cs3157 – Advanced Programming
Summer 2006, lab #4, 30 points
June 15, 2006

Follow these step-by-step instructions. This lab must be submitted electronically (see instructions at the end of the lab) by Wednesday June 21, 4 pm.

You need to include a README file with information about each file submitted and any other comments you want the TA's to see.

You are required to create a single Makefile to run each of the steps of the lab.
(Example, make step1
should compile and run step1 of the lab.) In addition you need to have a "make clean" step to clean up the object files, and executables generated by your make steps. Feel free to add comments to your makefiles

Step 1 – Magic square in C ! (10 points)

In this step, you will develop code that computes a magic square. A magic square is a 3 by 3 matrix filled with integers in the range [1 - 9] so that each entry in the matrix has a unique value. The sum of each of the rows and each of the columns and each of the two diagonals must be equal to 15. (Note that there is not a single solution to the magic square problem.) Remember that you are still working in c.

Create a file called **Maintest1.c/Magicsquare1.c/Magicsquare1.h** and (to help you get started) start by putting the following code/declarations in each file (where appropriate):

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef int MSQUARE_TYPE[3][3];

typedef MSQUARE_TYPE * MagSquare_PTR;

void initSquare( MagSquare_PTR );
void printSquare( MagSquare_PTR );

int main() {

MagSquare_PTR mptr= /* replace with something with malloc */;

srand( time( NULL ));  /* add comment for this */

initSquare( mptr );  /* will setup a random magic square */

printSquare( mptr ); /* will print out a square */

} // end of main()
```

Define the function **initSquare(`MagSquare_PTR`)** which takes the magic square pointer argument (as above) and fills it with values in the range [1 - 9]. These values must be assigned RANDOMLY, **but** make sure that no number appears more than once in the array.

Also Define the function **printSquare(`MagSquare_PTR`)** which takes the magic square pointer argument (as above) and prints out its contents to the screen in a nice 3 rows by 3 columns format.

You should test your code with just these 2 functions to make sure each is working (before adding the rest of the functions).

Hint: To get random numbers look up lookup rand() and srand() from the stdlib libary.

1. maintest1.c – this will contain the main function
2. magicsquare.h – this will contain the functions declarations that you will be using.
3. magicsquare.c – this will contain the function definitions from above including the following methods:
    a. void initSquare(`MagSquare_PTR`)
    (see above)
    b. void printSquare(`MagSquare_PTR`)
    (see above)
    c. int sumColumn(int column, `MagSquare_PTR`)
    The function takes the 2-dimensional magic square array and a column number and returns the sum of the 3 values in that column
    d. int sumRow(int row, `MagSquare_PTR`)
    see above.
    e. int sumDiagonal(int diagonal, `MagSquare_PTR`)
    The function takes the 2-dimensional magic square array and a diagonal number and returns the sum of the 3 values in that diagonal. Use diagonal = 0 to refer to the diagonal that runs from the northwest corner down to the southeast corner. Use diagonal = 1 to refer to the diagonal that runs from the northeast corner down to the southwest corner.
    f. Create a function called int isMagic(`MagSquare_PTR`) which takes the magic square pointer as an argument and checks each of the rows and columns and diagonals to see if all the sums equal 15. The function should return 0 if the argument is not a magic square and 1 if it is a magic square.
    g. Modify the above main() so that it calls isMagic() and prints out whether the square is a magic square or not.
    h. Modify the main() program so that after it prints out the square, it computes (using the functions above) and prints out the sum of each of the 3 rows and each of the 3 columns and each of the 2 diagonals.

Compile and test your program.
Here's a sample output:

