Overview

Bjarne Stroustrup (the inventor of C++ and former Texas A&M Professor) is jealous of Python's support for arithmetic with unbounded integers. Help Bjarne to feel better about C++ by building a calculator to do basic arithmetic operations with arbitrarily large numbers represented internally as strings, i.e., sequences of characters.

For this assignment, you will need to implement **addition** (same signs only) and **multiplication** using strings. The starter code from Mimir will provide a framework with comments in main.cpp and string_calculator.cpp where you will implement your code.

Example Run of the Calculator

```
String Calculator
"q" or "quit" or ctrl+d to exit

>> 1371195958099968000 * 191898783962510625

ans =

    263130836933693530167218012160000000

>> 82228386541779228177255628800000000 +
263130836933693530167218012160000000

ans =

    271353675587871452984943575040000000

>> quit

farvel!
```

Objectives

1. Practice using C++ string objects
2. Practice core C++ concepts
3. Practice problem-solving
    1. Implement standard arithmetic algorithms
    2. Create or discover novel algorithms

Requirements

1. The source files to submit are named

    A. main.cpp
    B. string_calculator.h (you should read this file)
    C. string_calculator.cpp

2. You are required to implement the following functions, which are declared and described in the string calculate header file, which you should read.

```
unsigned int digit_to_decimal(char digit)
```

```
digit_to_decimal('7') should return 7
```

```
digit_to_decimal('/') should throw std::invalid_argument
```

```
char decimal_to_digit(unsigned int decimal
```

```
decimal_to_digit(7) should return '7'
```

```
decimal_to_digit(36) should throw std::invalid_argument
```

```
std::string trim_leading_zeros(std::string num)
```

```
trim_leading_zeros("00123") should return "123"
```

```
trim_leading_zeros("-00123") should return "-123"
```

```
std::string add(std::string lhs, std::string rhs)  //same signs only, i.e., two positives or two negatives
```

```
add("1", "2") should return "3"
```

```
add("-1", "-2") should return "-3"
```

```
std::string multiply(std::string lhs, std::string rhs)
```

```
multiply(“2”, “3”) should return “6”

multiply(“2”, “-3”) should return “-6”

multiply(“-2”, “3”) should return “-6”

multiply(“-2”, “-3”) should return “6”
```

3. You may use the following includes:

   A. iostream
   B. limits
   C. sstream
   D. string
   E. string_calculator.h

4. The program must compile without warnings or errors.

```
g++ -std=c++17 -Wall -Wextra -pedantic -Weffc++ string_calculator.cpp main.cpp
```

5. The program must run without errors or unhandled exceptions.

   How to read:
   - user input in **bold red**; everything else is output
   - ␣ is a space character (‘ ’)
   - ↲ is a newline character (‘\n’, also displayed as)

   Sample run

```
$␣./a.out
String␣Calculator↲
"q"␣or␣"quit"␣or␣ctrl+d␣to␣exit↲
>>␣1␣+␣1↲
↲
ans␣=↲
↲
␣␣␣␣2↲
↲
>>␣quit↲
↲
farvel!↲
↲
```

The Problem-Solving Mindset

As with all homework assignments, the goal of String Calculator is to help you develop critical problem solving skills through applied programming experience. But you don't have to be a master programmer to start thinking like one!

At the root of each problem is a simple question: "What set of steps (or, algorithm) do I need to make this happen?" Oftentimes, a good place to start is with steps you naturally use when doing the problem yourself.

For example, in Mountains and Valleys, you had to determine if a sequence of digits represented a mountain or valley range based on direction changes. If you were given a number hundreds of digits long and asked to check if it were a mountain or valley range, what would you do? You would probably start checking digit-by-digit for direction changes. That is essentially the number slicing technique we recommended!

It's also the reason we discouraged solutions that were hard-coded for 3 or 4 digits. As humans, we might make simplifications like that, but you will find thinking at a large scale (e.g., hundreds of digits) can help you more clearly recognize the steps on a small scale (e.g., just three digits) and put you in a better mindset for developing algorithms.

