

CS 3844 Computer Organization - Lab #06 Name/abc123:

James Hernandez / jwg455

This lab focuses on the stack frame, calling a function, passing arguments, and return values. We call a recursive function “printArgv” which prints the arguments passed into the program. For this lab, enter the following list of words “bat”, “Elephant”, “LEOPARD”, “whale”, “bird” into the command arguments as shown in Figure 1 - Lab #6 Setup. You are free to create your own list but for your values to match the labs, this is the list to use.

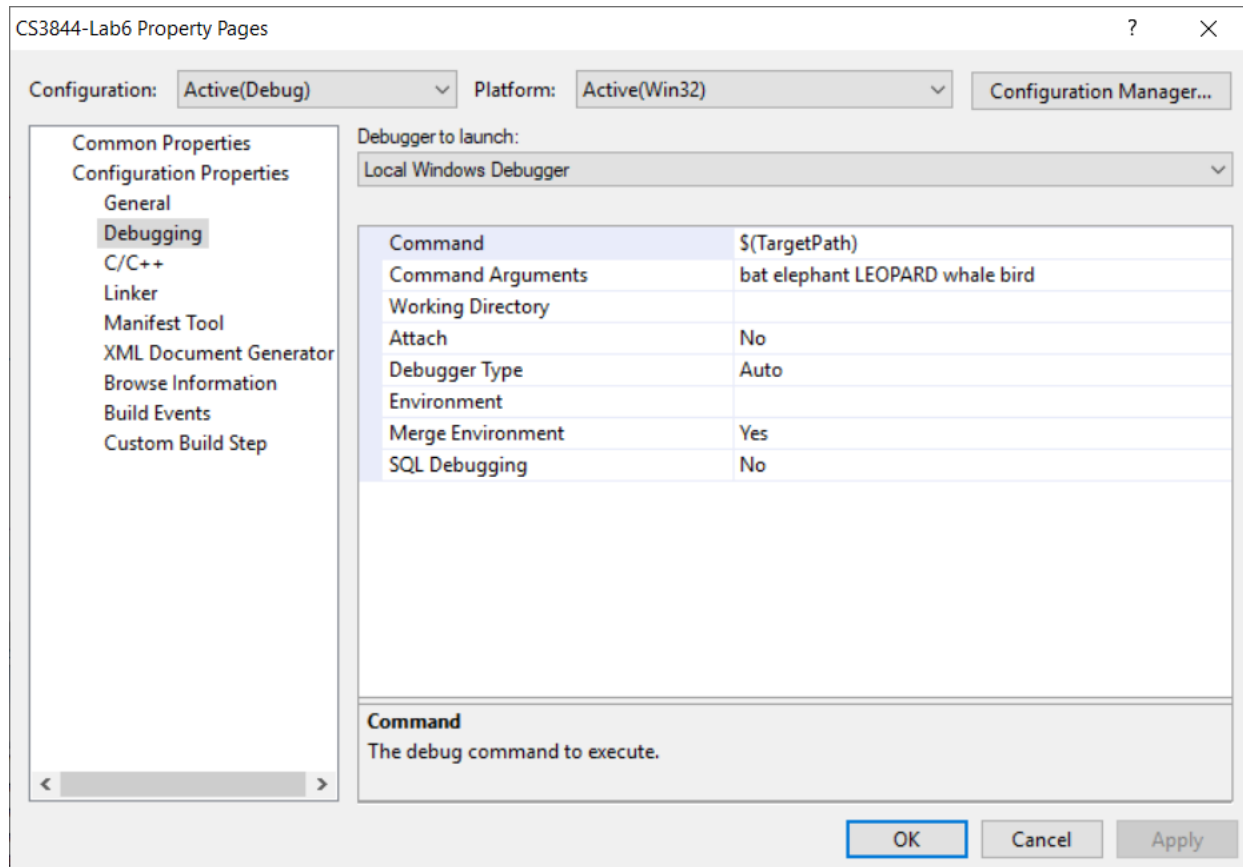


Figure 1 - Lab #6 Setup

The “main” function has two parameters: “int argc” and “char *argv[].” The argc parameter is the number of elements in the argv[] array. The argv[] array is an array of character pointers. So each element of the array is the address of a character – in this case, the first character in a null-terminated string. Since argv[0] is always the path and filename of the program we are running, argc is always at least one. NOTE: Since array indexes start at zero, argv[argc] is invalid.

Compile and run the program and observe what it does. It recursively prints the last argument down to argument #1, (recall, Argument #0 is the path/program name) as shown below.

Argument #5:

00982134: 62 69 72 64 00 bird

The address is the value of the argv[5] array element, the next 4 values are the hexadecimal numbers representing the four ASCII characters in the word, “bird.” Addresses may differ.

