COMP 1631 Assignment 3 (“Multiplication Tutor”)

Given: Friday, October 2, 2020
Electronic Copy Due: Wednesday, October 21, 2020 before 11:59 pm sharp

Objectives

- to continue building your problem-solving skills;
- to learn how to define, call and test value-returning and void functions in C++;
- to practice using integer division and modular arithmetic;

Overview

The primary objective of this assignment is to gain experience in writing and using functions so most of your focus should be on that objective.

The functions you must write are described in detail later on. Because these functions have been identified for you, this effectively takes you to the stage of having completed the top-down design. A top-down design always breaks a problem down into a number of smaller sub-problems that will be implemented as functions. These sub-problems can be solved independently, and each of them is much easier to solve than the problem as a whole.

Restrictions

You may not use any program statements that have not been covered in class. In particular, you may not use any "if" statements or loops in this assignment.

The Problem

One of the drawbacks of the electronic age is the loss of the knowledge of basic techniques in arithmetic, such as addition, subtraction, multiplication, and division. With the use of electronic calculators, arithmetic has become the mere process of punching the appropriate keys and reading the answer.

A good example is multiplication. Suppose one is given the following: `123456 * 78`. It is so easy to get to a calculator and enter the two numbers and then come up with `96269568`. How many would use paper and pencil to calculate the product?

Your job is to write a program that will implement a long multiplication tutor. Specifically, the purpose of the tutor is to show the user how to arrive at the product of a 6-digit multiplicand multiplied by a 2-digit multiplier. To verify the correctness of your result, you will compute the actual product of the two numbers. To be more generic your multiplication tutor will perform multiplication on values in bases between 2 and 16.

Input:

All input for this program is from the keyboard. For each input item the program should prompt the user and then read the corresponding user input. The first input item is an integer representing the base being used. The next input is the multiplicand, which is read as **6 individual digits**. The last input item is the multiplier, which is read as **2 individual digits**. The digits of both numbers must be separated by whitespace but should appear all on one line. Both numbers will be non-negative, but either number may have leading zeros. For example, the following illustrates what should appear on the screen as input for a calculation (the user will type what appears in bold):

```
Welcome to the Multiplication Tutor.  Ready to calculate...
Enter the base: 10
Enter six digits of multiplicand (separated by whitespace): 0 2 3 1 0 1
Enter two digits of multiplier (separated by whitespace): 3 3
```

You can assume that the input will be valid. In the interest of simplicity, the symbols for the digit values greater than 9 will be `10`, `11`, `12`, etc. in the input and should be shown that way in the output.

Output:

The output should be formatted similarly to the examples below. All digits should line up. Marks will be awarded for readable output with whitespace and blank lines as shown in the output example.

Example input:

```
10
9 8 7 6 5 4
3 2
```

Corresponding output:

```
For base 10:              9    8    7    6    5    4
                                         *    3    2
               ======================================
                     1    9    7    5    3    0    8
               2     9    6    2    9    6    2
               ======================================
               3     1    6    0    4    9    2    8

Verification:

      The multiplicand in base 10 is:  987654
      The multiplier in base 10 is:    32

      987654 * 32 is equal to: 31604928

      The tutor’s product in base 10 is: 31604928
```

Here are two more examples:
Input:

```
3
1 0 2 1 2 0
2 1
```

Output:

```
For base 3:              1    0     2    1    2     0
                                         *    2     1
              =====================================
                   0    1    0     2    1    2     0
              0    2    1    2     0    1    0
              =====================================
              0    2    2    2     2    2    2     0

Verification:

      The multiplicand in base 10 is:  312
      The multiplier in base 10 is:    7

      312 * 7 is equal to: 2184

      The tutor’s product in base 10 is: 2184
```

Input:

```
16
1 3 8 10 7 11
2 12
```

Output

```
For base 16:             1     3    8   10     7   11
                                         *     2   12
              =====================================
                   0   14    10    7   13    12    4
              0    2    7     1    4   15     6
              =====================================
              0    3    5    11   12   13     2    4

Verification:

      The multiplicand in base 10 is:  1280635
      The multiplier in base 10 is:    44

      1280635 * 44 is equal to: 56347940

      The tutor’s product in base 10 is: 56347940
```

Calculations – Getting Started

There are some fairly sophisticated formulas that you will have to devise in order to solve this problem. To help in your problem solving, you should look to the decimal number system as an analogous system.

