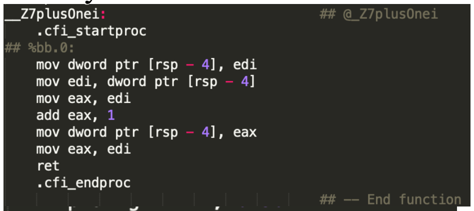
Yubin Lim yl2vv
March 28, 2019
Postlab8.pdf
CS2150-103

Parameter Passing

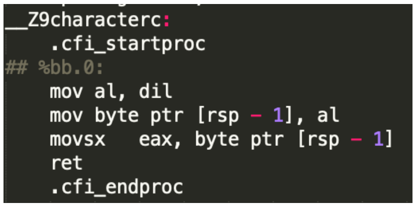
In order to examine my disassembled code of C++ code, I chose the option to have the C++ code output to an assembly file using `clang++ -S -mllvm --x86-asm-syntax=intel -fomit-frame-pointer fileName.cpp` , which created a .s file for me to view in a text editor.

1. When variables are passed by value, there is consistency on how they are passed. They are copied into the register at [rsp - _], and the number to subtract in [rsp - _], the pointer to destination, depends on the byte of the variable. The register depends on the memory it needs to allocate. When dealing with integers, which are 32 bits (4 bytes), instead of the 64 bit registers, the 32 bit registers are used such as eax. The 8 bit registers are used for characters, such as dil, and so on. When passed by reference, the local parameter is found always at [rsp – 8] and uses mov before the specific movss, movsd, etc. This is because the reference has to be found first.

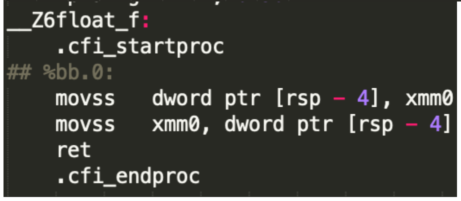
Pass by Value



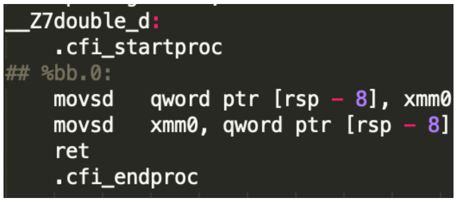
```
__Z7plusOnei:                           ## @_Z7plusOnei
    .cfi_startproc
## %bb.0:
    mov dword ptr [rsp - 4], edi
    mov edi, dword ptr [rsp - 4]
    mov eax, edi
    add eax, 1
    mov dword ptr [rsp - 4], eax
    mov eax, edi
    ret
    .cfi_endproc
                                        ## -- End function
```



```
__Z9characterc:
    .cfi_startproc
## %bb.0:
    mov al, dil
    mov byte ptr [rsp - 1], al
    movsx   eax, byte ptr [rsp - 1]
    ret
    .cfi_endproc
```

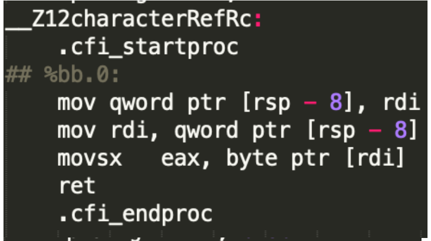


```
__Z6float_f:
    .cfi_startproc
## %bb.0:
    movss   dword ptr [rsp - 4], xmm0
    movss   xmm0, dword ptr [rsp - 4]
    ret
    .cfi_endproc
```

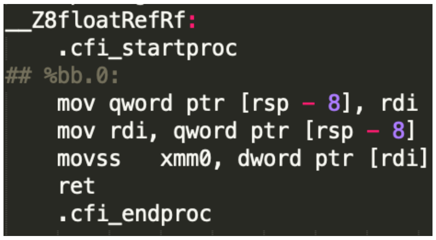


```
__Z7double_d:
    .cfi_startproc
## %bb.0:
    movsd   qword ptr [rsp - 8], xmm0
    movsd   xmm0, qword ptr [rsp - 8]
    ret
    .cfi_endproc
```

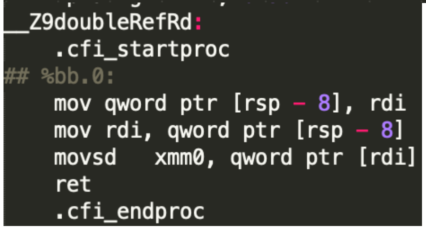
Pass by Reference



```
__Z12characterRefRc:
    .cfi_startproc
## %bb.0:
    mov qword ptr [rsp - 8], rdi
    mov rdi, qword ptr [rsp - 8]
    movsx   eax, byte ptr [rdi]
    ret
    .cfi_endproc
```



