Lab 1 - Calculator

Introduction

In this lab you will be writing your first program. This program will read user input, perform mathematical calculations, and then return the result to the user. It will rely on knowledge from outside of class such as data types and printf/scanf. A brief overview of these concepts is available in this document and there is additional information in K&R.

Due Date:

This lab will be due by 11:59pm Sunday April 10th.

Submission

This lab will consist of you filling in the provided lab1.c file. This is what you will submit using eCommons by the due date.

Assignment

Program Requirements

This assignment has the following requirements:

- Welcome the user to your calculator program with a greeting
- Prompt the user for first a mathematical operation to perform in the form of a single character. These must include at least the four basic math operations such as (*, /, +, -), though there are, of course, many others you could add in as well (for instance the mod operator, %). Your instructions should detail all of the operators available to the user.
- Prompt them further for two operands on which to perform the operation, one at a time.
- Finally return the result of the mathematical operation while displaying what operation it was that was performed, e.g. “Result of (3.25 * 4): 13”.
- Return to prompting the user for a mathematical operation to perform.
- Write well organized code with good variable names and following the style guidelines.

Provided Code

Courtesy of Andrea Di Blas and Cyrus Bazeghi
We have provided you with a program template to use in the form of lab1_stub.c. This sets up the PIC24F for running at 4MHz and using the serial port at 9600baud. This means you do not need to modify the Proteus simulator file before running it (Yay!).

**Code organization**

Lab1_stub.c provides areas for adding your code, which should be placed in between the long comments of asterisks. There are two sections: one for including any additional header files you require and another for writing your actual code. It also includes brief comments outlining the different actions your code will perform specified both in the requirements and in the program flow section further along in this document.

**Data types and casting**

This lab relies on a basic understanding of the different data types in C. This was touched on briefly in lecture, but you will need a little more understanding for this lab.

There are a few different data types, but for this lab you will only need to know about three of them: chars, floats, and doubles. Chars are 8-bit numbers that are generally used for working with text or raw bytes of data. This data type will be used for reading in which operation to perform. Float, known as single or single-precision in other languages) is the basic data type floating-point numbers. They are used for storing a larger range of numbers than integers, namely one's with fractional information, like 3.14 or 7.9. The double is the float's big sibling and gets its name from being referred to as double-precision over the float's single-precision. This distinction is important to note because the compiler always considers floats and doubles to be different variable types. This is somewhat annoying in an environment like on the PIC24F, where the double doesn't actually provide more precision (float and double are both 32 bits).

Now this annoyance can be mitigated somewhat through the use of casting. This term follows from the technique of shaping metal, turning it from one shape into another. In C, casting converts from one data type to another. Casting will be used in this program to convert between floats and doubles. Casting a variable looks like the following chunk of code:

```c
double g = 3.4;
float f = (float)g;
```

Placing a data type in parentheses in front of a variable allows you to treat it as if it was that data type. In the example above, it would not compile if you did not cast g to a float. More details on how casting will be used will be provided in the section.

**printf() and scanf()**

Both printf() and scanf() are functions that deal with streams, in this case streams of text coming over the serial port. Printf() is used for writing and can convert variables to human-readable strings suitable for sending over the serial port. Scanf() does the opposite and reads in strings of a specified format and stores them into variables. So printf() will be used for generating output and scanf() will be used for reading user input.

---

Courtesy of Andrea Di Blas and Cyrus Bazeghi
Printf() and scanf() both rely on formatting strings to specify input and output. An example of a formatting string is “%h %lf %04.3d”. The characters following the percent-sign specify what is expected in the string. For scanf() it will specify how to read the input string and for printf() it specifies how the output should be formatted. A caveat that I should mention is that %f used within a format string for scanf() uses a float while for printf() it uses a double and so floats will require a cast to a double, as shown in this example below.

Example usage of these functions follows:

```c
float j;
char g;
printf("Give me a letter and a number: ");
scanf("%c %f", &g, &j);
printf("You input %f %c", (double)j, g);
```

This code writes to standard output instructions for the user. If they respond with appropriate input then the variables j and g will have the values specified by the user. This data is then output to the user to confirm what they input.

Please note the ampersands in front of the variables passed as arguments to scanf(). These are very important. For the code that you're writing you will need one before all of the variable arguments to scanf(). You don't know or need to know the details of this right now and it will be covered later when we get to Array and Pointers. (Note for the curious: the ampersand corresponds to the memory address of the variable that follows it. So scanf() needs the memory location of the actual data in order to write the user input to these variables.)

Please also note that printf() uses a buffer to store the characters before they are then output to the appropriate location, in the case of standard out, it is normally the screen. What this means is that when printf() decides to write its buffer of characters to the screen is not necessarily when you would like it to.

Luckily, there are a few ways around this buffering problem. First, any use of a newline character ("\n") will trigger a flush of everything in the buffer before that newline. Depending on your application, you may not want to always use a newline everywhere, and that’s fine. There is a second option, setbuf(). Setbuf() takes two arguments, one is the stream handle (generally going to be stdout) and the second is a character buffer for it to use. The trick here is to use NULL as the character buffer. Setbuf() recognizes NULL as a special case that means that no buffering will then be used.

To learn more about these functions I suggest starting with section 7.2 and 7.4 of K&R. For more detail you can reference the 16-bit Language Tools Libraries from Microchip’s website, specifically section “2.13 – <stdio.h> input and output”. When writing in C the best place to find support is to read the official documentation for the compiler and any libraries you are using. In this case scanf() and printf() are from the stdio.h standard library provided by Microchip.
Program flow

You're program will loop continuously while reading and writing from the terminal. The basic outline of your program looks as follows:

Output greeting to the user
while (1)
    get operation
    get number 1
    get number 2
    do operation on numbers
    print result

The template code you have been provided in the form of lab1_stub.c includes comments that outline the different code segments listed above and should be followed when writing your code.

Program output

You may choose to tell the user as much or as little as you like but it must tell the user of all the functions your program is capable of. It needs to tell the user what mathematical operations it can perform and how to select said operation. It should also display what expression was evaluated when returning the result (as outlined in the program requirements).

Running the program

This program uses the same Proteus VSM simulator that was used for Lab 0. No alterations need to be made to the Proteus model in order to run this code. Output should be viewable in its virtual terminal as it was in Lab 0.