Liya Xu

lx2hy

03/27/2014

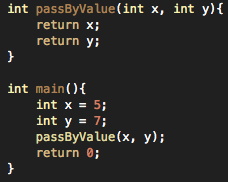
Section 103

**Postlab**

**Parameter passing**

* Int pass by value:

I wrote a small C++ program that contained a function taking integer value x and y and then return them.



Then I generated the assembly code, and found out that the caller code is:

*push EBP*

*mov EBP, ESP*

*sub ESP, 24*

*mov DWORD PTR [EBP - 4], 0*

*mov DWORD PTR [EBP - 8], 5*

*mov DWORD PTR [EBP - 12], 7*

*mov EAX, DWORD PTR [EBP - 8]*

*mov ECX, DWORD PTR [EBP - 12]*

*mov DWORD PTR [ESP], EAX*

*mov DWORD PTR [ESP + 4], ECX*

*call \_Z11passByValueii*

*mov ECX, 0*

*mov DWORD PTR [EBP - 16], EAX # 4-byte Spill*

*mov EAX, ECX*

*add ESP, 24*

*pop EBP*

*ret*

**It directly load the variables to the registers.**

The callee code is:

*sub ESP, 8*

*mov EAX, DWORD PTR [ESP + 16]*

*mov ECX, DWORD PTR [ESP + 12]*

*mov DWORD PTR [ESP + 4], ECX*

*mov DWORD PTR [ESP], EAX*

*mov EAX, DWORD PTR [ESP + 4]*

*add ESP, 8*

*ret*

* Int pass by reference:

The codes are all the same except that there are two more lines of code in the forth and fifth line:

*lea EAX, DWORD PTR [EBP - 8]*

*lea ECX, DWORD PTR [EBP - 12]*

Since the user is passing the memory location, these two lines serve to load the addresses of the value.

* Char pass by value: I changed the C++ function return type to char, and set x and y to chars. Then I generated the assembly code.

Caller code:

*push EBP*

*mov EBP, ESP*

*sub ESP, 24*

*mov DWORD PTR [EBP - 4], 0*

*mov BYTE PTR [EBP - 5], 97*

*mov BYTE PTR [EBP - 6], 98*

*mov AL, BYTE PTR [EBP - 5]*

*movsx ECX, AL*

*mov DWORD PTR [ESP], ECX*

*movsx ECX, BYTE PTR [EBP - 6]*

*mov DWORD PTR [ESP + 4], ECX*

*call \_Z11passByValuecc*

*mov ECX, 0*

*mov BYTE PTR [EBP - 7], AL # 1-byte Spill*

*mov EAX, ECX*

*add ESP, 24*

*pop EBP*

*ret*

**In the underlined part, 97 and 98 are ascii values of char a and b which I set for x and y. It also uses BYTE instead of DWORD, since a char take up a byte.**

Callee code:

*sub ESP, 2*

*mov AL, BYTE PTR [ESP + 10]*

*mov CL, BYTE PTR [ESP + 6]*

*mov BYTE PTR [ESP + 1], CL*

*mov BYTE PTR [ESP], AL*

*movsx EAX, BYTE PTR [ESP + 1]*

*add ESP, 2*

*ret*

* Char pass by reference: There are two more lines of code in the fourth and fifth line

*lea EAX, DWORD PTR [EBP - 5]*

*lea ECX, DWORD PTR [EBP - 6]*

**It works the same way as it is in int pass by reference – load the effective addresses of the char value.**

* Pointer by value: I changed the function to take in pointers and return them. Caller code:

*push EBP*

*mov EBP, ESP*

*sub ESP, 40*

*lea EAX, DWORD PTR [EBP - 12]*

*lea ECX, DWORD PTR [EBP - 8]*

*mov DWORD PTR [EBP - 4], 0*

*mov DWORD PTR [EBP - 8], 5*

*mov DWORD PTR [EBP - 12], 7*

*mov DWORD PTR [EBP - 16], ECX*

*mov DWORD PTR [EBP - 20], EAX*

*mov EAX, DWORD PTR [EBP - 16]*

*mov ECX, DWORD PTR [EBP - 20]*

*mov DWORD PTR [ESP], EAX*

*mov DWORD PTR [ESP + 4], ECX*

*call \_Z11passByValuePiS\_*

*mov ECX, 0*

*mov BYTE PTR [EBP - 21], AL # 1-byte Spill*

*mov EAX, ECX*

*add ESP, 40*

*pop EBP*

*ret*

**In the underlined code, it handled the addresses differently than the caller code of int pass by reference, since it is passed by the pointer.**

