CSC172 LAB

POINTERS TO POINTERS

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in his own work, but every student must list the name of the lab partner (if any) on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Pointers to pointers

The goal of this lab it to gain familiarity with pointers in the 'C' programming language.

1. Begin this lab by implementing a simple C program with a “main” function. Use the prototype for main that is used with command line arguments below. Include a “for” loop that prints out the command line arguments, one argument per line. You may refer to the code in Kernighan and Ritchie if you need help.
2. In past labs, you have become familiar with the equivalence of the array syntax `a[i]` and pointer syntax `*(a + i)` for single dimensional arrays. Begin this lab by declaring a two dimensional array of integers in C. Write nested looping code to fill in the array with a simple multiplication table and then print it out using the familiar array syntax `a[i][j].`

3. Declare an array of integer pointers `int *pi[size];` Write a loop that demonstrates the equivalence of the declaration by setting each pointer in your array of pointers to arrays in your two dimensional array `pi[i] = a[i] ;` Next, write a double loop that prints out the multiplication table using the mixed pointer and array syntax `*(pi[i] + j ).`
4. Declare a a pointer to integer pointers `int **ppi ; .` Demonstrate the equivalence of the syntax by setting the [pointer to the array of pointers] to [a pointer to integer pointers]: `ppi = pi; .` Write yet another double loop that prints the two dimensional array using pure pointer syntax. a pointer to integer pointers `int *(*ppi + k);.` Remember to update the pointer to pointers in the outer loop `ppi++ ;` It might help to draw a diagram of how the pointers are referring to actual integer locations.
5. What about the case where we want allocate memory dynamically (as we need it) rather than declare it beforehand in the code? In JAVA we have the familiar "`new`" operator for dynamic allocation. In C we have to "roll our own" by asking the operating system for more memory. We need a "memory allocation" function. In C, this is known as the "`malloc`" (memory allocate) function, and is found in the standard libray. (There is also "c`alloc`", with slightly different syntax, for allocating zeroed-out memory.) You will need to include `<stdlib.h>` in order to use the malloc function. Consider the following syntax and take a few moments to digest it:

   ```
   int size = 5;

   int **ppi2 = (int **) malloc(size * sizeof(int *));
   ```

   So, what is going on here? We are allocating some memory with a call to malloc. Malloc thinks in terms of bytes, so we have to send it the product of the number of things we want times the size of those things. Malloc returns a void pointer (no type) to a contiguous range of bytes, so it is good practice to cast it into the pointer type we want. Once you understand this declaration, put it in your code. Write a sentence which describes this statement. Again, drawing a diagram may help.

   Recall how allocate a ragged or triangular array in JAVA. We use a similar looping technique to build a two dimensional array in C using mallocs.

   ```
   for (i = 0 ; i < size; i++)
     *(ppi2 + i) = (int *) malloc (size * sizeof(int));
   ```

   Compare the statement above which allocates groups of integers and returns a pointer to an array of integers with the previous statement, which allocated a set of pointers to integers and returned a pointer to an array of pointers. Once you understand the distinction between these two statements, you are a long way on the road to understanding memory management through pointers in the C language. Put these statements into your code once you have a full understanding of them. Ask your lab partner to explain what is going on in simple English and you do the same for your partner. You should each be able to articulate, in your own words, the mechanism of this allocation of memory. Again, a simple drawing might help.
6. Finish up this lab by using a double loop to set the integer contents of the array you created using pointer syntax `*(*(ppi2 + i) + k).` Write a second loop to print out the values.

3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at

my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped”) file that contains the following.

1. A README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.).
2. The source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4 Grading

172/grading.html

Each section (1-6) accounts for 15% of the lab grade (total 90%)

(README file counts for 10%)