CS205 C/C++ Programming Project 1 A Simple Calculater

Name: 康耀中(KANG Yaozhong)

SID: 12110225

Part 1 - Analysis

1.1Input Filter

1.1.1 Input Container

The problem is to calculate the sum of two unclassified numbers `number_one` and `number_two`. `number_one` and `number_two` could be either integers or floating-point numbers.

- When the number is an integer, there is only one possible expression for a decimal number:

  $$(+/-)a[1]a[2]...a[n] \tag{1}$$

  Where `a` and `b` are simple 0~9 integers.
  For example, `114514`.

- When the number is a floating-point number, there could be several expressions:

  $$(+/-)(a[1]a[2]...a[n]).(b[1]b[2]...b[n]) \tag{2}$$

  Where `a` and `b` are simple 0~9 integers.
  For example, `19.1981`.

  $$(+/-)(a[1]a[2]...a[n]).(b[1]b[2]...b[n])(E/e)(+/-)(c[1]c[2]...c[n]) \tag{3}$$

  Where `a` and `b` are simple 0~9 integers.
  For example, `-19.1981e-100`.

At the same time, there should only be two numbers as inputs. If the size of the input string array is more of less than three, an error should be thrown.

There are several solutions to identify and calculate the numbers:

1. Transfer the input into specific kind of basic data package using `atoi()`,`atof()`.
2. Modify the input string directly.

If we want to hold an enormous input number,and at the same time, customize the input tolerance, directly modifying the input string is the better choice.

1.1.2 Regex Judgement

At first, to use a series of if-else judgments to distinguish which specific kinds of input numbers was preferred. However, because of enormous input possibilities, the judgments became overstaffed and it was hard to maintain the coding structure.

In case of that, regex judgement is a better choice. All the inputs can be classified into nine cases. The inputs that don't correspond to the nine cases will throw input errors.

1.1.3 Separation

After one input pass the regex judgement, it will be separated and be stored as two data `<effective part>` and `<exponential part>`, following the rules of its specific kind. The detailed descriptions of nine regex expressions will be listed in the source code part.

As the input pass the regex judgement, the program will check and record the `+` `-` `.` and `e`(or `E`) position in the string. Using those information, the program will store the effective part, the exponential part and the sign of number.
For example：
`123` will be saved as `123` and `0`,positive;
`+12.345` will be saved as `12345` and `-3`,positive;
`-12.345e+123` will be saved as `12345` and `120`,negative...

Actually, the program doesn't care what kind of data the input is. It will treat them as combinations of two integers equally.

1.2 Multiplication

1.2.1 The big integer multiplication

The multiplication reminds me of my java times when I had to write an calculator for big binary numbers. The basic idea is to use vertical operations.

e.g.

$$\begin{array}{r} 123 \\ \times 23 \\ \hline 369 \\ 246 \\ \hline 2829 \end{array} \tag{4}$$

Using "for" loop, we constantly do the multiplying, note down the reserving number and adding operations. The specific code will be listed below.

In case of the binaries, there are plenty of ways to reduce the time complexity. In decimal multiplication, I don't know how to do that. Those tasks should be considered in a **DSAA** class.

1.2.2 Add the exponential part

We add the exponential part of the two numbers contained in `long long`.

After we do the processing part, we will get two data. String `<effective part>` and Integer `<exponential part>` . The next thing we have to do is to print the result.

1.3 Print the result

The program will check the size of `<exponential part>`. When `<exponential part>` is smaller than `0` but bigger than `-10`, a `.` will be added at the specific position. When `<exponential part>` is smaller than `-10` or bigger than `0`, the result will be printed in scientific way, which will be added by `+` or `-` and `e`. When the `<exponential part>` is `0`, the program will print the `<effective part>` directly.