Legal sizes of single digits in the decimal number system range from `0` to `9`. Multiplication of two decimal numbers is typically carried out right to left, starting with the units digits of the multiplicand and the units digit of the multiplier. If the product of the two digits is larger than `9`, a carry is calculated and applied to the next digit of the multiplicand. For example, `6 * 8 = 48`, so we write `8` (`48 mod 10`) as the digit product and carry `4` (`48 div 10`) to the next column. If a non–zero carry remains at the end of number, the carry digit is simply prepended to the product line. The same method is followed using the hundredths digit of the multiplier. However, the second product line is indented one digit to the left. To get the final product, the corresponding columns are added, noting the rules of addition, that is, if the sum is greater than `9`, then a carry is calculated and added to the column on the left. In this case, the product will have more digits than either of the two numbers.

The above concept can be carried to multiplication in any base. Just keep in mind that the digits allowed in any base is from `0` to (`base-1`). Do some simple samples manually to convince yourself that this is indeed the case.

Because of the restrictions in this assignment, you may not use ***if*** statements or loops so you should provide for enough variables to cover all the digits you will need.

To check the correctness of the multiplication, the following operations are to be performed:

1. convert the 6 digits of the multiplicand to a single number in decimal
2. convert the 2 digits of the multiplier to a single number in decimal
3. multiply the two together to get the calculated decimal product
4. convert the digits of the tutor's product to a single number in decimal

When the numbers from steps 3 and 4 are printed, you should observe that they match, if multiplication was done properly. If they don't, congratulations – you have a bug (or 10_2) to wrestle with!

Detailed Specifications:

Design and code the following functions for your program.

1. int multiplyProduct (int multiplicand, int multiplier, int carry_in, int base)
   Receives four integers:

multiplicand	a single digit from the number being multiplied (the top number)
multiplier	a single digit from the number doing the multiplying (the bottom number)
carry_in	the excess from a previous multiplication
base	the base of the numbers being multiplied

   The function returns the ones digit of the product of this multiplication.

2. int multiplyCarry (int multiplicand, int multiplier, int carry_in, int base)
   The function receives the same inputs as function 2, but returns the carry out from the multiplication.

3. int addSum (int augend, int addend, int carry_in, int base)
   Receives four integers:

augend	a single digit from the top number being added
addend	a single digit from the bottom number being added
carry_in	the excess from the previous addition
base	the base of the numbers being added

   The function returns the ones digit of the sum of this addition.

4. int addCarry (int augend, int addend, int carry_in, int base)
   The function receives the same inputs as function 4, but returns the carry out from the addition.

5. int expand (int digit, int base, int position)
   Receives four integers:

digit	a single digit in a number
base	the base of the number that **digit** is part of
position	the position **digit** in the number

   The function returns the actual value of this digit within the number as a base 10 value.
   Example:

```
expand(3, 10, 3);     would return 3000

expand(1, 2, 5)       would return 32
```

No changes can be made to any function's interface – this is a fancy way of saying the parameters a function receives and value it returns (if any) *must* be as described!

The main program is responsible for calling the appropriate functions to generate the desired output; that is, it should:

1. Write out the introductory message.
2. Read in the data required by the program.
3. Perform the multiplication of the two numbers. This will require calling the appropriate functions to perform the desired calculations.
4. Write the output of the multiplication corresponding to the above examples.
5. Perform the calculation for the verification of the multiplication. This will require calling the appropriate functions to perform the desired calculations.
6. Write the output of the verification in the format prescribed in the above examples.

