Justin Choi (jc8mc)
CS 2150 -106
March 30th 2017

postlab8.pdf

Part 1: Parameter Passing

In this lab, I chose the parameter passing option. I examined the parameter passing question along with some x86 64-bit assembly code and C++ code to answer the guiding questions on Professor Bloomfield's CS 2150 website. I compiled it through the clang++ -m64 –mllvm –x86-asm-syntax=intel -S –fomit-frame-pointer param.cpp param.o to generate the assembly code in intel format as we have covered in class.

I. Pass by value vs. Pass by reference

```
// param.cpp
#include <iostream>
using namespace std;

void swapVal(int x, int y) {
   int temp = x;
   x = y;
   y = temp;
}
void swapRef(int &num1, int &num2) {
   int temp = num1;
   num1 = num2;
   num2 = temp;
}
int main() {
   int a = 6;
   int b = 7;
   swapVal(a, b);
   cout << a << " " << b << endl;
   swapRef (a, b);
   cout << a << " " << b << endl;
   return 0;
}
```

a) int

- Pass by value:

```
_Z7swapValii:                          # @_Z7swapValii
        .cfi_startproc
# BB#0:
        mov  dword ptr [rsp - 4], edi
        mov  dword ptr [rsp - 8], esi
        mov  esi, dword ptr [rsp - 4]
        mov  dword ptr [rsp - 12], esi
        mov  esi, dword ptr [rsp - 8]
        mov  dword ptr [rsp - 4], esi
        mov  esi, dword ptr [rsp - 12]
        mov  dword ptr [rsp - 8], esi
        ret
```

Pass by value caller:

```
mov  dword ptr [rsp + 36], 0
        mov  dword ptr [rsp + 32], 6
        mov  dword ptr [rsp + 28], 7
        mov  edi, dword ptr [rsp + 32]
        mov  esi, dword ptr [rsp + 28]
       call _Z7swapValii
```

- Pass by reference:

```
_Z7swapRefRiS_:                     # @_Z7swapRefRiS_
       .cfi_startproc
# BB#0:
       mov  qword ptr [rsp - 8], rdi
       mov  qword ptr [rsp - 16], rsi
       mov  rsi, qword ptr [rsp - 8]
       mov  eax, dword ptr [rsi]
       mov  dword ptr [rsp - 20], eax
       mov  rsi, qword ptr [rsp - 16]
       mov  eax, dword ptr [rsi]
       mov  rsi, qword ptr [rsp - 8]
       mov  dword ptr [rsi], eax
       mov  eax, dword ptr [rsp - 20]
       mov  rsi, qword ptr [rsp - 16]
       mov  dword ptr [rsi], eax
       ret
```

Pass by Reference caller:

```
       lea   rdi, [rsp + 32]
       lea   rsi, [rsp + 28]
       mov  qword ptr [rsp + 16], rax # 8-byte Spill
       call   _Z7swapRefRiS_
```

For integers, for pass by value, the 'ii' in the name represented two integer parameters The difference came into place when the pass by value used a 32-bit register names but the pass by reference used a 64-bit register names, excluding eax, which is the same as rax only the bit differences, which made the values to be subtracted from memory to be different when it was dereferenced in ordered to be copied. I think the most significant difference boils down to the caller. For pass by value, the parameter values were not stored in a register (saved on the stack) while for pass by reference it was saved in the register.

** For char and float, I simply changed the data types for the variables from the int C++ file

b) char
   - pass by value:

Pass by value:

```
       mov  al, sil
       mov  cl, dil
       mov  byte ptr [rsp - 1], cl
       mov  byte ptr [rsp - 2], al
       mov  al, byte ptr [rsp - 1]
       mov  byte ptr [rsp - 3], al
       mov  al, byte ptr [rsp - 2]
       mov  byte ptr [rsp - 1], al
       mov  al, byte ptr [rsp - 3]
       mov  byte ptr [rsp - 2], al
       ret
```

Pass by value caller:

```
       mov  dword ptr [rsp + 20], 0
       mov  byte ptr [rsp + 19], 97
       mov  byte ptr [rsp + 18], 98
       mov  al, byte ptr [rsp + 19]
       movsx       edi, al
       movsx       esi, byte ptr [rsp + 18]
       call   _Z7swapValcc
```

