Noah French (njf5cu)
Lab 8
10/31/17
Filename: inlab8.pdf
File Description: Lab 8 In-lab Parameter Passing Discussion

All the x86 code was generated by compiling with clang++ using the -m64 -mllvm –x86-asm-syntax=intel -S -fomit-frame-pointer flags.

First I created a C++ function called addOne that takes in an integer value and returns that value + 1. Then I created a function called addThree that takes in an integer value and calls addOne on the value and returns that output + 2. Both functions are pass by value. The addThree method's x86 code was generated as the following:

```
_Z8addThreei:                   # @_Z8addThreei
        .cfi_startproc
# BB#0:
        push    rax
.Ltmp2:
        .cfi_def_cfa_offset 16
        mov     dword ptr [rsp + 4], edi
        mov     edi, dword ptr [rsp + 4]
        call    _Z6addOnei
        add     eax, 2
        pop     rcx
        ret
```

The addOne method's x86 code is:

```
_Z6addOnei:                     # @_Z6addOnei
        .cfi_startproc
# BB#0:
        mov     dword ptr [rsp - 4], edi
        mov     edi, dword ptr [rsp - 4]
        add     edi, 1
        mov     eax, edi
        ret
```

In these functions, addThree is the caller and addOne is the callee. The parameter is passed into the caller in the edi register. When it calls the callee, the parameter is still passed in the edi register. The callee, addOne manipulates the parameter in edi, then moves it to the return register, eax, where the caller adds to the register and returns. The x86 code makes use of 32 bit registers for parameter passing because on my laptop, ints are 32 bits.

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 the addOne function. The function pass-by-reference function is void but should theoretically change the value of the parameter integer x. The parameter being used is still a 32 bit, albeit a reference to one, so the x86 code should still make use of 32 bit registers. Thus, the only real difference in parameter handling from the first example should be because this second example uses pass by reference. The x86 code for the caller function, valueFunc was:

```
_Z10valueFunc1Ri:                   # @_Z10valueFunc1Ri
        .cfi_startproc
# BB#0:
        push    rax
.Ltmp3:
        .cfi_def_cfa_offset 16
        mov     qword ptr [rsp], rdi
        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
```

In this example, the parameter is initially passed in through the rdi register rather than the edi register. This is probably because, though an int is 32 bits on this computer, a reference to an int is 64 bits on this computer. However, the function is passing 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:

```
_Z6addOnei:                         # @_Z6addOnei
        .cfi_startproc
# BB#0:
        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. It takes in its parameter in the edi register and puts the return value in the eax register.

In assembly, passing values by reference works the exact same way as passing values by pointer. There are obviously differences between pointers and references in C++, but in x86, taking in a pointer to a datatype is the same as taking in a reference to a datatype. The same procedures are done to ensure that the object being pointed at or referenced to undergoes the changes described in the routine.

When functions that have 16-bit datatype parameters, like a char (a lot of the time), the x86 code would presumably indicate that the 16-bit registers are being used.

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.

```
_Z9arrayFuncPi:                         # @_Z9arrayFuncPi
        .cfi_startproc
# BB#0:
        mov     qword ptr [rsp - 8], rdi
        mov     dword ptr [rsp - 12], 0
.LBB1_1:                                # =>This Inner Loop Header: Depth=1
        cmp     dword ptr [rsp - 12], 5
        jge     .LBB1_4
# BB#2:                                 #   in Loop: Header=BB1_1 Depth=1
        movsxd          rax, dword ptr [rsp - 12]
        mov     rcx, qword ptr [rsp - 8]
        mov     dword ptr [rcx + 4*rax], 2
# BB#3:                                 #   in Loop: Header=BB1_1 Depth=1
        mov     eax, dword ptr [rsp - 12]
        add     eax, 1
        mov     dword ptr [rsp - 12], eax
        jmp     .LBB1_1
.LBB1_4:
        ret
```

Subsequent elements in the array are accessed by incrementing in the stack, because the elements of the parameter array are stored in adjacent data blocks in the stack.