UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CSE013E: INTRODUCTION TO PROGRAMMING IN C

Lab 2: Calculator
12 Points

Introduction

In this lab you will be writing your first program from scratch. This program will read user input, perform mathematical calculations, and then print the results out to the user. This will be done using the serial port just as you did in part 3 of the last lab, and so will require use of a serial terminal program. See the Serial Communications document for more information on serial communications.

It will rely on knowledge from outside of class such as data types and `printf()` and `scanf()`; pay close attention to the reading list below. This lab will also introduce declaring, defining, and implementing functions. Aaa brief overview of these concepts is available in this document and there is additional information in K&R.

Reading

- K&R All of Chapter 1, 7.4, and Appendix B1.2
- Serial Communications handout
- Software design handout
- Code style handout

Concepts

- `printf()` and `scanf()`
- Functions and Prototypes
- C standard library
- Iterative Code Design

Graded Files:

Exactly one of each of these files must be contained inside the Lab02 folder of your repo, or in a subfolder in the Lab02 folder, in the commit that you submit. Be sure that the first line of each file contains your name and ucsc email address.

1) `Calculator.c`
2) `README.md`

Provided Files:

- **Edit these files:**
    - Calculator.c – (named as Calculator_template.c, you must rename this!) This file contains the program template where you will implement a simple calculator. The user interface functionality will be contained in `CalculatorRun()`. Calculations will be handled in functions that are prototyped before `CalculatorRun()` and defined after `CalculatorRun()` (comments indicate where you should add each function's definition).

      Some these prototypes and definitions have been started for you.
- **Do NOT edit these files:**
    - Lab2_main.c – This file contains a very brief main function that calls `CalculatorRun()`.

Assignment requirements

For all code that you submit: Follow the standard style formatting procedures for syntax, variable names, and comments. Be sure to check the CSE013E_StyleGuidlines document.

In this lab, you will write a simple calculator that will communicate with a user via a serial communication. This calculator has the following requirements:

- The program should:
    - Welcome the user to your calculator program with a nice greeting.
    - Prompt the user to choose a mathematical operation.
    - Prompt the user for one operand.
    - If appropriate, prompt the user for a second operand.
    - Display the result.
    - Return to the first prompt in an infinite loop.

- The user selects an operation by typing a single character. The prompt should also display all operations that are available to the user. These All of these operations must handle negative values correctly. The operations you must implement:
    - '*' - Performs multiplication on two operands.
    - '/' –Performs division by dividing the first operand by the second.
        - If the second operand is a 0, Your calculator should print an appropriate error message.
    - '+' - Performs addition on two operands.
    - '–' - Performs subtraction on two operands, subtracting the second operand from the first.
    - 'm' – Calculates the average of its two arguments.
    - 'a' – Calculates the absolute value of its argument.
    - 'c' - Treats its argument as a value in degrees Fahrenheit and converts it to degrees Celsius.
    - 'f' - Treats its argument as a value in degrees Celsius and converts it to degrees Fahrenheit.
    - 't' – Takes in a value in degrees and calculates the tangent value and returns the result.
    - 'q' – Exits the calculator run function using return. The code provided will exit the the program on the OS while will stall in infinite loop on the embedded system.
    - **Extra Credit:** 'r'. Rounds a number to the nearest integer, rounding away from zero if the fractional part is 0.5. Thus, 4.5 rounds to 5, and -4.5 rounds to -5.

- All of the calculations for the math operations above must be done inside of appropriate functions, and NOT inside `CalculatorRun()`.

- Input/output logic should be implemented in `CalculatorRun()` , and NOT inside of the calculation functions.

    - In other words, `scanf()`, `printf()`, or any other <stdio.h> function should NOT be found in any of your code besides the definition of `CalculatorRun()`.

    - This includes `Divide()`! Do not print from inside `Divide()`. Instead, `CalculatorRun()` should check the return value of `Divide()`, and react appropriately.