Part 2 - Code

> Check the input size

```
int main(int argc, char *numbers[])
{

    if(argc != 3)
    {
        cout << "You can only enter two numbers" << endl;
        return 0;
    }
    //store the first number
    string number_one = numbers[1];
    //store the essential part of the first number
    string number_one_critical;
    //store the exponential part of the first number
    long long number_one_tens = 0;
    //store the sign of the first number
    bool one_positive = true;
    string number_two = numbers[2];
    string number_two_critical;
    long long number_two_tens = 0;
    bool two_positive = true;
    //the nine kinds of input
    //1.positive number with 0 exponential part
    regex positive_int_reg("[+]?[1-9][0-9]*");
    //2.zero
    regex zero_reg("[+-][0][.]?[0]*");
    //3.positive number with 0 exponential part
    regex negative_int_reg("[-][1-9][0-9]*");
    //4.positive floating-point number
    regex positive_float_reg("[+]?[1-9][0-9]*[.][0-9]+");
    //5.negative floating-point number
    regex negative_float_reg("[-][1-9][0-9]*[.][0-9]+");
    //6.positive floating-point scientific expressed number with negative exponential
part
    regex positive_scientific_float("[+]?[1-9][0-9]*[.]?[0-9]*[Ee][-][0-9]+");
    //7.negative floating-point scientific expressed number with negative exponential
part
    regex negative_scientific_float("[-][1-9][0-9]*[.]?[0-9]*[Ee][-][0-9]+");
    //8.positive floating-point scientific expressed number with positive exponential
part
    regex positive_scientific_exp("[+]?[1-9][0-9]*[.]?[0-9]*[Ee][+]?[0-9]+");
    //9.negative floating-point scientific expressed number with positive exponential
part
    regex negative_scientific_exp("[-][1-9][0-9]*[.]?[0-9]*[Ee][+]?[0-9]+");
```

> some typical cases

```
    if(regex_match(number_one,positive_int_reg))
    {
        //cout<<"positive int"<<endl;
        if (number_one[0] == '+')
        {
            number_one.erase(0,1);
        }
        number_one_critical = number_one;
    }else if(regex_match(number_one,zero_reg))
    {
        //cout<<"zero_reg"<<endl;
        number_one_critical = "0";
    }else if(regex_match(number_one,negative_int_teg))
    {
        //cout<<"negative_int_teg"<<endl;
        number_one.erase(0,1);
        number_one_critical = number_one;
        one_positive = false;
    }else if(regex_match(number_one,negative_float_reg))
    {
        //cout<<"negative_float_reg"<<endl;
        number_one.erase(0,1);
        for(int i = 0; i < number_one.length(); i++)
        {
            if(number_one[i] == '.')
            {
                number_one_tens = -(number_one.length() - i - 1);
                number_one.erase(i,1);
            }
        }
        number_one_critical = number_one;
        one_positive = false;
    }else if(regex_match(number_one,positive_float_reg))
    {
        if(number_one[0] == '+')
        {
            number_one.erase(0,1);
        }
        for(int i = 0; i < number_one.length(); i++)
        {
            if(number_one[i] == '.')
            {
                number_one_tens = -(number_one.length() - i - 1);
                number_one.erase(i,1);
            }
        }
        number_one_critical = number_one;
        one_positive = false;
        //cout<<"positive_float_reg"<<endl;
    }
    ...
    else if(regex_match(number_one,positive_scientific_exp))
    {
        int dot_position = 0;
        int e_position;
        string e_string;
        if(number_one[0] == '+')
        {
            number_one.erase(0,1);
        }
        for(int i = 0; i < number_one.length(); i++)
        {
            if(number_one[i] == '.')
            {
                dot_position = i;
            }
            if(number_one[i] == 'e' || number_one[i] == 'E')
            {
                e_position = i;
                if (dot_position != 0)
                {
                    number_one_tens = - (e_position - dot_position - 1);
                }
                if(number_one[i+1] == '+')
                {
                    number_one.erase(i+1,1);
                }
                break;
            }
        }
        e_string = number_one.substr(e_position + 1, number_one.length() - e_position);
        for (int i = 0; i < e_string.length(); i++)
        {
            //cout << e_string << endl;
            number_one_tens += (e_string[i]-'0')*pow(10, e_string.length() - i -1);
            //cout << number_one_tens << endl;
        }
        if (dot_position != 0)
        {
            number_one.erase(dot_position,1);
            number_one_critical = number_one.substr(0, e_position - 1);
        }else
        {
            number_one_critical = number_one.substr(0, e_position);
        }
        cout<<"positive_scientific_exp"<<endl;
    }
```

