Joseph Elsisi

Lab 108

November 6, 2019

postlab.pdf

You must use (and cite!) at least TWO additional resources for this post-lab!

https://www.tutorialspoint.com/cplusplus/cpp_passing_arrays_to_functions.htm

https://stackoverflow.com/questions/2987876/what-does-dword-ptr-mean/2987916

Parameter passing

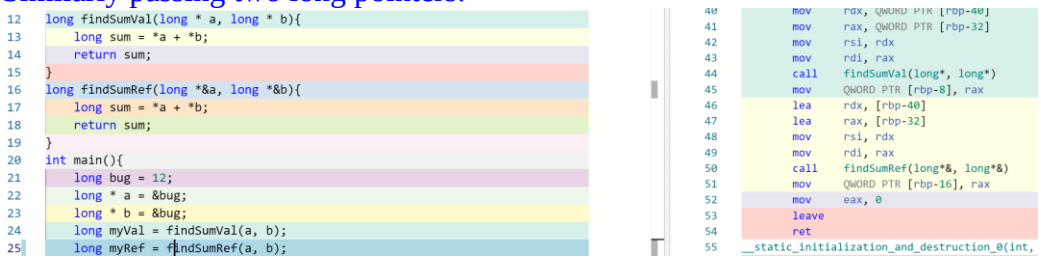
When comparing the functions I made and passing in several types, the difference between passing a value vs by reference is the way they are loaded into the parameter registers.

For understanding of qword ptr I searched https://stackoverflow.com/questions/2987876/what-does-dword-ptr-mean/2987916

Passing two longs: By reference By value

lea rdx, [rbp-32] mov rdx, QWORD PTR [rbp-32]
lea rax, [rbp-24] mov rax, QWORD PTR [rbp-24]

Similarly passing two long pointers:



```
12  long findSumVal(long * a, long * b){
13      long sum = *a + *b;
14      return sum;
15  }
16  long findSumRef(long *&a, long *&b){
17      long sum = *a + *b;
18      return sum;
19  }
20  int main(){
21      long bug = 12;
22      long * a = &bug;
23      long * b = &bug;
24      long myVal = findSumVal(a, b);
25      long myRef = findSumRef(a, b);
```

```
40      mov     rdx, QWORD PTR [rbp-40]
41      mov     rax, QWORD PTR [rbp-32]
42      mov     rsi, rdx
43      mov     rdi, rax
44      call    findSumVal(long*, long*)
45      mov     QWORD PTR [rbp-8], rax
46      lea     rdx, [rbp-40]
47      lea     rax, [rbp-32]
48      mov     rsi, rdx
49      mov     rdi, rax
50      call    findSumRef(long*&, long*&)
51      mov     QWORD PTR [rbp-16], rax
52      mov     eax, 0
53      leave
54      ret
55  __static_initialization_and_destruction_0(int,
```

It makes sense that instead of moving the value into the parameter register, the address is loaded instead when passing by reference.

```
lea     rdx, [rbp-112]
lea     rax, [rbp-64]
mov     rsi, rdx
mov     rdi, rax
call    catLoader(Cat&, Cat&)
mov     ebx, 0
```

```
lea     rdx, [rbp-208]
lea     rax, [rbp-112]
mov     rsi, rdx
mov     rdi, rax
call    Cat::Cat(Cat const&)
lea     rdx, [rbp-160]
lea     rax, [rbp-64]
mov     rsi, rdx
mov     rdi, rax
call    Cat::Cat(Cat const&)
lea     rdx, [rbp-112]
lea     rax, [rbp-64]
mov     rsi, rdx
mov     rdi, rax
call    catLoader(Cat, Cat)
lea     rax, [rbp-64]
mov     rdi, rax
call    Cat::~Cat() [complete object destructo
lea     rax, [rbp-112]
mov     rdi, rax
call    Cat::~Cat() [complete object destructo
```

The story is a little different for objects. In both cases, value and reference, the lea command is called. On the left I have passing by reference and on the right, value. It is clear that passing by value is more expensive. Instead of just loading the effective address of the object, a bit by bit copy is made (When the default constructor "call Cat::Cat(Cat const&)" is invoked) and then the address of that copy is loaded into a parameter register.