- Each calculation function must be implemented as outlined in the Functions section of this lab manual (declaration, implementation, and usage).
    - Each must have the same name and prototype as described in this lab manual.
    - All calculation functions should use double as both their input and output data types.
    - All function-based calculations should return the result using a return statement (no global or module-level variables!).

- Once all operands have been entered, print out the result of the mathematical operation along with what operation was performed. The following example will suffice for basic

operations: "`Result of (3.25 * 4): 13`". Notice that you will need several different `printf()` format strings for results that are calculated from a unary operator versus those from a binary operator. Unary operators will require a format that looks more like "`Result of |-5.3|: 5.3`" (that is an example of an absolute value calculation).

Example output:

- `Result of (4.5 deg->F): 40.099998`
- `Result of (57 deg->C):  13.888889`
- `Result of tan(3.7):     0.0647`
- `Result of round(5.8):   6.000000`
- `Result of (3 * 3):      9.000000`

- After printing the result, return to the initial prompt. This should result in an infinite loop of prompting the user for another calculation after displaying the results of the prior calculation.
- **Extra Credit:** implementation of a round function along with it being usable by the user of your calculator program.

Example:

```
Welcome to Max's calculator program! Compiled at Jul  2 2018 14:50:28

Enter a mathematical operation to perform (*,/,+,-,v,a,c,f,t,r): +
   Enter the first operand: -3
   Enter the second operand: 7.334
   Result of (-3.000000 + 7.334000): 4.334000

Enter a mathematical operation to perform (*,/,+,-,v,a,c,f,t,r): a
   Enter the first operand: -2000
   Result of |-2000.000000|: 2000.000000

Enter a mathematical operation to perform (*,/,+,-,v,a,c,f,t,r): t
   Enter the first operand: 45
   Result of tan(45.000000): 1.000000

Enter a mathematical operation to perform (*,/,+,-,v,a,c,f,t,r): r
   Enter the first operand: -3.5
   Result of round(-3.500000): -4.000000

Enter a mathematical operation to perform (*,/,+,-,v,a,c,f,t,r): /
   Enter the first operand: 4
   Enter the second operand: 0
   Divide by zero error!

Enter a mathematical operation to perform (*,/,+,-,v,a,c,f,t,r):
```

Doing this Lab on Linux:

This lab works very well on linux as it uses none of the special features of the embedded system. You can compile your project (and give it the name Calculator) using the command below.

```
$ gcc Calculator.c BOARD.c Lab2_main.c -lm -Wall -o Calculator
```

Like the first lab we are compiling all files in one step but we are adding the extra options explained below.

- "`-lm`": for lab 2 we need to use some of the math libraries provided to us by the compiler. In modern versions of GCC this options adds them to the process.
- "`-Wall`": This turns on the same warnings we have you enable in the MPLABX settings.
- "`-o Calculator`": Names the executable instead of using the default name of "`a.out`".

Lab writeup:

In your README, include:

- Your name
- The names of other students who you collaborated with
- Also, answer the following questions:
  - What happens if the line "`scanf(" ");`" executes? Why?
  - In this lab, we forbid you from using `printf()` or `scanf()` inside of certain functions. Explain why a rule like this is useful.
  - How long did this lab take you? Was it harder or easier than you expected?

Grading

This assignment consists of 12 points:

- **User Interface (3.25 points)** - User input is properly handled and results are output correctly.
- **Correct calculations (5.75 points) -** All operators are implemented correctly, helper functions for advanced calculations are also correct and named properly.
- **Code style (1 point) -** Your code follows the style guidelines and contains less than 10 errors total.

- **README.md (1 point) -** A README.md file was provided with the necessary contents.
- **GIT Hygene (1 point) –** You should have a minimum of 10 commits with significant differences and good commit messages on this lab
- **Extra credit (1 point) -** Correct implementation of a round function based on the implementation described below.

You will lose points for the following:

- Warnings displayed upon compilation. Make sure that "additional warnings" is turned on in Project Properties -> xc32-gcc -> preprocessing and messages.
- Using incorrect names for the calculation functions, or requiring the incorrect number of arguments. This could break our grading scripts!
- You may receive no credit at all if Calculator.c doesn't compile. It must also compile within MPLABX regardless of where you test it.

