ECS30 **Programming Assignment #3** **Spring 2015**

Due: 5-13-2015 (May 13,2015)

Filenames: pyramid.c, pumpkin.c, arithmetic.c, cipher.c, calc.c

Add your Name and UC Davis ID number as the first comment of each file to make it easier for TA's to identify your files.

All programs should compile with no errors. Warnings are okay, as long as execution is still correct. Compile programs using "gccx filename.c", then execute with "./a.out" or specify your executable filename during compilation using "gccx -o outputFile filename.c" and execute with "./outputFile".

Use *diff* with the *_output files to compare the output of your executable files with the official output files. (Syntax: "diff file1 file2") *diff* will compare two files line by line, and list the changes that need to be made to the first file to make it identical to the second. If there are no differences between the two files, then *diff* outputs nothing.

Submitting files

Usage: handin cs30 assignmentName files...

The assignmentNames are Proj1, Proj2, Proj3, Proj4, Proj5. **This is Proj3**. Do not hand in files to directories other than hw3, they will not be graded. The files are the names of the files you wish to submit. For this assignment, your files will be pyramid.c, pumpkin.c, arithmetic.c, cipher.c, calc.c. Therefore,

you would type from the proper directory:

>>handin cs30 Proj3 programName.c

Do NOT zip or compress the files in any way. Just submit your source code.

If you resubmit a file with the same name as one you had previously submitted, the new file will overwrite the old file. This is handy when you suddenly discover a bug after you have used handin. You have up to 4 late days to turn in files, with a 10% deduction for each late day. The deadline for each project (and late day) is 11:59pm.

TA comments:

If it makes sense to put code into a function, use function definition. Otherwise, you can write your code in main, you do not HAVE to use functions for each of these problems. Just to be clear, this does not mean hard code the print statements into your main function. It just meant that if it makes sense to move part of your code into a separate function, do so, otherwise, it's okay to keep all the code in main.

Some of you have been enabling c99 in your compilers so you can do things like:

```
for (int i=0; i< NUM; i++)
```

instead of:

```
int i;
```

for (i=0;i<NUM;i++)

Do NOT do this. For consistency we will not be enabling c99 during your grading scripts, so this will result in an error and you WILL LOSE POINTS. Please just write the extra line of code.

OUTPUT FILES:

For this program we are giving .out files for the non-graphics programs. These are EXECUTABLE FILES, NOT TEXT files. You will need to pipe the outputs of these files to an output text file as well as the outputs of your programs into a separate output text file for diff comparison.
For example:

Assume the official provided file is **abc.out**, and your .c file generates **a.out.**

To compare the outputs of the two .out files, you need to:

a.out > diff.txt
abc.out > refdiff.txt
diff diff.txt refdiff.txt

For the problem with required input file, do the following:

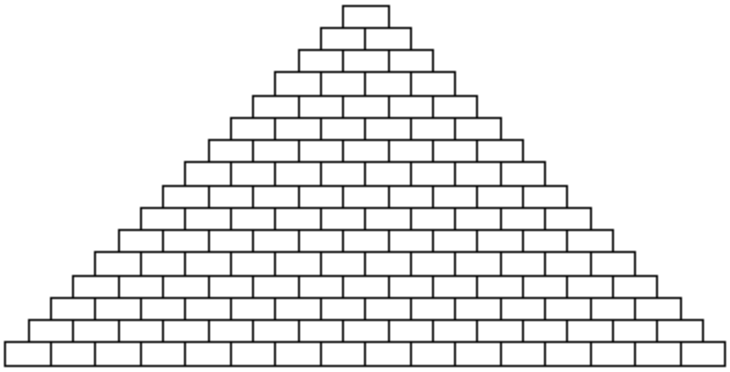
a.out < test1_input > diff.txt
abc.out < test1_input > refdiff.txt
diff diff.txt refdiff.txt

FOR THE GRAPHICS PROGRAMS: Your results will not be graded so there are no .out or output files. Please use the #define variables listed below.