Callee:

*sub ESP, 8*

*mov EAX, DWORD PTR [ESP + 16]*

*mov ECX, DWORD PTR [ESP + 12]*

*mov DWORD PTR [ESP + 4], ECX*

*mov DWORD PTR [ESP], EAX*

*mov EAX, DWORD PTR [ESP + 4]*

*mov EAX, DWORD PTR [EAX]*

*mov DL, AL*

*movsx EAX, DL*

*add ESP, 8*

*ret*

* Float pass by value:

Caller code:

*push EBP*

*mov EBP, ESP*

*sub ESP, 40*

*movss XMM0, DWORD PTR [.LCPI3\_0]*

*movss XMM1, DWORD PTR [.LCPI3\_1]*

*mov DWORD PTR [EBP - 4], 0*

*movss DWORD PTR [EBP - 8], XMM1*

*movss DWORD PTR [EBP - 12], XMM0*

*movss XMM0, DWORD PTR [EBP - 8]*

*movss XMM1, DWORD PTR [EBP - 12]*

*movss DWORD PTR [ESP], XMM0*

*movss DWORD PTR [ESP + 4], XMM1*

*call \_Z11passByValueff*

*fstp DWORD PTR [EBP - 16]*

*movss XMM0, DWORD PTR [EBP - 16]*

*mov EAX, 0*

*movss DWORD PTR [EBP - 20], XMM0 # 4-byte Spill*

*add ESP, 40*

*pop EBP*

*ret*

Callee code:

*sub ESP, 12*

*movss XMM0, DWORD PTR [ESP + 20]*

*movss XMM1, DWORD PTR [ESP + 16]*

*movss DWORD PTR [ESP + 8], XMM1*

*movss DWORD PTR [ESP + 4], XMM0*

*movss XMM0, DWORD PTR [ESP + 8]*

*movss DWORD PTR [ESP], XMM0*

*fld DWORD PTR [ESP]*

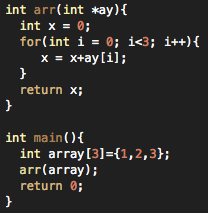
*add ESP, 12*

*ret*

**It changes the “mov” command to “movss”, which is a move command for floating point numbers.**

* Float pass by reference: it is similar to int by reference except the movss command.
* Object pass: the structure will change accordingly to the type of object users pass in. If you pass objects that hold chars, it will have similar codes as the char passing.
* Passing Arrays

I wrote a simple function to show how arrays are passed.



Accordingly, the assembly codes are:

Caller code:

*# BB#0:*

*push EBP*

*mov EBP, ESP*

*sub ESP, 24*

*mov DWORD PTR [EBP - 4], 0*

*mov EAX, .L\_ZZ4mainE5array*

*mov DWORD PTR [EBP - 16], EAX*

*mov EAX, .L\_ZZ4mainE5array+4*

*mov DWORD PTR [EBP - 12], EAX*

*mov EAX, .L\_ZZ4mainE5array+8*

*mov DWORD PTR [EBP - 8], EAX*

*lea EAX, DWORD PTR [EBP - 16]*

*mov DWORD PTR [ESP], EAX*

*call \_Z3arrPi*

*mov ECX, 0*

*mov DWORD PTR [EBP - 20], EAX # 4-byte Spill*

*mov EAX, ECX*

*add ESP, 24*

*pop EBP*

*ret*

**It seems that the array is set up like this:**

**.L\_ZZ4mainE5array:**

**.long 1 # 0x1**

**.long 2 # 0x2**

**.long 3 # 0x3**

**.size .L\_ZZ4mainE5array, 12**

**refer to the underlined code, the caller then set up these values in EAX, then correct the offset of EBP. The caller then move .L\_ZZ4mainE5array+4 into EAX and then moves EAX into the address of ESP to prepare to call test function.**

