ECS 30 **Homework #1 (37 points)** **Winter 2013**

Due: Wednesday, January 16th. Written: 4pm in 2131 Kemper. Programs: 11:55pm using handin to cs30, p1 directory.
Filenames: diameter.c, sugar.c, change.c, light.c, rounding.c ascii.c.

Written (6 points)

1. Which variables below are syntactically correct?
   income, 2time, int, Tom's, two fold, c3po, income#1, item

2. Stylistically, which of the following identifiers would be good choices for names of constant macros?
   gravity G MAX_SPEED Sphere_Size

3. (3 points) Write the data requirements, necessary formulas, and algorithm for Programming Project 9 (see grass.c below).
   This does NOT mean write the program. Look at page 37 in the text for an example.

4. List three standard data types of C.

zyante sections: 1.4 -1.8, 2.1, 2.2, 2.10, 2.11, 2.13, 2.16, 2.17, 9.3.

Programming (30 points)

All programs should be able to compile with no warnings when compiled with the –Wall option, e.g. gcc –Wall reimbursement.c. You should put your name(s) in a comment on the first line of each file. In the examples, the user input is in **bold.**

The prompts, and output format of each program must match my programs **<u>EXACTLY</u>**. You will find my executables and some testing files in ~ssdavis/30/p1 in the CSIF. Note that these are not accessible from the web. You must log into your CSIF account and use cp to copy them to your own directory, e.g. "cp ~ssdavis/30/p1/* p1", where you have already created a p1 directory.

You can use the UNIX diff utility to compare your executable with mine. diff will compare two files line by line, and list the changes needed that need to be made to the first file to make it identical to the second file. If there are no differences between the two files then diff outputs nothing.

In the following example of using diff I altered my diameter source code to produce an incorrect prompt, an incorrectly formatted number, and an incorrect value, and named the executable diameterErrors.out. To ensure that both executables receive the same input, I typed the input into a file named diameter.txt.txt, and then redirected it into each executable using the "<." To redirect the output from the monitor into files, I used the ">."

```
[ssdavis@lect1 p1]$ cat diameter1.txt
4.3

[ssdavis@lect1 p1]$ diameter.out < diameter1.txt > seans.txt
[ssdavis@lect1 p1]$ diameterErrors.out < diameter1.txt > errors.txt
[ssdavis@lect1 p1]$ cat seans.txt
Please enter a diameter: The circumference of a circle with diameter 4.300 is 13.51.
The area of a circle with diameter 4.30 is 14.522.
The surface area of a sphere with diameter 4.30000 is 58.0880.
The volume of a sphere with diameter 4.3000 is 41.62977.
[ssdavis@lect1 p1]$ cat errors.txt
please enter a diameter: The circumference of a circle with diameter 4.3000 is 13.51.
The area of a circle with diameter 4.30 is 14.522.
The surface area of a sphere with diameter 4.30000 is 46.4704.
The volume of a sphere with diameter 4.3000 is 41.62977.
[ssdavis@lect1 p1]$ diff seans.txt errors.txt
1c1
< Please enter a diameter: The circumference of a circle with diameter 4.300 is 13.51.
---
> please enter a diameter: The circumference of a circle with diameter 4.3000 is 13.51.
3c3
< The surface area of a sphere with diameter 4.30000 is 58.0880.
---
> The surface area of a sphere with diameter 4.30000 is 46.4704.
```

```
[ssdavis@lect1 p1]$
```

#1 Filename: diameter.c (6 minutes)

Write a program that reads in a diameter, and then prints out the four geometric values for circular objects with that diameter. Your program should not call any functions but printf and scanf. You should store PI as a constant macro with the value 3.14159265. The geometric formulas for circular objects are:

- Circumference of a circle = $2\Pi r$
- Area of circle = Πr^2
- Surface area of a sphere = $4\Pi r^2$
- Volume of a sphere = $4/3\Pi r^3$

