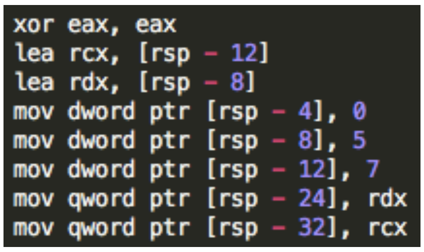
Jiwon Cha
CS 2150
11/02/2017

Post lab report

Parameter passing

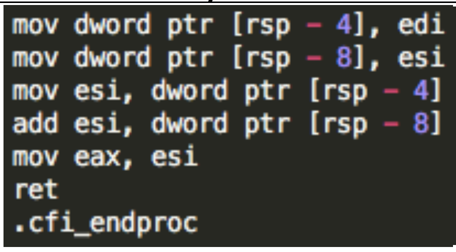
To test how assembly treats different data types including int, float, char, pointer and objects, I wrote a .cpp file with functions that take in different data types and tested them inside the main function. The first thing that I noticed is that inside the main function, the assembly initializes 'normal' data types (int, char, float) with mov and pointers (references) with lea. Inside the stack, the assembly first moves the memory addresses at rsp-12 and rsp-8 to rcx and rdx. Another point that I noticed is that the assembly assigns rdx as the pointer that was declared first and rcx as the second pointer. Then, rsp grows downwards to include all of the local variables; 0, 5, 7 and two pointers.



```
xor eax, eax
lea rcx, [rsp - 12]
lea rdx, [rsp - 8]
mov dword ptr [rsp - 4], 0
mov dword ptr [rsp - 8], 5
mov dword ptr [rsp - 12], 7
mov qword ptr [rsp - 24], rdx
mov qword ptr [rsp - 32], rcx
```

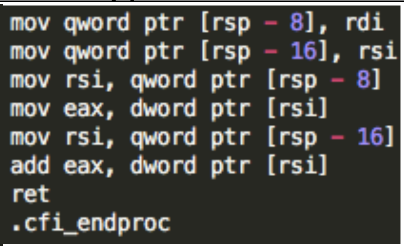
For my test cases, I wrote different functions to add the values of two parameters. First of all, the addition of two integers passed by value use eax, edi and esi as the registers. They are then pushed onto the stack and each spot is 4 bytes.
When the parameters are passed by reference, eax is used as the return value but rdi and rsi (64 bytes) are used to get the values of parameters, and the stack grows down by 8 bytes. Moreover, both values (stored in rsp-8 and rsp-16 respectively) are first stored in rsi and then added to the final eax value. The functions for pass by reference and pass by pointers were exactly the same, which makes sense because the assembly recognizes both reference and pointers as memory addresses. The functions written with char and float displayed similar behaviors as well.

Pass by value



```
mov dword ptr [rsp - 4], edi
mov dword ptr [rsp - 8], esi
mov esi, dword ptr [rsp - 4]
add esi, dword ptr [rsp - 8]
mov eax, esi
ret
.cfi_endproc
```

Pass by pointer & reference

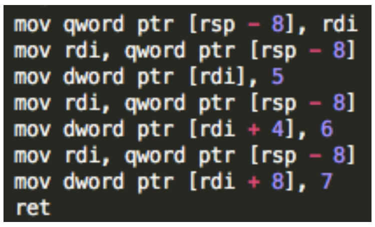


```
mov qword ptr [rsp - 8], rdi
mov qword ptr [rsp - 16], rsi
mov rsi, qword ptr [rsp - 8]
mov eax, dword ptr [rsi]
mov rsi, qword ptr [rsp - 16]
add eax, dword ptr [rsi]
ret
.cfi_endproc
```

When float values are passed by values, the registers used as parameters were xmm0 and xmm1, which are probably the registers used for floats and doubles (128 bits). Other than the registers used, the syntax is very similar to that of int function (passed by value) except xmm0 also acts as rax, and the commands used for floats are slightly different (movss, adds).

The function using char uses al, bl and cl as the registers (4 bits) because char is a small data type. The stack grows down by 1 byte. The syntax is also very similar to that of int function, except movsx is used to move the value stored in cl to eax (return value).

Finally, I wrote a C++ code to take in an array and assign 5,6, and 7 as its first three values. The assembly code for this is pretty simple; it first pushes the array that was passed in onto the stack at rsp-8. The first value (5) is stored at rdi, and two consecutive integers (6 and 7) are stored at rdi + 4 and rdi + 8 respectively.



```
mov qword ptr [rsp - 8], rdi
mov rdi, qword ptr [rsp - 8]
mov dword ptr [rdi], 5
mov rdi, qword ptr [rsp - 8]
mov dword ptr [rdi + 4], 6
mov rdi, qword ptr [rsp - 8]
mov dword ptr [rdi + 8], 7
ret
```

Objects

Question 1

Data layout: how are they kept in memory?