> multiplication section

```
    int sum_critical[100];
    for(int i = 0; i < 100; i++)
    {
        sum_critical[i] = 0;
    }

    for (int i = 0; i < number_one_critical.length(); i++)
    {
        int temp = 0;
        for (int j = 0; j < number_two_critical.length(); j++)
        {
            sum_critical[i+j] += (number_one_critical[number_one_critical
            .length() - i - 1]-'0') * (number_two_critical[number_two_critical.length()
- j - 1]-'0') + temp;
            temp = sum_critical[i+j]/10;
            sum_critical[i+j] %= 10 ;
        }
        sum_critical[i+number_two_critical.length()] += temp;
    }
    // calculate tens
    string sum;
    int index = number_one_critical.length() + number_two_critical.length();
    while(sum_critical[index]==0 && index>0) index--;
    cout << index << endl;
    for(int i=index; i >=0; i--)
    {
        sum.append(to_string(sum_critical[i]));
    }
```

> Recombination Part

```
    long long final_tens = number_one_tens + number_two_tens;
    if(final_tens < 0 && final_tens >= -10)
    {
        //cout<< sum.length()<<endl;
        sum.insert(sum.length()+final_tens,1,'.');
    }else if(final_tens < -10)
    {
        long long sum_length = sum.length();
        sum.insert(1,1,'.');
        final_tens += sum_length-1;
        for(int i = sum.length();;i--)
        {
            if (sum[sum.length()-1] == '0')
            {
                sum.erase(sum.length()-1);
                if (sum[sum.length()-2] == '.')
                {
                    break;
                }
                continue;
            }
            break;
        }
        sum.insert(sum.length(),1,'e');
        sum.insert(sum.length(),1,'+');
        string final_ten_string = to_string(abs(final_tens));
        //cout << int(log10(abs(final_tens))) << endl;
        for (int i = 0; i < (int)log10(abs(final_tens))+1; i++)
        {
            int a = pow(10,i+1);
            sum.append(to_string(final_ten_string[i]-'0'));
        }
    }else if (final_tens > 0)
    {
        long long sum_length = sum.length();
        sum.insert(1,1,'.');
        sum.insert(sum.length(),1,'e');
        sum.insert(sum.length(),1,'+');
        final_tens += sum_length-1;
        string final_ten_string = to_string(abs(final_tens));
        for (int i = 0; i < (int)log10(final_tens)+1; i++)
        {
            int a = pow(10,i+1);
            sum.append(to_string(final_ten_string[i]-'0'));
        }
    }
    if(one_positive^two_positive)
    {
        sum.insert(0,1,'-');
    }
```

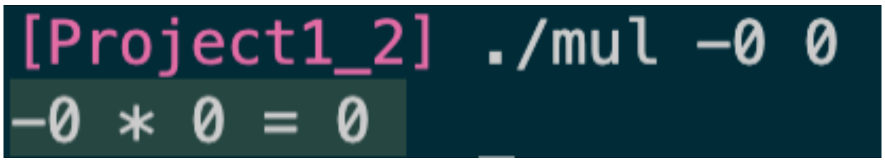
> Print the final result

```
cout << numbers[1] <<" * "<< numbers[2]<< " = " << sum << endl;
```

Part 3 - Result & Verification

Test case#1

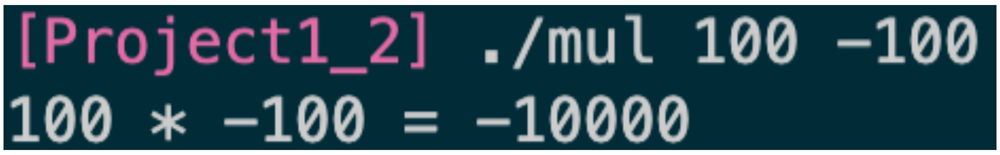
```
    Input: -0 0
    Output: -0 * 0 = 0
```



```
[Project1_2] ./mul -0 0
-0 * 0 = 0
```

