

This lab covers loops and string instructions. You will be stepping through some code using some string instructions and writing code to do a loop.

First, examine the code in Lab7.cpp. at the top of the file, you will find “`#define STRLEN_TEST`” which is choosing the code that uses the C strlen library function. There are two global character arrays defined, n and q. The two prototypes for the solution functions are included – you will get those files later. FUNC1_Student and FUNC2_Student are ready for your modification. In this lab, you will write code for FUNC1_Student. In Lab #8, you will write code for FUNC2_Student.

Next is FUNC0 which takes a function pointer as an argument. The program as posted, uses the address of the C library function “strlen” here. You are going to write assembly code functionally equivalent to strlen and put that in FUNC1_Student.

Set a BreakPoint (BP) on the “return 0” at the end of main and on the “return len” at the end of FUNC0.

Run the program inside the debugger and view the output window. You should see the following: “FUNC: Length = 14.” Then click the green arrow and continue to the end. Your output window should now have this, “NewString: Exams are fun! But this lab sux!” displayed after the first statement.

This program calls FUNC0 with the address of a function that gets the length of the global string “n.” Then it runs the assembly code which concatenates “n” with string “q.” Continue until the program ends.

Add a BP to this statement in main “`phraseLength = FUNC0((DWORD *) strlen);`”, run the program, and view the disassembly. Note: If you are not seeing the machine code values in the disassembly window, right-click and select “Show Code Bytes.”

We are passing the address of strlen into FUNC0 and typecasting it as a DWORD pointer. The reason for this is simple. If we passed an actual function pointer, we’d have to define FUNC0 as taking a pointer to a function with a specific prototype and the prototype for strlen is different than the prototype for FUNC1_Student and FUNC2_Student. This way, we can pass any function address.

1. Single-step into the call to FUNC0, continue past the extra jmp, and get to the push ebp in FUNC0. Then step until you get to “push ebx” and stop there. In a memory window, type in ebp. Note, some values may vary so use my stack below to answer the question. EBP = 0x18FED4.

```
0x0018FED4  34 ff 18 00 d3 d6 42 00 f9 b6 42 00 00 00 00 00  4ÿ..ÓÖB.ùŒB....
0x0018FEE4  00 00 00 00 00 e0 fd 7e e0 21 43 00 d9 21 43 00  ....àÿ~à!C.Û!C.
```

- a. Looking at my stack, what is the value of the prior EBP.

0x0018FF34

- b. What is the return address?

0x0042D6D3

- c. What is the address of strlen?

0x0042B6F9

- Step to the “0042D680 8D 4D FC lea ecx,[len]” instruction and note the value in ECX before executing it. Now, execute the instruction. What is the hex value in ECX? The description would be the address of the local variable “len” which is at displacement -4 in EBP. **0x0018FE34**
- Single step to the call instruction “0042D68B FF 55 08 call dword ptr [funcAddress].” It is actually a call [ebp+8] which means it will add 8 to EBP, use that address to retrieve the address to call. Before taking the call, what is the destination address? **0x0042F73A**

Address: 
EBP+8 = 0x0018FE48 3a f7 42 00 00 00 00 00 00 00 00 00 e0 fd 7e

- Since this is a call to strlen, no need to step into it. Step over it to the “mov dword ptr [len], eax” instruction. What is this instruction doing? NOTE: We only do this for strlen, your function will populate len directly.
This is creating size DWORD PTR [LEN] and moving it into EAX for it to be populated.

- What is the length of the string?

0x0000000E = 14

- Type “&len” in a memory window. The current value is zero since it was initialized. Step to the next instruction and see the red **0e** in the memory window as eax was written. What is the offset of the local variable “len” from EBP? **EBP - 12 = 0e 00 00 00**
- The next few instructions call printf, step over all that until you get to “mov eax,dword ptr [len].” Why are we doing this?

