CS205 Project 1 - A Simple Calculator

Name: Wang Haoyu

SID: 11911134

Part 1 - Analysis

1.1 Requirements

In this project, we aim to implement a calculator which can multiply two numbers. When we run the program, it is supposed to output the expression of the mutiplication and the result.

There are some basic requirements for the calculator.

> The two numbers to be multiplied should be input through the command line arguments.
>
> It can multiply two integers.
>
> It can multiply floating-point numbers.
>
> It can multiply an integer and a floating-point number.
>
> It can multiply two big integers.
>
> It can multiply two big floating-point numbers.
>
> It can tell and give a warning when the input is not a number.

1.2 Solving Idea & Basic Steps

1.2.1 Step 1 - Write restricted conditions.

Overall Logic

To achieve these requirements, I chose to input the two numbers as strings. Firstly, I wrote some restricted conditions for the input. The requests for the input strings are shown below:

- Each char in the string must be a number `0`-`9` or `e` or point `.` or the negative sign `-`.
- The string can only have one point `.` and one `e` and the position of the point `.` or `e` cannot be at the first position of the string.
- The string cannot be a single point `.` or a single `e` or `.e` or `e.` .
- If the string have both point `.` and `e`, the position of the point `.` must be in front of the position of `e` .
- The number of the input to be multiplied must be two, not one or three or more.
- The negtive sign can only at the fisrt place of the string or a bit after the `e` .

Details of Implements

- **Each char in the string must be a number `0`-`9` or `e` or point `.` or the negative sign `-`.**

  I refered the ASCII Table(7-bit) to find the corresponding integer to the char. I used `for` loops to judge each char of the strings. If the integer value of the char is not belong to the decimal number below, this request will not be satisfied. If all the chars satisfy the condition, the request is satisfied. For the negative sign, I used the `‘-’` directly.

DEC	OCT	HEX	BIN	Symbol	HTML	Discription
48	060	30	00110000	0	&#48;	Zero
49	061	31	00110001	1	&#49;	One
50	062	32	00110010	2	&#50;	Two
51	063	33	00110011	3	&#51;	Three
52	064	34	00110100	4	&#52;	Four
53	065	35	00110101	5	&#53;	Five
54	066	36	00110110	6	&#54;	Six
55	067	37	00110111	7	&#55;	Seven
56	070	38	00111000	8	&#56;	Eight
57	071	39	00111001	9	&#57;	Nine
101	145	65	01100101	e	&#101;	Lowercase e
46	056	2E	00101110	.	&#46;	Period, dot or full stop

- **The string can have only one point `.` and one `e` and the position of the point `.` or `e` and the position of the point `.` or `e` cannot be at the first position of the string.**

  Use `if` statements to judge this condition. If the string has `.` , test the position of the first point`.` and the position of the last point`.` . If the first position is the same as the last position, it means that there is only one point in the string, which satisfy the request. The judgement of `e` is similar to point `.` .

- **The string cannot be a single point `.` or a single `e` or `.e` or `e.` .**

  If the string is one of the four expressions, the request is not satisfied.

- **If the string have both point `.` and `e`, the position of the point `.` must be in front of the position of `e` .**

  Judge the position of `.` and `e`. If the position number of `.` is less than the position number of `e`, the request is satisfied.

- **The number of the input to be multiplied must be two, not one or three or more.**

  Since we input the two numbers through the command line arguments, we can easily get the number of the input strings. Refer to the code `int main(int argc, char **argv)`, if `argc` is not equal to 3, the request is not satisfied.

- **The negtive sign can only at the fisrt place of the string or a bit after the `e` .**

  Do judgements for each string. If the string has more than two negative signs then output the warning `The input cannot be inerpret as numbers!`. If the string has one negative sign, it can only be at the first place or one place after `e`. If the string has two negative sign and the string has `e`, the first negative sign must be at the first place and the last negative sign must be one place after `e`.

1.2.2 Step 2 - Make judgement.

If the input strings do not satisfy the requests, the program will output `The input cannot be interpret as numbers!` or `Please input two numbers.` if the number of input to be multiplied is not two. If the input strings satisfy all the requests, the program will do the calculation part.

1.2.3 Step 3 - Calculate.

Overall Logic

I divided the possible conditions into three main parts:

- The first part is the multiplication of two integers or two big integers. Because using `int` type may sometimes cause an overflow so I used `long long` type to initialize the variables.
- The second part is the multiplication of two floating-point numbers or an integer and a floating-point number.
- The third part is the multiplication of big numbers, containing `e` that represent powers of ten. It can do multiplication of two big floating-point numbers or two big integers. It can also multiply an integer or a floating-point number with a big floating-point number or a big integer, which use `e` to represent.

Details of Implement

- **Part 1**

  Firstly we can make a judgement. According to the constraints in the first step, if there is no point `.` or `e` in the strings, the two strings are both composed of numbers, then transform the strings from `string` type to `long long` type and do the integer multiplication. At first, I was going to judge whether the first number is zero or not. However, I found the compiler will ignore the zeros before the first non-zero number and then do the calculation. So I did not do this judgement.

