

# Project1 A Simple Calculator

Name: 周思呈  
SID: 12110644

## Part 01 - Analysis

要求实现一个简单的乘法器，从命令行中读取参数然后判断输入合法性并进行计算。  
主要包括以下几个方法：

1. 判断命令行输入数据是否合法。合法输入只能包含两个用于计算的数据。合法数据包括普通数字和科学计数法表示的数字，其他均为不合法。
2. 从命令行中读取数据。命令行输入的参数默认为字符数组类型，需要转化为数字。
3. 乘法计算。分为普通计算和模糊计算，后者用于处理输入数据过大的情况，主要思路是转化为科学计数法表示，然后分别对底数和指数部分进行计算。

## Part 02 - Code

### 判断输入是否合法

合法输入包含数字、小数点、负号和e（E）。由于在cpp中一切数据类型都是数，判断时可直接使用ASCII码。

```
bool legal(char *input)
{
    int k = 0;
    while (input[k] != 0)
    {
        if ((input[k] >= 30 && input[k] <= 57)
            || input[k] == '.' || input[k] == '-'
            || input[k] == 'e' || input[k] == 'E')
        {
            k++; //合法就往后挪
        }
        else
        {
            return false;
        }
    }
    return true;
}
```

### 将字符串转为数字

用a1存整数部分，用a2存小数部分。最后二者存入double数组中分别返回。这种分别存储的方式解决了大数据高精度计算的问题（详见Part04 “大数据的精度”部分）。

中间如果遇到e或者E就进入科学计数法读取，调用`scientific_info`，该方法详细描述见下文。如果指数部分大于150，则返回inf，在main中进入`fuzzy_mul`模糊计算模式。如果指数部分没有超过150，则调用`scientific_num`将科学计数法转化为小数形式返回。为什么以150作为`fuzzy_mul`的分界呢？因为double数据类型的最大值是1.79769e+308，两个1e+150相乘正好在溢出的边缘大鹏展翅。

```

double *get_num(char *input)
{
    double *output = new double[2];
    double a1 = 0;    // a的整数位
    double a2 = 0;    // a的小数位
    int k = 0;
    double dec = 0.1;
    bool flag = false;

    while (input[k] != 0)
    {
        if (input[k] == '.')
        {
            flag = true; //来到小数点之后
            k++;
            continue;
        }
        else if (input[k] == 'e' || input[k] == 'E') //进入科学计数法
        {
            double *result = scientific_info(input);
            if (result[2] > 150)
            {
                output[0] = 1.0 / 0.0;
                output[1] = 1.0 / 0.0;
                return output;
            }
            else
            {
                double temp = scientific_num(result[0], result[1], result[2]);
                output[0] = floor(temp);
                output[1] = temp - output[0];
                return output;
            }
        }

        if (!flag)
        {
            a1 = a1 * 10 + input[k] - '0';
        }
        else
        {
            a2 = a2 + (input[k] - '0') * dec;
            dec = dec * 0.1;
        }
        k++;
    }
    output[0] = a1;
    output[1] = a2;
    return output;
}

```

`scientific_info` 方法返回三个值，分别是科学计数法表示下的底数整数、底数小数和指数。为了可扩展性，方法实现包含两部分，分别为将普通数字输入转化为上述三值，以及将科学计数法输入转化为上述三值。

```

double *scientific_info(char *input)
{
    double *result = new double[3];
    long a1 = 0;    // a的整数位
    double a2 = 0;  // a的小数位
    int k = 0;      // 遍历字符数组的每个元素
    double dec = 0.1; // 计算小数
    bool flag = false; // 标记是否来到小数点之后
    double a = 0;
    int digit = -1;
    while (input[k] != 0)
    {
        if (input[k] == '.')
        {
            flag = true; // 来到小数点之后
            k++;
            continue;
        }
        else if (input[k] == 'e') // 取exp
        {
            k++;
            int exp = 0;
            bool neg = false; // 正数
            while (input[k] != 0)
            {
                if (input[k] == '-')
                {
                    neg = true;
                    k++;
                    continue;
                }
                else
                {
                    exp = exp * 10 + input[k] - '0';
                    k++;
                }
            }
            if (neg)
            {
                exp = -exp;
            }

            result[0] = a1;
            result[1] = a2;
            result[2] = exp;
            return result;
        }

        if (!flag)
        {
            digit++;
            a1 = a1 * 10 + input[k] - '0';
        }
        else
        {
            a2 = a2 + (input[k] - '0') * dec;
            dec = dec * 0.1;
        }
    }
}

```

```

    }
    k++;
}
a = a1 + a2;

a1 = floor(a / pow(10, digit));
a2 = (a - a1 * pow(10, digit)) * pow(0.1, digit);
result[0] = a1;
result[1] = a2;
result[2] = digit;
return result;
}

```

## 普通计算

用乘法分配律提高计算精度。分别计算整数部分和小数部分，作为字符串返回。

```

string mul(double *inputs[])
{
    long a1 = (long)inputs[0][0] * (long)inputs[1][0];
    double a2 = inputs[0][0] * inputs[1][1] + inputs[0][1] * inputs[1][0] + inputs[0]
[1] * inputs[1][1];

    stringstream ss;
    ss << setprecision(15) << a2;
    string output;

    if (a2 > 0.0000001)
    {
        output = to_string(a1 + a2);
    }
    else if (a2 == 0)
    {
        output = to_string(a1);
    }
    else
    {
        output = to_string(a1) + "+" + ss.str();
    }
    return output;
}

