**ECS 30** **Homework #4 (50 points)** **Fall 2013**

Due: Wednesday, October 30th, Written: 4pm in 2131 Kemper. Program: 11:59pm using handin to cs30, p4 directory.
Filenames: data.c, beer.c, line.c

Written (7 points): pp. 303-304: 1, 5, 6, 7 pp. 363-364: 3, 4, 6.
pp.303-304:
1. In what ways are the initialization, repetition test, and update steps alike for a sentinel-controlled loop and an endfile-controlled loop? How are they different?

5. Rewrite the program segment that follows using a for loop:
```
count = 0;
i = 0;
while (i < n) {
  scanf("%d", &x);
  if (x == i)
    ++count;
  ++i;
}
```

6. Rewrite this for loop heading, omitting any invalid semicolons.
```
for (i = n;
     i < max;
     ++i;);
```

7. Write a do-while loop that repeatedly prompts for and takes input until a value in the range 0 through 15 inclusive is input. Include code that prevents the loop from cycling indefinitely on input of a wrong data type.

pp. 363-364:
3. Explain the allocation of memory cells when a function is called. What is stored in the function data area for an input parameter? Answer the same question for an output parameter?

4. Which of the functions in the following program outline *can* call the function grumpy()? All function prototypes and declarations are shown; only executable statements are omitted.

```
  int grumpy(int dopey);
  char silly(double grumpy);
  double happy(int char greedy);

int main(void) {
  double p, q, r;
  …
}

int grumpy(int dopey) {
  double silly;
  …
}

char silly(double grumpy) {
  double happy;
  …
}

double happy(int goofy, char greedy) {
  char grumpy;
  …
}
```

6. Present arguments against these statements:
   a. It is foolish to use function subprograms because a program written with functions has many more lines than the same program written without functions.

   b. The use of function subprograms leads to more errors because of mistakes in using argument lists.

**Programming (43 points)**

All should be able to compile with no warnings when compiled with the -Wall option. You should put your name(s) in a comment on the first line of each file. The prompts, and output format of each program must match the examples exactly. You will find my executable in ~ssdavis/30/p4 in the CSIF.

Since Chapter 5 is exploring loops, the testing values can be quite long. Rather than have you have to enter data tediously, your Chapter 5 programs will read their data from files. To allow you to specify different file names, we are going to introduce the command line parameters of main() well before you should understand them clearly. I want you to accept the following explanation of how to use them, without worrying about the need to understand what is going on. Rest assured we will get to them in about a month! For those curious students, look up "command line arguments" in the index of the txt.

Here is what you need to know for this assignment. main(), like all functions, can take parameters. In the case of main(), the parameters are supplied by the operating system based on what you type on the command line when you invoke your program. When you wish to utilize these parameters, you declare "int main(int argc, char *argv[])" instead of "int main()". argc contains the number of parameters on the command line, including the name of the executable in the count. argv is a list containing each separate word on the command line. For this assignment, you will be supplying the input filename after the name of the executable, e.g., "a.out nums.txt." In this format, argv[1] contains the filename. All you will need to do to open the file is: `FILE *fp = fopen(argv[1], "r");`

From there on use fp, just like you have already grown accustom to using it with fscanf().

1. p. 305 #4. (7 minutes, 5 points) Filename: data.c

Write a program that will find the smallest, largest, and average values, and standard deviation in a collection of N numbers. Get the value of N before scanning each value in the collect of N Numbers. To compute the standard deviation, accumulate the sum of the squares of the data values (**`sum_squares`**) in the main loop. After loop exits use the formula:

$$\textit{standard deviation} = \sqrt{\frac{sum\_squares}{N} - average^2}$$

The first line of the file will contain how many numbers will be in the file as an int. The numbers themselves will be doubles.

```
[ssdavis@lect1 p4]$ cat data1.txt
5
4.34 23.4 18.92 -78.3 17.9
[ssdavis@lect1 p4]$ data.out data1.txt
Smallest: -78.30
Largest:   23.40
Average:   -2.75
Standard deviation: 38.309
[ssdavis@lect1 p4]$
[ssdavis@lect1 p4]$ cat data2.txt
4
10.0 15.0 25.0 30.0
[ssdavis@lect1 p4]$ data.out data2.txt
Smallest:  10.00
Largest:   30.00
Average:   20.00
Standard deviation:  7.906
[ssdavis@lect1 p4]$
```

2. p. 307 #9. (10 minutes, 8 points) Filename: beer.c

"Suppose you own a beer distributorship that sells Piels (ID number 1), Coors (ID number 2), Bud (ID number 3), and Iron City (ID number 4) by the case. Write a program to

a. Get the case inventory for each brand for the start of the week.
b. Process all weekly sales and purchase records for each brand.
c. Display out the final inventory.