- **Part 2**

  First of all, make a judgement whether the strings have a floating point and `e`. If at least one string has a point and both of them do not contain`e`, then execute the following codes in this part.

  After entering this part, judge whether the string is an integer. I make the judgement separately. If the string does not have a point and only has numbers, then it is an integer and its format will be changed. In order to do multiplication of floating-point numbers, we change the format from integer to floating-point number through adding `.` at the end of the string. In that case, the numbers in the strings are floating-point numbers.

  When the numbers in the two strings are both floating-point numbers, change the strings from `string` type to `double` type. Then do the multiplicaion of two floating-point numbers and we can get the result.

- **Part 3**

  If at least one string has a char `e` , then execute the following codes in this part. I divided this part into three subparts. The first subpart is that the first string do not has `e` and the second string has `e`. The second subpart is that the first string has `e` and the second string do not has `e`. The third subpart is that both two strings have `e`. Since the first subpart and the second subpart are similar, I just illustrate the first subpart and the third subpart. In each subparts, I judge the position of `e` in the string which contains `e` . If `e` is the last char, then output the warning `The input cannot be inerpret as numbers!`. If not, the program will execute the following codes.

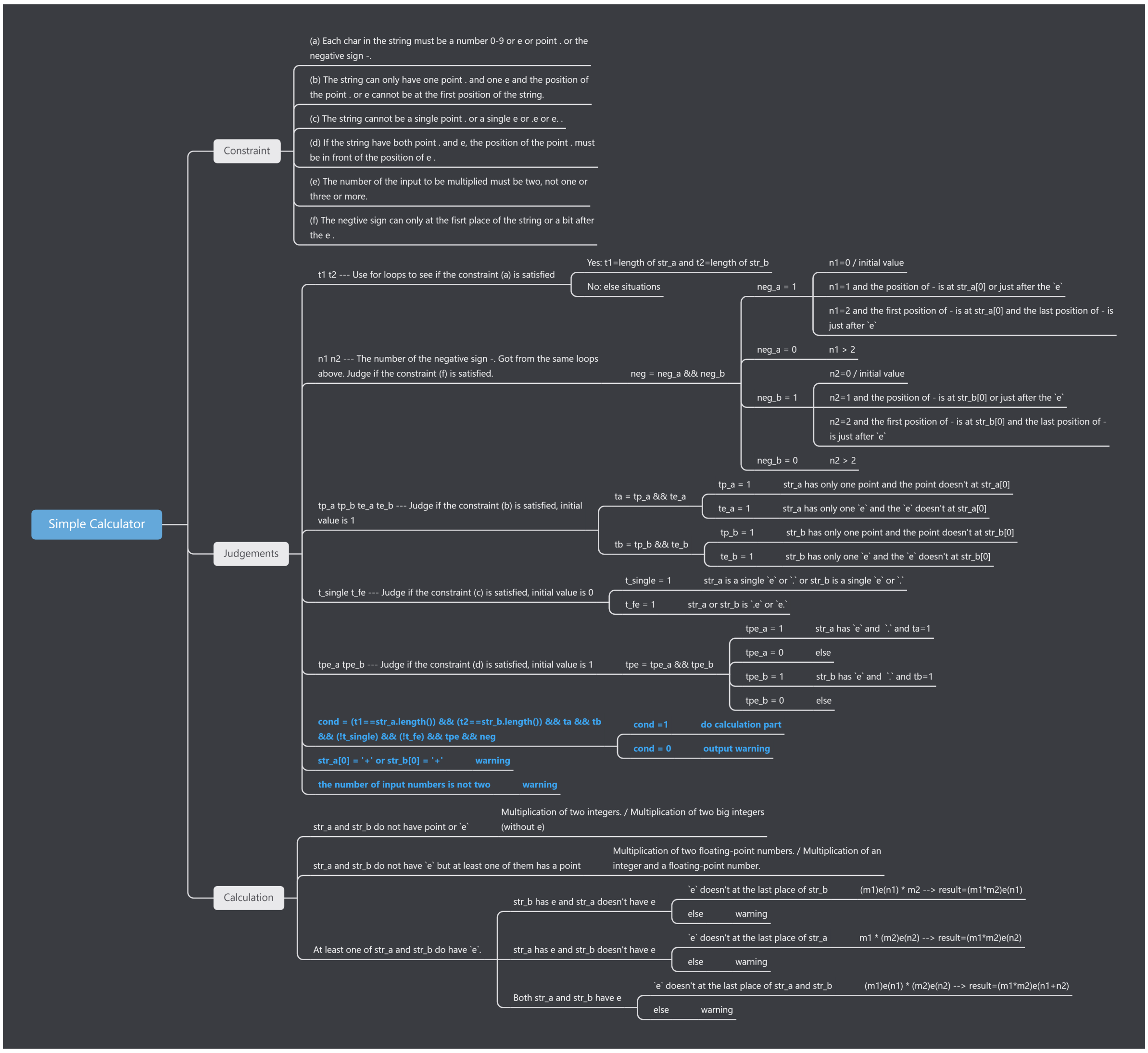
For the first subpart, because the first string does not have `e` , so the number in the first string can only be an integer or a floating-point number. We can divide the second string into two parts. The first part is the substring before `e` and the second part is the substring started from `e` to the last char. Then transform the first string and the first substring into `double` type and do the multiplication of the two floating-point numbers. After that, we can transform the result from `double` to `string`. Finally, connect the result string and the second substring together then we get and output the final result string

