

**Due: 6-5-2015 (June 5, 2015)**

Filename: **polygon.c, currencies.c, fibonacci.c, towers.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 Proj5.** Do not hand in files to directories other than Proj5, they will not be graded. The files are the names of the files you wish to submit. For this assignment, your files will be **polygon.c, currencies.c, fibonacci.c, towers.c**. Therefore,

you would type from the proper directory:

**>>handin cs30 Proj5 programName.c**

**Do NOT zip or compress the files in any way.** Just submit your source code.  
\$ handin cs30 Proj5 polygon.c, currencies.c, fibonacci.c, towers.c

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 project we are giving .out files for all the problems, and input files when needed. 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:*

If you are working on hello.c:

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

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

```
$ ./hello > myHello.txt
```

```
$ ./hello.out > hello.txt
```

```
$ diff myHello.txt hello.txt
```

**If there were no differences, nothing will print when you run ‘diff’. If you get a “Permission denied” error for any of the files (ex. hello.out), try `$ chmod 777 hello.out` in the console/terminal.**

For the problem with required input file, do the following (ex. inchtocm.c):

```
$ ./inchtocm < inchtocm_input > myInchtocm.txt
```

```
$ ./inchtocm.out < inchtocm_input > inchtocm.txt
```

```
$ diff myInchtocm.txt inchtocm.txt
```

```
$ diff -w myInchtocm.txt inchtocm.txt (ignore differences of white spaces) - used for grading
```

**Checking with diff is very important as we will be using scripts to grade your homeworks, and even the smallest difference will result in your not getting points for a problem.**

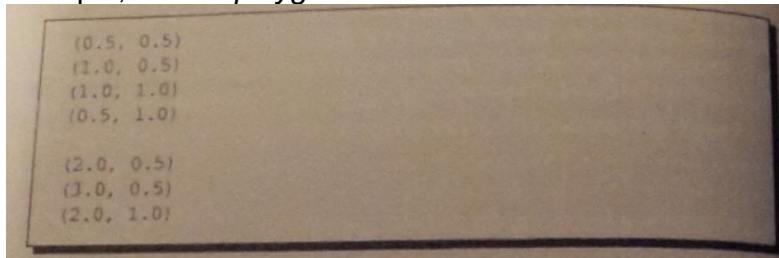
### Programming project #5: Ch. 15: 9; Ch. 16: 7; Ch. 17: 1, 4

Programs reproduced below for your convenience: (from the Programming Exercises section of each chapter)

#### CH.15: (pg. 549-556)

9. filename: **polygon.c** (polygon.dat, polygon.out, and polygon.png provided)  
**(output text from polygon.out does not matter for this problem)**

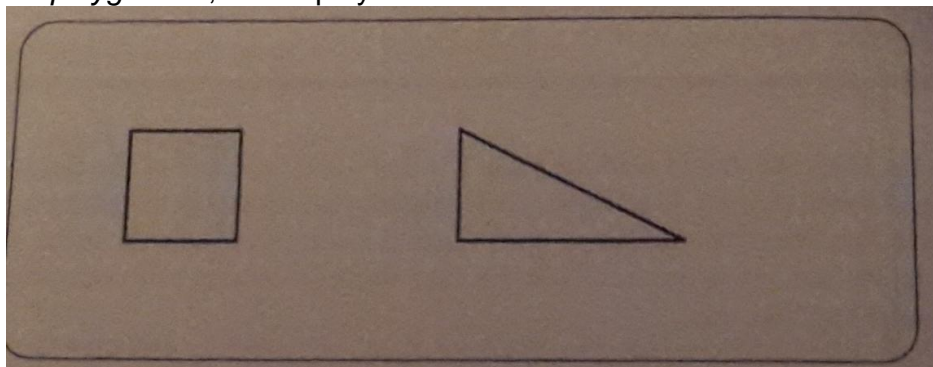
Using the graphics library from Chapter 7, write a program that displays polygons on the screen. The coordinates for the vertices of a polygon are stored in a data file, one coordinate per line. Each coordinate consists of two floating point numbers enclosed in parenthesis and separated by a comma. The input file may contain vertex coordinates for several different polygons; each polygon is separated from the preceding one by a blank line in the file. For example, the file *polygon.dat* contains the coordinates for a square and a triangle, as follows:



```
(0.5, 0.5)
(1.0, 0.5)
(1.0, 1.0)
(0.5, 1.0)

(2.0, 0.5)
(3.0, 0.5)
(2.0, 1.0)
```

Your program should read the input file and draw the polygons on the screen. Given the input file *polygon.dat*, the display should look like this:



***TA comment: Try using GetLine() to get polygon.dat from the user. You should also look into stdio.h file related functions, including fopen and fclose.***

#### CH.16: (pg. 593-600)

7. filename: **currencies.c** (exchange.dat, currencies.out, currencies\_input provided)

Suppose you have been hired as a programmer for a bank to automate the process of converting between different foreign currencies at the prevailing rate of exchange. Every day, the bank receives a data file called *exchange.dat* containing the current exchange rates. The file is composed of lines in the following form:

dollar	1.00
yen	0.0078
franc	0.20
mark	0.68
pound	1.96

Each line consists of the name of a particular kind of currency, at least one space, and the dollar equivalent of one unit of that currency. Thus, the sample input file tells us that the British pound is worth \$1.96 and the German mark is worth 68 cents. (Note that the file includes a line for dollars, for which the exchange rate is always 1.00. The presence of this line means that the U.S. dollars need not be treated as a special case.)

