Justin Choi (jc8mc)
CS 2150 -106
Lab 8: Assembly, Part 1

inlab8.pdf

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.

The C++ code that I used to compare parameter passing is the following for integers:

```
// 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;
}
```

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.

For int:
Abbreviated Assembly code for swapVal method, which is 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
```

Justin Choi (jc8mc)
CS 2150 -106
Lab 8: Assembly, Part 1

Abbreviated Assembly code for swapRef method, which is 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
```

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 while for pass by reference it was saved in the register.

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 caller:

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

For the rest, chars, pointers, floats, and objects, I changed the above C++ code and matched with the different types in order to generate assembly code for different data types. The assembly code for the different types followed a similar manner for the functions.

Justin Choi (jc8mc)
CS 2150 -106

For char,
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. The difference between integers is the byte size differences.

For floats,
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, which was for two floating number parameters. Also, there was a new xmm0, which I think is a register. (I am not sure what is it?) 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 pass by reference passing ints, chars, and floats was similar. The actual value is passed in pass by value while memory location is passed in pass by reference. In pass by reference, the caller saved the stack pointer first, before pushing the parameters on the stack as well as when the callee is called. The callee, then can access it by popping the stack in reverse order.

For pointers,

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
```

```
        call    _Z7swapValPiS_
```

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
```

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 pas 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 calling conventions of pass by reference using the rax register while pass by pointer did not.

For objects and arrays, it followed a very similar pattern from the rest. For arrays, I noticed that there was a unique trait to it since it needs to move from one memory location to another in order to move within the array to get to the next element. Also when I looked up online in stackoverflow, it said arrays are always passed by reference. So, I am assuming that

the values are moved onto the stack in the caller function. In the callee, it can then be accessed through dereferencing for both call by reference and call by value.
For objects, for pass by value, it appeared very similar to passing integers.