CS205 C/ C++ Programming - Project

name: 廖铭骞

SID: 12012919

Part 1 - Analysis

高精度乘法实现的思想：

乘法步骤概述

依次将乘数的每一个位乘以被乘数，并将产生的结果依次写在竖式下方，并随着乘数的每一个位产生的结果每次将结果左移一个位之后再与上面产生结果之和相加，最后产生的即为乘法的结果。

单个位数与被乘数相乘的实现

先考虑简单的情况，即被乘数有很多位，而乘数只有一位的情况。此时，从最右端开始遍历被乘数的所有数字，依次与长度为一的乘数相乘，由于字符串的基本单位是char，而char在c++当中是以整数存储的，依据字符串的数值编码规则，数字的字符串可以通过减去'0'得到对应数字的整数形式。两个数相乘会产生进位，在循环遍历被乘数数位的过程中，使用carry变量存储两数相乘产生的进位，使用一个sum变量存储乘积的和位。可以得到下列关系式：(假设两个相乘的数字分别为num1和num2)

```
int carry = 0;
string res = "";
//遍历被乘数的每一个数位
for(int i = num.size() - 1; i >= 0; i--){
    int mul = num1 * num2 + carry;//两数相乘的结果加上上一次相乘产生的进位（carry的初始值为0）
    int sum = mul % 10;
    res.append(sum + '0');//res作为结果字符串倒序存储产生的和
    carry = mul / 10;//维护新的carry进位值，供给下一次mul的计算
}
res.append(carry);//切记不要漏了最后的进位也要加入结果字符串res
```

以上就是单个数位的乘数乘以多个数位的被乘数的基本实现代码

多个数位被乘数与多个数位乘数的相乘

在分析了单个数位乘数与多个数位被乘数的相乘过程之后，我们可以进一步分析多个数位的被乘数与多个数位乘数相乘的情况，其实这种情况可以看成遍历乘数的每一个位，将每一个位看成单个位数的乘数与被乘数的每一个位数相乘。

只不过这种情况还涉及到两次结果字符串的错位相加问题，为此，我编写了一段可以将任意长度字符串相加的子方法，这个方法返回两个任意长度字符串相加结果的字符串。

addString 方法的实现

这个方法的声明为

```
string addString(string, string);
```

首先为了存储任意长度的字符串，我使用了栈这一数据结构，将两个字符串的所有元素分别压入两个栈当中，每次取出栈顶的元素进行相加，维护一个变量carry用来存储两数相加产生的进位，将产生的和位压入存储结果的栈result当中。当一个栈为空，那么直接将另一个栈的剩余元素与进位变量carry相加，动态维护carry并将和位直接压入结果数组当中。
最后将result栈中所有元素输出到一个字符串当中，最后返回该字符串。

多个位数乘数与被乘数的相乘

介绍完如何将两个代表整数的字符串相加之后，下面回到我们的主方法，通过对乘数每一个数位与被乘数每一个数位相乘结果做加法运算的观察，每次加法运算都会空出一位不作运算，此时不妨直接将这一位数字作为最终的输出结果的一部分先存储起来，每次错位相加实际上就是被加数去掉了个位数“右移”后的结果与加数相加，以此类推，每一次错位运算都可以将被加数的个位数字“丢弃”，将其先存入结果栈当中，之后的运算就不考虑这一位了。
这样就实现了加数与被加数的在竖式乘法运算当中的”错位运算“

高精度乘法实现的细节部分

任意长度数字高精度乘法的实现

通过使用字符串存储输入的数字，可以有效防止因为数字过大导致的超过数据类型存储范围的问题，并使用两层循环，第一层是遍历乘数的每一个数位，第二层是遍历被乘数的每一个数位，这样就通过实现多次单个数位乘数与多数位被乘数的相乘的基本子问题解决任意数位乘数与任意数位被乘数的相乘问题。

命令行参数输入两个乘数的乘法实现

通过给 main 函数设置命令行参数可以实现project的基本要求，注意到一个细节，通过命令行参数输入两个数字的时候程序并不会提示诸如

`Please input two integers:`

的结果，即不需要用户再次输入两个数字进行运算，而是直接运算出结果。此处可以通过判断命令行参数的个数来判断是否需要用户输入数字。

代码思路如下所示：

```
int main(int argc, char **argv)
{
if (argc <= 1 )
    {
    //提示用户输入数字
    }
}
else{
    //直接将命令行参数赋给两个字符串，用于之后的运算
    for (int i = 1; i < argc; i++)
        {
            if (i == 1)
            {
                str1 = argv[i];
            }
            else
            {
                str2 = argv[i];
            }
        }
    //直接运算出结果
}
```