Programming project #3: Ch. 7: 2, 7; Ch. 8: 8; Ch. 9: 8; Ch. 10: 10
Programs reproduced below for your convenience: (from the Programming Exercises section of each chapter)

CH.7: (pg. 252-258)

2. filename: pyramid.c **(This graphics program is OPTIONAL and will NOT be graded)**
Write a program that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as your move up the pyramid, as shown in the following diagram:

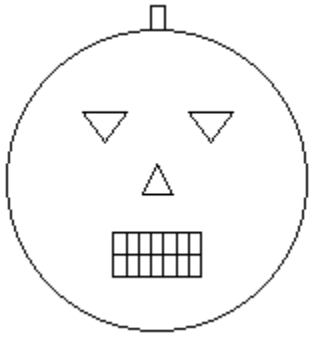


Your implementation should use the constant *NBricksInBase* to specify the number of bricks in the bottom row and the constants *BrickWidth* and *BrickHeight* to specify the dimensions of each brick.

TA comment:
Use these constants:
#define NBricksInBase 16
#define BrickWidth 0.3
#define BrickHeight 0.15

7. filename: pumpkin.c **(This graphics program is OPTIONAL and will NOT be graded.)**

Write a program that draws a picture of the Halloween pumpkin shown in the following diagram:



As in the *house.c* program shown in Figure 7-7, your picture should be controlled by several constants:

```
#define   HeadRadius      1.0
#define   StemWidth       0.1
#define   StemHeight      0.15
#define   EyeWidth        0.3
#define   EyeHeight       0.2
#define   NoseWidth       0.2
#define   NoseHeight      0.2
#define   NTeethPerRow    7
#define   ToothWidth      0.083333
```

```
#define    ToothHeight        0.15
```

These values are the ones used to produce the pumpkin shown in the diagram, and you should be able to figure out what each constant means by looking at the picture. Your program must be written so that changing any of these constants changes the picture in the appropriate way. For example, if you change NTeethPerRow to 4, the new diagram should have only four teeth in each row, but the mouth should still be centered horizontally. The two eyes and the mouth of the pumpkin face should be drawn halfway from the center to the edge of the circle in the appropriate direction, so that changing *HeadRadius* also changes the positions at which these features are drawn. The center of the circle representing the pumpkin should appear at the center of the screen.

TA comment:

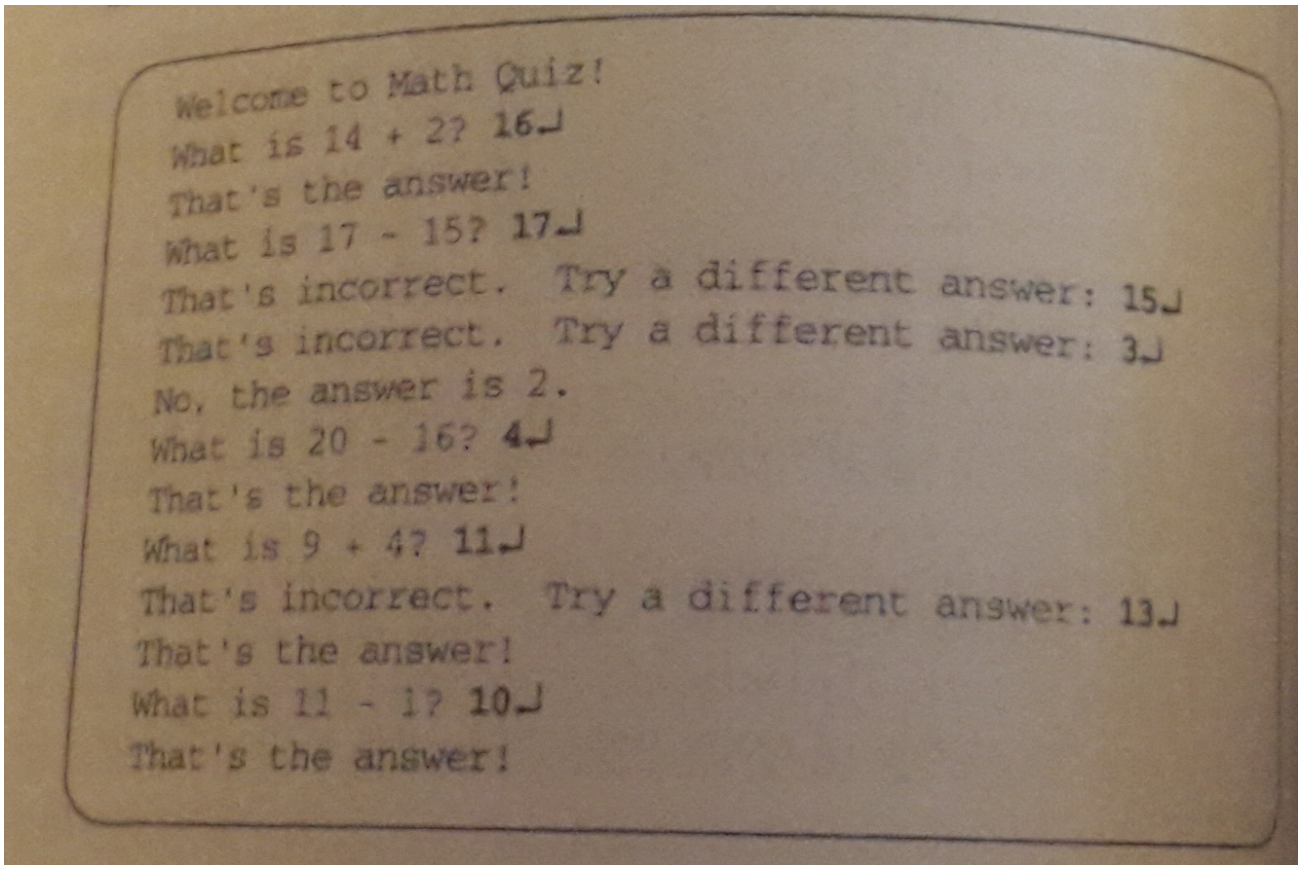
```
#define    HeadRadius         1.0
#define    StemWidth          0.1
#define    StemHeight         0.15
#define    EyeWidth           0.3
#define    EyeHeight          0.2
#define    NoseWidth          0.2
#define    NoseHeight         0.2
#define    NTeethPerRow       7
#define    ToothWidth         0.083333
#define    ToothHeight        0.15
```

CH.8: (pg. 292-300)

8. filename: arithmetic.c

arithmetic.out is the output file, please do not diff the results of the output file, it will NOT be the same. You may use the output file to check for formatting consistencies. THERE are NO OUTPUT or INPUT file provided by TAs.

As computers become more common in schools, it is important to find way to use the machines to aid in the teaching process. This need has led to the development of an educational software industry that has produced many programs that help teach concepts to children.
As an example of an educational application, write a program that poses a series of simple arithmetic problems for a student to answer, as illustrated by the following sample run:



Your program should meet these requirements:

- It should ask a series of five questions. As with any such limit, the number of questions should be coded as a *#define* constant so that it can easily be changed.
- Each question should consist of a single addition or subtraction problem involving just two numbers, such as —"What is 2 + 3?" or —"What is 11 – 7?". The type of problem—addition or subtraction—should be chosen randomly for each question.
- To make sure the problems are appropriate for students in the first or second grade, none of the numbers involved, including the answer, should be less than 1 or greater than 20. This restriction means that your program should never ask questions like —"What is 11 + 13?" or —"What is 4 – 7?" because the answers are outside the legal range. Within these constraints, your program should choose the numbers randomly.
- The program should give the student three chances to answer each question. If the student gives the correct answer, your program should indicate that fact in some properly congratulatory way and go on to the next question. If the student does not get the answer in three tries, the program should give the answer and go on to another problem.