For the third subpart, both the two input strings have `e`. I firstly judge the position of `e`. If the position of `e` is at the last position of the strings, the program will give warning `The input cannot be inerpret as numbers!`. If not, the program will go on to calculate the result. In this subpart, I firstly divided the strings into three substrings. The first substring is the substring before `e`. The second substring is `e`. The third substring is the substring after `e`. Secondly, I change the two first subtrings into `double` type and the two third substrings into `long long` type. Thirdly, multiply the two `double` type numbers and change the result into *string1*. Add the two `long long` type numbers and change the result into *string2*. Finally, connect the *string1* and `e` and *string2* to get the final result and output it.

To be more clear, the main method can be expressed as:

$$(m1)e(n1) * (m2)e(n2) \rightarrow result = (m1 * m2)e(n1 + n2)$$

The mind map is shown below:


Simple Calculator
Constraint
(a) Each char in the string must be a number 0-9 or e or point . or the negative sign -.
(b) The string can only have one point . and one e and the position of the point . or e cannot be at the first position of the string.
(c) The string cannot be a single point . or a single e or .e or e. .
(d) If the string have both point . and e, the position of the point . must be in front of the position of e .
(e) The number of the input to be multiplied must be two, not one or three or more.
(f) The negtive sign can only at the fisrt place of the string or a bit after the e .
Judgements
t1 t2 --- Use for loops to see if the constraint (a) is satisfied
Yes: t1=length of str_a and t2=length of str_b
No: else situations
n1 n2 --- The number of the negative sign -. Got from the same loops above. Judge if the constraint (f) is satisfied.
neg = neg_a && neg_b
neg_a = 1
n1=0 / initial value
n1=1 and the position of - is at str_a[0] or just after the `e`
n1=2 and the first position of - is at str_a[0] and the last position of - is just after `e`
neg_a = 0
n1 > 2
neg_b = 1
n2=0 / initial value
n2=1 and the position of - is at str_b[0] or just after the `e`
n2=2 and the first position of - is at str_b[0] and the last position of - is just after `e`
neg_b = 0
n2 > 2
tp_a tp_b te_a te_b --- Judge if the constraint (b) is satisfied, initial value is 1
ta = tp_a && te_a
tp_a = 1
str_a has only one point and the point doesn't at str_a[0]
te_a = 1
str_a has only one `e` and the `e` doesn't at str_a[0]
tb = tp_b && te_b
tp_b = 1
str_b has only one point and the point doesn't at str_b[0]
te_b = 1
str_b has only one `e` and the `e` doesn't at str_b[0]
t_single t_fe --- Judge if the constraint (c) is satisfied, initial value is 0
t_single = 1
str_a is a single `e` or `.` or str_b is a single `e` or `.`
t_fe = 1
str_a or str_b is `.e` or `e.`
tpe_a tpe_b --- Judge if the constraint (d) is satisfied, initial value is 1
tpe = tpe_a && tpe_b
tpe_a = 1
str_a has `e` and `.` and ta=1
tpe_a = 0
else
tpe_b = 1
str_b has `e` and `.` and tb=1
tpe_b = 0
else
cond = (t1==str_a.length()) && (t2==str_b.length()) && ta && tb && (!t_single) && (!t_fe) && tpe && neg
cond =1
do calculation part
cond = 0
output warning
str_a[0] = '+' or str_b[0] = '+'
warning
the number of input numbers is not two
warning
Calculation
str_a and str_b do not have point or `e`
Multiplication of two integers. / Multiplication of two big integers (without e)
str_a and str_b do not have `e` but at least one of them has a point
Multiplication of two floating-point numbers. / Multiplication of an integer and a floating-point number.
At least one of str_a and str_b do have `e`.
str_b has e and str_a doesn't have e
`e` doesn't at the last place of str_b
(m1)e(n1) * m2 --> result=(m1*m2)e(n1)
else
warning
str_a has e and str_b doesn't have e
`e` doesn't at the last place of str_a
m1 * (m2)e(n2) --> result=(m1*m2)e(n2)
else
warning
Both str_a and str_b have e
`e` doesn't at the last place of str_a and str_b
(m1)e(n1) * (m2)e(n2) --> result=(m1*m2)e(n1+n2)
else
warning

