CS 3844 Computer Organization - Lab #06 Name/abc123: Ariel Guerrero aes604

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.

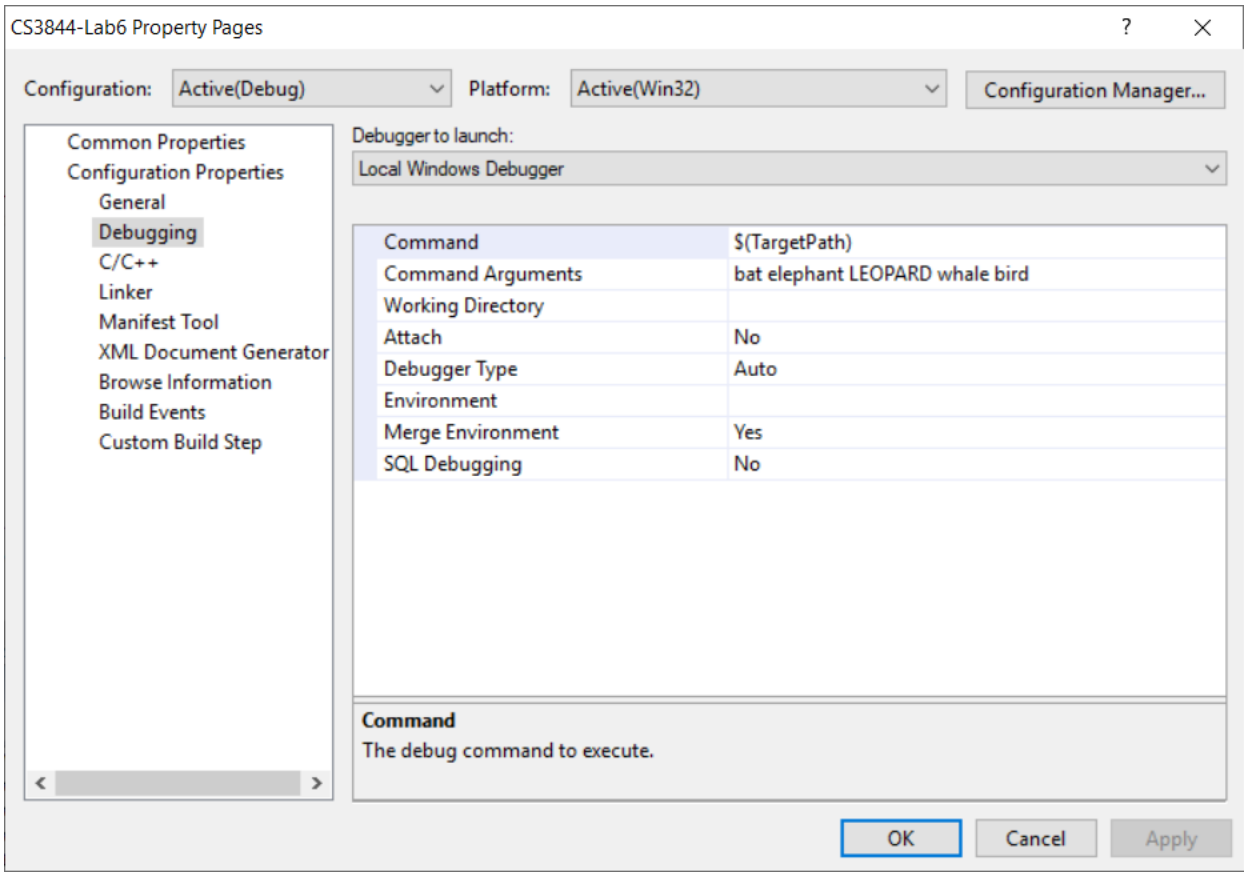


Figure 1 - Lab #6 Setup

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.