The Utility Library

You are being provided with a programmer created library, called *utility.o*, that contains two functions. The first function, called prettyPrint, will print out an integer right–justified within a specified field width. The documentation below explains how to use this function.

```
// Function Name:  prettyPrint (width, value)
// Input:     an integer representing the field width
//            an integer representing the number to be outputted
// Action:    the number is outputted to the standard output
//            right-justified in the specified width (leading
//            blanks are printed if necessary)
//          e.g.  prettyPrint (5, 2); would produce the output
//                       □□□□2     where □ represents a blank

//                prettyPrint (4, -8); would produce the output
//                       □□-8

//                prettyPrint (5, 2);
//                prettyPrint (5, -8);   would produce the output
//                       □□□□2□□□-8
```

The second function, called intPower, performs integer exponentiation, whose documentation is:

```
// Function Name:  intPower (value, exp)
// Input:  an integer representing the base value
//         an integer representing exponent value
// Return: the base value raised to the exponent
//   e.g.  intPower (5, 2); would return the value 25 (i.e. 5^2)
//         intPower (2, 5); would return the value 32
//         intPower (-5, 3); would return the value -125
//         intPower (5, -8); is invalid since the result is not
//                   an integer value and will return the value of 0
```

To use these functions, you will need to copy the files *utility.h* and *utility.o* from the course directory (`/users/library/comp1631/assignments/a3/`) to your home directory. Then, include *utility.h* in your program file by including the following line before the #include of iostream:

```
#include "utility.h"
```
this line goes at the top of your program

When compiling the executable *utility.o* needs to combined with your program file. This is done by using the following command:

```
g++ myprog.cpp utility.o
```
this is how to compile your program

Suggested Approach to Development:

When it comes to coding the program, it is important to code, debug and test in small pieces. Thus, always break the problem into pieces. Use the list of required functions as your starting point. When creating a program using functions, you can choose to work **top-down** or **bottom-up**. You are free to choose whichever method you feel more comfortable using – in fact, it is not uncommon to use both methods as you are working towards a functioning solution.

For the **top-down** approach, you will start with the main program. The main problem breaks down naturally into a sequence of steps. Begin by writing down, at a high level, the steps of the main program. When a function needs to be called, write the actual function call using the correct name and the actual input arguments. At this stage, don't worry about *how* that function will perform the required task—to get your program to work, write each function as a ***stub***. Each ***stub*** will have the correct function name and formal arguments, but the body will be a statement outputting the function name (and if it is a value-returning function it will return an appropriate default value).

Once you are satisfied with the sequencing and flow of control for the main program, tackle the design for each function *one at a time.* Repeat the same process used for the main program: write down the sequence of steps for the function, verifying that you have identified all the required inputs. This is the time to make changes to the main program plan if you discover missing input arguments. Realize that when coding a function you may need to create additional ***stubs***, i.e. when implementing function A, which calls two additional functions B and C, you would need a stub for each of them.

For the **bottom-up** approach, you will start with some function from the list of required functions. You will need to create a dummy main program - a ***driver***. For each new function, the ***driver*** will have to be altered to include one or more calls to the new function that provides appropriate actual parameters so that the function logic can be tested.

Regardless of whether you use a **top-down** or **bottom-up** approach, you should realize that dependencies between functions will impose an order on the creation of the functions. For example, suppose function A depends on function B, which depends on function C which depends on function D. This dependency chain does not specify the order in which these functions should be implemented - they can be done in any order. However, the order you choose may require the use of stubs for any unimplemented functions.

Note: Start work on the assignment early. Don't wait until the last minute; leave yourself plenty of time to solve unexpected problems with the C++ language.

Hand In:

Specific submission instructions and test data will be provided a couple of days before the assignment is due.