Part 2 - Code

```cpp
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

int main(int argc, char **argv)
{
    //input
    string str_a;
    string str_b;

    if (argc==3)
    {
        str_a = argv[1];
        str_b = argv[2];
    }

    string str_a0 = str_a;
    string str_b0 = str_b;

    int t1 = 0;
    int t2 = 0;
    int n1 = 0;
    int n2 = 0;
    for (int i = 0; i < str_a.length(); i++)
    {
        if ((str_a[i]>=48 && str_a[i]<=57) || (str_a[i]==46) || (str_a[i]==101) || (str_a[i]=='-'))
        {
            t1 = t1 + 1;
            if (str_a[i]=='-')
            {
                n1 = n1 + 1;
            }
            continue;
        }
        else break;
    }
    for (int j = 0; j < str_b.length(); j++)
    {
        if ((str_b[j]>=48 && str_b[j]<=57) || (str_b[j]==46) || (str_b[j]==101) || (str_b[j]=='-'))
        {
            t2 = t2 + 1;
            if (str_b[j]=='-')
            {
                n2 = n2 + 1;
            }
            continue;
        }
        else break;
    }

    bool tp_a = 1;
    bool te_a = 1;
    bool tp_b = 1;
    bool te_b = 1;
    if (str_a0.find('.')!=string::npos)
    {
        tp_a = (str_a0.find_first_of('.')==str_a0.find_last_of('.')) && (str_a0.find_first_of('.')!=0);
    }
    if (str_a0.find('e')!=string::npos)
    {
        te_a = (str_a0.find_first_of('e')==str_a0.find_last_of('e')) && (str_a0.find_first_of('e')!=0);
    }
    if (str_b0.find('.')!=string::npos)
    {
        tp_b = (str_b0.find_first_of('.')==str_b0.find_last_of('.')) && (str_b0.find_first_of('.')!=0);
    }
    if (str_b0.find('e')!=string::npos)
    {
        te_b = (str_b0.find_first_of('e')==str_b0.find_last_of('e')) && (str_b0.find_first_of('e')!=0);
    }

    bool ta = tp_a && te_a;
    bool tb = tp_b && te_b;
    bool t_single = (str_a0 == ".") || (str_a0=="e") || (str_b0 == ".") || (str_b0=="e");
    bool t_fe = (str_a0 == ".e") || (str_a0=="e.") || (str_b0 == ".e") || (str_b0=="e.");

    bool tpe_a = 1;
    bool tpe_b = 1;
    if ((str_a0.find('.')!=string::npos) && (str_a0.find('e')!=string::npos) && ta)
    {
        tpe_a = str_a0.find_first_of('.') < str_a0.find_first_of('e');
    }
    if ((str_b0.find('.')!=string::npos) && (str_b0.find('e')!=string::npos) && tb)
    {
        tpe_b = str_b0.find_first_of('.') < str_b0.find_first_of('e');
    }
    bool tpe = tpe_a && tpe_b;

    bool neg_a = 1;
    bool neg_b = 1;
    bool neg   = 1;

    if (n1==1)
    {
        if (str_a[0]=='-')
            neg_a = 1;
        else
        {
            if ((str_a.find('e')!=string::npos) && te_a)
                neg_a = (str_a.find_first_of('-') == (str_a.find_first_of('e')+1) );
            else
```

```
                neg_a = 0;
        }
    }
    else if (n1==2)
        neg_a = (str_a.find_first_of('-')==0) && (str_a.find('e')!=string::npos) && te_a && (str_a.find_last_of('-') == (str_a.find_first_of('e')+1));
    else if (n1>2)
        neg_a = 0;

    if (n2==1)
    {
        if (str_b[0]=='-')
            neg_b = 1;
        else
        {
            if ((str_b.find('e')!=string::npos) && te_b)
                neg_b = (str_b.find_first_of('-') == (str_b.find_first_of('e')+1) );
            else
                neg_b = 0;
        }
    }
    else if (n2==2)
        neg_b = (str_b.find_first_of('-')==0) && (str_b.find('e')!=string::npos) && te_b && (str_b.find_last_of('-') == (str_b.find_first_of('e')+1));
    else if (n2>2)
        neg_b = 0;

    neg = neg_a && neg_b;

    bool cond = (t1==str_a.length()) && (t2==str_b.length()) && ta && tb && (!t_single) && (!t_fe) && tpe && neg;

    if ((str_a0[0]=='+') || (str_b0[0]=='+'))
    {
        cout << "Please input numbers without the positive sign." << endl;
    }
    else if(!cond)
    {
        cout << "The input cannot be inerpret as numbers!" << endl; //输入不符合要求
    }
    else if (argc!=3)
    {
         cout << "Please input two numbers." << endl;
    }
    else
    {
        //(大)整数乘法
        long long a_long;
        long long b_long;
        long long result_long;//output

        if (str_a.find('.')==string::npos && str_a.find('e')==string::npos && str_b.find('.')==string::npos && str_b.find('e')==string::npos)
        {
            a_long = stoll(str_a);
            b_long = stoll(str_b);
            result_long = a_long * b_long;

            cout << a_long << " * " << b_long << " = " << result_long << endl;
        }

        //浮点数乘法(浮点数*浮点数 and 整数*浮点数)
        double a_double;
        double b_double;
        double result_double;//output
        if ((str_a.find('.')!=string::npos || str_b.find('.')!=string::npos) && str_a.find('e')==string::npos && str_b.find('e')==string::npos)
        {
            //如果检测到是整数，则转换为浮点数格式，最后直接进行浮点数乘法
            if (str_a.find('.') == string::npos)
            {
                str_a = str_a + '.';
            }

            if (str_b.find('.') == string::npos)
            {
                str_b = str_b + '.';
            }

            int a_firstpoint = str_a.find_first_of('.');
            int a_lastpoint  = str_a.find_last_of('.');
            int b_firstpoint = str_b.find_first_of('.');
            int b_lastpoint  = str_b.find_last_of('.');


            a_double = stod(str_a);
            b_double = stod(str_b);
            result_double = a_double * b_double;

            int pa = str_a.length() - a_firstpoint - 1;
            int pb = str_b.length() - b_firstpoint - 1;
            int pr;
            if((pa+pb)<=15 && (pa+pb)>0)
            {
                pr = pa + pb;
            }
            else if((pa+pb)==0)
            {
                pr = 1;
            }
            else pr = 15;

            cout.setf(ios_base::fixed, ios_base::floatfield);
            cout << setprecision(min(pa,15)) << a_double << " * ";
            cout << setprecision(min(pb,15)) << b_double << " = ";
            cout << setprecision(pr) << result_double << endl;