(saving cause its a pain to keep looking up)
My data:

address of bird (argv[5]) = 0065C009
value = `62 69 72 64 00 = bird`

address of msg = 0048FCE8

value = "You have reached the goal!"

```
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.

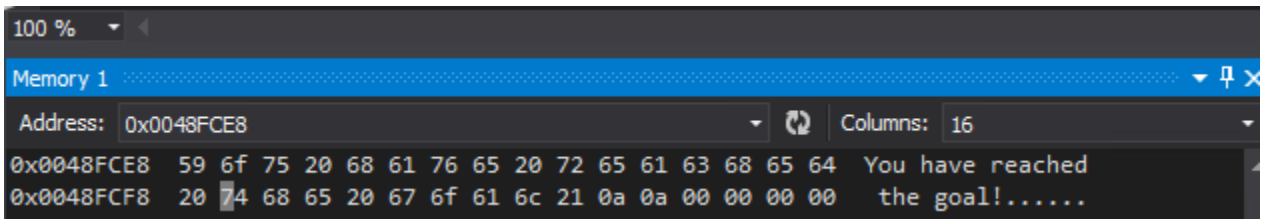
Run the program inside the debugger and view disassembly. This should look very familiar since it is almost identical to HW#4. There is an extra parameter to printArgV, the message which is passed in as a literal string. Note: If you are not seeing the machine code values in the disassembly window, right-click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

1 Describe what is meant by a literal string.
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")

The string "You have reached the goal!\n\n"

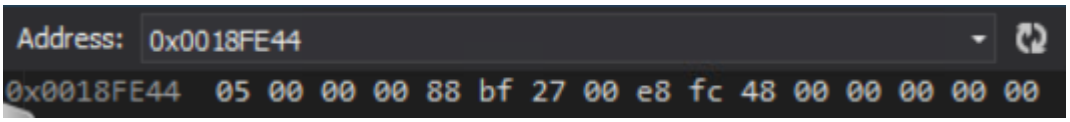
2 Single-step once to execute the push instruction. Get the address of the string from the stack. Type that address into a memory window. What is the address? Cut/paste the data in the memory window here for your answer.

Address of String: 0048FCE8



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.

ESP

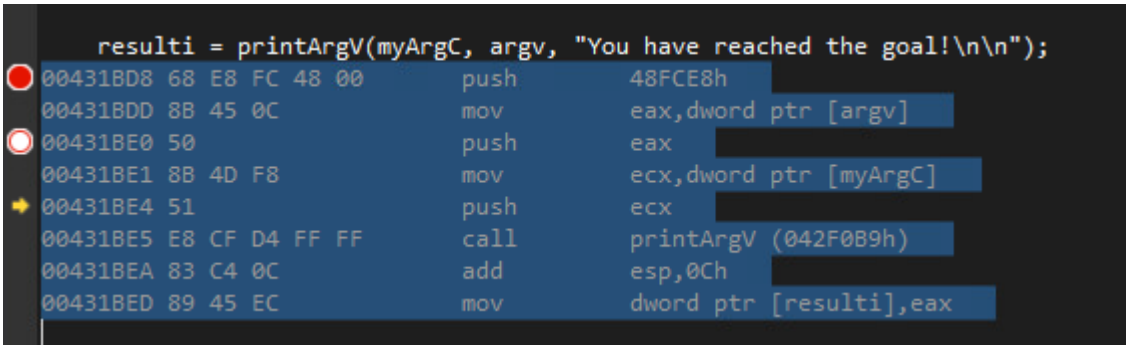


3.a What is the value of 5 representing?

It represents the number of inputs

3.b What is the return address?

return address is 00431BEA