```

## 模糊计算

遇到特别大数据时调用，舍弃一些精度，大幅度提高可计算的数据范围。

获取科学计数法表示的大数据信息，将两个底数相乘，将两个指数相加，作为字符串返回。还要注意有进位的情况。

```

string fuzzy_mul(char *inputs[])
{
    double *result1 = new double[3];
    double *result2 = new double[3];
    result1 = scientific_info(inputs[1]);
}

```

```

    result2 = scientific_info(inputs[2]);

    double a = (result1[0] + result1[1]) * (result2[0] + result2[1]);
    double exp = result1[2] + result2[2];
    if (a >= 10)
    {
        a = a / 10;
        exp++;
    }
    string output = to_string(a) + "e" + to_string(exp);
    return output;
}

```

## 主函数

先判断是否合法，再获取数据。数据过大则进入模糊计算，否则进入普通计算。

```

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        cerr << "Please input exactly two numbers!" << endl;
        return 0;
    }

    if (!(legal(argv[1]) && legal(argv[2])))
    {
        cerr << "The input cannot be interpret as numbers!" << endl;
        return 0;
    }
    else
    {
        double *a = get_num(argv[1]);
        double *b = get_num(argv[2]);

        if (isinf(a[0]) || isinf(b[0])) //超范围，进入fuzzy_mul
        {
            cout << "The input is so big that fuzzy multiplication is invoked!" <<
endl;
            cout << argv[1] << " * " << argv[2] << " = " << fuzzy_mul(argv) << endl;
        }
        else if (a[0] > 1e10 || b[0] > 1e10)
        {
            cout << argv[1] << " * " << argv[2] << " = " << fuzzy_mul(argv) << endl;
        }
        else
        {
            double *input[2] = {a, b};
            cout.precision(15);
            cout << argv[1] << " * " << argv[2] << " = " << mul(input) << endl;
        }
    }
    return 0;
}

```

## Part 03 - Result & Verification

### Test Case 1

If the two numbers are integers, the program will multiply them in integer format.

```
• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 2 3
2 * 3 = 6
```

### Test Case 2

If the input contains some non-integer numbers, the program will try to interpret the input as floating-point numbers.

```
• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 3.1416 2
3.1416 * 2 = 6.283200

• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 3.1415 2.0e-2
3.1415 * 2.0e-2 = 0.062830
```

### Test Case 3

It can tell that the input is not a number.

```
• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out a 2
The input cannot be interpret as numbers!
```

### Test Case 4

If you input some big integers, it will calculate correctly.

```
• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 1234567890 1234567890
1234567890 * 1234567890 = 1524157875019052100
```

其实上面这组test case，long也完全能完成正确计算。但是如果数字大到long无法解决问题，我们就触发模糊计算来实现功能。

```
• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 12345678900 12345678900
12345678900 * 12345678900 = 1.524158e20.000000

• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 1.0e200 1.0e200
The input is so big that fuzzy multiplication is invoked!
1.0e200 * 1.0e200 = 1.000000e400.000000
```

## Part 04 - Difficulties & Solutions

### 大数据的精度

第一版calculator使用double存储所有读入的合法数据，但是测试过程中发现，用这种存储方式会导致大数据精度的丢失。比如一个数字同时拥有很大的整数和很小的数，那个很小的数在计算过程中就会被自动忽略。例子见下图。

```
• (base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 10000000000000.0000000001 20
100000000000000 * 20 = 2e+15
```

经过分析和思考发现，实际上所有合法数据都能被表示为整数部分+小数部分。注意这部分处理的数据和fuzzy\_mul处理的数据和关注的重点是不同的：此处主要关注整数部分小于1e150的数据，要求提高这部分数据的计算精度；而fuzzy\_mul关注的是大于1e150的超大数据，允许精度的丢失。在处理中，我们用乘法分配律将这几个部分交叉相乘后相加，以提高计算精度。具体计算方法如下：

$$1.23e^{45} * 6.7e^8 =$$

$$(1 + 0.23) \times (6 + 0.7)$$

$$= 1 \times 6 + 1 \times 0.7 + 6 \times 0.23 + 0.23 \times 0.7$$

最后在转成字符串输出的时候出现了一点小问题。上述`1000000000000000.00000000001 * 20`的例子中，经过计算，小数部分是`2e-10`，如果表示成小数形式好像没有什么现成的方法，并且10位小数可读性也不高，于是采用“+”连接整数部分与小数部分。这部分方法实现包含在`mul`当中。最后输出结果如下图：

```
(base) zhousicheng@zhousichengdeMacBook-Pro project_1_simple_calculator % ./a.out 1000000000000000.00000000001 20
1000000000000000.00000000001 * 20 = 2000000000000000+2e-10
```

但是之后在测试中又遇到了另一个问题，就是处理这种精度问题时，如果把`1e150`设置成模糊计算的边界，那么在`1e18`以上`1e150`以下这个区间中，`long`无法实现计算要求。如果需要计算这个范围内的数，需要调用类似模糊计算那种使用科学计数法进行计算的方法，但是一旦使用科学计数法，数据精度便难免丢失。或者其实还有一种解决方法，就是手写乘法进位，但是这必然带来效率的大幅度降低。此处尚未思考出两全其美的解决办法，只能留待后续思考。

当前实现的代码当中保留了大数据计算，选择了丢失精度。如果要想实现保留精度的办法，将`main`函数`else if (a[0] > 1e10 || b[0] > 1e10)`的这部分注释掉即可。