In this homework, you will be forced to think at scale because we are working with strings, some of which will be *very* long. You need to implement algorithms that can iterate through strings of any size and apply the appropriate operations. So what to do?

Let's consider how you might tackle this problem on your own before thinking about the code. Note: this is not a new technique. Recall that we talked about the simple software development process: **Analyze** → **Design** → **Implement**. All we are doing here is to analyze the problem by thinking through the steps you might want to perform. From there, you can sketch out your design before beginning implementation details.

Long Addition

It's probably been some time since you sat down and performed an addition problem by hand, but the same steps still apply. When you perform long addition, you stack the numbers, add column-by-column, and carry forward a digit when the sum is 10 or more.

For example, let's add the numbers 47349 and 6232:

```
          1     1   ← Carry
        4 7 3 4 9
      +   6 2 3 2
      -----------
Result:  5 3 5 8 1
```

If we look at each individual step, we can see that we need only to traverse the two operands from right-to-left, adding as we go. We'll need something to store the carry, and a place to store the result.

Now that we have the general idea, translating to code is a matter of identifying the variables and mechanisms needed to do these steps. Think about data types, loops, and the conditional logic you might need at each step. If you have to write down the actions in more detail to get a better understanding, go for it!

If you want more practice, CalculatorSoup provides an excellent [Online Long Addition Calculator]

which breaks down the steps for any operands you provide.

Long Multiplication

At first, multiplication might sound more difficult than addition, but it will be much the same in that you are still going to iterate over the operands digit-by-digit and apply the operation (in this case, multiply). The main difference in long multiplication is the inclusion of partial products.

Let's consider the product of 4512 and 28. We start by computing the partial product for the 1's digit. Note that at each step, we're just multiplying the 1's digit of the lower number, 8, with each digit of the upper number, and adding the carry.

```
        3 4   1   ← Carry for the 1's digit: 8
    4 5 1 2
*       2 8
-----------
  3 6 0 9 6 ← Partial Product for the 1's digit: 8
```

Next, we'll repeat the operation for the 10's digit, 2:

```
          1       ← Carry for the 10's digit: 2
    4 5 1 2
*       2 8
-----------
  3 6 0 9 6 ← Partial Product for the 1's digit: 8
  9 0 2 4 0 ← Partial Product for the 10's digit: 2
```

By this point, we've uncovered a couple of special considerations:

1. It's evident that we are doing two levels of iteration, so we might need more than just a single loop to:

    1. multiply a digit in the bottom number by each digit of the top number and
    2. traverse all digits in the bottom number.
2. With each new digit of the bottom number, we have to account for its order (1's, 10's, etc.) by placing the correct number of zeros at the right.

We're basically at the end of the calculation. All that remains is to add the partial products, but we already have an add function!

```
          3 6 0 9 6
      + 9 0 2 4 0
      -----------
Result: 1 2 6 3 3 6
```

By leveraging the addition function that is already complete, we can sum the partial products as we go, simplifying the multiplication function greatly.

Again, CalculatorSoup has a helpful Online Long Multiplication Calculator

with these steps and visualizations.

Handling the Signs

The examples we've considered are with positive numbers, and that will get you nearly all the points in Mimir. For the base requirements, handling signs requires only very little work.

In multiplication, you only need to check if signs are the same (result is positive) or different (result is negative). In addition, we will only test same-sign numbers in the base requirements. If both are positive, the result is positive, and if both are negative, the result is negative.

The algorithms themselves are the same. You will just need to check if the number begins with '-' to determine negative or not, and remember it to update the final result.

Next Steps

These examples demonstrate one set of operations for adding and multiplying a digit at a time. There are many alternatives; for instance, look up Left-to-Right Addition.

You may have your own idea for a novel algorithm to do the task. Pursue it! Implementing your ideas is a great way to develop your problem solving skills. If you

want an additional challenge, consider writing the subtraction algorithm. Not only can you earn bonus points, but you can make a more full-featured calculator.