Take a few minutes to study the source code. MAKE SURE you understand what and why it does what it does because if you don't understand the source code, the assembly is going to be worse! Trust me, you will spend more time if you skip this step. On the exam, you won't even get source code for some questions.

Set a breakpoint on the line in main: `resulti = printArgV ...` Also set a breakpoint on the return instruction after that. And finally, one more BP at the line inside the `printArgV` function: `"printf("%s", msg);"` Set up at least one memory window with 16 bytes displayed.

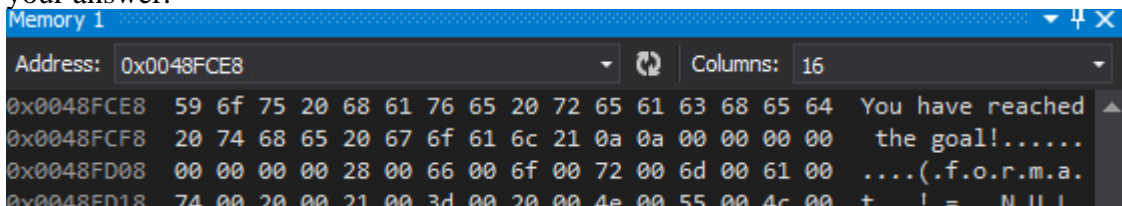
Run the program inside the debugger and view disassembly. This should look very familiar since it is almost identical to HW#4. There is an extra parameter to `printArgV`, the message which is passed in as a literal string. Note: If you are not seeing the machine code values in the disassembly window, right-click and select "Show Code Bytes."

1. Describe what is meant by a literal string.

The address of the string is the

Notice the literal string is not pushed, but rather the address of that string. (Do NOT be confused by the word "offset" – in this situation it just means "address")

2. Single-step once to execute the push instruction. Get the address of the string from the stack. Type that address into a memory window. What is the address? Cut/paste the data in the memory window here for your answer.

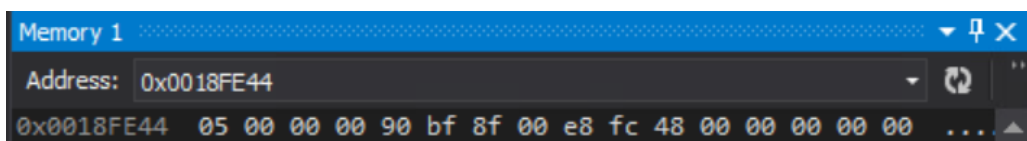


```

    resulti = printArgV(myArgC, argv, "You have reached the goal!\\n\\n");
004318D8 68 E8 FC 48 00      push     48FCE8h
004318DD 8B 45 0C            mov     eax,dword ptr [argv]
004318E0 50                push     eax
004318E1 8B 4D F8            mov     ecx,dword ptr [myArgC]
004318E4 51                push     ecx
004318E5 E8 CF D4 FF FF      call    printArgV (042F0B9h)
004318EA 83 C4 0C            add     esp,0Ch
004318ED 89 45 EC            mov     dword ptr [resulti],eax

```

3. Step to the "call" instruction (Takes two steps because of the intermediate jmp). Put esp in the memory window. Show the top 16 bytes on the stack here.



- a. What is the value of 5 representing?

The number of elements also, argc.

- b. What is the return address?

0x00431BEA

Notice that for the declaration “int x;” there is no assembly code. Reserving space on the stack is all that is required.

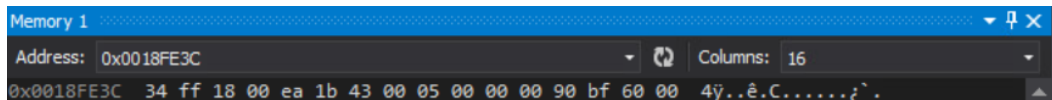
4. Single-step down to the jne instruction.

Jump Not Equal means it will jump when the zero flag is not equal to one (i.e. $Z = 0$). In human terms, the jne simply means that the jump will be taken when the two things being compared are not equal to each other. If they are, then zero is the result, and $Z = 1$. Look at the CMP (compare) instruction and the value of argc and determine if the jump is taken or not. CMP will perform this subtraction: $\text{argc} - 0$; The answer is not stored but the flags are set so $Z = 1$ only when $\text{argc} = 0$.

- a. The parameter argc is stored at what displacement from ebp?

ebp+8 because it is the first parameter.

- b. What is its value? Find it on your stack.



05 00 00 00

- c. In the debugger you can hover over it to see the value, but not on the exam. What is the value in ebp and at what address is argc stored on the stack?