Step over the next call to printf and all the way until you are out and back in main. Should be on an “add esp,4” instruction. Single step up to the “push 400h” instruction. Since you have the source code, you should be able to easily determine what this part of the code does. On the next exam, you could be required to write the C code for something like this.

```

0042D6D9 68 00 04 00 00    push    400h
0042D6DE E8 1B E0 FF FF    call   @ILT+1785(_malloc) (42B6FEh)
0042D6E3 83 C4 04          add     esp,4
0042D6E6 89 45 FC          mov     dword ptr [ptrNewString],eax
0042D6E9 83 7D FC 00       cmp     dword ptr [ptrNewString],0
0042D6ED 75 14            jne     main+43h (42D703h)

0042D6EF 68 98 2C 48 00    push    offset string "Error - Could not allocate 1024 "
0042D6F4 E8 19 EA FF FF    call   @ILT+4365(_printf) (42C112h)
0042D6F9 83 C4 04          add     esp,4
0042D6FC 6A FF            push    0FFFFFFFFh
0042D6FE E8 31 EB FF FF    call   @ILT+4655(_exit) (42C234h)

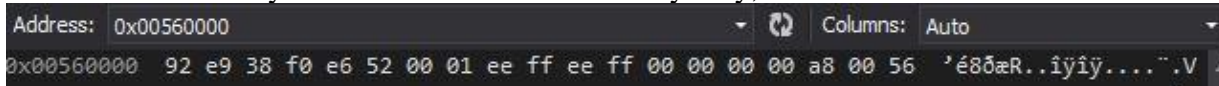
```

- Step over all of the instructions until you get past the call to _exit. Stop on instruction “mov edi,dword ptr [ptrNewString]” EDI gets the address of the newly allocated memory – this is heap memory – which is the *destination* for the string “n.” ESI gets the *source* address. Step to the “loads” (load string) instruction. What are the values of EDI, ESI, and ah? **EDI = 0x0056FE68 , ESI = 0X0042F73A , AH = 0x00**
- You should be familiar with the string instructions from the narrations and slides. If you skipped all that you are going to have trouble with this – I recommend that you watch the narrations and review the slides before you continue. After executing lods/stos what value do you expect to be in EDI/ESI/ah/al? For each of those

registers, explain why the value of each is what it is. **EDI = 0x0056FE68, and ESI = 004A3000 will be incremented by 1 as it loops. And AH will be equal to DH, AL will remain the same.**

10. “**Test al,al**” will not set the zero flag at this point, so **jne** will go to REPEAT_OP. How many times will this loop and why is it that specific number? **This will loop 14 times as that is the length of string n.**

11. Set a BP on the “**test ah,ah**” instruction. Hit continue. In a memory window, get the contents of the allocated memory and show it here. Address may vary, but data should match solutions.



```
Address: 0x00560000 Columns: Auto
0x00560000 92 e9 38 f0 e6 52 00 01 ee ff ee ff 00 00 00 00 a8 00 56 'é8ðæR..îÿÿ....'.V
```

12. Based on the value of AH, will the **jne** be taken?

Yes as the value of AH is 0x00

13. Single step to the “**jmp REPEAT_OP**” instruction. We **inc AH** to set a flag so this will not be repeated. We load the effective address of “q” into ESI. We are going to concatenate “n” and “q.” Next we “**dec EDI**” – why is that? Hint: look at EDI -2 in a memory window. **This preserves the CF state and allows EDI to be updated without messing with the CF.**

14. Continue again to the “**test ah,ah**” instruction. Type “ptrNewString” in a memory window. How long is the new string in bytes? Once you know, continue and exit the program.

The new string is 32 bytes.

15. Now, to complete the lab, comment out “**#define STRLEN_TEST**” and uncomment the source “**phraseLength = FUNC0((DWORD *) FUNC1_Student);**” in main. You need to write an assembly routine to do a **strlen** EXCEPT the result will be stored in the 2nd parameter rather than the return value. For lab #7 do not use string instructions, that will be lab #8.

If your code is correct, your output will look exactly the same as the one when passing **strlen**.