`printf()` and `scanf()`

You will be using both of these functions in your program to interact with the user. This is done through standard input and output. Both functions are included within the C standard library (part of the C language). They are declared in the header file `stdio.h`. You will need to add an `#include` statement to include the `stdio.h` standard library.

Example usage of these functions follows:

```
char g;
printf("Type in any character:");
scanf("%c", &g);
printf("You input '%c'", g);
```

Please note the *reference operator* (&) in front of the variables passed as arguments to `scanf()`. These are very important! You will need one before all of the variable arguments to `scanf()`. (You'll learn more about what this operator does soon, when we cover pointers. If you're curious now, refer to chapter 5 of K&R).

Note that `scanf()` is a little finicky about how it handles input. If you use `scanf("%f", &x)` to read in a double and type a number and press Return, not all of the characters will be processed. All of the numbers will end up parsed and placed into the `x` variable, but the newline character will not have been processed, and will be captured by future calls to `scanf()`.

One way to solve this is to use a *whitespace specifier*, which is simply a space: `" "` The whitespace specifier will instruct scanf to discard zero or more whitespace characters, so if you use `scanf(" %f", &x)`, any newlines from the previous call will be consumed before the %f is read.

Note that a warning will be generated if the specifer `"%f"` is used to read into a variable of type `double`. `"%lf"` is the appropriate specifier for double floats.

Also, note that scanf may require a *heap.* To allocate heap space, go to "project properties -> xc32-ld" and enter a number in the "heap size" field (2000 is plenty). Later in the course, you'll learn about what the heap is and how to use it.

Functions

- Declaring functions – before a function can be used, it must first be declared (just like a variable). Usually declarations are made with *function prototypes*. They are used to describe the function's name, inputs, and return type. In short, the protype describes everything about the function EXCEPT what it actually does.

  These declarations need to occur in the source code BEFORE the function is first referenced. This means if you call a function in `main()`, but the function is implemented after `main()`, you'll need to put a function prototype before `main()`.

  An example of a function prototype is as follows:

```
double HarmonicMean(double op1, double op2);
```

  This prototype states that the function `HarmonicMean()` takes in two values of type double and returns a double as well.

- Implementing Functions — the *definition* describes what the function does, or in other words, the code that the function executes. This includes the same information as the function prototype (without the `;`), and additionally has code between two curly braces (`{}`) . A function ends when it hits a return statement or the closing curly brace. It can have temporary variables declared after the opening `{`.

  An example of a function definition is as follows:

```
double HarmonicMean(double op1, double op2)
{
      double denominator = 1/op1 + 1/op2;
      return 1/denominator;
}
```

This creates a function that is called with two variables of type `double`. The function returns a value of type `double` that is the harmonic mean of the values passed in as its arguments.

- Using Functions — Functions can be used for various things, but in this lab all that is necessary to know is how to store the return value of a function into a variable. This is done just like storing any value into a variable. On the left-hand side of the assignment operator (=) is the variable that will hold the value, and on the right-hand side is the function call.

  An example, of how this is done is as follows:

  ```
  double result, operand1 = 5, operand2 = 10;
  result = HarmonicMean(operand1, operand2);
  ```

  The expression `HarmonicMean(operand1, operand2)` is a *call* to the HarmonicMean() function. `operand1` and `operand2` are the *arguments* that are *passed* to HarmonicMean() during the call. The *return value* is stored in the variable `result`.

Operations with functions

The functions you will need to implement in this lab are listed below. DO NOT change their names! Again, none of these functions perform input or output using using `scanf()` or `printf()`, that functionality belongs in `CalculatorRun()`. All input arguments are of type double, and all functions return a double.