The local variables are pushed onto the stack. As local variables are declared inside of the main function, and the stack pointer grows down by the size of the data type. For example, if the local variable is an integer type, the stack grows down by 4 bytes and the integer will be stored at [rsp-4]. The parameters are often stored in the registers (rsi, rdi…) and on the stack if there are values that may not be modified by the subroutine callee or if there are more than 6 parameters. When a variable is declared using new, which means that the memory is dynamically allocated, it is stored on the heap instead of the stack.

How does C++ keep different fields of an object "together"?

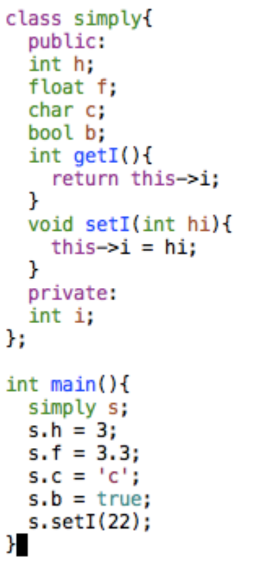
In C++, we can declare public and private members (fields) inside a class or a struct. For example, when we want to access those fields inside of the main method, we must access them using a dot (.) for normal data structures and an arrow (->) for pointers. In this way, every field is always associated with the class. (class.member or class->member)

How does assembly know which data member to access? How does the assembly know which object it is being called out of?

Upon declaring variables and initializing them to a value, each variable is assigned to a spot on the stack. (memory) Therefore, when there are operations involving that particular variable - for example, when a class member is initialized inside the main function – the assembly simply has to look for that particular spot on the stack and perform the operations. Moreover, assembly and C++ use the notion of a 'hidden pointer'. When we call member functions or members of an object, a pointer to the object this is implicitly passed. It also is passed as a first 'hidden' parameter.[1] Each call has its own version of these hidden parameters.

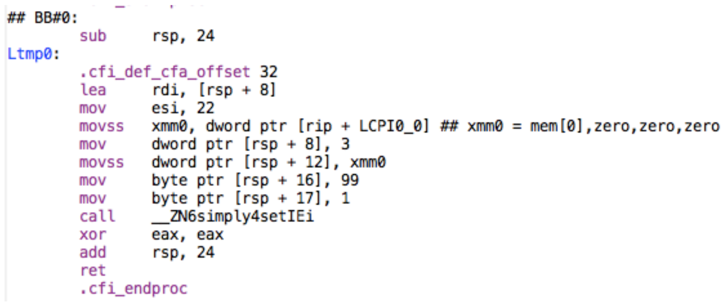
Question 2

Where is the data laid out in memory?



```
class simply{
  public:
  int h;
  float f;
  char c;
  bool b;
  int getI(){
    return this->i;
  }
  void setI(int hi){
    this->i = hi;
  }
  private:
  int i;
};

int main(){
  simply s;
  s.h = 3;
  s.f = 3.3;
  s.c = 'c';
  s.b = true;
  s.setI(22);
}
```



```
## BB#0:
        sub     rsp, 24
Ltmp0:
        .cfi_def_cfa_offset 32
        lea     rdi, [rsp + 8]
        mov     esi, 22
        movss   xmm0, dword ptr [rip + LCPI0_0] ## xmm0 = mem[0],zero,zero,zero
        mov     dword ptr [rsp + 8], 3
        movss   dword ptr [rsp + 12], xmm0
        mov     byte ptr [rsp + 16], 99
        mov     byte ptr [rsp + 17], 1
        call    __ZN6simply4setIEi
        xor     eax, eax
        add     rsp, 24
        ret
        .cfi_endproc
```

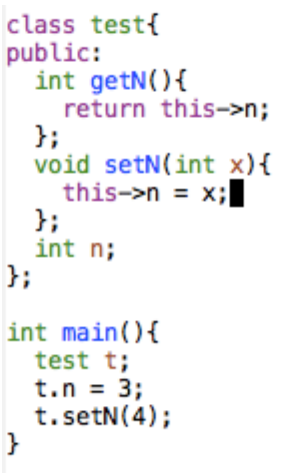
To observe how assembly processes a class with multiple variables, I've generated an assembly code from a sample C++ program. From this, we can observe that the assembly first allocates 24 bytes of memory (subtract 24 from the stack pointer) First of all, the assembler sets the private member i to 22 via the setI function. Notice that 22 is moved into esi. There seems to be no apparent difference between private and public members. Then, the assembler promptly allocates memory from the bottom of the frame. (Note that we are starting from [rsp-24] and going upwards) s.h, which is an int type, is placed at [rsp+8] and take up 4 bytes until [rsp+12]. Float takes another 4 bytes of the stack from [rsp+12] to [rsp+16]. Char and Bool are both one byte, therefore they each occupy 1 byte of the stack (the char value occupies from 16 to 17) Then, I created another class, within the same file, called example which contains one public member, int hi. Then, I made example a public field of

