CPP_Project1_Report$_{by@Jiko}$

Ⅰ程序启动

启动方式

方式一：命令行带输入

情况①：正常输入

```
/mul 1.1234 346435
```

情况②：无输入或输入字符串个数不为2，转到启动方式二

```
/mul 12 34 245
```

方式二：命令行无输入（或不规范）

```
/mul 12 34 245
```

此时程序会输出特定语句，输入语句后的前两个字符串，

```
/mul 12 34 245
Please input two numbers need to be multiply(split by " "):
123 35 123
```

即第一个数为"123"，第二个数为"35"

代码实现：

```
if (argc == 3){        //方式1
        a = argv[1];
        b = argv[2];
    }else{             //方式2
        cout << "Please input two numbers need to be multiply(split by \" \"):" << endl;
        string A,B;
        cin >> A >> B;
        a = const_cast<char*>(A.c_str());//把String转成Char*再存入全局变量a、b中
        b = const_cast<char*>(B.c_str());
    }
```

II数据的输入与处理

数据的输入

启动方式1使用`int main(int argc, char* argv[])`直接输入两个数据；启动方式2先使用string存放两个数据再利用`a = const_cast<char*>(A.c_str());`转为char*。

数据的存放

> 因为之前只学过java，所以不知道怎么选择最优的存放方式，在经过一段时间的探索之后决定使用两个全局的char*来存放输入的字符串，并在之后直接在此上操作。

数据类型的判断

在这一步中使用`int judgeType(char* obj)`来判断输入数据的类型，因为输入的类型是无穷无尽的，所以我仅挑选了其中几个比较常见的典型案例，例如中文（一千一百五十一、陆拾壹）、科学计数法（1e8，1e-5，4e-20）等等，不同类型的数据对应不同的数字输出。

后续有更多的类型只要补充正则以及对应的输出数字就行。

代码实现：

```
int judgeType(string obj){
    int Ecounter = 0;
    for (int i = 0; i < obj.length(); ++i) {
        if (obj[i]>=0 && obj[i]<=127){//在ascii码表内
            if (obj[i]<'0' || obj[i]>'9'){//不为数字
                if (obj[i] == 'e'){//计算e的数量
                    Ecounter++;
                }
            }
        } else{//含全角字符
```

```
            return 1;
        }
    }
    if (Ecounter == 1){
        return 2;
    }
    return 0;
}
```

不同数据类型的处理

翻译

> 将所有数据变为只由“0-9”以及小数点“.”和负号“-”组成，将这一数据定为**标准数据**。

return -1：非法输入

使程序直接输出“The input cannot be interpret as numbers!”

例如：

```
/mul 我不是数字 我是汉字
The input cannot be interpret as numbers!
```

```
/mul %*# 2
The input cannot be interpret as numbers!
```

return 0：合法字符

即输入数据已经满足为**标准数据**，直接进入下一步的运算。

return 1：中文字符

使用了网上的代码进行转换

https://blog.csdn.net/u010412858/article/details/80354996

```
int chineseToInt(wstring s)
{
    map<wchar_t, int> chineseNum;
    chineseNum.insert(pair<wchar_t, int>(L'零', 0));
    chineseNum.insert(pair<wchar_t, int>(L'一', 1));
    chineseNum.insert(pair<wchar_t, int>(L'二', 2));
    chineseNum.insert(pair<wchar_t, int>(L'两', 2));
    chineseNum.insert(pair<wchar_t, int>(L'俩', 2));
    chineseNum.insert(pair<wchar_t, int>(L'三', 3));
    chineseNum.insert(pair<wchar_t, int>(L'四', 4));
    chineseNum.insert(pair<wchar_t, int>(L'五', 5));
    chineseNum.insert(pair<wchar_t, int>(L'六', 6));
    chineseNum.insert(pair<wchar_t, int>(L'七', 7));
    chineseNum.insert(pair<wchar_t, int>(L'八', 8));
    chineseNum.insert(pair<wchar_t, int>(L'九', 9));
    chineseNum.insert(pair<wchar_t, int>(L'十', 10));
    chineseNum.insert(pair<wchar_t, int>(L'百', 100));
    chineseNum.insert(pair<wchar_t, int>(L'千', 1000));
    chineseNum.insert(pair<wchar_t, int>(L'万', 10000));
```

```
    chineseNum.insert(pair<wchar_t, int>(L'亿', 100000000));
    chineseNum.insert(pair<wchar_t, int>(L'壹', 1));
    chineseNum.insert(pair<wchar_t, int>(L'贰', 2));
    chineseNum.insert(pair<wchar_t, int>(L'叁', 3));
    chineseNum.insert(pair<wchar_t, int>(L'肆', 4));
    chineseNum.insert(pair<wchar_t, int>(L'伍', 5));
    chineseNum.insert(pair<wchar_t, int>(L'陆', 6));
    chineseNum.insert(pair<wchar_t, int>(L'柒', 7));
    chineseNum.insert(pair<wchar_t, int>(L'捌', 8));
    chineseNum.insert(pair<wchar_t, int>(L'玖', 9));
    chineseNum.insert(pair<wchar_t, int>(L'拾', 10));
    chineseNum.insert(pair<wchar_t, int>(L'玖', 100));
    chineseNum.insert(pair<wchar_t, int>(L'仟', 1000));
    chineseNum.insert(pair<wchar_t, int>(L'萬', 10000));
    chineseNum.insert(pair<wchar_t, int>(L'億', 100000000));

    int result=0, tmp = 0, hnd_mln=0;
    wchar_t curr_char;
    int curr_digit;
    for (int i = 0; i < s.length(); ++i)
    {
        curr_char = s.at(i);
        if (chineseNum.find(curr_char) == chineseNum.end())
            return NULL;
        curr_digit = chineseNum.at(curr_char);

        if (curr_digit == pow(10, 8))//meet 「亿」 or 「億」
        {
            result = result + tmp;
            result = result * curr_digit;
            //get result before 「亿」 and store it into hnd_mln
            //reset `result`
            hnd_mln = hnd_mln * pow(10, 8) + result;
            result = 0;
            tmp = 0;
        }
        else
        {
            if (curr_digit == pow(10, 4))//meet 「万」 or 「萬」
            {
                result = result + tmp;
                result = result * curr_digit;
                tmp = 0;
            }
            else
            {
                if (curr_digit >= 10)//meet 「十」, 「百」, 「千」 or their
traditional version
                {
                    if (tmp == 0)
                        tmp = 1;
                    result = result + curr_digit * tmp;
                    tmp = 0;
                }
                else
```

```
                {
                    tmp = tmp * 10 + curr_digit;
                    /*if (curr_digit != NULL)

                    else
                    {
                        return result;
                    }*/
                }
            }
        }
    }
    result = result + tmp;
    result = result + hnd_mln;
    return result;
}
```