```
the square is not a magic square
here is your square:
        2     8     3
        1     4     5
        7     9     6
sum of row 1 = 13
sum of row 2 = 10
sum of row 3 = 22
sum of column 1 = 10
sum of column 2 = 21
sum of column 3 = 14
sum of diagonal 0 = 12
sum of diagonal 1 = 14
```

Step 2 get the square (8 points)

Now copy maintest1.c to **maintest2.c** and magicsquare1.c to **magicsquare2.c**, and magicsquare1.h to **magicsquare1.h** modify them as follows:

1. Create a function called **permuteSquare(`MagSquare_PTR`)** which takes the magic square pointer as an argument and randomly switches two entries in the array. Do this by randomly picking two sets of row and column indices (in the range [0 - 2]) and then swapping the entries located at each pair of indices. Check to make sure you aren't picking the same spot .. that is swapping the same location with itself ☺
2. Modify the **main()** so that after it calls **isMagic()**, if the square is not magic, then it calls **permuteSquare()** to switch around two entries in the square and then test again to see if the square is magic. Do this repeatedly until a magic square is found. When a magic square is found, call **printSquare()** again to print out the magic square.
3. Have the program count the number of times it has to permute the square in order to find a magic square. Print that out too (Hint: count it where you call method permutesquare). Remember this can take a while.

Compile and test your code.

Step 3 CGI (4 points)

CGI!
Create a file called **maintest3.c** and **maintest3.html** which together allow your program to over the web. The html page should provide some information on what a magic square is, and allow the user to click on a button to get started. The program should run as in the previous step, and then output an html page with all the information (you should put the magic square values in an html TABLE tag (look it up if necessary). Also you need to time the running time of your program so output the time when the program was called, and how long it took to get the correct answer.

Remember that to run a cgi c program you need to perform the following steps:

1. code your c program
2. successfully compile the c program on the correct platform…..you will probably run it from your own desktop. For cygwin, it means compiling a exe file and having cygwin in your path to get it to run.
3. rename the executable to something.cgi and call it from a webpage locally (make sure its executable).
4. you can also test the cgi script separately, by running it on the command line.

Here is a sample c CGI program:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  printf( "Content-type: text/plain\n\n" );
  printf( "QUERY_STRING = [%s]\n",getenv( "QUERY_STRING" ));
} // end of main()
```

to get the time:

```
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main() {
      time_t t1,t2;
      (void)time(&t1);
      printf("current time is %s",ctime(t1));
```

<u>Shell Programming Section.</u>

In this lab, you'll use several UNIX tools which we discussed in class on Wednesday. The tools are:

- 'cut' — outputs sections of each line in input
- 'grep' — outputs lines matching a pattern
- 'sed' — stream editor
- 'sort' — outputs sorted concatenation of input
- 'wc' — outputs the number of bytes, words and/or lines in input
- 'ps' — 'outputs process id of all running processes

The man pages for each of these can be accessed by typing, for example **`man cut`**, at the UNIX prompt.

We mentioned (briefly), how send the output of these commands to a file as well as to stdout (which is the default). We also discussed how you can pipe the output of one command into the input of another.

Here's an review/overview in brief:
The greater-than sign > is used to redirect the *output* of a command from stdout to a file. This is like opening a file for writing in C. If the file exists, it will be overwritten. If the file does not exist, then it will be created. Here's an example:

```
grep "hello" myfile > hello.dat
```

This example will search for all lines containing the word "hello" in a file called "myfile", and it will output those lines (containing the word "hello") to the output file "hello.dat".
The double greater-than sign >> is used to ***<u>append</u>*** the redirected output to the end of an existing file, rather than creating a new file. If the file does **not** exist, then it will create it. This:

```
grep "hello" myfile >> hello.dat
```

will behave just like the example above, except that the output will be added to the end of the "hello.dat" output file.
The vertical bar | is used in between commands to pipe the output of the first command into the input of the second command. For example:

```
grep "hello" myfile | wc -l
```

will look for all the lines in "myfile" that contain the word "hello" and will output those lines, but instead of outputting them to the screen, the lines will be read as input by the wc command, which (since it is invoked with the -l option) will output the number of lines it receives.

Step 4 – Getting Statistics from a file (8 point)

Create a Bourne shell (/bin/sh) script file called **step4.sh**. The shell script should use the above UNIX commands to print out statistics from the file **tcm.dat**, answering the questions below.
Label each of the statistics output.
Hint: you can use the “echo” command in a shell script to print text to the screen, for example:

```
echo hello world
```

to view the contents of the file you can type either of the following commands:

```
more tcm.dat
less tcm.dat
```

1. How many movies are in the data file that are being shown in the evening ?
   By morning, I mean any movie whose starting time is listed as “PM”.
   It doesn’t have to be a unique list – if one movie is shown three different times during the month, you should then count that movie 3 times.
2. How many movies are in the data file all together ?
   Again, it doesn’t have to be a unique list – if one movie is shown three different times during the month, you should then count that movie 3 times.
3. How many times did John Wayne appear in a movie on TCM in February ?
   If the same movie is shown twice, then count it twice.
   You can safely assume that John Wayne is the only actor listed in the file whose name contains “Wayne”.
4. How many movies made in the 1940’s are in the data file ?
5. How many movies were not made in the 1930’s in the data file ?

Submit your lab.

See instructions online