#2. Filename: sugar.c (12 minutes)

A government research lab has concluded that certain chemicals commonly used in foods will cause death in laboratory mice. A friend of yours is desperate to lose weight, but cannot give up soda pop. Your friend wants to know how much diet soda pop it is possible to drink without dying as a result. Write a program to supply the answer. The input to the program is the amount of artificial sweetener needed to kill a mouse (a real in kilograms), the weight of the mouse (a real in kilograms), and the weight of the dieter (an integer in pounds). To ensure the safety of your friend, be sure the program requests the weight at which the dieter will stop dieting, rather than the dieter's current weight. Assume that the diet soda contains 1/10th of one percent artificial sweetener. Use a defined constant to express this percent. You may want to express the percent as the real value 0.001. One pound equals 0.453592 kilograms. Use a constant macro to express this conversion ratio.

```
[ssdavis@lect1 p1]$ sugar.out
Amount of sweetener needed to kill the mouse (kg): 0.43
Weight of the mouse (kg): 1.23
Goal weight of the dieter (pounds): 135
For a goal weight of 135 pounds, you may drink 21407.3 kg of soda.
[ssdavis@lect1 p1]$ sugar.out
Amount of sweetener needed to kill the mouse (kg): 0.019
Weight of the mouse (kg): 0.98
Goal weight of the dieter (pounds): 105
For a goal weight of 105 pounds, you may drink 923.4 kg of soda.
[ssdavis@lect1 p1]$
```

#3. Filename: change.c (4 minutes)

Write a program that displays the minimum number of coins required to make change for an amount between 0 and 99 cents. The value input will be between 0 to 99 as an integer. The output should list the number pennies, nickels, dimes, and quarters necessary to create the value with the minimum number of coins. Note that integer division makes this pretty easy.

```
[ssdavis@lect1 p1]$ change.out
Please enter the amount of change (0-99): 97
That would be 3 quarters, 2 dimes, 0 nickels, and 2 pennies.
[ssdavis@lect1 p1]$ change.out
Please enter the amount of change (0-99): 99
That would be 3 quarters, 2 dimes, 0 nickels, and 4 pennies.
[ssdavis@lect1 p1]$ change.out
Please enter the amount of change (0-99): 74
That would be 2 quarters, 2 dimes, 0 nickels, and 4 pennies.
[ssdavis@lect1 p1]$ change.out
Please enter the amount of change (0-99): 19
That would be 0 quarters, 1 dimes, 1 nickels, and 4 pennies.
[ssdavis@lect1 p1]$
```

#4. Filename: light.c (8 minutes)

Surprisingly, the speed of light now impacts the top speed of computers. The speed of light is 299,792,458 meters per second. (Electricity flows in wires between 97% and 66% of the speed of light, but we will ignore this for this problem.) The central processing units (CPUs) of computers are now able to can do operations, e.g add two numbers, as frequently as four billion times a second. You are to write a program that inputs the frequency of a CPU, in Hz (cycles per second), and prints out the distance that can be covered by light in one cycle of the CPU in millimeters. The input will be between one million, and four billion. Since an “int” can only hold values from -2^{31} to 2^{31} = 2,147,483,648 to 2,147,483,648 we will need to use an “unsigned int” which can hold values from 0 to 2^{32} = 0 to 4,294,967,296. To read an unsigned int using scanf you will use “%u” instead of “%d”. Use a constant macro to store the speed of light without a decimal point.

Because the speed of light and the frequency of CPUs are so close to the limit of values represented in ints, it would be quite easy to write arithmetic expressions with ints that would produce incorrect results either because an value exceeds the capacity of an int, or an intermediate integer division losses many sign digits, or is even zero. To overcome this problem, you should use a double and/or casting to doubles in your calculations.

