Fr2md   Farid Rajabi Nia   lab 11:00 am

I hate this course since it makes me miserable.

Passing by value:

I have created a nonsense function in C++ and tried to change the variable from int to float to char to a pointer to observe the difference in the created s file. here is the command I am using to create my .s file (clang++ -m64 -S -mllvm --x86-asm-syntax=intel -fomit-frame-pointer c.cpp -o c.s) It is more like a x86 version of assembly rather than 64Here is my C++:

```
 int fun(int x){
return x;}
int main (){
 int x;
fun(x);
return 0;}
```

It uses mov and 4byte for integer and then transfer it to rax in x64. When I change it to a float function it used movss still keeping the 4 byte. This time it does not store it in rax.

```
movss  dword ptr [rsp - 4], xmm0
movss  xmm0, dword ptr [rsp - 4] # xmm0 = mem[0],zero,zero,zero
ret
```

When I changed float to char, I noticed that the 4 has changed to 1 byte for char and it is using byte pointer which I am confused about.

```
mov     al, dil
mov     byte ptr [rsp - 1], al
movsx eax, byte ptr [rsp - 1]
ret
```

It seems that when we are passing parameters by value, compiler likes to hold it on the register or in some cases push it on the stacks. "Then the subroutine called simply accesses the right register to get its hands on the parameter. Of course the subroutine will have to have been written in such a way as to expect its argument to be in a given register."
(http://www.cs.mcgill.ca/~cs573/fall2002/notes/lec273/lecture15/15_4.htm)

The caller simply moves the parameter to the rdi register and then calls the subordinate.

```
mov     dword ptr [rsp + 20], 0
mov     rdi, qword ptr [rsp + 8]
call    _Z3funPi
xor     ecx, ecx
mov     qword ptr [rsp], rax    # 8-byte Spill
mov     eax, ecx
add     rsp, 24
ret
```

Passing by Reference:
Observing the behavior of the compiler while passing by value, I conclude that when passing by reference, it holds the memory address of the parameter rather than its value. "By putting the address of some parameter in memory in an address register, this technique can be used to implement passing by reference. Likewise given some parameter, if we put a copy of its value in some data register, then we are in effect implementing passing by value." (http://www.cs.mcgill.ca/~cs573/fall2002/notes/lec273/lecture15/15_4.htm)
My assumption was right and we have instances of sharing memory address in the. s file.

```
lea     rdi, [rsp + 16]
mov     dword ptr [rsp + 20], 0
call    _Z3funRi
```

Passing values by reference vs. passing by pointer:
Finally, pointers are reparented in a strange way in assembly. When I feed int pointer to the function my complier dedicates 8 bytes for each pointer. Also, it uses qword ptr instead of byte pointer for char. The first part is the representation of the function "fun" with one int pointer, and later in the main I all call the function and pass a pointer to it.

```
# BB#0:
        mov     qword ptr [rsp - 8], rdi
        mov     rdi, qword ptr [rsp - 8]
        mov     eax, dword ptr [rdi]
Ltmp0:
        lea     rax, [rsp + 16]
        mov     dword ptr [rsp + 20], 0
        mov     dword ptr [rsp + 16], 2
        mov     qword ptr [rsp + 8], rax
        mov     rdi, qword ptr [rsp + 8]
        call    _Z3funPi
        xor     ecx, ecx
        mov     dword ptr [rsp + 4], eax # 4-byte Spill
        mov     eax, ecx
        add     rsp, 24
        ret
```

when I changed the function to pass with reference instead of pointer, I got similar assembly code. They both use rdi before calling the function to store the memory address of the variable. Although pointer first started with rax.

```
.Ltmp0:
        lea     rdi, [rsp + 16]
        mov     dword ptr [rsp + 20], 0
        mov     dword ptr [rsp + 16], 2
        call    _Z3funRi
        xor     ecx, ecx
        mov     dword ptr [rsp + 12], eax # 4-byte Spill
        mov     eax, ecx
        add     rsp, 24
        ret
```

Object pass by Value and Reference:
I assumed that the object passing by value should not be different from a parameter passing by value. And my guess was right. I have created an object of a class and then passed it by value (an int and a float) to observe. The object actual values are copied around in stacks and registers as you can see below. The numbers 4, 20, 24 are the offsets from the base pointer letting the complier know where to look for when the value is being used. I have defined all my variables and function public in my object. When I changed some of it to private, I did not see a particular change in the code.

```
movss  xmm0, dword ptr [.LCPI0_0] # xmm0 = mem[0],zero,zero,zero
mov    dword ptr [rsp - 4], 0
mov    dword ptr [rsp - 20], 5
movss  dword ptr [rsp - 24], xmm0
mov    ecx, dword ptr [rsp - 20]
mov    dword ptr [rsp - 16], ecx
movss  xmm0, dword ptr [rsp - 24] # xmm0 = mem[0],zero,zero,zero
movss  dword ptr [rsp - 12], xmm0
```

When I passed in by reference, instead of copying the values, it was using memory address of the object components. In general, the object behavior was almost the same as parameters, but more complicated and verbose. My understanding of the .s file about objects is that since assembly is not OO, it treats the objects as a variable which can either be passed by value or reference.

```
lea    rcx, [rsp - 32]
lea    rdx, [rsp - 28]
movss  xmm0, dword ptr [.LCPI0_0] # xmm0 = mem[0],zero,zero,zero
mov    dword ptr [rsp - 4], 0
mov    dword ptr [rsp - 28], 5
movss  dword ptr [rsp - 32], xmm0
mov    qword ptr [rsp - 24], rdx
mov    qword ptr [rsp - 16], rcx
```

Array:
I have created an array of int and then a function feeding on that array and spiting the first element of the array. As you can see from the assembly code, it stores the array by its memory address meaning that arrays are mutable in assembly. I was hoping to see that array is stored in sequential number offset by the base pointer, but I do not see a clear image of that happening. However, the arrays components are just being moved around a lot. It seems that the array components are stored in stacks and then their memory address is copied into rcx and rdx.

```
lea    rcx, [rsp - 32]
lea    rdx, [rsp - 28]
movss  xmm0, dword ptr [.LCPI0_0] # xmm0 = mem[0],zero,zero,zero
mov    dword ptr [rsp - 4], 0
mov    dword ptr [rsp - 28], 5
movss  dword ptr [rsp - 32], xmm0
mov    qword ptr [rsp - 24], rdx
mov    qword ptr [rsp - 16], rcx
ret
```