            //小数位太多会不精确，尝试多次发现小数位为15位时仍保持精准，15位以上开始丢失精度，所以精度设置为最大15

        }
        //大浮点数乘法(包含大整数，科学计数法表示，含e)
        if (str_a.find('e')!=string::npos || str_b.find('e')!=string::npos) // a b 至少一个含e
        {
            if (str_a.find('e')==string::npos && str_b.find('e')!=string::npos) //a不含e，b含e
            {
                if (str_b.find_first_of('e')==(str_b.length()-1))
                {
                    cout << "The input cannot be inerpret as numbers!" << endl;
                }
                else
                {
                    if (str_a.find('.')!=string::npos || str_b.find('.')!=string::npos)
                    {
                        string str_b1 = str_b.substr(0,str_b.find_first_of('e'));
                        string str_b2 = str_b.substr(str_b.find_first_of('e'),(str_b.length()-str_b.find_first_of('e')));
                        double a1 = stod(str_a);
                        double b1 = stod(str_b1);
                        double re1 = a1 * b1;

                        string result1 = to_string(re1) + str_b2;

                        cout << str_a0 << " * " << str_b0 << " = " << result1 << endl;
                    }
                    else
                    {
                        string str_b1 = str_b.substr(0,str_b.find_first_of('e'));
                        string str_b2 = str_b.substr(str_b.find_first_of('e'),(str_b.length()-str_b.find_first_of('e')));
                        long long a1 = stoll(str_a);
                        long long b1 = stoll(str_b1);
                        long long re1 = a1 * b1;

                        string result1 = to_string(re1) + str_b2;

                        cout << str_a0 << " * " << str_b0 << " = " << result1 << endl;
                    }
                }
            }
            else if (str_a.find('e')!=string::npos && str_b.find('e')==string::npos)
            // a含e, b不含e
            {
                if (str_a.find_first_of('e')==(str_a.length()-1))
                {
                    cout << "The input cannot be inerpret as numbers!" << endl;
                }
                else
                {
                    if (str_a.find('.')!=string::npos || str_b.find('.')!=string::npos)
                    {
                        string str_a1 = str_a.substr(0,str_a.find_first_of('e'));
                        string str_a2 = str_a.substr(str_a.find_first_of('e'),(str_a.length()-str_a.find_first_of('e')));
                        double a2 = stod(str_a1);
                        double b2 = stod(str_b);
                        double re2 = a2 * b2;

                        string result2 = to_string(re2) + str_a2;

                        cout << str_a0 << " * " << str_b0 << " = " << result2 << endl;
                    }
                    else
                    {
                        string str_a1 = str_a.substr(0,str_a.find_first_of('e'));
                        string str_a2 = str_a.substr(str_a.find_first_of('e'),(str_a.length()-str_a.find_first_of('e')));
                        long long a2 = stoll(str_a1);
                        long long b2 = stoll(str_b);
                        long long re2 = a2 * b2;

                        string result2 = to_string(re2) + str_a2;

                        cout << str_a0 << " * " << str_b0 << " = " << result2 << endl;
                    }
                }
            }
            //相同格式进行大浮点数运算
            else if((str_a.find('e')!=string::npos) && (str_b.find('e')!=string::npos))
            {
                int a_firste = str_a.find_first_of('e');
                int a_laste  = str_a.find_last_of('e');
                int b_firste = str_b.find_first_of('e');
                int b_laste  = str_b.find_last_of('e');

                bool test_b = (str_b.find_first_of('e')==(str_b.length()-1)) || (str_a.find_first_of('e')==(str_a.length()-1));

                if (test_b)
                {
                    cout << "The input cannot be inerpret as numbers!" << endl;
                }
                else
                {
                    if (str_a.find('.')!=string::npos || str_b.find('.')!=string::npos)
                    {
                        string a_mul = str_a.substr(0,(a_firste));
                        string b_mul = str_b.substr(0,(b_firste));
                        string a_add = str_a.substr((a_firste+1),(str_a.length()-a_firste-1));
                        string b_add = str_b.substr((b_firste+1),(str_b.length()-b_firste-1));

                        double    a_d = stod(a_mul);
                        double    b_d = stod(b_mul);
                        long long a_l = stoll(a_add);
                        long long b_l = stoll(b_add);

                        double    r1 = a_d * b_d;
                        long long r2 = a_l + b_l;

                        string result_string = to_string(r1) + "e" + to_string(r2);

                        cout << str_a0 << " * " << str_b0 << " = " << result_string << endl;
                    }
                    else
                    {
                        string a_mul = str_a.substr(0,(a_firste));
                        string b_mul = str_b.substr(0,(b_firste));
                        string a_add = str_a.substr((a_firste+1),(str_a.length()-a_firste-1));
                        string b_add = str_b.substr((b_firste+1),(str_b.length()-b_firste-1));

                        long long a_d = stoll(a_mul);
                        long long b_d = stoll(b_mul);
                        long long a_l = stoll(a_add);
                        long long b_l = stoll(b_add);

                        long long r1 = a_d * b_d;
                        long long r2 = a_l + b_l;

                        string result_string = to_string(r1) + "e" + to_string(r2);

                        cout << str_a0 << " * " << str_b0 << " = " << result_string << endl;
                    }
                }
            }
        }
    }
    return 0;
}
```