EBP = 0x0018FE3C

argc is stored at = 0x0018FE44 [ebp+8]

- d. Based on that, will the jne be taken?

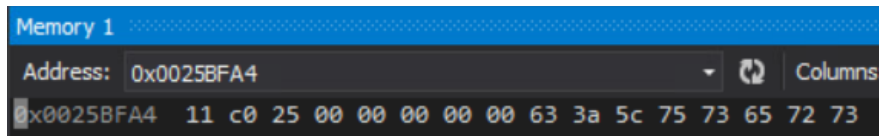
No

5. Single-step to the “`lea edx, [ecx+eax*4]`” instruction. Ecx is the base register and has a value of the address of argv. (0x662120 for my lab run, may vary). Eax is equal to argc. (5).

- a. What will edx equal after executing this instruction given my numbers above?

EDX = 0x0025BFA4

- b. If this were a “`mov edx, [ecx+eax*4]`” instruction, instead of copying the calculated value into edx, it would take the additional step of using that value as an address and copying the contents of that memory location into edx. Execute the lea instruction, type edx in the memory window, and show the contents of memory here:



6. Now go ahead and press the continue button (green arrow) and look at the contents of the output window:

There are 6 arguments to this program.

Argument #5:

00662134: 62 69 72 64 00 bird

Argument #4:

00662130: 77 68 61 6C 65 00 whale

Argument #3:

0066212C: 4C 45 4F 50 41 52 44 00 LEOPARD

Argument #2:

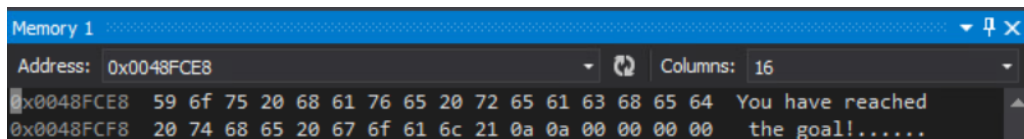
00662128: 45 6C 65 70 68 61 6E 74 00 Elephant

Argument #1:

00662124: 62 61 74 00 bat

Your screen should look similar to the above output, though addresses may vary. In the assembly code, this time, that `jne` was not taken.

- a. Show `msg` in a memory window. The value of `msg` is the address: `0x582C98`. It's contents are the ASCII characters you see.



Visual Studio is a little confusing. It shows this:

```
"0052D60F 8B 45 10 mov eax,dword ptr [msg]"
```

in the disassembly window. However, it is actually this:

```
"0052D60F 8B 45 10 mov eax,dword ptr [ebp + 0x10]."
```

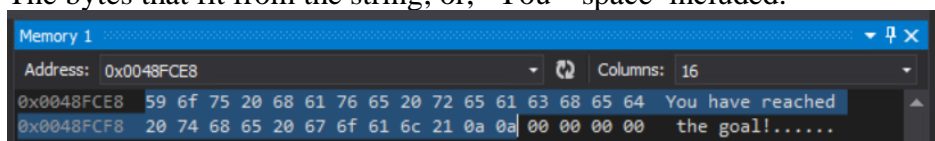
So we are moving the value of the 3rd argument into `eax` which is `0x582C98`. But it looks like we are using `msg` as a displacement like this:

```
"0052D60F A1 98 2C 58 00 mov eax,dword ptr [msg (582CA1h)]."
```

Note how the address is directly embedded in the machine code.

- b. If we did this: `mov edx, dword ptr [msg]` (`msg=0x582CA1` used as displacement) what value would be in `edx`?

The bytes that fit from the string, or, "You " space included.



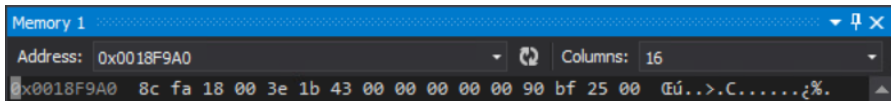
- c. If we did this: `lea edx, dword ptr [msg]` what value would be in `edx`?

“Y o u _ “
59 6F 75 20

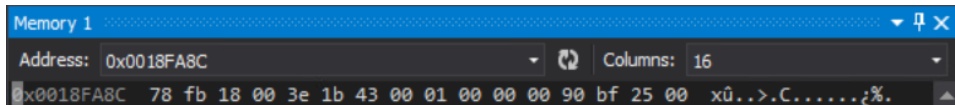
“You “ (space included)

7. We are going to walk the stack. Each stack frame has 0x44 bytes for local variables, 4 bytes for the prior `ebp`, 4 bytes for the return address, 12 bytes for the arguments, and 12 bytes for saving registers, for 0x64 bytes total. Put `ebp` in the memory window.

