CS205-2022Fall Project 1 Reprot

A Simple Calculator

Name: 陈康睿

SID: 12110524

Part 1 - Analysis

To implement a multiplicator which can calculate big integers and big float-point numbers, this program is designed to process strings inputed from command line arguments and calculate the answer using high precesion multiplication.

Part 2 - Code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <algorithm>

using namespace std;

/*
anslen : the length of the precise digits of the answer
anse : the exponent of the answer (based on 10)
len[] : the length of the precise digits of the inputs
dot[] : the index of '.' of the inputs
sp[] : the index of simplified signs for exponents
e[] : the index of 'e'/'E' of the inputs
ex[] : the value of the exponents
digit[][] : the precise digits of the inputs
ans[] : the precise digits of the answer
*/
int anslen, anse, len[2], dot[2], sp[2], e[2], ex[2], digit[2][10000], ans[20000];
bool minus[2], em[2];

/*
Only used when dubugging
*/
void check()
{
    for (int i = 0; i <= 1; i++)
    {
        for (int j = len[i] - 1; ~j; j--)
            printf("%d", digit[i][j]);
        printf("\nex = %d\n", ex[i]);
    }
    printf("anslen = %d\n", anslen);
    for (int i = anslen - 1; ~i; i--)
        printf("%d", ans[i]);
```

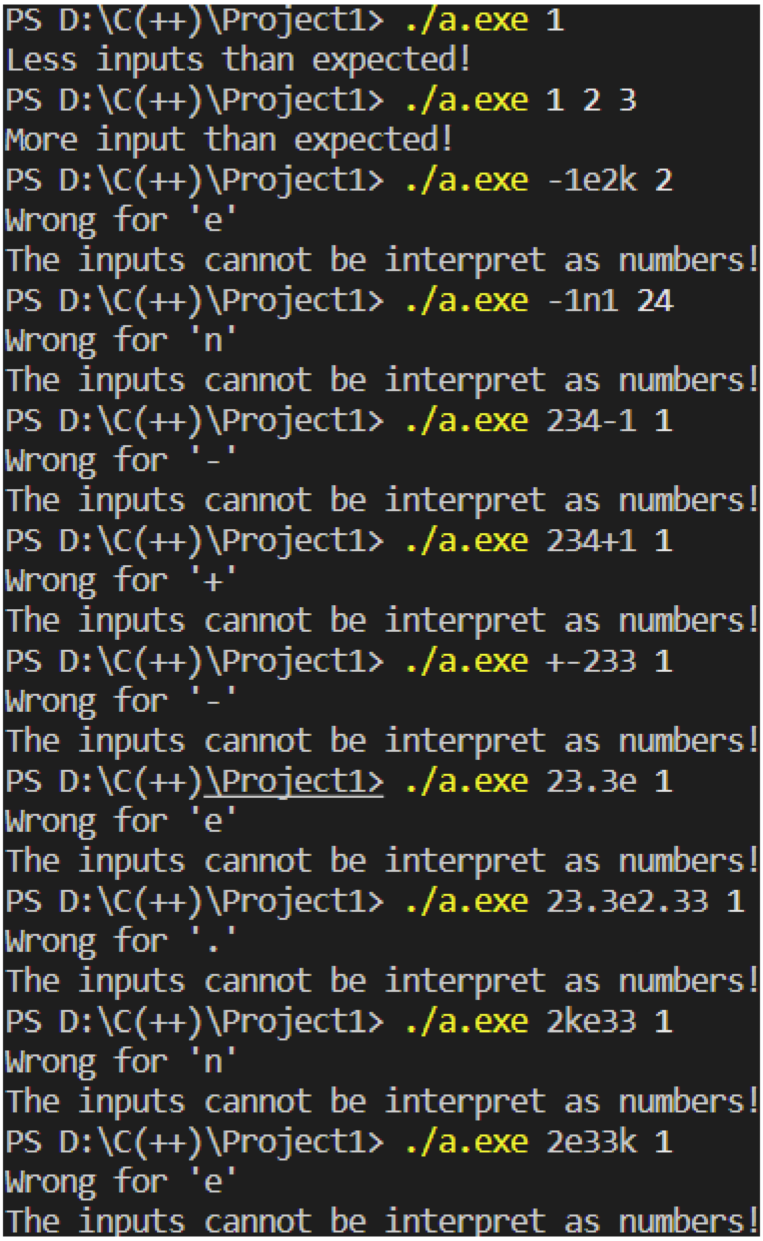
```c
    printf("\nanse = %d\n", anse);
}