- pass by reference

```
    mov  qword ptr [rsp - 8], rdi
    mov  qword ptr [rsp - 16], rsi
    mov  rsi, qword ptr [rsp - 8]
    mov  al, byte ptr [rsi]
    mov  byte ptr [rsp - 17], al
    mov  rsi, qword ptr [rsp - 16]
    mov  al, byte ptr [rsi]
    mov  rsi, qword ptr [rsp - 8]
    mov  byte ptr [rsi], al
    mov  al, byte ptr [rsp - 17]
    mov  rsi, qword ptr [rsp - 16]
    mov  byte ptr [rsi], al
    ret
```

Pass by reference caller:

```
    lea    rdi, [rsp + 19]
    lea    rsi, [rsp + 18]
    mov  qword ptr [rsp + 8], rax # 8-byte Spill
    call   _Z7swapRefRcS_
```

Similar to integers, pass by reference stores values in registers while pass by value does not. There is also a difference in terms of the data types in that due to the different types, the allocation of memory is different, which is highlighted by the use of "byte" for chars to differentiate between integers and char byte size differences. Integers use 4 bytes while chars only use 1 byte, 8 bits of memory in a register, so we also see a difference in how much it subtracts when it is dereferenced.

c) float
   - pass by value:

```
_Z7swapValff:                        # @_Z7swapValff
        .cfi_startproc
# BB#0:
        movss       dword ptr [rsp - 4], xmm0
        movss       dword ptr [rsp - 8], xmm1
        movss       xmm0, dword ptr [rsp - 4] # xmm0 = mem[0],zero,zero,zero
        movss       dword ptr [rsp - 12], xmm0
        movss       xmm0, dword ptr [rsp - 8] # xmm0 = mem[0],zero,zero,zero
        movss       dword ptr [rsp - 4], xmm0
        movss       xmm0, dword ptr [rsp - 12] # xmm0 = mem[0],zero,zero,zero
        movss       dword ptr [rsp - 8], xmm0
        ret
```

Pass by value caller:

```
        movss       xmm0, dword ptr [.LCPI3_0] # xmm0 = mem[0],zero,zero,zero
        movss       xmm1, dword ptr [.LCPI3_1] # xmm1 = mem[0],zero,zero,zero
        mov  dword ptr [rsp + 36], 0
        movss       dword ptr [rsp + 32], xmm1
        movss       dword ptr [rsp + 28], xmm0
        movss       xmm0, dword ptr [rsp + 32] # xmm0 = mem[0],zero,zero,zero
        movss       xmm1, dword ptr [rsp + 28] # xmm1 = mem[0],zero,zero,zero
        call    _Z7swapValff
```

- pass by reference

```
_Z7swapRefRfS_:                    # @_Z7swapRefRfS_
        .cfi_startproc
# BB#0:
        mov  qword ptr [rsp - 8], rdi
        mov  qword ptr [rsp - 16], rsi
        mov  rsi, qword ptr [rsp - 8]
        movss       xmm0, dword ptr [rsi]   # xmm0 = mem[0],zero,zero,zero
        movss       dword ptr [rsp - 20], xmm0
        mov  rsi, qword ptr [rsp - 16]
        movss       xmm0, dword ptr [rsi]   # xmm0 = mem[0],zero,zero,zero
        mov  rsi, qword ptr [rsp - 8]
        movss       dword ptr [rsi], xmm0
        movss       xmm0, dword ptr [rsp - 20] # xmm0 = mem[0],zero,zero,zero
        mov  rsi, qword ptr [rsp - 16]
        movss       dword ptr [rsi], xmm0
        ret
```

Pass by reference caller:

```
        lea    rdi, [rsp + 32]
        lea    rsi, [rsp + 28]
        mov    qword ptr [rsp + 16], rax # 8-byte Spill
        call   _Z7swapRefRfS_
```

For floats, when passed by reference, rather than having the ii for parameters, which stated as two integers, there was ff, were for two floating number parameters. Similar to integers, pass by reference stores values in registers while pass by value does not. There was also difference in byte sizes compared to integers and floating numbers.