Write a program that performs these steps:

- Read in the *exchange.dat* data file into a suitable internal data structure.
- Ask the user to enter two currency names: that of the old currency being converted and of the new currency being returned.
- Ask for a value in the original currency.
- Display the resulting value in the second currency. The easiest way to compute this value is to convert the original currency to dollars and then convert the dollars to the target currency.

The following sample run illustrates the operation of this program using the data in the *exchange.dat* file:

```

Convert from: mark.
Into: yen.
How many units of type mark? 200.
200 mark = 17435.9 yen

```

**TA comment:** Use %g to print the values of your final currency numbers.

CH.17: (pg. 639-645)

1. filename: **fibonacci.c** (**fibonacci.out**, **fibonacci\_input** provided)

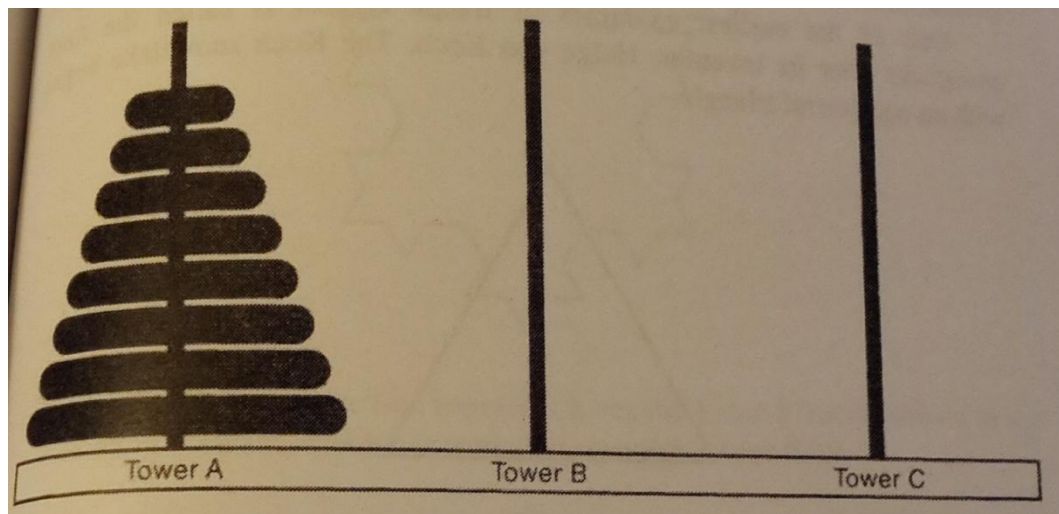
Exercise 8 of Chapter 4 introduced you to the Fibonacci series, in which the first two terms are 0 and 1 and every subsequent term is the sum of the two preceding terms. The series therefore begins with

$$\begin{aligned}
 F_0 &= 0 \\
 F_1 &= 1 \\
 F_2 &= 1 \quad (F_0 + F_1) \\
 F_3 &= 2 \quad (F_1 + F_2) \\
 F_4 &= 3 \quad (F_2 + F_3) \\
 F_5 &= 5 \quad (F_3 + F_4) \\
 F_6 &= 8 \quad (F_4 + F_5)
 \end{aligned}$$

and continues in the same fashion for all subsequent terms. Write a recursive implementation of the function  $Fib(n)$  that returns the  $n^{\text{th}}$  Fibonacci number. Your implementation must depend only on the relationship between the terms in the sequence and may not use any iterative constructs such as *for* and *while*.

4. filename: **towers.c** (**towers\_input**, **towers.out** provided)

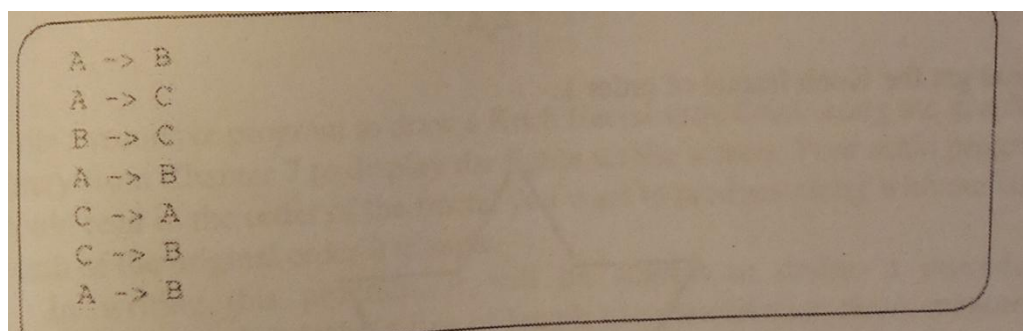
In almost any computer science course that covers recursion, you will learn about a nineteenth-century puzzle that stands as the archetypal recursive problem. This puzzle, which goes by the name *Towers of Hanoi*, consists of three towers, one of which contains a set of disks – usually eight in commercial version of the puzzle – arranged in decreasing order of size as you move from the base of the tower to its top, as illustrated in the following diagram:



The goal of the puzzle is to move the entire set of disks from Tower A to Tower B, following these rules:

- You can only move one disk at a time.
- You can never place a larger disk on top of a smaller disk.

Write a program to display the individual steps required to transfer a tower of  $N$  disks from Tower A to Tower B. For example, your program should generate the following output when  $N$  is 3:



The key to solving this problem is finding a decomposition of the problem that allows you to transform the original Tower of Hanoi problem into a simpler problem of the same form.

T.A comment:

You will get full credit for the print statements. The animation is extra credit.  
Use “%c” for printing