/*
Assign dot[], e[] with -1, which means the corresponding signs haven't appear
*/
void init()
{
    memset(dot, -1, sizeof(dot));
    memset(e, -1, sizeof(e));
}

/*
Used when found invalid input string
Print error character signs:
    '+' : invalid number/positions for character '+'
    '-' : invalid number/positions for character '-'
    '.' : invalid number/positions for character '.'
    'e' : invalid number/positions for character 'e'/'E' or colliding with simplified signs for exponents (e.g. 'k'/'G')
    'n' : unexpected appearance of characters
Print error notice and exit the whole program with value 1.
*/
void exc(char c)
{
    printf("Wrong for '%c'\n", c);
    puts("The inputs cannot be interpret as numbers!");
    exit(1);
}

/*
Check whether character c is a digit
*/
bool isdigit(char c) {
    return c >= '0' && c <= '9';
}

/*
Standardize input strings into precise digits with displacements, similar to storing floating-point numbers.
*/
void pre(int idx, char *s)
{
    int l = strlen(s);
    if (l > 10000) // Only support inputs shorter than 10000
    {
        printf("The inputs have too many bits!");
        exit(1);
    }
    sp[idx] = l - 1;
    switch (s[l - 1]) // Check whether the inputs used simplified signs for exponents and translate the values
    {
        case 'k':
```

```
        case 'K':
            ex[idx] = 3;
            break;
        case 'm':
        case 'M':
            ex[idx] = 6;
            break;
        case 'g':
        case 'G':
            ex[idx] = 9;
            break;
        default:
            l++;
            sp[idx] = -1;
            break;
    }
    if (!--l) // Only a sign
        exc('n');
    for (int i = 0; i < l; i++)
    {
        switch (s[i])
        {
            case '+': // '+' can only appeares at the beginning or right after 'e'/'E'
                if (i && e[idx] != i - 1)
                    exc('+');
                break;
            case '-': // '-' can only appeares at the beginning or right after 'e'/'E'
                if (i)
                    if (e[idx] == i - 1 && i != l - 1)
                        em[idx] = true;
                    else
                        exc('-');
                else
                    minus[idx] = true;
                break;
            case '.': // '.' can only appears befor 'e' and can't be the head
                if (!i || i == l - 1 || ~dot[idx] || ~e[idx])
                    exc('.');
                dot[idx] = i;
                break;
            case 'e':
            case 'E': // 'e'/'E' can't be the head or the tail or right afer a sign
                if (!i || i == l-1 || ~e[idx] || ex[idx] || !isdigit(s[i-1]))
                    exc('e');
                e[idx] = i;
                break;
            default: // Record the precise digits and the exponents
                if (s[i] < '0' || s[i] > '9')
                    exc('n');
                if (~e[idx])
```

```
                    ex[idx] = ex[idx] * 10 + s[i] - '0';
                else
                    digit[idx][len[idx]++] = s[i] - '0';
                break;
        }
    }
    reverse(digit[idx], digit[idx] + len[idx]); // Reverse the precise digits for the convenience of calculation
    l = strlen(s);
    if (em[idx]) // Calculate the exponents using index of signs
        ex[idx] = -ex[idx];
    if (~dot[idx])
    {
        if (~sp[idx])
            ex[idx] -= sp[idx] - dot[idx] - 1;
        if (~e[idx])
            ex[idx] -= e[idx] - dot[idx] - 1;
    }
}