Overall, for int, char, and float, pass by value and pass by reference is different in the way that pass by reference copies (mov) the address of the variable rather than copying the value. For similarities for int, char, and float, Passing by value takes the value in the register and pushes it onto the stack. Pass by reference moves the address of the variable.

d) pointer
   - pass by value:

```
_Z7swapValPiS_:                    # @_Z7swapValPiS_
        .cfi_startproc
# BB#0:
        mov  qword ptr [rsp - 8], rdi
        mov  qword ptr [rsp - 16], rsi
        mov  rsi, qword ptr [rsp - 8]
        mov  eax, dword ptr [rsi]
        mov  dword ptr [rsp - 20], eax
        mov  rsi, qword ptr [rsp - 16]
        mov  eax, dword ptr [rsi]
        mov  rsi, qword ptr [rsp - 8]
        mov  dword ptr [rsi], eax
        mov  eax, dword ptr [rsp - 20]
        mov  rsi, qword ptr [rsp - 16]
        mov  dword ptr [rsi], eax
        ret
```

Pass by value caller:

```
        lea    rdi, [rsp + 28]
        lea    rsi, [rsp + 32]
        mov    dword ptr [rsp + 36], 0
        mov    dword ptr [rsp + 32], 6
        mov    dword ptr [rsp + 28], 7
```

- pass by reference

```
_Z7swapRefRiS_:                       # @_Z7swapRefRiS_
        .cfi_startproc
# BB#0:
        mov   qword ptr [rsp - 8], rdi
        mov   qword ptr [rsp - 16], rsi
        mov   rsi, qword ptr [rsp - 8]
        mov   eax, dword ptr [rsi]
        mov   dword ptr [rsp - 20], eax
        mov   rsi, qword ptr [rsp - 16]
        mov   eax, dword ptr [rsi]
        mov   rsi, qword ptr [rsp - 8]
        mov   dword ptr [rsi], eax
        mov   eax, dword ptr [rsp - 20]
        mov   rsi, qword ptr [rsp - 16]
        mov   dword ptr [rsi], eax
        ret
```

Pass by reference caller / callee:

```
        lea    rdi, [rsp + 32]
        lea    rsi, [rsp + 28]
        mov    qword ptr [rsp + 16], rax # 8-byte Spill
        call   _Z7swapRefRiS_

        xor    ecx, ecx
        mov    qword ptr [rsp + 8], rax # 8-byte Spill
        mov    eax, ecx
        add    rsp, 40
        ret
```

When comparing pass by int pointer by value and pass by reference, it has a very similar code to when it is passed by reference for integers. A slight difference is the offsets for passing by int pointer and passing by reference.

e) object
- pass by value: takes the value in the register and pushes it onto the stack
- pass by reference: moves the address of the variable

II. Arrays

For arrays, as they are a list of memories, they are passed by pushing the address of the first element. Then to access the next, in the callee, it adds 4 bytes to the base address pointer to find and access each and next element to dereference and perform various operations and pushes on to the stack.

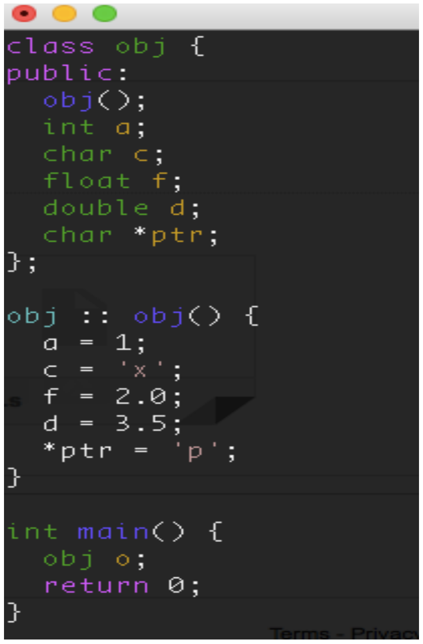
III. Pass by reference vs. pass values by pointer

Pass by pointer and pass by reference differs in the high-level. When passed by reference, the formal parameter is able to change the value of the actual argument. Thus, the subroutine can modify the actual parameter. In assembly, the caller function stores the memory address of the variable in a register. The callee function is able to access the stored value by the square brackets ([]), which is similar to dereferencing a pointer. For pass by pointer, the assembly code for it was very similar to pass by reference. I was surprised at first. However, as pass by pointer can modify the pointee in the subroutine, I thought it could be similar. There wasn't a difference in the functions itself. However, there was a difference in the offset.