乘法结果正负的判断

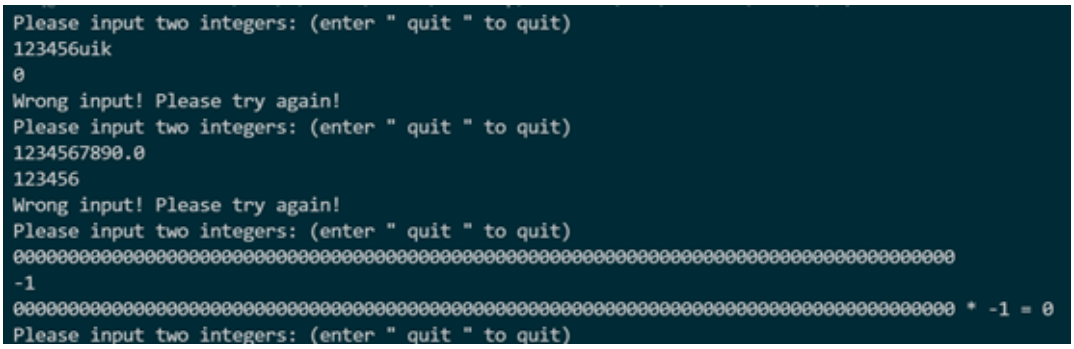
由于上述程序仅限于正整数的乘法计算，对于负数，在字符串上面最基本的体现就是字符串第一个字符并不是数字，而是一个 '-'的字符，所以在用户输入的数字之后要进行字符串首位字符类型的判断，如果是'-'就要采取相关措施，即：

如果两个字符串首位都是'-'，那么根据"负负得正"原则，输出的结果是整数，此时只要对两个字符串切除首位之后的字符串进行接下来的运算即可

如果恰有一个字符串首位是'-'，那么把结果标记为负数，如果最终计算结果不是0，那么就在输出结果之前，先输出'-'，再进行结果的输出。

程序的鲁棒性

程序的鲁棒性在这次project中的体现主要是在用户不小心输入了输了数字之外的字符串或是字符的时候要及时提示用户这次输入是不合规范的，这可以通过在遍历乘数与被乘数的过程中实现，一旦发现有一个字符不是数字，就退出运算，并提示用户这是一个不合法的输入。抵抗不合法的输入示例如图所示：



Some Improvements

1. 作为一个乘法器，考虑到用户使用这一种程序的感受，肯定不希望运行一次就只能做一次乘法运算，一定是希望自己能够控制它停止或者继续，重复不断地完成自己想要的乘法运算。为了满足用户的这一需要，本程序通过使用 goto语句精确控制程序的运行顺序，分别在程序运算结束之后与提示用户输入不符合规范之后回到程序的初始阶段重新开始读取用户的输入，直到用户输入了"quit"为止，实现了重复次数的乘法运算。
2. 在前面的篇幅中也介绍过，通过分析两数通过竖式作乘法运算的过程可以发现，每次错位相加都可以省略被加数的个位进行运算，这样就可以很大程度节省addString方法的运算时间与空间，也方便这一字符串相

加程序的编写，也不用考虑错位的位数关系，每次都是直接进行运算即可而且运算的最大位数即为加数的位数。

3. 通过使用了栈这一数据结构可以实现任意长度的两个数字的相乘运算，而不用考虑数组的长度是否足够的问题，实现了高精度的两数相乘运算。使用栈使得实现两个字符串相加的时候不用考虑两个字符串长度是否一致的问题，因为如果通过字符串实现这种运算的话，需要找到字符串的最高位，甚至如果两个字符串长度不一致还可能导致内存管理方面的问题，而使用栈存储数字的话，在运算时都是统一从栈顶弹出元素进行最低位的运算，提高了运行的效率。