Test case#2

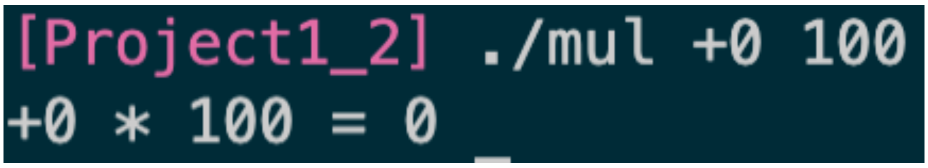
```
    Input: 100 -100
    Output: 100 * -100 = -1000
```



```
[Project1_2] ./mul 100 -100
100 * -100 = -10000
```

Test case#3

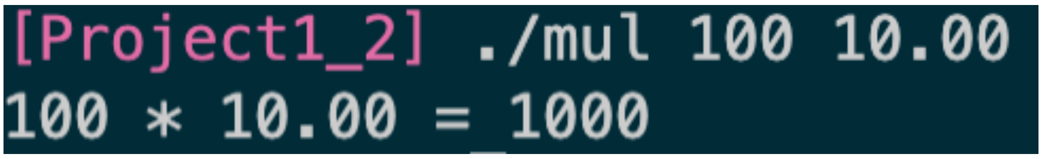
```
    Input: +0 100
    Output: +0 * 100 = 0
```



```
[Project1_2] ./mul +0 100
+0 * 100 = 0
```

Test case#4

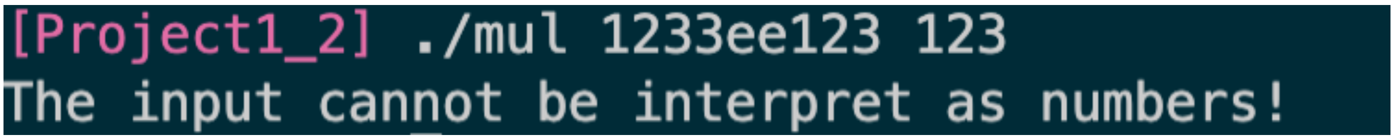
```
    Input: 100 10.00
    Output: 100 * 10.00 = 1000
```



```
[Project1_2] ./mul 100 10.00
100 * 10.00 = 1000
```

Test case#5

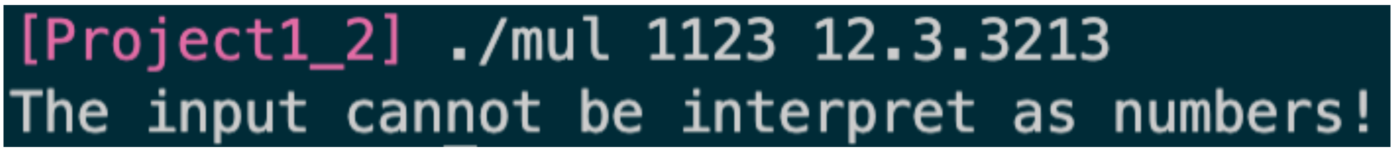
```
    Input: 1233ee123 123
    Output: The input cannot be interpret as numbers!
```



```
[Project1_2] ./mul 1233ee123 123
The input cannot be interpret as numbers!
```

Test case#6

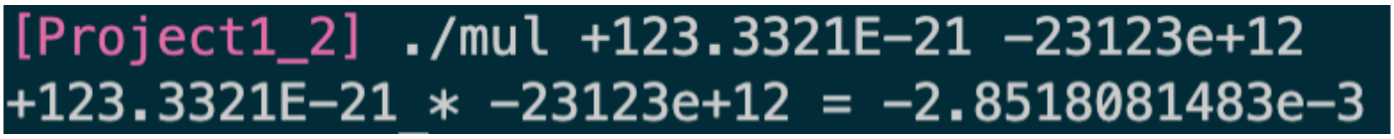
```
    Input: 1123 12.3.3213
    Output: The input cannot be interpret as numbers!
```



```
[Project1_2] ./mul 1123 12.3.3213
The input cannot be interpret as numbers!
```

