HW5: Pointers

Assigned: 2 July 2018 **Due:** 10 July 2018

1. Print pointer values:

Copy the file explore.c from my hw5 directory to your hw5 directory. Edit the explore.c file to add printf statements to print both variable values and values of the pointers to variables and/or array[0] where indicated. Note that if you print a pointer value you should use %p as the format (see K&R P.154 for the description of this format). For all other values, you should use %x format. Remember to build your program using the –m32 option for a 32-bit application.

You should find that as automatic variables are allocated memory or as arguments are pushed onto the stack to be passed to a function (foobar) that the addresses of the locations in memory are decreasing.

In the main program printf loop for array[i], it should be sufficient to print for i = 0 to i < 6. Can you find the locations that have been assigned for the automatic variables and can you see their values?

In the foobar function printf loop for array[i], it should be sufficient to print for i = 0 to i < about 40 or 50. A sample output is provided in hw directory. Can you find the locations that have been assigned for the automatic variable and the argument list variables? Can you see the same data that you printed above in the main program further back on the stack?

Write a memo.txt file to document your findings.

2. Do K&R Exercise 5-13:

The program takes lines from standard input and keeps the last n of them in memory as it goes through standard input. When it gets to an EOF, it prints the last n lines out. You may assume n is less than 2000 and each individual line is not longer than 1000 including the newline and the null terminator.

Use two source files plus a header file for this, just to show you know how to make multiple source files work together.

tail.c:	interprets the command line argument. Calls init_lineholder(int nlines) with the number from the command line. Does a loop calling getline and insert_line(char *line). When getline returns 0 (indicating EOF on stdin), it calls print_lines().
lineholder.c:	contains a static array of pointers for lines. Implements init_lineholder, insert_line, and print_lines. Init_lineholder initializes the "first" slot and related variables. Insert_line adds a line to the array. It must allocate memory for the new line. It must free the memory for a line no longer needed, if any. Print_lines prints the lines in the array and frees the memory used for them.

lineholder.h: just has prototypes for the three calls with appropriate comments explaining what they do for the caller. (Any comments on *how* they do it belong in lineholder.c.)

Write a makefile that compiles tail.c and lineholder.c, with the appropriate dependencies, and builds the executable "tail". (tail should be the default target.) Also provide "make clean" to remove the intermediate object files. Also include the option to build the executable using the –m32 switch for a 32-bit application.

As indicated in the book, the challenge here is to hold the lines in memory efficiently. Only the last n lines should be held in memory, not all the lines. You should set up an array of char * pointers as on page 108. Instead of alloc, use malloc (see p. 143 for an example of the use of malloc.). When you are done with a line, use free(pointer) to release the memory previously allocated with malloc. You only need one buffer of length 1000. All the mallocs should be of just the right length to hold the actual line.

Be sure you explain your method of adding the n+1st line and freeing lines no longer needed in a good header comment at the beginning of lineholder.c. You have a choice of methods for adding the n+1'st line to the array of n lines. Here are two ideas:

Ripple the pointers down one slot in the array and put the new one at slot n-1. First you need to free the one at slot 0. Similarly treat each succeeding line.

Maintain a moving "first" slot variable, that stays at 0 for the first n line additions, now moves to slot 1. Free the old line at slot 0 and put the new one there. When another line comes in, release the one at slot 1 and put the new one there, and make "first" be 2. When you print them all out, wrap around from slot n-1 to slot 0 and back to slot first-1.

You can generate your own test1.in, test2.in, etc. files, or to use test files provided tail0.in, tail1.in etc., to test all special cases that your tail program might encounter and generate test1.out, test2.out, etc. files. Use n = 10 (your program should save the last 10 lines of the input). Using "script typescript2", demonstrate "./tail" on your test files.

NOTE: Since "tail" is a UNIX command, doing "tail" by itself will use the UNIX command instead of your program, but "./tail" unambiguously specifies your own tail program. You can determine which program will actually execute by using the UNIX command "which": try "which tail" and "which ./tail".

3. Deliverables:

1. Typescript file showing cat and gcc of explore.c, your experiments with it, and a memo.txt file explaining what you think you have learned and how.

2. Files to deliver:
tail.c, lineholder.c, lineholder.h, makefile, testn.in files, testn.out files
Typescript

3. Print out your typescript.