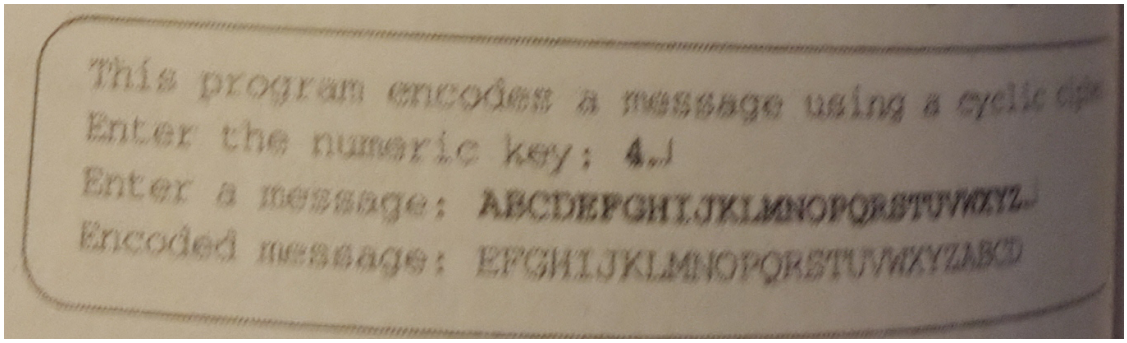
TA comment

Use "RandomInteger(x,y)" (from random.h library)

CH.9: (pg. 334-338)
8. ciper.c : **(diff with "cipher.out" for this problem, our example uses**
5 (Numeric Key)
abcdefgh (Message))

One of the simplest types of codes used to make it harder for someone to read a message is a **letter-substitution cipher**, in which each letter in the original message is replaced by some different letter in the coded version of that message. A particularly simple type of letter-substitution cipher is a **cyclic cipher**, in which each letter is replaced by its counterpart a fixed distance ahead in the alphabet. The word *cyclic* refers to the fact that if the operation of moving ahead in the alphabet would take you past Z, you simply circle back to the beginning and start over again with A.

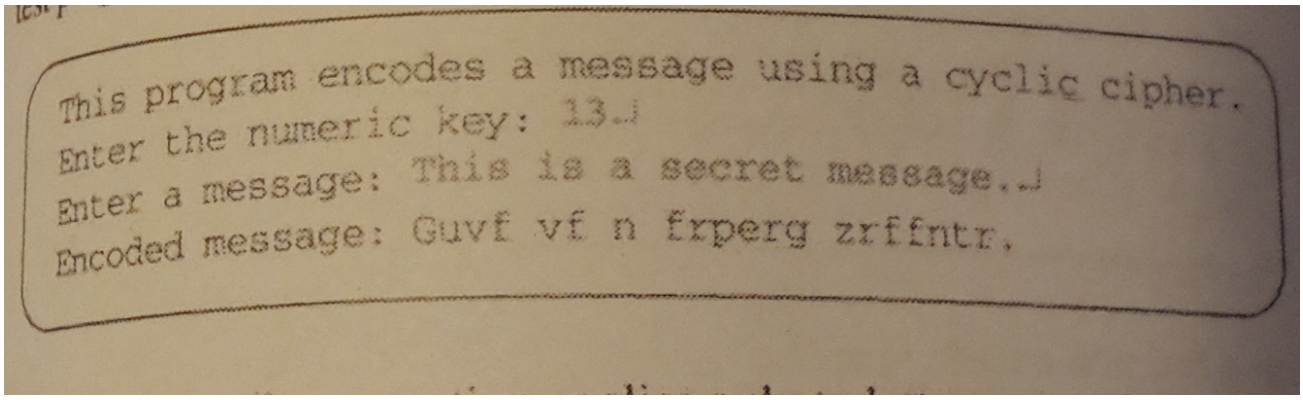
As an example, the following sample run shows how each letter in the alphabet is changed by shifting it ahead four places. The A becomes E, the B becomes F, the Z becomes D (because it cycles back to the beginning), and so on:



To solve this problem, you should first define a function *EncodeString* with the prototype

string EnocdeStirng (string str, int key);

The function returns the new string formed by shifting every letter in *str* forward the number of letters indicated by *key*, cycling back to the beginning of the alphabet if necessary. After you have implemented *EncodeString*, write a test program that duplicates the examples shown in the following sample run:



Note that the coding operation applies only to letters; any other character is included unchanged in the output. Moreover, the case of letters is unaffected: lowercase letters come out as lowercase, and uppercase letters come out as uppercase.

T.A comment:
Use “%s” for printing

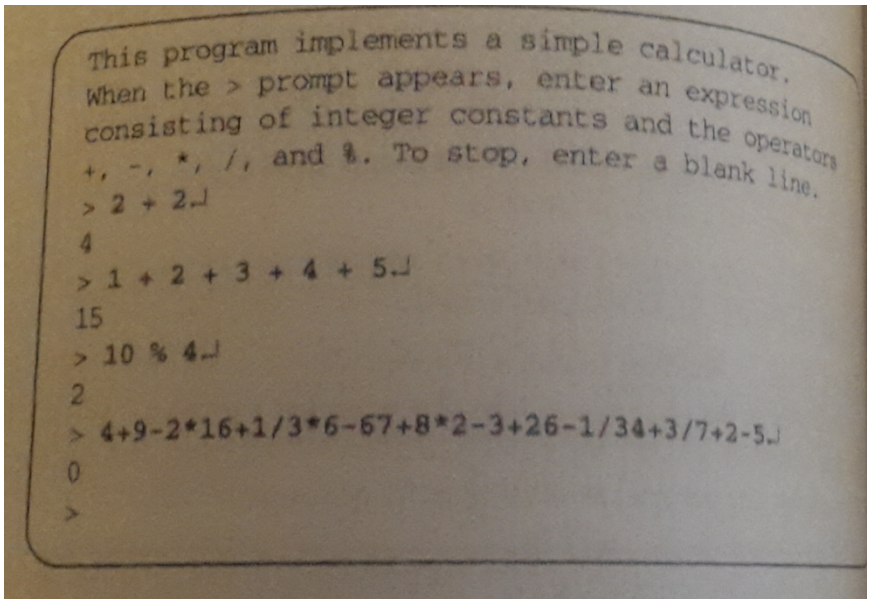
Write your program so that a negative value of key means that letters are shifted toward the beginning of the alphabet instead of toward the end, as illustrated by the following sample run:

CH.10: (pg. 367-372)
10. calc.c **(diff with calc.out, use the calc_input file for the input)**

Write a program *calc.c* that implements a simple arithmetic calculator. Input to the calculator consists of lines composed of integer constants separated by the five arithmetic operators used in C: +, -, *, /, and %. For each line of input, your program should display the result of applying the operators to the surrounding terms. To read the individual values and operators, you should use the scanner module as extended in exercise 9, so that spacing is ignored. Your program should exit when the user enters a blank line.
To reduce the complexity of the problem, your calculator should ignore C‟s rules of precedence and instead apply the operators in left-to-right order. Thus, in your calculator program, the expression

2 + 3 * 4

has the value 20 and not 14, as it would in C.
The following is a sample run of the calc.c program:



TA comment:

THE SCANNER MODULE IS PROVIDED IN THE RESOURCES, USE THEM FOR THIS PROGRAM. PLEASE DO NOT WRITE A NEW SCANNER.H AND LINK THEM.

Importing scanner.h:

#include "scanner.h", [THIS SHOULD BE BEFORE THE MAIN FUNCTION]

and now start using the functions of "scanner.h" in your calc.c program,

Also include

#define TRUE 1

#define FALSE 0