Justin Choi (jc8mc)
CS 2150 -106
March 30th 2017

Part 2: Objects

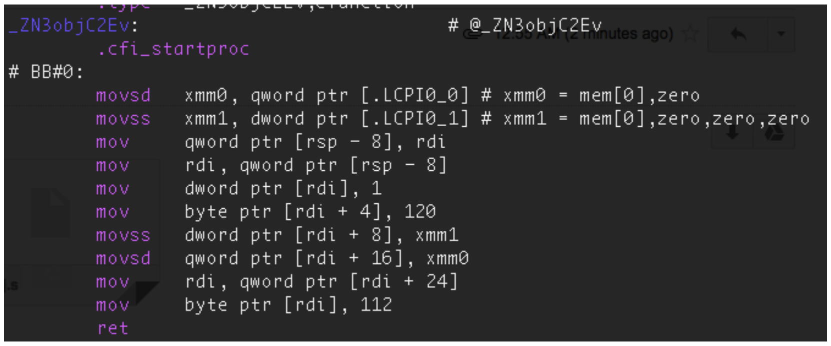
For the object part of the lab, in order to see how objects are translated and implemented in assembly code, I created my own user-defined object, called obj. In the object, it included five data members of different types, including ints, chars, floats, doubles, and a char pointer.



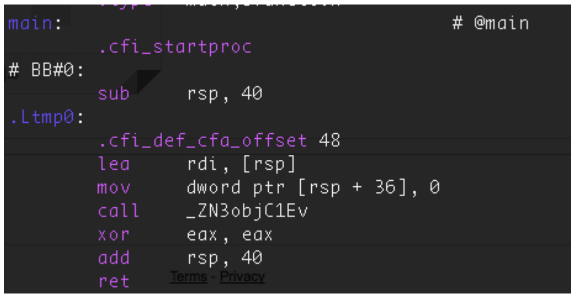
```
class obj {
public:
   obj();
   int a;
   char c;
   float f;
   double d;
   char *ptr;
};

obj :: obj() {
   a = 1;
   c = 'x';
   f = 2.0;
   d = 3.5;
   *ptr = 'p';
}

int main() {
   obj o;
   return 0;
}
```



```
_ZN3objC2Ev:                            # @_ZN3objC2Ev
        .cfi_startproc
# BB#0:
        movsd   xmm0, qword ptr [.LCPI0_0] # xmm0 = mem[0],zero
        movss   xmm1, dword ptr [.LCPI0_1] # xmm1 = mem[0],zero,zero,zero
        mov     qword ptr [rsp - 8], rdi
        mov     rdi, qword ptr [rsp - 8]
        mov     dword ptr [rdi], 1
        mov     byte ptr [rdi + 4], 120
        movss   dword ptr [rdi + 8], xmm1
        movsd   qword ptr [rdi + 16], xmm0
        mov     rdi, qword ptr [rdi + 24]
        mov     byte ptr [rdi], 112
        ret
```



```
main:                                   # @main
        .cfi_startproc
# BB#0:
        sub     rsp, 40
.Ltmp0:
        .cfi_def_cfa_offset 48
        lea     rdi, [rsp]
        mov     dword ptr [rsp + 36], 0
        call    _ZN3objC1Ev
        xor     eax, eax
        add     rsp, 40
        ret
```

I.

When my user-defined object.cpp code was translated into assembly code, it consisted of data type fields in the constructor. The fields in the constructor were pushed onto the stack in reverse order due to the property of stack growing down in assembly. For accessing data member access, the callee must find the data members in which the pointer to the first field on the stack and then it can be accessed in reverse order. And, all of the data fields can be accessed by their respective offsets with the square brackets for dereferencing. For data access, the fields are pushed in reverse order of variable initialization, depending on the offset of the data types. For methods in classes, assembly finds the first field and stores it into another register to use it later when it is called. Then the callee uses it to dereference based on the offsets to find it.

Justin Choi (jc8mc)
CS 2150 -106
March 30th 2017

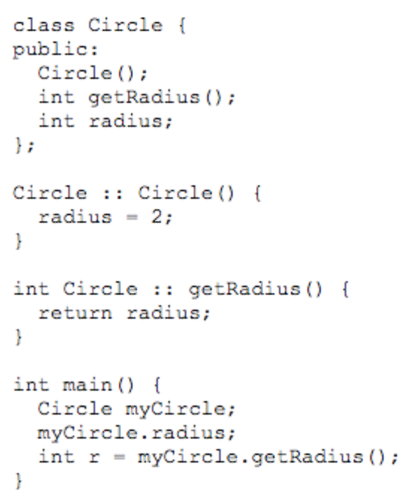
II.