as you can see the return address is the address directly after the call

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.

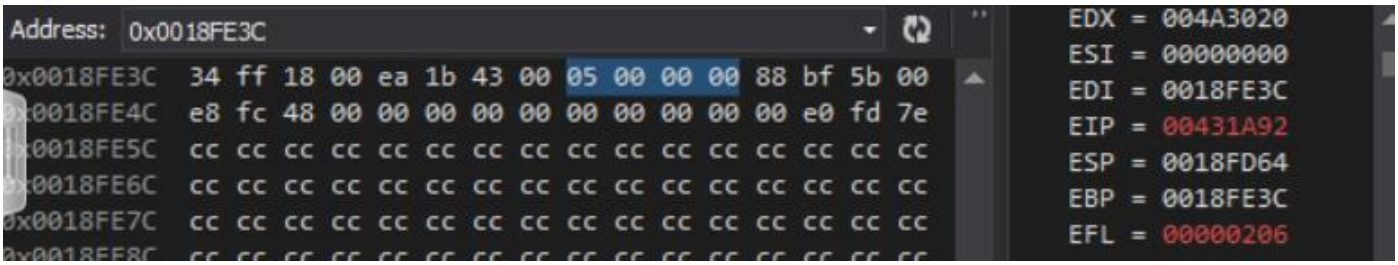
Jump Not Equal means it will jump when the zero flag is not equal to one (i.e. Z = 0). In human terms, the jne simply means that the jump will be taken when the two things being compared are not equal to each other. If they are, then zero is the result, and Z= 1. Look at the CMP (compare) instruction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

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

It is stored at ebp+0Ch

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

+04 +8 + 12 +16
Return address | myArgC | argv | address of string literal



00 05 00 00 is at ebp + 8

4.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?

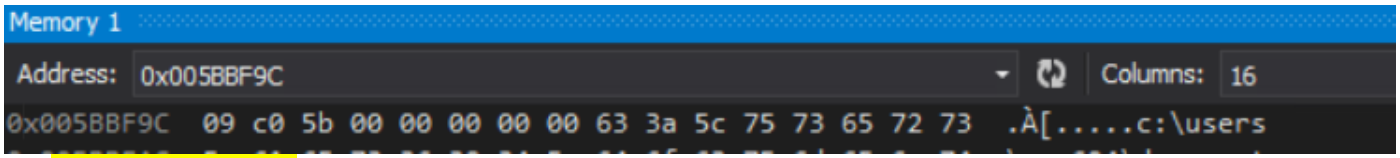
ebp = 0018FE3C
Argc is at [ebp+8], or, 0018FE44

4.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).

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



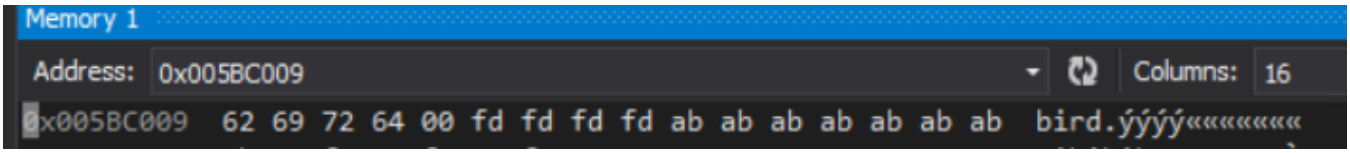
EDX = 005BBF9C

the address of the word bird

5.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:

005BC009

the word "bird"

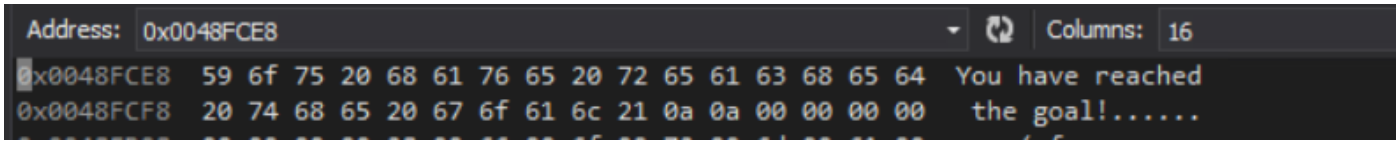


6 Now go ahead and press the continue button (green arrow) and look at the contents of the output window:

```
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.

6.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]"

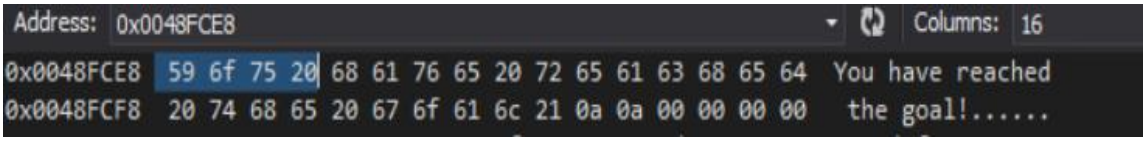
in the disassembly window. However, it is actually this:
"0052D60F 8B 45 10 mov eax,dword ptr [ebp + 0x10]."

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.

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

A part of the string in msg , specifically “You “ 59 6F 75 20

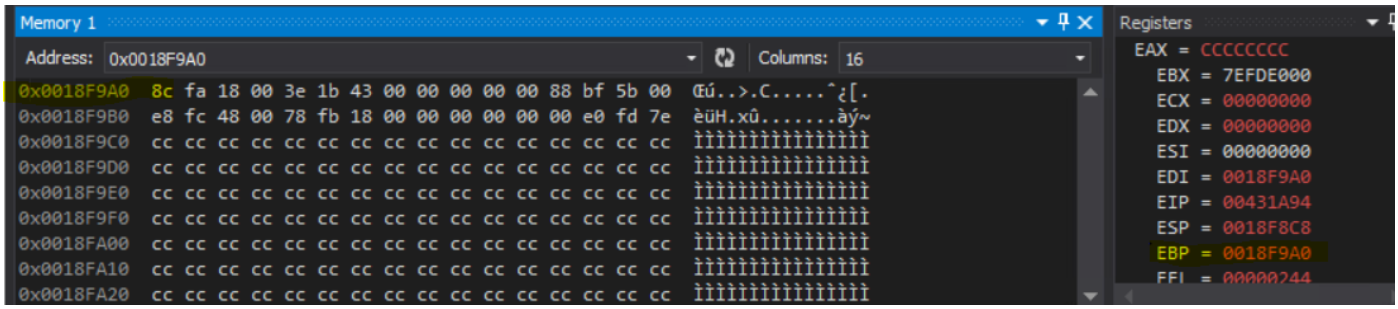


6.c If we did this: lea edx, dword ptr [msg] what value would be in edx?

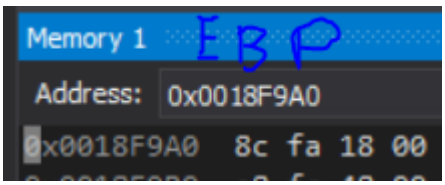
0x004FCE8

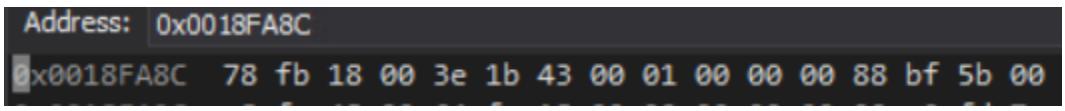
7 We are going to walk the stack. Each stack frame has 0x44 bytes for local variables, 4 bytes for the prior ebp, 4 bytes for the return address, 12 bytes for the arguments, and 12 bytes for saving registers, for 0x64 bytes total. Put ebp in the memory window.

7.a Show 16 bytes of the memory window.



7.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.

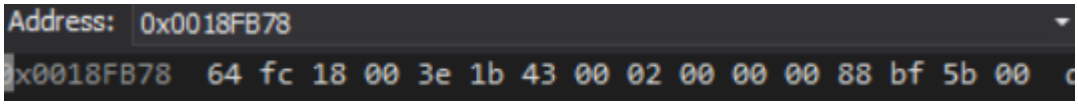




7.c At what address is the first argument on this stack frame and what does it equal?

Its equal to 1 and it is at 0x0018FA894

7.d Show the next stack frame here.



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

0018FC64

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]
```

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

Clearing out the data in the address of x , setting it to 0 for the for loop

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

Storing the value of eax in address of x incrementing eax by 1
then updating value in address of x
in other words x++

8.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.)

0018FC64

8.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?

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

Printing out the messages

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

argc

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

Because of the possibility of a negative while subtracting

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.

9.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

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

Because you are using byte pointers

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

byte located at [edx+eax]