- a. Show 16 bytes of the memory window.



- b. `EBP` is pointing to the prior `ebp`, so type the address shown in the memory window and show the top 16 bytes below. Note that $0x19FCC0 + 0x64 = 0x19FD24$.



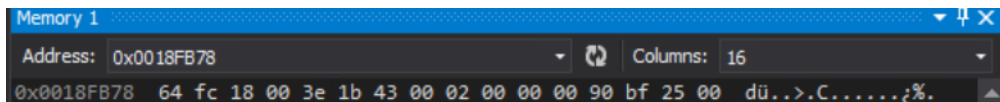
- c. At what address is the first argument on this stack frame and what does it equal?

The first argument on the stack would be at `EBP`'s value address + 8 .

So, 0x0018FA94

The value would be 01

- d. Show the next stack frame here.



- e. What is the return address for this stack frame?

0x0018FC64

8. You can continue until you reach the top stack frame if you like, but otherwise stop the debugger. Put a breakpoint on the “for” loop and run the debugger again. You can use the debugger to step through this code to help (if needed). Rather than comment all of this code, I am going to ask specific questions.

```
0052D645 C7 45 FC 00 00 00 00 mov     dword ptr [x],0
0052D64C EB 09                jmp     printArgV+57h (52D657h)

0052D64E 8B 45 FC                mov     eax,dword ptr [x]
0052D651 83 C0 01                add     eax,1
0052D654 89 45 FC                mov     dword ptr [x],eax

0052D657 8B 45 08                mov     eax,dword ptr [argc]
0052D65A 8B 4D 0C                mov     ecx,dword ptr [argv]
0052D65D 8B 14 81                mov     edx,dword ptr [ecx+eax*4]
0052D660 52                    push    edx
0052D661 E8 89 E0 FF FF        call    @ILT+1770(_strlen) (52B6EFh)
0052D666 83 C4 04                add     esp,4

0052D669 39 45 FC                cmp     dword ptr [x],eax
0052D66C 7F 20                jg      printArgV+8Eh (52D68Eh)

0052D66E 8B 45 08                mov     eax,dword ptr [argc]
0052D671 8B 4D 0C                mov     ecx,dword ptr [argv]
0052D674 8B 14 81                mov     edx,dword ptr [ecx+eax*4]
0052D677 8B 45 FC                mov     eax,dword ptr [x]
0052D67A 0F BE 0C 02            movsx   ecx,byte ptr [edx+eax]
0052D67E 51                    push    ecx
0052D67F 68 74 2C 58 00        push    offset string "%02X " (582C74h)
0052D684 E8 7A EA FF FF        call    @ILT+4350(_printf) (52C103h)
0052D689 83 C4 08                add     esp,8

0052D68C EB C0                jmp     printArgV+4Eh (52D64Eh)
0052D68E 8B 45 08                mov     eax,dword ptr [argc]
```

- a. What is the instruction at 0x52D645 doing?

Zeroing out X.

- b. What are the 3 instructions at 0x52D64E doing?

Storing address of x into EAX, then incrementing EAX by one, then updates value of x.

- c. For instructions at 0x52D657, assume that argc = 3 and argv = 0x22120, from what address does edx get its value? (Hint: you can type ecx+eax*4 in the memory window.)

It is moving the value of EAX + ECX*4 into EDX, we get this from EAX being equal to argc = 3 and EDX being the base register and has the value of argv. So we can calculate what EDX will be.

- d. If this was an array of “short int” values instead of “char *” values, how would you write this instruction “mov edx,dword ptr [ecx+eax*4]” differently?

I would use WORD PTR to move 2 bytes respectively into EDX.

- e. What are the instructions at 0x52D669/6C doing?

Comparing the contents of EAX to X and executing a jump-if-greater-than statement.

f. Where does the value in `eax` come from?

The value of `EAX` comes from `argc`.

g. “`jg`” (jump if greater than) is a signed jump – why is it signed?

Because we subtracting 1 from `argc`, and there is a possibility of a negative value.

9. Argv is an array of pointers (a double pointer). Each element contains an address of a character. In this case, it is a character string. After this instruction “**52D674: mov edx,dword ptr [ecx+eax*4]**” edx contains the address of the character string corresponding to the argc element number.
- a. Describe what is eax being used to do in this instruction “**movsx ecx, byte ptr [edx+eax]**?”

EAX would be the base register for ECX.

- b. The scale factor is one (not shown) but why is it one?

Because we are using a byte ptr.

- c. Describe what ecx contains after the above instruction executes?

This would contain the stack frame and parameters for the next loop.

10. This concludes Lab #6