EE 1305, Fall 2015.

Lab 07: Pointers and dynamic memory allocation

You must work with at least one partner on this lab!

Preparation:

Read Chapter 10, "Pointers and Dynamic Memory Allocation". You may skip the material on classes (p. 427-428 and section 3).

Topics you should understand by the end of this lab:

pointers
& operator
* operator

aiming a pointer at a single value/variable
aiming a pointer at an array
    iterating through an array with a pointer

Passing pointers to functions

dynamically allocating individual memory spaces (Like a single variable)
dynamically allocating a block of memory spaces (like an array)

The following exercises are more or less the equivalent of practicing scales on the piano. They are not at all interesting, but if you work through them and think about how they are working, you should have a good understanding of pointers and how to use them when you are done. This will make your future C++ work easier and, I hope, more interesting.

I encourage you to work with a partner on this lab; it should help considerably to have two heads to scratch when things don't make sense. As for all the labs, two can work together, but you must both submit the programs you write. More importantly, you must both **understand** what you turn in. So if your partner tries to whiz through before you're sure you've understood what you just did, don't let her (or him) get ahead of you...

Part I: basics of Pointers

1. Compile and run this program. Working together with your lab partner, determine why it gives the results it does, and answer the questions posed in the output. Make a text file with answers to all the questions, and submit that as part of your lab work.

Name your* file pointers_questions.txt *(*or pointers_questions.docx*)

Here's the basic program for next few parts of this lab:

```
#include <iostream>
using namespace std;

int main()
{
```

```
    int i = 5, j=51, k=62;
    int data[5] = {10, 20, 30, 40, 50};
    char my_cstring[8] = "the fox";

    int *p = NULL;
    char *pc = NULL;

// do something useful with the arrays and the pointers
//    . . .

// and put some clean-up code to delete any
// dynamically allocated variables

    cout << "done"<<endl;
    #ifdef WIN32
    system("pause");
    #endif
    return 0;
}
```

A side note on: `#ifdef ... #endif`:
These lines create a "conditional compile". Visual Studio defines a precompiler constant called WIN32, while other compilers, like the Gnu C++ compiler, do not. Thus, the line `system("pause");` is compiled in Visual Studio, but left out of a gnu compile. This line lets you run with the debugger in VS and pause your program before exiting, so the window doesn't disappear before you can read the output.

Using a pointer to access data in memory:

2. Starting from the basic program:
   a. assign p the address of i, and output the value where it points.
   b. move it to point to j and k and output those values.
   c. point it to the first element of the array data and output it
   d. point it to the 3rd element of data and output it.
   e. using your pointer variable, change the values of the i and the 1st and 3rd elements of data.

Now switch partners (if you've got one).

3. Starting from the previous (part 2) program:
   a. initialize pc to the beginning of my_cstring.
   b. use it to capitalize the 1st and 5th characters of my_cstring. (hint: toupper(c) returns the upper-case equivalent of c)
   c. NOTE:
      a. Either of the following will point to the 1st char:
         i. pc = my_cstring; // using name of array
         ii. pc = &my_cstring[0]; // or taking address of 1st element
      b. You could use either of the following to point to the 5th element:
         i. pc = my_cstring+4; // 4 elements over from beginning of array
         ii. pc = &my_cstring[4]; // or taking address of 5th element.

   d. (you'll obviously want to output those values after you change them...)
   e. use your char pointer to output the integer value of each element of my_cstring (including the 8th element, my_cstring[7]). (hint: you could copy each element to an integer and output it, or use static_cast<int>(...)).
   f. NOTE:
      a. The following might be helpful. (hmmm. why does "cout << *pc" print more than 1 char?):
         i. pc = my_cstring;
            cout <<*pc << " "<< pc[0] << " " <<setw(3)<<static_cast<int>(*pc)<<endl;

Name your program **basic_pointers.cpp**

Part II. Pointers and Arrays

4. Pointers and arrays: Now, starting from the (original) basic program
   a. create a pointer, initialize it to the beginning of the array data
   b. output the elements of data by **indexing the array data[ ]** using a loop (and the indexing operator []).
   c. output the elements of data by **indexing your pointer** using a loop, **with the pointer pointed at the beginning of the array and using [ ]**
   d. output the elements of data by **incrementing the pointer** each time through the loop. your loop control could look like this (why does this work?):

```
for (p=data; p < data+5; p++)
```

   e. reset your pointer to point at the beginning of the array again, then output the elements of data by adding i, a loop counter, to the pointer and dereferencing the summed address. Use something like:

```
cout << *(p+i)
```

   (Be sure to point back at the beginning of the array before the loop!)

Q: how is **`cout << *(p+i)`** different from **`cout << *p+i`** ? If you're not sure, try both and see what happens. To verify your answer, look at the operator precedence table in the textbook and find the precedence of * (for dereferencing) and +. Which is higher precedence?

Name your program **pointers_arrays.cpp**

Part III: Dynamic memory allocation, and passing pointers to functions

5. Dynamic memory allocation:

C++ let's you request memory from the OS using the **new** operator, and use it like individual variables or arrays. Using it for single variables is more important for object-oriented programming and data structures (courses such as CS 1412 and CS 2413), but we'll do it here with ints and floats. Using it for dynamically allocated arrays is VERY useful, however, so be sure you understand this part of the lab when you're done!