Each transaction will consist of two data items. The first item will be the brand ID number (an integer). The second will be the amount purchased (a positive integer value) or the amount sold (a negative integer value). For now you may assume that you always have sufficient foresight to prevent depletion of your inventory for any brand. (*Hint:* Your data entry should begin with four values representing the case inventory, followed by the transaction values.)"

The first line of the file will have the inventory for the four brands arranged by ID number. All succeeding lines will have one transaction per line. You should use macros for the ID numbers.

```
[ssdavis@lect1 p4]$ cat beer1.txt
234 120 4521 93
1 -100
4 -40
1 -49
2 40
3 -2500
[ssdavis@lect1 p4]$ beer.out beer1.txt
Piels:         85
Coors:        160
Bud:         2021
Iron City:     53
[ssdavis@lect1 p4]$
```

```
[ssdavis@lect1 p4]$ cat beer2.txt
239 539 1200 140
4 -29
3 -674
4 -41
2 -302
1 49
3 -93
4 10
2 -150
3 -75
3 500
2 100
1 -200

[ssdavis@lect1 p4]$ beer.out beer2.txt
Piels:         88
Coors:        187
Bud:          858
Iron City:     80
[ssdavis@lect1 p4]$
```

3. pp.366-368 #4. (40 minutes, 30 points) Filename: line.c

"The table below summarizes three commonly used mathematical models of nonvertical straight lines.

| **Mode** | **Equation** | **Given** |
|---|---|---|
| Two-point form | m = (y2 – y1) / (x2 – x1) | (x1, y1), (x2, y2) |
| Point-slope form | y – y1 = m(x –x1) | m, (x1, y1) |
| Slope-intercept from | y = mx + b | m, b |

Design and implement a program that permits the user to convert either two-point form or point-slope form into slope-intercept form. Your program should interact with the user as follows:

```
Select the form that you would like to convert to slope-intercept form:
1) Two-point form (you know two points on the line)
2) Point-slope form (you know the line's slope and one point)
=> 2

Enter the slope=> 4.2
Enter the x-y coordinates of the point separated by a space=> 1 1

Point-slope form
  y - 1.00 = 4.20(x - 1.00)

Slope-intercept form
```

```
  y = 4.20x - 3.20

Do another conversion (Y or N)=> Y

Select the form that you would like to convert to slope-intercept form:
1) Two-point form (you know two points on the line)
2) Point-slope form (you know the line's slope and one point)
=> 1

Enter the x-y coordinates of the first point separated by a space=> 4 3
Enter the x-y coordinates of the second point separated by a space=> -2 1

Two-point form
      (1.00- 3.00)
  m = -------------
      (-2.00 - 4.00)

Slope-intercept form
  y = 0.33x + 1.66
Do another conversion (Y or N)=> N
```

Implement the following functions:

**`get_problem`**—Displays the user menu, then inputs and returns as the function value the problem number selected.

**`get2_pt`**—Prompts the user for the x-y coordinates of both points, inputs the four coordinates, and returns them to the calling function through output parameters.

**`get_pt_slope`**—Prompts the user for the slope and x-y coordinates of the point, inputs the three values and returns them to the calling function through output parameters

**`slope_intcpt_from2_pt`**—Takes four input parameters, the x-y coordinates of two points, and returns through output parameters the slope (*m*) and y-intercept (*b*).

**`intcpt_from_pt_slope`**—Takes three input parameters, the x-y coordinates of one point and the slope, and returns as the function value the y-intercept.

**`display2_pt`**—Takes four input parameters, the x-y coordinates of two points, and displays the two-point line equation with a heading.

**`display_pt_slope`**—Takes three input parameters, the x-y coordinates of one point and the slope, and displays the point-slope line equation with a heading.

**`display_slope_intcpt`**—Takes two input parameters, the slope and y-intercept, and displays the slope-inercept line equation with a heading. "

You may assume that the user will only enter valid values. The x-y coordinates will be entered as ints. Because of differences in rounding, your slope and intercepts may differ by .01 from mine. Note that intercept = y – mx.

Be careful when calling scanf() with an output parameter. Since the parameter is already an address, you will not need to use the '&' operator in the call to scanf().

My suggested order of development:

1. Write the prototypes for all of the above functions based on their descriptions, and a main() with just a return statement, and then compile. It should compile without warnings.
2. Copy the prototypes as a group below main(). Make each prototype into a function stub by adding '{' and a properly commented '}'. For the functions that return values, write a simple "return 0;". This file should also compile without warnings.
3. Declare the variables in main(), and write it completely without filling in any stubs. Use the output to the screen as a guide of the order of function calls, and the need for control structures. Your file should compile without warnings at most intermediate steps, so compile frequently while writing main().
4. Write the functions in the order in which they are called in main(). Compile and run the program after writing each function. If there are problems you will know from where they must have arisen, and you will be quite familiar with your most recent code. Presentation of the slope-intercept form is tricky because the sign is presented separately from the intercept itself. (A good little puzzle for you.)