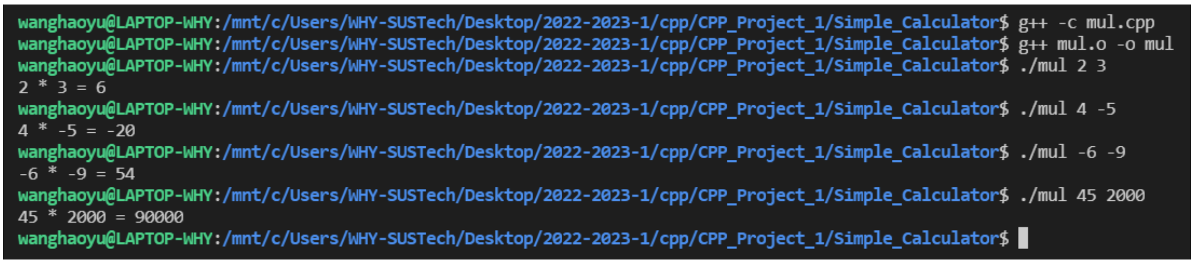
Part 3 - Result & Verification

My final program can do these functions below:

- Multiplication of two integers.
- Multiplication of two big integers (without `e`). *(The result must be in the scope of* `long long` *type.)*
- Multiplication of two floating-point numbers. *(The result must be in the scope of* `double` *type. If the input is represented by* `e` *, the result will also be represented by* `e` *.)*
- Multiplication of an integer and a floating-point number.
- Multiplication of big floating-point numbers (containing `e`).
- Multiplication of big integers (containing `e`).
- Multiplication of a big floating-point number and a big integer (both containing `e`).
- Tell the input is not a number.
- Tell the number of the input numbers is not two.
- Tell if the positive number has a positive sign. To simplify the calculation, it require the input cannot have positive sign.
- The calculation includes the multiplication with negative numbers.

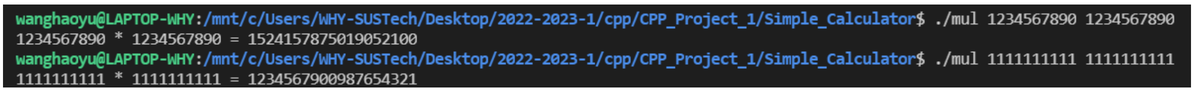
I wrote some test cases to verify the function of the simple calculator.

Test case #1: Multiplication of two integers.