Here is a sample program which allocates a double and uses it, then allocates a block of 10 doubles and uses them as an array. It also demonstrates how to pass pointers by value and by reference to functions. Compile and run it, discuss it with your partner, and make sure you understand how it works.

Write a program which:
   a. dynamically allocates an array of 5 integers
   b. uses a loop to set the elements to 5, 10, 15, 20 and 25
   c. uses **a second loop** to output the elements of the array.
      a. be sure you use a second loop for this!
   d. deletes the array.

change partners and continue:
   e. Move the array-printing to a function print_array
   f. Write a function create_array with the following prototype:

```
void create_array(int * &p, int size, int initval);
```

g. this function should dynamically allocate an array and set all elements to initval. ***Note: you'll need to use a loop to set all the values.***
h. use both these functions in your program. Have your program a dynamically allocated array of 10 elements and initialize it to all zeros (and print it out), then have it create another dynamically allocated array of 20 elements, and initialize it to all 999's (and then print it).

Name your program **dynamic_arrays.cpp**

6. Here are 4 files: numbers1.txt , numbers2.txt and numbers3.txt and numbers4.txt . The first two contains a "header" line telling you how many data items follow. (The first one is very small, so you can easily verify you are getting the right results; the second is larger, too large to verify by hand.) The second pair are like the first two, except they are just a list of numbers (no "header", i.e., no count at the top).

a. Write a program that can calculate the average, min and max of the numbers in the first 2 files. It should read the first value (# of points), allocate an array the correct size, read the data into that array, then analyze the array and print the count, average, min & max of the array.
    a. Use a function to read the data, a second function to find the average, and a third function to find the min & max. Read the count and allocate the array itself in the read_data(...) routine -- your read_data(...) routine will have to pass back both the count and the pointer after it allocates the array.
    b. I recommend that you print out the first few elements and the last few elements of the data file when you read them in to make sure you get all the values. Some print statements in the various routines would also be helpful while writing/testing to make sure things are working properly.
    c. Make sure you delete the dynamic array properly and set your pointer back to NULL.
b. OPTIONAL: Modify your program to work with an array of filenames, and read and print the statistics for each file in the list. You can use an array of strings for the filenames:
```
string filenames[] = {"numbers1.txt", "numbers2.txt"};
```

c. OPTIONAL: Write a modified version of the program from part a. that can work with the files numbers3.txt and numbers4.txt . The only difference is that the file doesn't tell you the size upfront. Your program will have to read the file twice – once to count how many values there are, and then (after allocating an array), read in the data and get the min, max and average.
    a. Use a function count_values(filename) that reads the file and counts the number of values.
    b. Note: if you're not careful, you can end up reading and counting the last element twice! Make sure you don't do that...remember, cout is your friend here...
    c. You can use an array of strings for the filenames:
```
string filenames[] = {"numbers3.txt", "numbers4.txt"};
```

Name your programs read_nums.cpp (part b) and read_nums_nh.cpp (part c)

7. Complete this program, which you will recognize as the PGM negation program from earlier labs, using functions and dynamic memory allocation. Be sure to delete [ ] the array before exiting.
Question: How is it that the function negatePGM(...) can change the actual pixel array, even though the pointer pix is passed by value?

To test your program, I recommend testing initially with a small PGM, such as your initials.pgm from the beginning of the semester. When you have verified that it is working, try it on some of the images from the image archive, available on the course materials web page.

Name your program **pgm_negate.cpp**

Don't take on the challenge this semester! ian. Work on the project instead.

Challenge

Write a routine to rotate an image by 90 degrees. Copy your part III program, and replace the call to negatePGM with a call to your rotate90PGM(...) function. Note that the #cols & #rows will change, although the actual pixel array will not. So, you'll need to pass #cols & #rows by reference, so the function can change them. Rotating in place is actually rather complex, so it is simpler to allocate a temporary array, copy the pixels to the temporary array, then move them back to their new locations.

Super Challenge

Modify your rotate90PGM function to rotatePGM, so it can rotate by 90, 180 or 270 degrees. You'll obviously need to pass in the degrees to rotate by, and have it do something sensible if the degrees passed is not evenly divisible by 90.

UberChallenge

Modify your rotatePGM to rotate the image to any arbitrary angle. A few helpful hints. **First**, you'll either need to write the results to a new, larger PGM image buffer, or else drop any pixels that are rotated to outside the original image size. **Second**, regardless of whether you expand the image or keep the same size, whenever the image is rotated by anything other than a multiple of 90 degrees, you'll have some pixels at the corners of the output image which do not come from any pixel in the input image; Set these either to black, white, or medium gray. **Third**, output pixels will generally come from non-integer pixel locations, and you'll have to decide how you want to handle that. An arbitrary-rotator will typically do a weighted average of the nearest pixels to create the output pixels, but you could also simply use either the nearest pixel or some other choice.

What to submit

Submit your text file answering the questions in the first sample program, and the other questions asked in this lab.

Submit the final part 1 program (from 3.).	**basic_pointers.cpp**
Submit the final part 2 program (from 4.e)	**pointers_arrays.cpp**
submit the program for parts 5.	**dynamic_arrays.cpp**
submit the programs for part 6.	**read_nums.cpp read_nums_nh.cpp**
submit the program for parts 7.	**pgm_negate.cpp**
If you did any of the challenges, submit them	**challenge_07.cpp, super_07.cpp, uber_07.cpp**

If you worked with a partner, be sure each of you submits your lab work to Blackboard, and be sure to note in the text file who you worked with on this lab.