Callee code:

*# BB#0:*

*sub ESP, 12*

*mov EAX, DWORD PTR [ESP + 16]*

*mov DWORD PTR [ESP + 8], EAX*

*mov DWORD PTR [ESP + 4], 0*

*mov DWORD PTR [ESP], 0*

*.LBB3\_1: # =>This Inner Loop Header: Depth=1*

*cmp DWORD PTR [ESP], 3*

*jge .LBB3\_4*

*# BB#2: # in Loop: Header=BB3\_1 Depth=1*

*mov EAX, DWORD PTR [ESP + 4]*

*mov ECX, DWORD PTR [ESP]*

*mov EDX, DWORD PTR [ESP + 8]*

*add EAX, DWORD PTR [EDX + 4\*ECX]*

*mov DWORD PTR [ESP + 4], EAX*

*# BB#3: # in Loop: Header=BB3\_1 Depth=1*

*mov EAX, DWORD PTR [ESP]*

*add EAX, 1*

*mov DWORD PTR [ESP], EAX*

*jmp .LBB3\_1*

*.LBB3\_4:*

*mov EAX, DWORD PTR [ESP + 4]*

*add ESP, 12*

*ret*

*.Ltmp10:*

*.size \_Z3arrPi, .Ltmp10-\_Z3arrPi*

*.globl main*

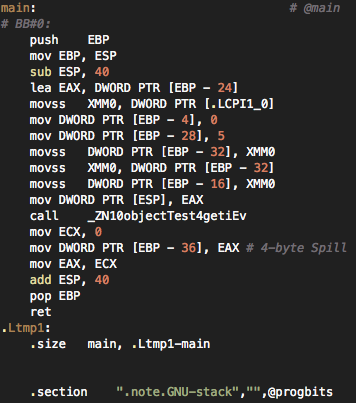
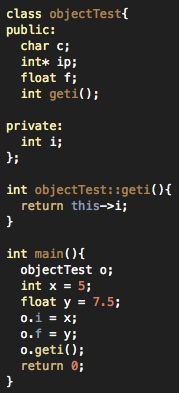
*.align 16, 0x90*

*.type main,@function*

* The assembly code for pointer and reference passing is the same. They both push a pointer to the stack when passed.

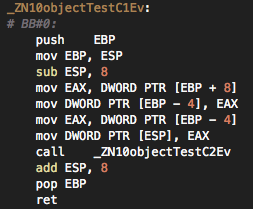
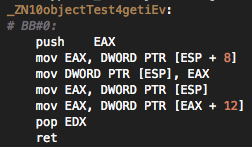
**Objects**

I wrote a C++ program to create objects and a method. Accordingly, I generated the assembly code for this program.



In the main, first it subtracts 40 from ESP to make room for the variables. The number it subtracts is decided by the size of class. Then it moves the address of [EBP-24] to EAX. The 24 here comes from the size of the types in the fields. Char takes up 4 bits, int\* takes up 8, float takes up 8 and int takes up 4. Next, it uses the “movss” command to store the float into XMM0. As I discussed in the previous section, “movss” is the command for float. Then the next three lines set up the parameters with EBP-4, EPB-28 and EBP-32. Before calling the geti() method, it sets the int of the object to x, and the float of the object to y. Then after this it moves the value of EAX to ESP, which clears the unused space that the program leaves earlier.

In order to access the objects in the private field, a “get” method must be called in the main. For objects in the public field, they can be directly accessed. When accessing from the inside we need to use “this” pointer. The assembly code on the left is the geti() method, and it shows how “this” pointer is implemented. The code on the right is the assembly code for the object I created in the main.



References:

<http://en.wikipedia.org/wiki/X86>

<http://en.wikipedia.org/wiki/X86_calling_conventions>

<http://en.wikipedia.org/wiki/X86_instruction_listings>

and the CS 2150 TA