- `Add()`, `Subtract()`, `Multiply()` — These functions are binary. They return the sum, difference, and product of their operands, respectively.
- `Divide()` – A binary function. If its second argument is not zero, it returns the quotient of its two operands. If the second argument is zero, it returns the math.h constant `HUGE` to signal a divide-by-zero error. It does not print under any circumstances.
- `AbsoluteValue()` – A unary function. It returns the absolute value of its argument. Note: for this function, DO NOT utilize the absolute value function from math.h.
- `FahrenheitToCelsius()` and `CelsiusToFahrenheit()` – These functions are both unary. They convert from °F to °C and from °C to °F, respectively. These should be easy to implement if you finished Lab 1!

- `Tangent()` – This function is unary. It implements the tangent function. It should interpret the input argument in degrees. You must use the standard math library to implement this function.

  The standard math library's version of tangent() uses radians, so you must perform a conversion. To do this conversion, you must use the constant M_PI (also defined in the standard math library). A quick search through the XC32 help docs should help you find all the details you need to implement this function.

- `Average()` – This function is binary and returns the average of its two inputs.

- **For Extra Credit:** `Round()` – This function is unary and takes in a double and returns a double. This function rounds the input to the nearest integer. If the fractional part of the input is 0.5, it should round *away* from zero, so -4.5 rounds to -5, and 4.5 rounds to 5. You may not use any of the functions within the standard library or Microchip's peripheral library to implement this!

  You may have to think a little bit about how this can be done. One method utilizes type casting (described in section 2.7 of K&R, page 42). Another (much less efficient) method could use a while loop that counts down to find the fractional part.

  To get the extra credit, you must implement `Round()`, *and* your calculator must incorporate it as well.

Program flow

Your program will loop continuously while reading and writing from the terminal. This concept is outlined for you within the `` `while (1)` `` loop (which will loop forever) in the pseudo code below. The basic outline of your program looks as follows:

```
Output greeting to the user
while (1)
	get operator
	if operator is invalid:
		set operator to 0
	if operator is valid (ie, not 0):
		get operand1
		if operator is a binary operator:
			get operand2

		if operator is +:
			result = Add()
		else if operator is -:
			result = Subtract()
		    .
```

```
            .

            .

        else if operator is a ‘r’
              result = round

        if operator is a unary operator:
              print the result of a unary operation
        else:
              print the result of a binary operation
  else if operator is invalid:
        print invalid operator message
```

Doing this lab

The CSE_SoftwareDesign handout describes a very powerful way to approach any programming project. Make sure you read it and understand it—this will be useful to you far beyond this class.

Below you will see we have given you an example method of completing this lab loosely following the practices described in the Iterative Code design handout. Remember, it is important to stop and test your code for correct functionality at each step before moving on. Note that at each of these steps you should commit before and after you finish that step at a minimum. One step not included is handling the ‘q’ operator since it can be done at almost any step.

Step 1

- Display a greeting message using `printf()`.

Step 2

- Prompt the user to input a character.
  - There is no need to print this character back out to test this, part of BOARD code automatically echoes back to the user whatever they typed in. So if you see the input you typed in the terminal, that means it was successfully received.
  - Add in an echo back message: “Character received was: ‘x’”

Step 3

- Prompt the user for a character within an infinite loop
  - These characters should all be echoed like they were in the above step.

Step 4

- Now add an invalid operator checker

- Checking for this should set your operator variable to a standard error value (-1) if operator is not one of your valid operator's (at this point you can just use '+').
- Now print the operator if it is not equal to your standard error value, and print an error message otherwise ("Error, not a valid operator").
- At a minimum, test your code with all valid operators and a large number of invalid ones to convince yourself that it works (this is called unit testing).

Step 5

- Continuously prompt the user for an operator and two operands.
  - Echo the results ("You input 3.5 + 4.7") to ensure you are capturing the correct values.
  - If the user enters an invalid operator, print the error message
  - Test this extensively

Step 6

- Continuously prompt the user for an operator and two operands.
  - If the user enters a '+', calculate the result and print it
  - If the user enters an invalid operator, print the error message
  - Test and ensure that your addition is correct. Make sure you use both + and – numbers.

