Noah French (njf5cu)
Lab 8
10/31/17
Filename: postlab8.pdf
File Description: Exploration of parameter passing and object representation in assembly

Parameter Passing

1.)

To demonstrate how ints are passed by value in assembly, I created a C++ function that takes in an int and returns that int plus 1. In the main method, I created an int x = 1 and passed it into that function. I compiled the .cpp file using the following flags: -m64 -mllvm –x86-asm-syntax=intel -S -fomit-frame-pointer. The caller, in this case the main method, looked like the following in x86:

```
mov     dword ptr [rsp + 4], 1
mov     edi, dword ptr [rsp + 4]
call    _Z7intFunci
xor     edi, edi
mov     dword ptr [rsp], eax    # 4-byte Spill
mov     eax, edi
pop     rcx
ret
```

The parameter x = 1 is created by inserting the value 1 into the place in the stack designated for local variables: 4 bytes above the top of the stack. This makes sense, because ints are 32 bits (4 bytes) on my machine, so only 4 bytes are necessary to hold the int. That variable is then stored in the edi register to be passed into the callee function. The caller puts the parameter in edi (rather than rdi) because – as mentioned – ints are 32 bits on my machine. 32 bit standard registers are therefore used.

The callee function (which simply takes in an int parameter by value and returns the value plus one) is the following in x86:

```
mov     dword ptr [rsp - 4], edi
mov     edi, dword ptr [rsp - 4]
add     edi, 1
mov     eax, edi
ret
```

The parameter is passed into the function in the edi register. 1 is added to the parameter within edi, and then the value is moved to eax, where it can then be used by the caller.

Passing in other data types by value is largely the same as for ints, but different registers are used depending on the size of the data type. For example, the x86 code for a void callee function that simply takes in a char as a parameter is:

```
mov     al, dil
mov     byte ptr [rsp - 1], al
ret
```

Because a char is only 8 bits, the 8 bit registers are used for parameter passing, and the parameters are stored on the stack only one byte away from the pointer.

To see how the registers are used for pass by reference, I created a function called valueFunc that takes in a reference to an integer x. The function sets the integer x to equal x + 1 by passing it into a pass by value callee function. The callee function simply returns x + 1. Here is the x86 output for the caller function:

```
mov     rdi, qword ptr [rsp]
mov     edi, dword ptr [rdi]
call    _Z6addOnei
mov     rcx, qword ptr [rsp]
mov     dword ptr [rcx], eax
pop     rax
ret
```

The parameter is initially passed in through the rdi register rather than the edi register, because, though an int is 32 bits on this computer, a reference to an int is 64 bits on this computer. However, the caller must pass an int into the callee function. An int is 32 bits, so the caller function puts an int value in the edi register before calling the callee. The callee looks like the following in x86:

```
mov     dword ptr [rsp - 4], edi
mov     edi, dword ptr [rsp - 4]
add     edi, 1
mov     eax, edi
ret
```

The callee function looks the same as it did before, when the caller function also used 32 bit ints! It takes in its parameter in the edi register and puts the return value in the eax register.

2.)

In C++ passing an entire array into a function as a parameter isn't really doable. Instead, you pass in a pointer to the first element in the array. To demonstrate how passing an array as a parameter works in assembly, I created a C++ function called arrayFunc that takes in an int array of size 5, and iterates through it, setting each element to a value of 0. The function is pass-by value, so obviously it's not changing the actual values of these elements, but it is accessing each of them.

```
        mov     qword ptr [rsp - 8], rdi
        mov     dword ptr [rsp - 12], 0
.LBB1_1:
        cmp     dword ptr [rsp - 12], 5
        jge     .LBB1_4
        movsxd          rax, dword ptr [rsp - 12]
        mov     rcx, qword ptr [rsp - 8]
        mov     dword ptr [rcx + 4*rax], 2
        mov     eax, dword ptr [rsp - 12]
        add     eax, 1
        mov     dword ptr [rsp - 12], eax
        jmp     .LBB1_1
.LBB1_4:
        ret
```

The pointer to the first element is passed into the function through the rdi register. That memory location is then put into the stack 8 bytes below the stack pointer, and a counter is initiated 12 bytes below. One key line demonstrates how the computer gets from one element to the next:

```
mov     dword ptr [rcx + 4*rax], 2
```

The value at the current position in the array is stored in rcx, and the counter number (0-5) is stored in rax. The location in memory of any given element is determined by the value of the integer + 4 times the iteration that the counter is on.

3.)

Passing by reference generates the same assembly code as passing by pointer. In C++ there are subtle differences between pointers and references. A pointer can be reassigned to a different memory location whereas a reference cannot, for instance. But for parameter passing on an assembly code level, the two are essentially the same. They both hold memory addresses, and when they are passed into functions, the function is using that memory address. To illustrate this, I created two void functions. One simply takes in a reference to an int. The other simply takes in pointer to an int. Both functions have the same x86 code:

```
mov     qword ptr [rsp - 8], rdi
ret
```

Additionally, the caller function calls the callees similarly.

Objects

1.)

Object oriented programming languages like C++ are complicated tools that allow their user to bundle many different components (ints, arrays, methods, other objects) into a single class. Assembly and x86 are not object oriented. Assembly code is made up of a series of discrete operations in registers and memory. Assembly code must therefore do very specific things to maintain the structure of the higher-level code it represents.

A basic class is made up of two parts: the fields (variables) and the methods. When the constructor of a class is called, the variables are placed into the object being created, and therefore the memory, in sequential order. The variables are usually put into memory in multiples of 2 bytes so they can be easily located. When the program wants to find one of the variables of this object, it only needs needs to know the offset of that variable with respect to the first variable of that object (https://en.wikibooks.org/wiki/X86_Disassembly/Objects_and_Classes). In this way, the fields of an object stay close to one another in memory and can be easily accessed when necessary.

Just as there's no real difference between a reference and a pointer at a machine-code level, there is no big difference between a method and a function at a machine-code level. But there must be something to tie that method to its object or they would be useless. To solve this problem, when the method is called in assembly, the object whose method is being used is part of the method call. (http://www.ethanjoachimeldridge.info/resources/mipsoob.pdf).

2.)

I created a sample C++ class called Person with some public and private fields of different data types.

```
class Person {
public:
  bool human;
  long num;
```

```
private:
  int age;
  char initial;
  double weight;
};
```

In my main method I changed human to false, num to 24, then num to 25. The main method x86 code for those operations is the following:

```
mov    byte ptr [rsp - 40], 0
mov    qword ptr [rsp - 32], 24
mov    qword ptr [rsp - 32], 25
```

The two pieces of data are located next to one another in memory! They are separated by 8 bytes.

3,4.)

The previous example shows how data members can be accessed from outside a member function (within the main method). The assembly code directly uses the memory locations of the data members and directly alters them by using the mov command. To explore how data members are accessed within a member function, I created a public member function called setAge(). In the main method, I called the private method on a Person object and set the age to 17. These operations were represented as the following in assembly:

```
mov    esi, 17
call   _ZN6Person6setAgeEi
```

The caller function seems to have stored the value in the esi register before passing it off to the setAge method. The setAge callee method is made of up these commands:

```
mov    qword ptr [rsp - 8], rdi
mov    dword ptr [rsp - 12], esi
mov    rdi, qword ptr [rsp - 8]
mov    esi, dword ptr [rsp - 12]
mov    dword ptr [rdi + 16], esi
ret
```

Again, the memory address of the data member is used to directly change the value at that memory address. The only difference is that using the member method takes one extra step: it puts the value in a register, then moves that value to the memory location of the data member.