From tutorialspoint.com, "C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.". It is not possible to pass an array by its value, instead only the address of the first block of memory will be passed. Since I declared a long array of size 2, the program first reserves 16 spots of memory. (I struggle to understand the reduntent commands, maybe clang is inefficient)

```
long arr [2] = {1, 0};
takeArr(arr);
return 0;
}
```

```
13      mov     rbp, rsp
14      sub     rsp, 16
15      mov     QWORD PTR [rbp-16], 0
16      mov     QWORD PTR [rbp-8], 0
17      mov     QWORD PTR [rbp-16], 1
18      lea     rax, [rbp-16]
```

The first element in the array is one at address [rbp-16]. This address is loaded, through command lea, into rdi, the first parameter register.

```
takeArr(arr);
return 0;
}
```

```
16      mov     QWORD PTR [rbp-8]
17      mov     QWORD PTR [rbp-16
18      lea     rax, [rbp-16]
19      mov     rdi, rax
20      call    takeArr(long*)
21      mov     eax, 0
```

(I again see redundant commands, why use rax at all?) The subsequent elements in the array are accessed through arithmetic shown below.

```
void takeArr(long (arr)[]){
    long risky = arr[1];
    long riskier = arr[2];
    long riskiest = arr[5];
}

int main(){
    long arr [2] = {1, 0};
    takeArr(arr);
    return 0;
}
```

```
2     push    rbp
3     mov     rbp, rsp
4     mov     QWORD PTR [rbp-40], rdi
5     mov     rax, QWORD PTR [rbp-40]
6     mov     rax, QWORD PTR [rax+8]
7     mov     QWORD PTR [rbp-8], rax
8     mov     rax, QWORD PTR [rbp-40]
9     mov     rax, QWORD PTR [rax+16]
10    mov     QWORD PTR [rbp-16], rax
11    mov     rax, QWORD PTR [rbp-40]
12    mov     rax, QWORD PTR [rax+40]
13    mov     QWORD PTR [rbp-24], rax
```

Arr[5] is 5 * 8 = 40 spots in memory away from where the base of the array, which was loaded into rax, is. I added this example to show that assembly and C++ lack bounds checking. I don't know what is at that memory location but it is not the array I made.

In C++ passing by reference is absolutely different from passing by pointer. Notice how c, a cat, is treated differently in these two implementations.

```
void catLoader(Cat * c, Cat * b){
    c = new Cat;
    cout << "MEOW, i have " << c -> numLegs << endl;
}
void catLoader(Cat &c, Cat & b){
    c = new Cat;
    cout << "MEOW, i ate this many " << c.numLegs << endl;
}
```

References are implicitly dereferenced, meaning c is not the pointer to the cat, but the cat itself. Curiously, in assembly the implementations are the same for these methods.

```
catLoader(Cat*, Cat*):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     esi, OFFSET FLAT:.LC0
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream
        mov     rdx, rax
        mov     rax, QWORD PTR [rbp-8]
        mov     rax, QWORD PTR [rax]
        mov     rsi, rax
        mov     rdi, rdx
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(long)
        mov     esi, OFFSET FLAT:_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_ostream<char, std::char_traits<char> >
        nop
        leave
        ret
```

```
catLoader(Cat&, Cat&):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream
        mov     rdx, rax
        mov     rax, QWORD PTR [rbp-8]
        mov     rax, QWORD PTR [rax]
        mov     rsi, rax
        mov     rdi, rdx
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(long)
        mov     esi, OFFSET FLAT:_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_ostream<char, std::char_traits<char> >
```

Without their headings, these functions would be impossible to tell apart. In both cases, the memory location of the cats is loaded into the parameter registers.

Objects

Spending time changing around the number of data members in my class, I noticed that the data in a class is treated just like an array. In fact it seems that objects are treated almost identically to arrays the only difference being their methods are declared in a separate part of the program, and when used they are simply called.

```
    int sum = c -> numLegs + b -> numLegs;
}
void catLoader(Cat &c, Cat & b){
    cout << "MEOW, i ate this many " << c.numLegs << endl;
}

int main(){
    Cat test;
    Cat bruh;
    Cat backed;
```

```
43          mov     QWORD PTR [rbp-32], 4
44          mov     QWORD PTR [rbp-24], 12
45          mov     DWORD PTR [rbp-16], 9
46          mov     QWORD PTR [rbp-8], 69
47          mov     QWORD PTR [rbp-64], 4
48          mov     QWORD PTR [rbp-56], 12
49          mov     DWORD PTR [rbp-48], 9
50          mov     QWORD PTR [rbp-40], 69
51          mov     QWORD PTR [rbp-96], 4
52          mov     QWORD PTR [rbp-88], 12
53          mov     DWORD PTR [rbp-80], 9
```

Three new cats are basically three new arrays.