(5 points extra credit) Write a second set of calculations that only use ints and/or unsigned ints and no casting, and yet consistently comes within 2% of the correct answer as long as the frequency entered is between one million and four billion. If you do the extra credit, print out the “integer” answer on the line after the “double” answer. Note that the value of your answer for the extra credit need not match mine.

```
[ssdavis@lect1 p1]$ light.out
Frequency of the CPU (Hz): 1000000
Electricity can travel 299792 millimeters in one cycle of a CPU that has
a frequency of 1000000 Hz.
Using only integers, that is 299792 millimeters.
[ssdavis@lect1 p1]$ light.out
Frequency of the CPU (Hz): 1001000
Electricity can travel 299493 millimeters in one cycle of a CPU that has
a frequency of 1001000 Hz.
Using only integers, that is 299492 millimeters.
[ssdavis@lect1 p1]$ light.out
Frequency of the CPU (Hz): 4000000000
Electricity can travel 75 millimeters in one cycle of a CPU that has
a frequency of 4000000000 Hz.
Using only integers, that is 74 millimeters.
[ssdavis@lect1 p1]$
```

#5. Filename: rounding.c (4 minutes)

You would think that representing money would be easy in a computer. After all, you only need two digits to the right of the decimal point. However, since the double data type is stored using base 2, some simple numbers for us are difficult for the computer. Much the same as 1/3 is an endless decimal number 0.33333… in base 10, so too is 1/10 in base 2. Also, since a double uses 52 bits to hold its mantissa, it can only retain 15 to 17 decimal digits. This program will permit you to detect the rounding error of doubles. Write a program that reads in a double, and then multiplies it by 1,000,000, stores the product in a double, then divides the product by 1,000,000 and stores that in a double named result, and then prints out that final result.

Print out the result in three formats below each other with the decimal points aligned: 1) with two digits to the right of the decimal point, 2) with 10 digits to the right of the decimal point, and 3) and with 20 digits to the right of the decimal point. You must use only printf statement to print out all three lines of the results, and that printf may not contain any spaces in the format string.

Try experimenting with inputting monetary values. Which of the following values can a double accurately represent? 534.10, 70.20, 920.25, 345330.61, 1234567890123456.50, 9234567890123456.50, 12345678901234567.50, and 1234567890123456.75. Why?

```
[ssdavis@lect1 p1]$ rounding.out
Please enter a monetary value: 534.10
          534.10
          534.1000000000
          534.1000000000002273737
[ssdavis@lect1 p1]$ rounding.out
Please enter a monetary value: 920.25
          920.25
          920.2500000000
          920.2500000000000000000
[ssdavis@lect1 p1]$ rounding.out
Please enter a monetary value: 1234567890123456.50
1234567890123456.50
1234567890123456.5000000000
1234567890123456.5000000000000000000
[ssdavis@lect1 p1]$ rounding.out
Please enter a monetary value: 9234567890123456.50
9234567890123456.00
9234567890123456.0000000000
9234567890123456.0000000000000000000
[ssdavis@lect1 p1]$
```

#6. Filename: ascii.c (6 minutes)

The char data type is able to hold values from -128 to 127, inclusive. It is sort of miniature int! While you would normally use “%c” with printf to print out a char, you can also use “%d” to print out its ASCII value. For this program you will read in two chars and an int, and then print out their values and the results of a variety of arithmetic operations among the three variables. Please note that most of the ASCII values between 0 and 31 will not print anything on a monitor.

```
[ssdavis@lect1 p1]$ ascii.out
Please enter two characters and an integer: $A2
c1 + c2 = e (char), 101 (int).
c1 - c2 = ã (char), -29 (int).
c1 + num = & (char), 38 (int).
 [ssdavis@lect1 p1]$ ascii.out
Please enter two characters and an integer: 50-2
c1 + c2 = e (char), 101 (int).
c1 - c2 =  (char), 5 (int).
c1 + num = 3 (char), 51 (int).
[ssdavis@lect1 p1]$
```