Test case#7

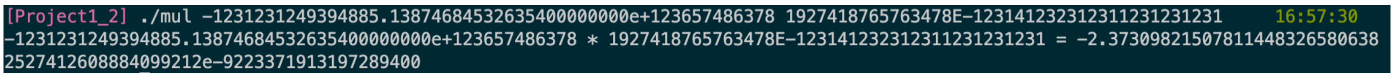
```
    Input: +123.3321E-21 -23123e+12
    Output: +123.3321E-21 * -23123e+12
    = -2.8518081483e-3
```



```
[Project1_2] ./mul +123.3321E-21 -23123e+12
+123.3321E-21 * -23123e+12 = -2.8518081483e-3
```

Test case#8

```
    Input: -1231231249394885.13874684532635400000000e+123657486378 1927418765763478E-
1231412323123112312312310
    Output:
    -1231231249394885.13874684532635400000000e+123657486378 * 1927418765763478E-
123141232312311231231231 = -2.37309821507811448326580638252741260888409921 2e-
9223371913197289400
```



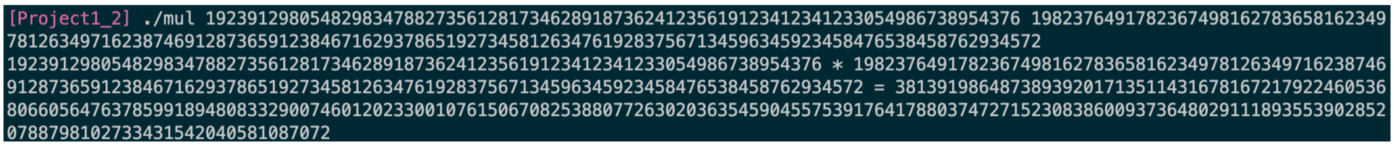
```
[Project1_2] ./mul -1231231249394885.13874684532635400000000e+123657486378 1927418765763478E-123141232312311231231231      16:57:30
-1231231249394885.13874684532635400000000e+123657486378 * 1927418765763478E-123141232312311231231231 = -2.373098215078114483265806382527412608884099212e-9223371913197289400
```

Test case#9

The length of the critical answer is set as 10000 digit.(Actually, the digits cannot be more than the size of long long)

```
    Input:
    192391298054829834788273561281734628918736241235619123412341233054986738954376

198237649178236749816278365816234978126349716238746912873659123846716293786519273458126
347619283756713459634592345847653845876293457 2
    Output:
    192391298054829834788273561281734628918736241235619123412341233054986738954376 *
198237649178236749816278365816234978126349716238746912873659123846716293786519273458126
3476192837567134596345923458476538458762934572 =
381391986487389392017135114316781672179224605368066056476378599189480833290074601202330
010761506708253880772630203635459045575391764178803747271523083860093736480291118935539
0285207887981027334315420405810870 72
```



```
[Project1_2] ./mul 192391298054829834788273561281734628918736241235619123412341233054986738954376 1982376491782367498162783658162349
781263497162387469128736591238467162937865192734581263476192837567134596345923458476538458762934572
192391298054829834788273561281734628918736241235619123412341233054986738954376 * 198237649178236749816278365816234978126349716238746
912873659123846716293786519273458126347619283756713459634592345847653845876293457 2 = 381391986487389392017135114316781672179224605 36
806605647637859918948083329007460120233001076150670825388077263020363545904557539176417880374727152308386009373648029111893553902852
0788798102733431542040581087072
```

Part 4 - Difficulties & Solutions

The first difficulty is how to judge the correctness of the input. For example, input can contain `E` or `e` and you should throw an error when there's a second `e`, and for the next digit you can allow `+` . There are desperately too much of the situations, but thankfully, it can be solved by using `regex`.

The second difficulty is how to do the calculation between different kinds of numbers, which is solved by treating them the same and turn them into the same kind of data.

The third difficulty is to do the big number calculation, which is solved by doing big integer calculation following the memory of doing those in java.

The fourth difficulty is to consider the output style of the result. For example, `0.000` should be shortened as `0`. By considering a lot of situations, the result became reasonable.

The source code contains lots of repeating parts, which can be solved by using functions. However, it is hard for functions to return more than one value in c++. Thus I kept my source code verbose so that I can get control of everything.