[1] https://stackoverflow.com/questions/8202203/how-does-member-function-knows-that-it-is-called-for-specific-object

class simply. When I instantiated s.exp.hi as 34, the assembler simply put 34 to the spot [rsp] after zeroing out eax (before adding 24 to rsp). Therefore, we know that the assembly does not distinguish between different classes, but it simply pushes the variables onto the stack.

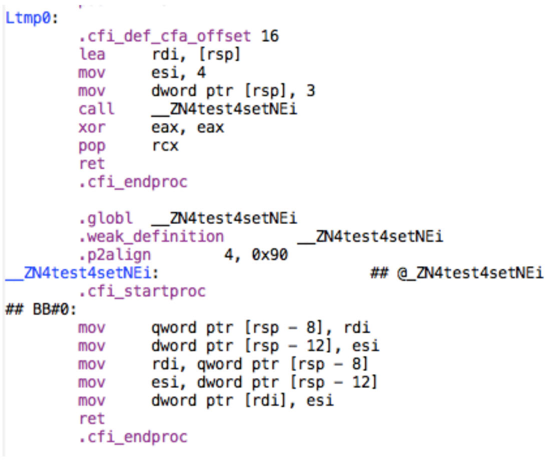
Question 3

To test how data members are accessed from inside and outside of the function, I wrote a simple test file.



```
class test{
public:
  int getN(){
    return this->n;
  };
  void setN(int x){
    this->n = x;
  };
  int n;
};

int main(){
  test t;
  t.n = 3;
  t.setN(4);
}
```

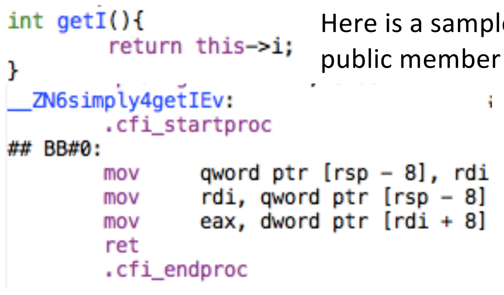


```
Ltmp0:
        .cfi_def_cfa_offset 16
        lea     rdi, [rsp]
        mov     esi, 4
        mov     dword ptr [rsp], 3
        call    __ZN4test4setNEi
        xor     eax, eax
        pop     rcx
        ret
        .cfi_endproc

        .globl  __ZN4test4setNEi
        .weak_definition        __ZN4test4setNEi
        .p2align        4, 0x90
__ZN4test4setNEi:                       ## @_ZN4test4setNEi
        .cfi_startproc
## BB#0:
        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], esi
        ret
        .cfi_endproc
```

To facilitate the comparison, I made int n a public member of class test. Then, n will be instantiated both directly and through a setter to see if there is a difference. From the code above, we see that when n is set to 3, 3 is directly moved to the spot [rsp] on the stack. However, when n is set to 4 via the setter, the function first gets the input from rdi (parameter) and then accesses the spots [rsp-8] and [rsp-12] respectively.

Question 4



```
int getI(){
        return this->i;
}
__ZN6simply4getIEv:
        .cfi_startproc
## BB#0:
        mov     qword ptr [rsp - 8], rdi
        mov     rdi, qword ptr [rsp - 8]
        mov     eax, dword ptr [rdi + 8]
        ret
        .cfi_endproc
```

Here is a sample implementation of accessing a private member i through a public member function getter, which uses this pointer to return the value of i. This method is translated into assembly in the following way. First of all, the value of i is stored in rdi, then that value is pushed onto the stack at the spot [rsp-8]. The getter method simply moves the pointer to the address [rdi+8] to the return value. I believe that this pointer is not explicitly stored on the stack, but rather implied by the assembler. However, I believe that this pointer is stored differently in each language and system. For example, in Visual studio, this pointer is stored in ecx when a method is called using an instance.[2] Since this pointer is rather an expression than a variable, I believe that this pointer cannot be modified nor updated.

Reference

https://stackoverflow.com/questions/16585562/where-is-the-this-pointer-stored-in-computer-memory
http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
https://stackoverflow.com/questions/33556511/how-do-objects-work-in-x86-at-the-assembly-level
https://understandingartofprogramming.wordpress.com/tag/this-pointer-in-assembly/

[2] https://understandingartofprogramming.wordpress.com/tag/this-pointer-in-assembly/