As you can see in catLoader, data members are accessed by their offset from the base pointer.

```
class Cat{
    public:
        long numLegs = 4;
        long bruh = 12;
        int letsgo= 9;
        //long surgfs = 198;
        //float murica = 99.9;
    private:
    long num = 69;
        //long numLives = 9;
        //string str = "hi";
};
void catLoader(Cat * c, Cat * b){
    int sum = c -> numLegs + b -> numLegs;
}
```

```
1   catLoader(Cat*, Cat*):
2           push    rbp
3           mov     rbp, rsp
4           mov     QWORD PTR [rbp-24], rdi
5           mov     QWORD PTR [rbp-32], rsi
6           mov     rax, QWORD PTR [rbp-24]
7           mov     rax, QWORD PTR [rax]
8           mov     edx, eax
9           mov     rax, QWORD PTR [rbp-32]
10          mov     rax, QWORD PTR [rax]
11          add     eax, edx
12          mov     DWORD PTR [rbp-4], eax
13          nop
14          pop     rbp
15          ret
16  .LC0:
17          .string "MEOW, i ate this many "
18  catLoader(Cat&, Cat&):
```

In order for assembly to know which object it is calling, it is dependent on the address (offset from the base pointer) that was passed in. Again, to find which data member in that specific object, the offset from the beginning address of the object is used.

An example of this is that when “Cat test;” is declared, its “bruh” data member is offset by 40 from the base pointer. When it is called “test.bruh”, the assembly simply looks at what is at [rbp – 40].

It is a little more complicated when accessing data members within a subroutine. As you can see in catLoader, data members are accessed by their offset from the address of the beginning of the object. Then, the 8 byte long that we are looking for is found by rax, the register that holds the beginning address of the object.

```
1   catLoader(Cat*, Cat*):
2           push    rbp
3           mov     rbp, rsp
4           mov     QWORD PTR [rbp-24], rdi
5           mov     QWORD PTR [rbp-32], rsi
6           mov     rax, QWORD PTR [rbp-24]
7           mov     rax, QWORD PTR [rax+8]
8           mov     edx, eax
9           mov     rax, QWORD PTR [rbp-32]
10          mov     rax, QWORD PTR [rax+8]
11          add     eax, edx
12          mov     DWORD PTR [rbp-4], eax
13          nop
14          pop     rbp
15          ret
```

Member functions are called using the call syntax, which accepts a label of an address in the program. In my class, the label catLoader(Cat*, Cat*): will store the location of that subroutine. When called, the parameter for the object in question will have it’s beginning address stored.

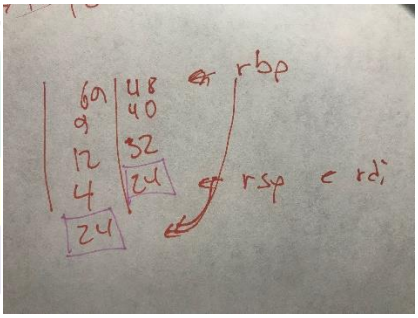
In the case of ‘this’, the stack pointer is the memory address stored parameter rdi that was lea from the first element in the object array.

```
mov     QWORD PTR [rbp-48], 4
mov     QWORD PTR [rbp-40], 12
mov     DWORD PTR [rbp-32], 9
mov     QWORD PTR [rbp-24], 69
lea     rax, [rbp-48]
mov     rdi, rax
```

The address is then stored right under that first element [rbp-8] by an offset of 8 bytes, and is used as the ‘this’ pointer to reference all other data in the object that was passed into the method. In this implementation ‘this’ was stored in rax.

[rax+8] yields 12, the data member we want.

```
catLoader(Cat*):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     rax, QWORD PTR [rax+8]
        pop     rbp
        ret
main:
```



Thank you for coming to my ted talk.