/*
Calculate the answer
For the precise digits, using high precision multiplication.
For the exponent, just used int for reality consideration.
*/
void calculate()
{
    for (int i = 0; i < len[0]; i++)
        for (int j = 0; j < len[1]; j++)
            ans[i + j] += digit[0][i] * digit[1][j];
    for (int i = 0; i < len[0] + len[1]; i++)
    {
        ans[i + 1] += ans[i] / 10;
        ans[i] %= 10;
    }
    for (int i = len[0] + len[1]; ~i; i--)
        if (ans[i])
        {
            anslen = i + 1;
            break;
        }
    anse = ex[0] + ex[1];
}

/*
Print the answer: (-)A(.B)((-)eC)
*/
void print(char *argv[])
{
    printf("%s * %s = ", argv[1], argv[2]);
    if (!anslen)
    {
        putchar('0');
        return;
    }
    if (minus[0] ^ minus[1])
        putchar('-');
    switch (anslen)
    {
    case 0:
        putchar('0');
        break;
    case 1:
        printf("%d", ans[0]);
        if (anse)
            printf("e%d", anse);
        break;
    default:
        anse += anslen - 1;
        int low = 0;
        for (int i = 0; i < anslen; i++)
            if (ans[i])
            {
                low = i;
                break;
            }
        printf("%d.", ans[anslen - 1]);
        if (anslen-1 > low) putchar('.');
        for (int i = anslen - 2; i >= low; i--)
            printf("%d", ans[i]);
        if (anse)
            printf("e%d", anse);
        break;
    }
}

int main(int argc, char *argv[])
{
    if (argc != 3) // Check the inputs numbers
    {
        printf("%s", argc < 3 ? "Less inputs than expected!" : "More input than expected!");
        return 1;
    }
    init();
    pre(0, argv[1]);
    pre(1, argv[2]);
    calculate();
    print(argv);
    return 0;
}
```

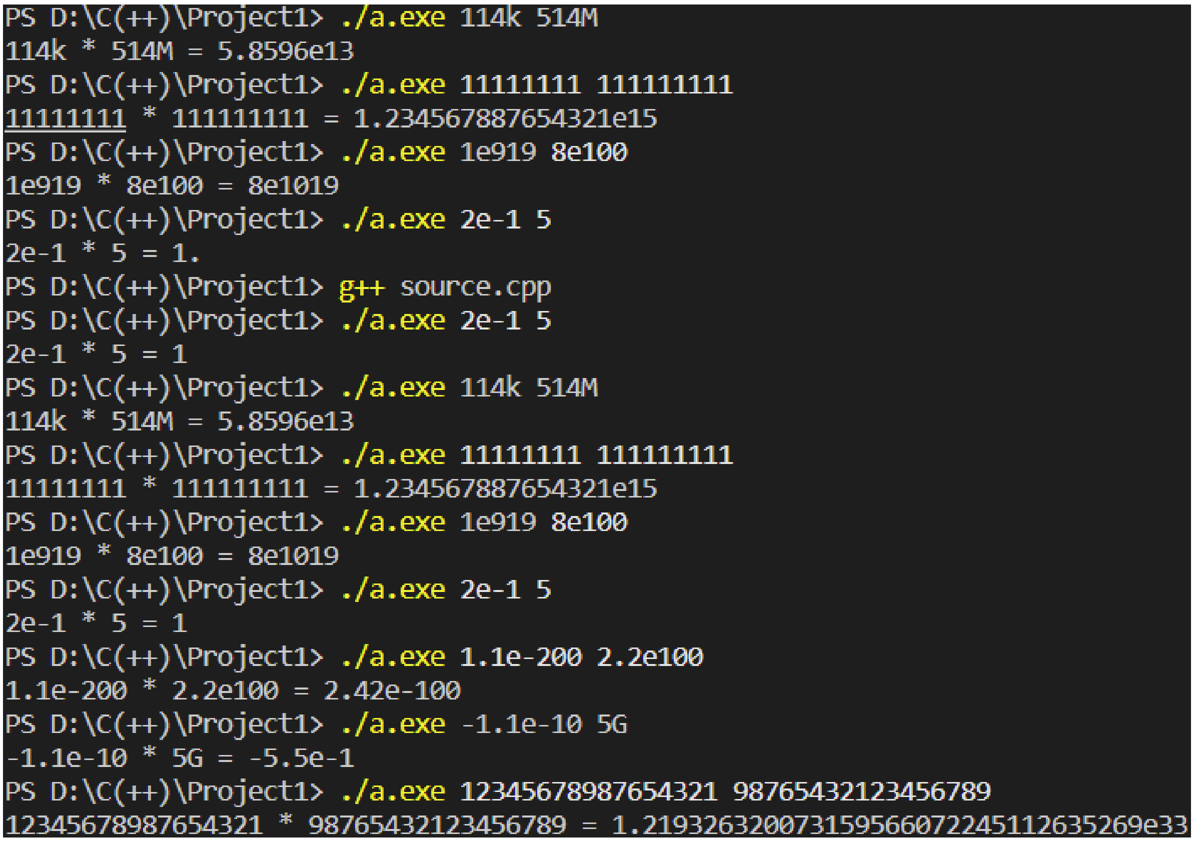
Part 3 - Result & Verification

Invalid Cases