将对应的字赋予对应的值最后相乘相加，在此基础上我加入了其他口语表达习惯例如“俩”“两”。需要注意的是中文输入需要非常遵守规范，十百千万都需说明清楚，否则判为无法转换。例如：

```
/mul 一十 十一
The input cannot be interpret as numbers!
/mul 二百五 二百五十
205 * 250 = 51250
/mul 贰佰 叁佰五十
200 * 350 = 70000
```

转换后再进入下一步计算。

return 2：科学计数法

将科学计数法分为两类，一类是+一类是-。

主要思路还是把科学计数法转成**标准数据**，将数据分为两部分，前一部分为part1，后一部分为part2。part2先把后面的数字拼起来然后使用`stoi()`变成int类型，名字叫`Enumber`，如果是正的就开一个长度为Enumber+1的数组，初始化所有格子为0，从前往后填充part1；如果是负的就开一个长度为Enumber+2的数组，初始化所有格子为0，第二位改为小数点，从后往前填充part1。如此一来就可以得到**标准数据**之后再进行下一步计算。

代码实现：

```
else if (type == 2){//科学计数法
            int eLocation = 0;
            for (int i = 0; i < strlen(obj); ++i) {
                if (obj[i] == 'e'){
                    eLocation = i;
                }
            }
            if (obj[eLocation+1] == '-'){
                char* linshi;
                for (int i = eLocation,j=0; i < strlen(obj); ++i) {
                    if (obj[i]>='0'&&obj[i]<='9'){
                        linshi[j++] = obj[i];
                    }
                }
```

```
                int Enumber = stoi(linshi);
                char result[Enumber+2] = {0};
                result[1] = '.';
                for (int i = eLocation,j=Enumber+1; i >=0; --i) {
                    if (obj[i]>='0'&&obj[i]<='9'){
                        result[j--] = obj[i];
                    }
                }
                obj = result;
            } else{
                char* linshi;
                for (int i = eLocation,j=0; i < strlen(obj); ++i) {
                    if (obj[i]>='0'&&obj[i]<='9'){
                        linshi[j++] = obj[i];
                    }
                }
                int Enumber = stoi(linshi);
                char result[Enumber+1] = {0};
                for (int i = 0,j=0; i < eLocation; ++i) {
                    if (obj[i]>='0'&&obj[i]<='9'){
                        result[j++] = obj[i];
                    }
                }
                obj = result;
            }
```

Ⅲ计算

数组做乘法

正负判断

翻译之后所有数据将变为只由“0-9”以及小数点“.”和负号“-”，此时再使用函数来处理数据的正负，若为正，则函数返回true，若为负，则函数返回false，并移除负号。在这时候数据变为无符号数，接下来将两数进行计算。最后在打印时如果两数对应的布尔值不一致则先打印一个负号，反之不打印。

