rax register.
4) The process core accessing a data member from inside a member function has been described in detail in 2) above. The process for access outside a member function is exactly the same, assuming the data member is public (else it is impossible). Below is an example of accessing a public data member ouside of a member function. The data member is the first object in the array that stores the objects information so only the front of the array is passed. It uses register rax to store the value temporarily.

```
mov     rax, QWORD PTR [rbp-32]
mov     eax, DWORD PTR [rax]
```

5) Member functions are explained extensively in part 3) of this section. The first parameter for all of the member functions is the address of the object. This means that for our object Dog d, every time a member function is called, the rdi parameter contains the address of d ([rbp-32]). The “this” pointer in assembly and the actual object “this” is pointing too are one and the same. This can be seen in the method call snippet below.

```
mov     rdx, QWORD PTR [rbp-24]
mov     rax, QWORD PTR [rbp-32]
mov     rsi, rdx
mov     rdi, rax
```

Sources Used:

https://hackr.io/blog/pass-by-reference-vs-pass-by-pointer

https://courses.cs.washington.edu/courses/cse351/13su/lectures/12-structs.pdf

For commands:
http://www.cs.cmu.edu/afs/cs/academic/class/15213-s12/www/lectures/08-machine-data-1up.pdf