```
PS D:\C(++)\Project1> ./a.exe 1
Less inputs than expected!
PS D:\C(++)\Project1> ./a.exe 1 2 3
More input than expected!
PS D:\C(++)\Project1> ./a.exe -1e2k 2
Wrong for 'e'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe -1n1 24
Wrong for 'n'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe 234-1 1
Wrong for '-'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe 234+1 1
Wrong for '+'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe +-233 1
Wrong for '-'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe 23.3e 1
Wrong for 'e'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe 23.3e2.33 1
Wrong for '.'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe 2ke33 1
Wrong for 'n'
The inputs cannot be interpret as numbers!
PS D:\C(++)\Project1> ./a.exe 2e33k 1
Wrong for 'e'
The inputs cannot be interpret as numbers!
```

Valid Cases



```
PS D:\C(++)\Project1> ./a.exe 114k 514M
114k * 514M = 5.8596e13
PS D:\C(++)\Project1> ./a.exe 11111111 111111111
11111111 * 111111111 = 1.234567887654321e15
PS D:\C(++)\Project1> ./a.exe 1e919 8e100
1e919 * 8e100 = 8e1019
PS D:\C(++)\Project1> ./a.exe 2e-1 5
2e-1 * 5 = 1.
PS D:\C(++)\Project1> g++ source.cpp
PS D:\C(++)\Project1> ./a.exe 2e-1 5
2e-1 * 5 = 1
PS D:\C(++)\Project1> ./a.exe 114k 514M
114k * 514M = 5.8596e13
PS D:\C(++)\Project1> ./a.exe 11111111 111111111
11111111 * 111111111 = 1.234567887654321e15
PS D:\C(++)\Project1> ./a.exe 1e919 8e100
1e919 * 8e100 = 8e1019
PS D:\C(++)\Project1> ./a.exe 2e-1 5
2e-1 * 5 = 1
PS D:\C(++)\Project1> ./a.exe 1.1e-200 2.2e100
1.1e-200 * 2.2e100 = 2.42e-100
PS D:\C(++)\Project1> ./a.exe -1.1e-10 5G
-1.1e-10 * 5G = -5.5e-1
PS D:\C(++)\Project1> ./a.exe 12345678987654321 98765432123456789
12345678987654321 * 98765432123456789 = 1.219326320073159566072245112635269e33
```

Part 4 - Difficulties & Solutions

1. Using primary data types to calculate large numbers sometimes leads to precision loss or even `NaN`.
   - Using high precision multiplication.
2. Inputs has to many possible valid as well as invalid forms.
   - Using preprocessing to filter invalid cases and standerdize the strings into high precision float-point format.
   - Using function `isdigit()` to simplify the condition expressions, for many of the invalid cases come from sign collisions.
3. Special cases of valid inputs and outputs.
   - Add special judgements to allow some simplified forms of input (e.g. `K/k` `M/m` `G/g`).
   - Be careful to print special cases and avoid outputing unnecessary stuff (e.g. `0` alone, needless `e`, prefix/postfix `0`).