小数点处理

先移除小数点，最后在数组乘法结束后再加上。

例如：$346.134613471 * 2313461.234234 = 346134613471 * 2313461234234/1e15$

数组乘法

在经过前面的**数据处理**、**正负判断**、**小数点处理**后余下的只有0-9共10个数字组成的数，因为可能遇到数据类型无法存放的超大数据，所以我决定采用数组来计算乘法。其原理与列乘法竖式类似，将两数分别存入两个数组，长度分别为`len1`和`len2`，n位数与m位数相乘最大为m+n位数（使用计算器99*999简单验证），因此可以开一个存放结果的数组3，长度为m+n，数1的i位与数2的j位相乘，结果加在数3的i+j位，每次运算后把多于10的部分进位，个位数保留，依此类推可以计算非常大的数字。经过小数点处理后也可以处理高精度的浮点数。代码实现：

```
void mul(char* number1,char* number2){
    int len1 = strlen(number1);
```

```
    int len2 = strlen(number2);
    int len3 = len1+len2;
    bool n1 = true;
    bool n2 = true;
    if (number1[0] == '-'){
        number1[0] = '0';
        n1 = false;
    }
    if (number2[0] == '-'){
        number2[0] = '0';
        n2 = false;
    }
    revstr(number1);//翻转char*
    revstr(number2);
    //查找小数点位置
    int point = 0;
    point = findPoint(number1,len1) + findPoint(number2,len2);
    if (point == 0){
        ifPoint = false;
    }else{
        ifPoint = true;
    }

    int first[len1] = {0};
    int second[len2] = {0};
    int result[len3] = {0};
    for (int i = 0,j = 0; i < len1; ++i) {
        if (number1[i] != '.'){
            first[j++] = number1[i] - '0';
        }
    }
    for (int i = 0,j = 0; i < len2; ++i) {
        if (number2[i] != '.'){
            second[j++] = number2[i] - '0';
        }
    }

    for (int i = 0; i < len1; ++i) {
        for (int j = 0; j < len2; ++j) {
            result[i+j] += first[i] * second[j];
        }
    }
    int up = 0;
    for (int i = 0; i < len3; ++i) {
        result[i] += up;
        up = result[i] / 10;
        result[i] = result[i]%10;
    }

    char solution[len3];//最终结果的数组
    for (int i = 0,j = 0; i < len3; ++i) {
        if (point == 0){
            solution[i] = result[j++] + '0';
        }else{
            if (i!=point){
```

```
                solution[i] = result[j++] + '0';
            }else{
                solution[i] = '.';
            }
        }
    }
    //掐头去尾
    if (ifPoint){
        cut(solution,len3);
    }
    revstr(solution);
    cut(solution,len3);
    if (n1 != n2){
        cout << "-";
    }
    for (int i = 0; i < strlen(solution); ++i) {
        if (solution[i]>='0' && solution[i]<='9' ){
            cout << solution[i];
        } else if (solution[i] == '.'){
            cout << solution[i];
        }
    }
    cout << endl;

    revstr(number1);
    revstr(number2);
    return;
}
```

案例：

```
/mul 12345 457458536589
12345 * 457458536589 = 5647325634191205
/mul 346.134613471 2313461.234234
346.134613471 * 2313461.234234 = 800769010.091728182766214
```

掐头去尾

数组乘法结束后头尾会有多余的0，此时需要从头到尾和从尾到头一直去除直到该0的下一位是小数点。

代码实现：

```
void cut(char* solution,int len3){
    for (int i = 0; i < len3; ++i) {
        if ((solution[i] <= '0' || solution[i] > '9' )&& solution[i+1] != '.'){
            solution[i] = '&';
        }else{
            break;
        }
    }
}
```

刚经过数组乘法的结果还没有翻转，此时使用cut()即从尾到头去0。

之后使用revstr翻转结果，此时使用cut()即从头到尾去0。

需要注意的是若存在小数点且小数点后全部为0，会保留十分位的0来说明进行了浮点运算。

例子：

```
/mul 0.2 5
0.2 * 5 = 1.0
/mul 1.23 23.2
1.23 * 23.2 = 28.536
```

IV输出

先使用`printAB()`来输出经过`converter()`处理后的两个数据及乘号，接着使用`mul()`（参照数组乘法中的代码，在计算完后会直接输出结果）来输出整个算式：

```
void printAB(){
    for (int i = 0; i < strlen(a); ++i) {
        cout << a[i];
    }
    cout << " * ";

    for (int i = 0; i < strlen(b); ++i) {
        cout << b[i];
    }
    cout << " = ";
}
```

V一些测试数据

```
/mul 2.1e5 1.1e5
210000 * 110000 = 23100000000
```

```
/mul 345os 029845f
The input cannot be interpret as numbers!
```

```
/mul 134561032451451235 96029436023945620
134561032451451235 * 96029436023945620 = 12921820057112706828443579721840700
```

```
/mul 六千五百一十三 五千两百二十
6513 * 5220 = 33997860
```

```
/mul 1e5 2.5
100000 * 2.5 = 250000.0
```