For this part, by using the code that I developed above, I created a class that contains 5 different data types in the constructor: int, char, float, double and a char pointer. For these data types, as stack grow downward in assembly language, once they are pushed, they must be popped in reverse order. I also noticed a size difference in memory allocation for these different types. According to the x86-64 Machine-Level Programming from Carnegie Melon, ints have 4 bytes, char have 1 byte, float have 4 byte, double have 8 byte, and an char pointer have 8 bytes. (https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf).

When I changed fields to private, and had getters, it changed only a bit. However, I noticed that for private fields, I had to use the getX method due to its private visibility and had added additional calls. I wondered how a friend keyword would be implemented in assembly since it can bypass the requirements of visibility for accessing it.

III.

I created a circle class for this part to compare the differences of inside and outside member functions. myCircle.radius is the inside member function while myCircle.getRadius is the outside member function.

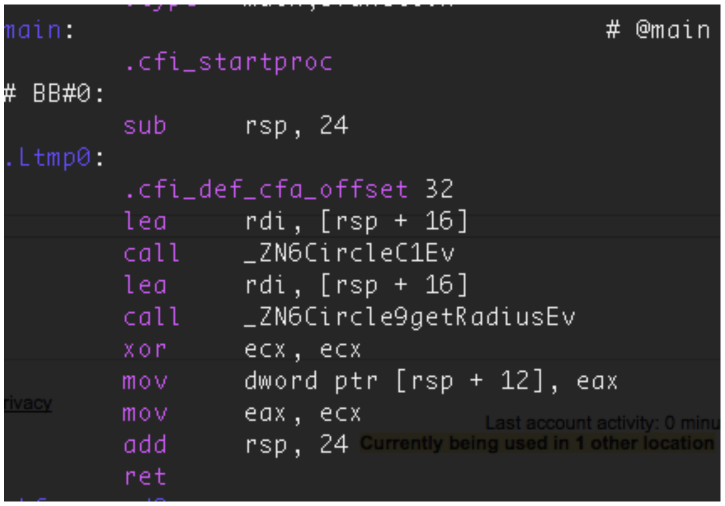


```
class Circle {
public:
  Circle();
  int getRadius();
  int radius;
};

Circle :: Circle() {
  radius = 2;
}

int Circle :: getRadius() {
  return radius;
}

int main() {
  Circle myCircle;
  myCircle.radius;
  int r = myCircle.getRadius();
}
```



```
main:                                   # @main
        .cfi_startproc
# BB#0:
        sub     rsp, 24
.Ltmp0:
        .cfi_def_cfa_offset 32
        lea     rdi, [rsp + 16]
        call    _ZN6CircleC1Ev
        lea     rdi, [rsp + 16]
        call    _ZN6Circle9getRadiusEv
        xor     ecx, ecx
        mov     dword ptr [rsp + 12], eax
        mov     eax, ecx
        add     rsp, 24
        ret
```

As fields of objects are pushed in reverse order, accessing these would make the callee to point to the right memory dependent on the offset. When inside a function, it had the lea keyword, which is for load effective address. The "lea instruction places the address specified by its second operand into the register specified by the first operand. The content of the memory location are not loaded, but the effective address is computed and placed in the register" (http://www.cs.virginia.edu/~evans/cs216/guides/x86.html). However, for outside a function, it did not have the lea keyword since it just directly stores in the return register, eax to be ready to be popped.

Justin Choi (jc8mc)
CS 2150 -106
March 30th 2017

IV. D

In order to evaluate what the “this” pointer does, I changed the constructor so it had a “this” pointer: this -> radius = 2; since only member functions have a “this” pointer. The “this” pointer is an implicit parameter to all member functions, and they are used to refer invoking objects. (https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm). As I retrieved the assembly code, there was no difference between the assembly codes from the part above and when implemented with “this”. So, I am guessing that it is not stored nor accessed nor passed to a member function nor updated since it is supposed to only invoke objects but not actually a real pointer that points to an address in memory.