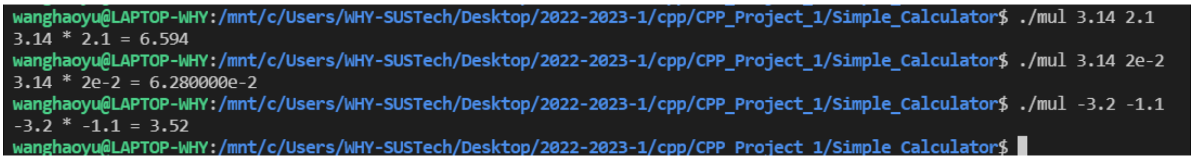
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ g++ -c mul.cpp
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ g++ mul.o -o mul
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2 3
2 * 3 = 6
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 4 -5
4 * -5 = -20
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul -6 -9
-6 * -9 = 54
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 45 2000
45 * 2000 = 90000
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$
```

Test case #2: Multiplication of two big integers (without `e`).



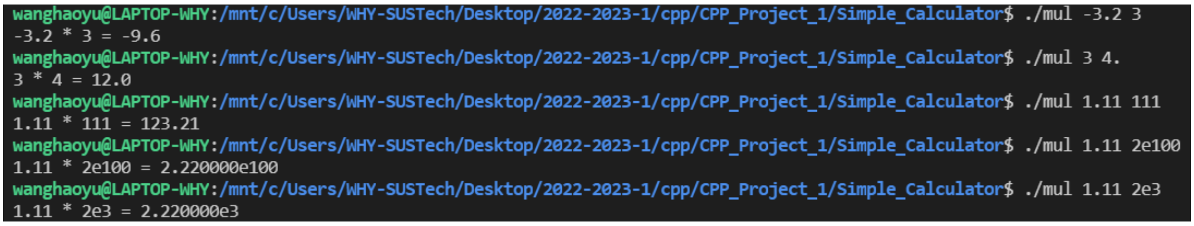
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1234567890 1234567890
1234567890 * 1234567890 = 1524157875019052100
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1111111111 1111111111
1111111111 * 1111111111 = 1234567900987654321
```

Test case #3: Multiplication of two floating-point numbers.



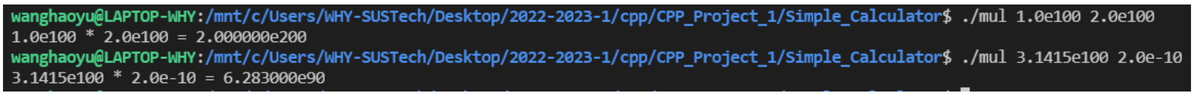
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3.14 2.1
3.14 * 2.1 = 6.594
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3.14 2e-2
3.14 * 2e-2 = 6.280000e-2
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul -3.2 -1.1
-3.2 * -1.1 = 3.52
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$
```

Test case #4: Multiplication of an integer and a floating-point number.



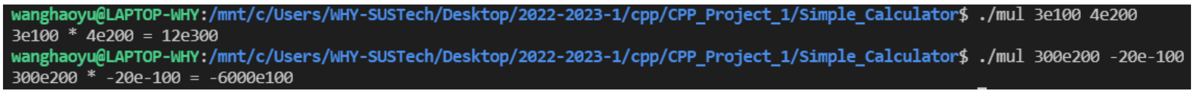
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul -3.2 3
-3.2 * 3 = -9.6
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3 4.
3 * 4 = 12.0
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1.11 111
1.11 * 111 = 123.21
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1.11 2e100
1.11 * 2e100 = 2.220000e100
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1.11 2e3
1.11 * 2e3 = 2.220000e3
```

Test case #5: Multiplication of big floating-point numbers (containing `e`).



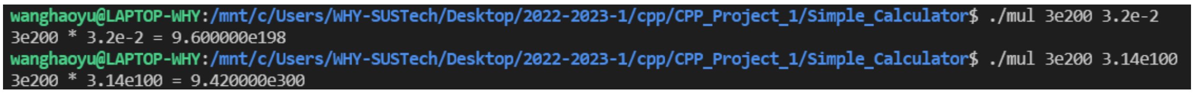
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1.0e100 2.0e100
1.0e100 * 2.0e100 = 2.000000e200
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3.1415e100 2.0e-10
3.1415e100 * 2.0e-10 = 6.283000e90
```

Test case #6: Multiplication of big integers (containing `e`).



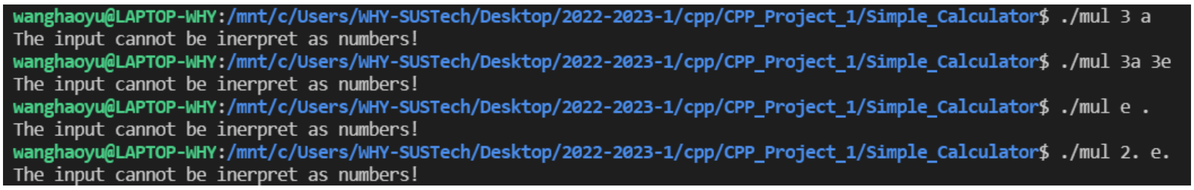
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3e100 4e200
3e100 * 4e200 = 12e300
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 300e200 -20e-100
300e200 * -20e-100 = -6000e100
```

Test case #7: Multiplication of a big floating-point number and a big integer (both containing `e`).