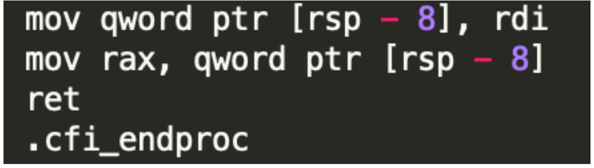
```
__Z8floatRefRf:
    .cfi_startproc
## %bb.0:
    mov qword ptr [rsp - 8], rdi
    mov rdi, qword ptr [rsp - 8]
    movss   xmm0, dword ptr [rdi]
    ret
    .cfi_endproc
```



```
__Z9doubleRefRd:
    .cfi_startproc
## %bb.0:
    mov qword ptr [rsp - 8], rdi
    mov rdi, qword ptr [rsp - 8]
    movsd   xmm0, qword ptr [rdi]
    ret
    .cfi_endproc
```

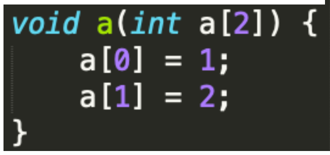
2. Assembly is not object oriented, meaning the way we manipulate data is different from languages like Java. When you pass a variable of a type object, it takes up 4 bytes and when you pass by reference, they take up 8 bytes. When passed by value, the parameter register contains

the actual value of the object. When passed by reference, the parameter register contains the memory of the object, like a pointer and works with the object in the address.

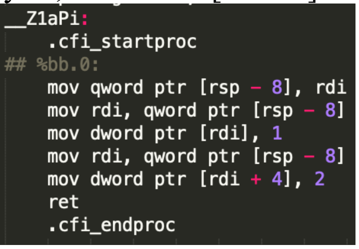


```
mov qword ptr [rsp - 8], rdi
mov rax, qword ptr [rsp - 8]
ret
.cfi_endproc
```

3. When the array is passed into functions, memory is created for the elements. The first element is a pointer the array and each next array is separated by the same amount of memory. The first element is where the array is placed in memory. The array is passed in by passing a pointer to the first element of the array. The callee accesses the parameter by calculating the address of the nth element in the array and returns that memory address. In this case, because it is an array of ints, the memory are spaced by four bytes, as seen on [rdi + 4] in the picture down below.

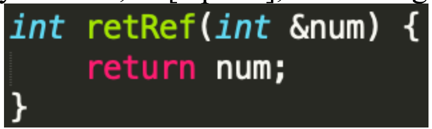


```
void a(int a[2]) {
    a[0] = 1;
    a[1] = 2;
}
```

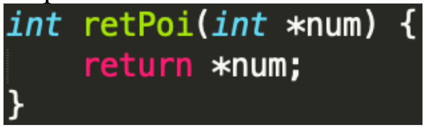


```
__Z1aPi:
    .cfi_startproc
## %bb.0:
    mov qword ptr [rsp - 8], rdi
    mov rdi, qword ptr [rsp - 8]
    mov dword ptr [rdi], 1
    mov rdi, qword ptr [rsp - 8]
    mov dword ptr [rdi + 4], 2
    ret
    .cfi_endproc
```

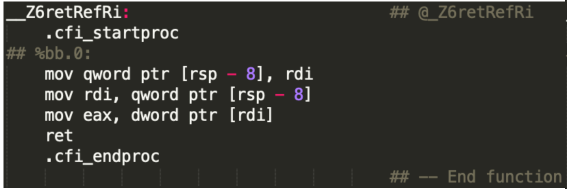
4. Comparing the two assembly code provided below for passing values by reference and passing values by pointers, it shows that there is no difference between the two in ways such as parameter passing and such. The two codes are identical. Passed into the parameter register is the memory address, in [rsp – 8], located right after rsp in the stack.



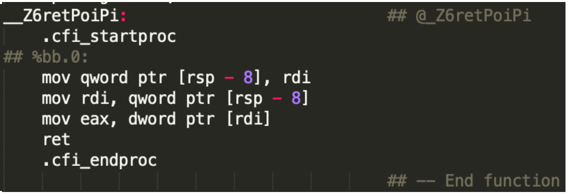
```
int retRef(int &num) {
    return num;
}
```



```
int retPoi(int *num) {
    return *num;
}
```