Step 7

- Expand code to work for all 4 basic operators: +, -, /, *
  - Note: this will require you do update your valid operator checker as well.
  - Ensure that your division traps the divide by zero
  - Test to make sure that all the functions work correctly. Lots of testing here.

Step 8

- Display the results nicely as the requirements describe.

Step 9

- Add an operator for an absolute value calculation 'a'.
- Add checking for one or two operands. This checking should make it so your program only prompts for one operand when given the absolute value operator (don't do the calculation just print something to show it works).

Step 10

- Define an absolute value function.
- Test that it works with hard coded inputs. (I.e.: use test cases: -3, -8.63, 0, and 13.67)
  - `printf("%f\n", AbsoluteValue(-3)); // result should be 3`
  - `printf("%f\n", AbsoluteValue(-8.63)); // result should be 8.63`
  - `printf("%f\n", AbsoluteValue(0)); // result should be 0`
  - `printf("%f\n", AbsoluteValue(13.67)); // result should be 13.67`
- Test this extensively with other inputs and make sure you have the correct results. Try to think of inputs that would be problematic and see how it handles them.

Step 11

- Implement the absolute value operator in your calculator by updating your operator checkers, and calling the function in the appropriate place.
- You will now need a new result message with a printf() formatted to display a calculation with only one operand (by now you should know how to do this with an operator checker).

Step 12

- Define an Average function.
- Test to see if it works with hard coded inputs. I.e.: (55.5, 0), (0.00, -10), (-36.49, 36.49)
    - Your outputs should be 27.75, -5, and 0.0 correspondingly.

Step 13

- Implement the Average operator in your calculator.
- Test it extensively
- Check that it now works within the full loop with the other operators

Step 14

- Define a Celsius to Fahrenheit conversion function.
- Test to see that it works with hard coded inputs. I.e: inputs 32, -27, 0
    - Your outputs should be 89.599995, -16.599998, and 32 correspondingly.

At this point you should see a pattern. Implement a new function, test it using hard coded inputs (that you know the corresponding outputs). This tests the functionality. Extensive testing at this stage can reveal bugs which are easy to fix. Try to find inputs that break your function—these are known as "corner cases." Once you have thoroughly tested the function, put it into the rest of the code, and make sure it still works within the new program flow. Test some more, and fix the bugs you find.

Step 15

- Implement the Celsius conversion function in your calculator.
- Test it extensively.

Step 16

- Define a Fahrenheit to Celsius conversion function.
- Test with hard coded inputs (i.e.: 98, -12, 0)
    - Your output should be 36.666668, -24.444445, and -17.777779 correspondingly.

Step 17

- Implement the Fahrenheit conversion function in your calculator.
- Test, test, test. Squash bugs

Step 18

- Define a Tangent in Degrees function.
    - Implement a helper function to convert degrees to radians. Test it.
    - Put together the full function.

    - Test it with hard coded inputs (i.e.: 57, 1.5, -33, 0)
    - Your outputs should be 1.5399, 0.0262, -0.6494, and 0 correspondingly.

Step 19

- Implement the Tangent function in your calculator.
- Test, test, test. Squash bugs, retest.

Step 20

- Implement the Round function in your calculator.

Step 21

- Remove any dead code and comment out any leftover test code. Keeping your test code in the project is generally a good idea, as you can easily uncomment it and test your code again after making changes. Later in the quarter, you will be shown a more elegant way to do this.
- Double-check that you met all program requirements listed in this document.
- Compare your code to the examples in the Style Guidelines document, fixing any errors you see.
- **Submit your finished lab2.c and README.txt through the online submission tools.**

Frequently Asked Questions:

I get warning: format '%f' expects type 'float *', but argument 2 has type 'double *' ***when compiling.***

"`%lf`" is the appropriate format specifier for double floats.

After testing an operation once, and seeing the result, I can no longer select an operation, as it appears to be chosen automatically, but results in an invalid operator.

This is due to not consuming the extra newline character that is the result of pressing ENTER after typing in your operands. See the printf() and scanf() section.