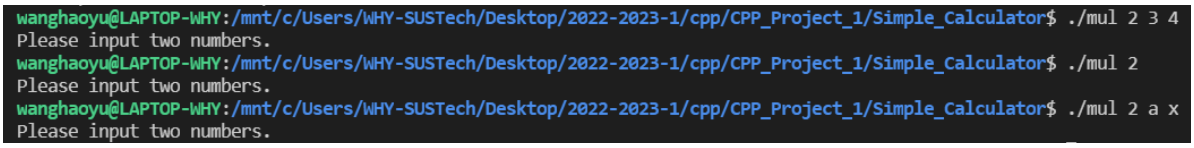
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3e200 3.2e-2
3e200 * 3.2e-2 = 9.600000e198
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3e200 3.14e100
3e200 * 3.14e100 = 9.420000e300
```

Test case #8: Tell the input is not a number.



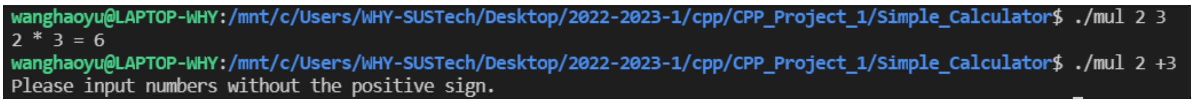
```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3 a
The input cannot be inerpret as numbers!
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 3a 3e
The input cannot be inerpret as numbers!
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul e .
The input cannot be inerpret as numbers!
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2. e.
The input cannot be inerpret as numbers!
```

Test case #9: Tell the number of the input numbers is not two.



```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2 3 4
Please input two numbers.
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2
Please input two numbers.
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2 a x
Please input two numbers.
```

Test case #10: Tell if the positive number has a positive sign. To simplify the calculation, it require the input cannot have positive sign.



```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2 3
2 * 3 = 6
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 2 +3
Please input numbers without the positive sign.
```

Part 4 - Difficulties & Solutions

4.1 Problem about the judgement of the input

Difficulty :

Firstly, I just divided the calculation in three parts and did judgements in each part. It would lead the judgements and loops repeated each time, which occupied many lines and easy to be confused. Apart from considering the satisfied chars, we also have to consider the number of each satisfed char.

Solution :

I make the judgements before all the calculation parts. If all the requests are satisfied, the proram can go to the calculation parts. If not, the program will output a warning to remind the person to input numbers in the correct form.

4.2 Problem about the number of decimal digits

Difficulty :

When I was writing the multiplication of two floating-point numbers, I found that if the number of decimal digits is more than 15, the floating-point number will loss value and be not accurate.

Solution :

I restricted the number of decimal digits of the result, which can be 15 at most.

4.3 Problem about the multiplication of big floating-point numbers

Difficulty :

I tested the multiplication of big floating-point numbers, its result may be `inf`.

Solution :

I uniformly divided the string to do calculation separately. The details are described in the analysis part. The main method can be expressed as:

$$(m1)e(n1) * (m2)e(n2) \to result = (m1 * m2)e(n1 + n2)$$

4.4 Problem about the coding logic

Difficulty:

There were many judging statements and loops. The coding logic is of vital importance to program. The code has been modified for six or seven time to change the logic order to make the code be less redundant and make the logic be clearer.

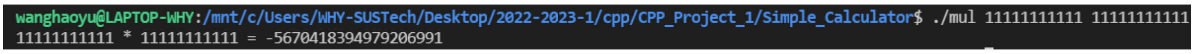
Solution:

Briefly draw a mind map to analyze the program logic. In the mind map, write down the possible situations and then extend to the possible solutions or implements. After finish the overall logic, look through the mind map to see if some logics can be optimized or combining. If so, modify the mind map to get a better logic net. Then write the code and our brain will be clearer and we can also code more efficiently.

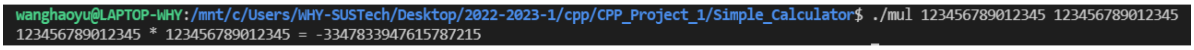
4.5 Problem about the overflow even if using the `long long` or `double` type

Difficulty:

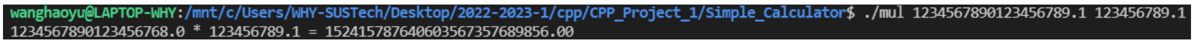
Although when we use `long long` type or `double` type to do the multiplication, the calculator can do multiplication for a large scope of numbers, it will happen overflows when the numbers are too big. In my test, the big integer multiplication can only get the correct result when the result is in the scope from -2^{63} to $(2^{63} - 1)$. And for the multiplication of the big floating-point numbers not using `e`, it can only get the correct result when the result is in the approxiamate scope from $4.94065645841246544e^{-324}$ to $1.79769313486231570e^{308}$ for positive number and from $-1.79769313486231570e^{308}$ to $-4.94065645841246544e^{-324}$.



```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 11111111111 11111111111
11111111111 * 11111111111 = -5670418394979206991
```



```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 123456789012345 123456789012345
123456789012345 * 123456789012345 = -3347833947615787215
```



```
wanghaoyu@LAPTOP-WHY:/mnt/c/Users/WHY-SUSTech/Desktop/2022-2023-1/cpp/CPP_Project_1/Simple_Calculator$ ./mul 1234567890123456789.1 123456789.1
1234567890123456768.0 * 123456789.1 = 152415787640603567357689856.00
```

Possible Solution:

I searched it on the Internet and found that high precision multiplication might be useful. However, I do not have enough time to modify my original code to implement this. I will try it by myself even though I have submitted the project.