Gabriel Groover (gtg3vv)

Postlab8.pdf

Passing Parameters

In order to test how parameters are passed between the caller and callee in c++, I created a simple main() and method for each parameter type. The main method would initialize a value of each type, and then call the method that simply accepts and returns that variable.

Int – When passing an int into a function, the values are placed at addresses relative to ebp. You will see that DWORD is used to allow 4 bytes of space for the int. The addresses are also four apart to account for this. The callee then moves the value from its address into the eax register so that it can be returned, and then returns it.

```
Caller:  Mov DWORD PTR [ebp-4], 1
         Mov eax, DWORD PTR [ebp-4]
         Mov DWORD PTR [esp], eax
Callee: Mov ebp, esp
         Mov eax, DWORD PTR [ebp+8]
```

When passing an int by reference, the only difference in the caller is that the address must be moved into eax using lea. The callee must move the address to eax, and then get the value at the address before moving that into eax.

```
Caller: Lea eax, [ebp – 8]
         Mov DWORD PTR [ebp-4], eax
         Mov eax, DWORD PTR [esp], eax
Callee: Mov eax, DWORD PTR [ebp+8]
         Mov eax, DWORD PTR [eax]
```

Char- Char behaves identically to int for reference and value. The only difference is that chars require only a single byte of space and are declared using BYTE instead of DWORD. The acii value is placed into the register instead of the actual character.

```
Caller: Mov BYTE PTR [ebp-1], 99
         Movsx eax, BYTE PTR [ebp-1]
         Mov DWORD PTR [esp], eax
Callee: Mov eax, DWORD PTR [ebp+8]
         Mov BYTE PTR [ebp-4], al
```

Movzx eax, BYTE PTR [ebp-4]

Pointer – If we pass a pointer by value into a function, it will be exactly the same as passing by reference as above. This will pass an address into the function and the callee will have to retrieve the value at the address before placing it in eax. Passing the address using a pointer by value is the same as passing some other type by reference.

Float – The storage of a float is the same as an int. In this case, they take up the same amount of space and will be placed at addresses 4 bytes apart. The notation is slightly different to denote that the value is a float. The fld command allows the float to be loaded onto the stack. If a float is passed by reference it will be identical to passing an int by value with only the added fld command.

Caller: Mov eax, DWORD PTR .LC1

Mov DWORD PTR [ebp-4] ,eax

Mov eax, DWORD PTR [ebp-4]

Callee: Mov DWORD PTR [ebp-4], eax

Fld DWORD PTD [ebp-4]

Arrays – The array base address is passed into the function and loaded into eax using the lea command as above. The rest of the caller function will match that of whatever data type the array holds. In my example I used integers. The values are placed 4 bytes apart and relative to the array base address. By passing the array base address, we allow the callee to access any element of the array.

Caller: Mov DWORD PTR [ebp-16], 1

Mov DWORD PTR [ebp-12] , 2

Mov DWORD PTR [ebp-8],3

Mov DWORD PTR [ebp-4],4

Lea eax, [ebp-16]

Objects

Objects and their corresponding values seem to be stored very similarly to arrays. Arrays store all of the values relative to a base address, and objects group values together the same way. I created a simple object that contained an int, *int, char, float and getI() function. The int is a private member. When creating the object in assembly, 36 is subtracted from esp. This is presumably to make room for all of the member variables that will now be stored in that space. Each individual value is then moved into a location relative to ebp. This step is similar to the passing caller for each of the above data types. The difference here is there is more space between the variables than the size of the variable. Each instance of this object will have a new set of variables at a new location in memory. The function geti() however, will only be placed in a single location. When the function is called, it receives a “this” pointer to the object it is being

called on to allow it to access that object's fields. This is a key distinction and we should be able to see this in the assembly code generated for my program.

```
Callee: sub esp, 36
        Mov DWORD PTR [ebp-20], 3
        Mov BYTE PTR [ebp-12] , 99
        Lea eax, [ebp-20]
        Mov DWORD PTR [ebp-16], eax
        Mov eax, DWORD PTR .LC0
        Mov DWORD PTR [ebp-8], eax
        Lea eax, [ebp-16]
        Mov DWORD PTR [esp] ,eax
```

We see that all of the values are moved to addresses relative to the base pointer. The int, char, pointer and float are all handled and stored as we would expect from above. It seems to me that this could be done with fewer instructions by skipping the eax command at each point. If this is being generated line by line however, the eax step could be necessary to return the proper values. The last step here is the most important one. The lea command is used to load the address of the object base into esp before calling geti(). This allow geti() to know what object is being called on.

```
Caller:  mov ebp, esp
         Mov eax, DWORD PTR [ebp+8]
         Mov DWORD PTR [eax+12], 3
         Mov eax, DWORD PTR [ebp+8]
         Mov eax, DWORD PTR [eax +12]
```

The geti() function first sets I, and then returns it. As we had hoped, we see that the address previously put in esp, is now moved to ebp to access each of the objects fields. Once this is done, the process is identical to setting an int in any other scenario.

Sources:

http://stackoverflow.com/questions/12378271/what-does-an-object-look-like-in-memory

http://stackoverflow.com/questions/1658294/whats-the-purpose-of-the-lea-instruction