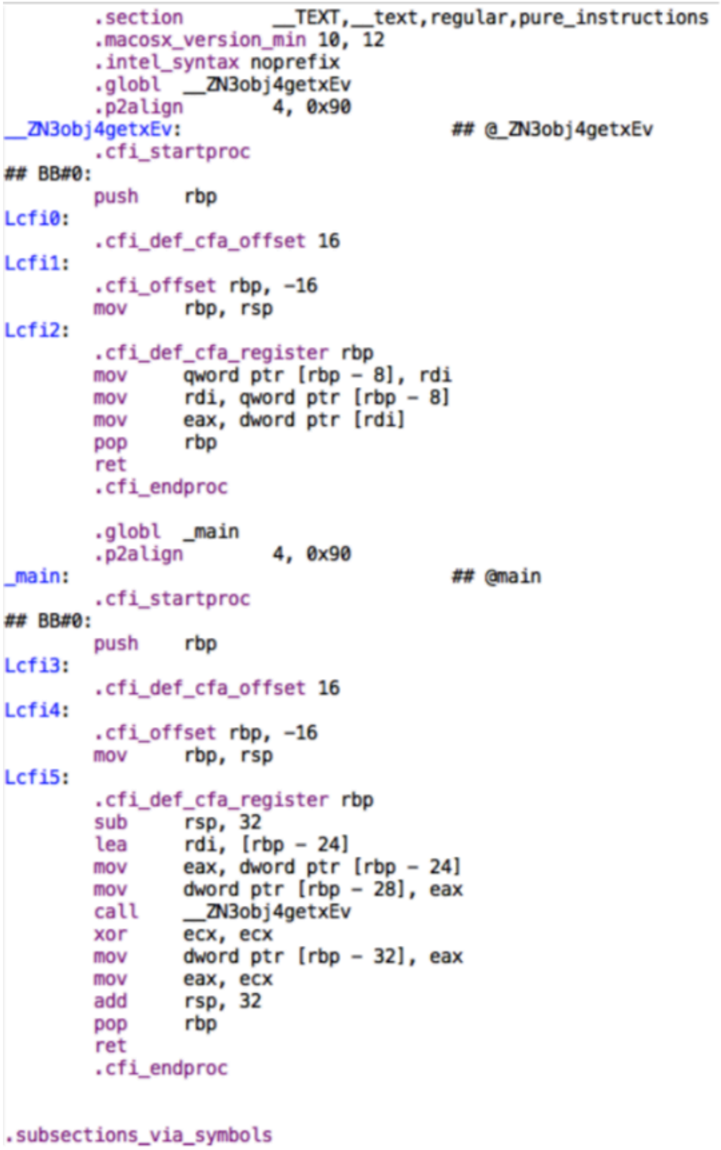
```
__Z6retRefRi:                           ## @_Z6retRefRi
    .cfi_startproc
## %bb.0:
    mov qword ptr [rsp - 8], rdi
    mov rdi, qword ptr [rsp - 8]
    mov eax, dword ptr [rdi]
    ret
    .cfi_endproc
                                        ## -- End function
```



```
__Z6retPoiPi:                           ## @_Z6retPoiPi
    .cfi_startproc
## %bb.0:
    mov qword ptr [rsp - 8], rdi
    mov rdi, qword ptr [rsp - 8]
    mov eax, dword ptr [rdi]
    ret
    .cfi_endproc
                                        ## -- End function
```

Object

1. Using dynamic memory, objects have more freedom in being made as long as we keep track of where the variables are stored. We first allocate the memory of the object and then load it into memory. Then we have to initialize the fields to be the right addresses for the functions it includes. The final stage is to call the object functions.



```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 12
        .intel_syntax noprefix
        .globl  __ZN3obj4getxEv
        .p2align        4, 0x90
__ZN3obj4getxEv:                        ## @_ZN3obj4getxEv
        .cfi_startproc
## BB#0:
        push    rbp
Lcfi0:
        .cfi_def_cfa_offset 16
Lcfi1:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Lcfi2:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     eax, dword ptr [rdi]
        pop     rbp
        ret
        .cfi_endproc

        .globl  _main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        push    rbp
Lcfi3:
        .cfi_def_cfa_offset 16
Lcfi4:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Lcfi5:
        .cfi_def_cfa_register rbp
        sub     rsp, 32
        lea     rdi, [rbp - 24]
        mov     eax, dword ptr [rbp - 24]
        mov     dword ptr [rbp - 28], eax
        call    __ZN3obj4getxEv
        xor     ecx, ecx
        mov     dword ptr [rbp - 32], eax
        mov     eax, ecx
        add     rsp, 32
        pop     rbp
        ret
        .cfi_endproc


.subsections_via_symbols
```

2. In terms of the data member access, when the data is compiled into assembly code, the data of the objects are stored in order. The function as well as the data members are able to be stored in the same place.

3. Assembly is not object oriented. The assembly knows which object it is being called out of by the order that they are stored. If one object had two integers, it would store eight bytes and store the fields of the next object right after. The registers that are reserved for storing data, such as rsi, have an order in which objects are stored. Also, when running the assembly, arguments of the function are stored on a stack in order and the location of the return is allocated on the stack.

4. To access members outside of the member function, a class has to be declared as a friend class of another. These classes can be access, but they are not directly inside the function.

To access data members inside the member function, the data was placed in a stack.

5. In order to access private fields, the get function had to be used. While private methods require the proper call using get, public items were accessible with simple calls. The this keyword was used to access private items. It allows us to access members from inside the class. In order to access member functions for my class, the this command was used. Memory was left for each of

the variable that used the this pointer. It is not placed on the stack, but rather was given a memory address. The values around “this” have memory addresses that are based on the location of where the this pointer is located. Memory is saved, and if the space is needed, it is there ready to be used.

Sources

- https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture
- https://medium.freecodecamp.org/understanding-by-reference-vs-by-value-d49139beb1c4
- https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm
- http://www.drdobbs.com/embedded-systems/object-oriented-programming-in-assembly/184408319