Part 2 - Code

此次project的代码如下所示：

```
#include <iostream>
#include <string>
#include <stack>


typedef long long ll;

using namespace std;

string addString(string, string);
int main(int argc, char **argv)
{
    bool first_time = true;//判断是否是第一次进入程序，
    //因为涉及到命令行输入的问题，只有第一次才可以通过命令行输入两个数字
loop:

    stack<char> res;
    string f, s;//代表起始输入的两个字符串
    if (argc < 3 || !first_time)
    {

        cout << "Please input two integers: (enter \" quit \" to
quit)" << endl;
        cin >> f;
        //程序会一直循环进行指导用户输入quit为止
        if (f == "quit")
        {
```

```
            return 0;
        }
        cin >> s;
    }
    else
    {
        first_time = false;//通过命令行参数传入第一次运算的数字时，改变first_time的值为 false;
        for (int i = 1; i < argc; i++)
        {
            if (i == 1)
            {
                f = argv[i];
            }
            else
            {
                s = argv[i];
            }
        }
    }

    //当输入的第一个字符串为quit时，退出程序
    if (f == "quit")
    {
        return 0;
    }

    //判断结果的正负
    bool isNegative = false;
    if ((f[0] == '-' && s[0] != '-') || (f[0] != '-' && s[0] == '-'))
    {
        isNegative = true;
    }

    ll fsize = f.size();
    ll ssize = s.size();
    string first = "", second = "";

    //调整成为竖式乘法上端较大，下端较小，便于之后的计算
    if (fsize < ssize)
    {
        first = s;
        second = f;
    }
    else
    {
```

```
        first = f;
        second = s;
    }

    string last = "";

    //重新维护两个字符串长度，防止出现内存空间分配问题
    ssize = second.size();
    fsize = first.size();

    if (first[0] == '-')
    {
        fsize--;
        first = first.substr(1, first.size() - 1);//截取除了负号后面的一段字符串做之后的计算
    }
    if (second[0] == '-')
    {
        ssize--;
        second = second.substr(1, second.size() - 1);//截取除了负号后面的一段字符串做之后的计算
    }

    for (ll i = ssize - 1; i >= 0; i--) //i = 0
    {
        if (i == 0 && i == '-')
        {
            break;
        }
        int carry = 0;
        string sum1 = "";
        for (ll j = fsize - 1; j >= 0; j--)
        {
            int num1 = first[j] - '0';
            int num2 = second[i] - '0';

            if (j == 0 && second[j] == '-')
            {
                break;
            }
            //一旦检测到不是数字的字符就提示用户并且跳转到程序开头使程序重新运行
            if ((num1 > 9 || num1 < 0) || (num2 > 9 || num2 < 0))
            {
                cout << "Wrong input! Please try again!" << endl;
                goto loop;
            }
```

```
            int sum = num1 * num2 + carry;//存储两个单位数进行乘法运算的结果

            carry = sum / 10;//存储进位

            int s = sum % 10;//存储和位

            sum1 += s + '0';
        }

        sum1 += carry + '0';//最后不要忘了加上剩余的进位

        //翻转字符串
        string temp = sum1;
        for (ll k = 0; k < sum1.size(); k++)
        {
            temp[k] = sum1[sum1.size() - k - 1];
        }
        sum1 = temp;

        //last存储上一次进行加法运算的结果
        //初始情况last为空，此时不进行运算，直接将第一次运算的结果赋值给last
        if (last.empty())
        {
            last = sum1;
            res.push(last[sum1.size() - 1]);
            continue;
        }


        last = addString(last.substr(0, last.size() - 1), sum1);
        res.push(last[last.size() - 1]);//将个位数先存进结果栈当中，之后便不再考虑这一数位
    }
    for (ll i = last.size() - 2; i >= 0; i--)
    {
        res.push(last[i]);
    }

    string ans = "";
    //将 res 栈顶元素逐一输入到ans字符串当中，并且去掉前导零
    bool flag = false;//标记是否有第一个非零值出现
    while (!res.empty())
    {
        if (res.top() != '0')
```

```
        {
            flag = true;
        }
        if (!flag && res.top() == '0')
        {
            res.pop();
            continue;
        }
        ans += res.top();
        res.pop();
    }

    cout << f << " * " << s << " = ";
    //如果结果为负数且答案不为零，就先输出负号
    if (isNegative && ans.size() != 0)
    {
        cout << '-';
    }
    //如果答案为零，则在前面的"去掉前导零"的操作中并不会给字符串添加任何值，即此时字符串为空，需要手动输出'0'
    if (ans.size() == 0)
    {
        cout << "0" << endl;
    }
    //如果答案非零，则输出答案
    else
    {
        cout << ans << endl;
    }

    goto loop;//运算完成之后回到程序起始，再次进行新一轮运算
}

//add two string if both of them are composed of numbers
string addString(string str1, string str2)
{
    string res = "";
    stack<char> result;
    //考虑到字符串的长度可能非常长，长度可能超过所有的基本数据类型的大小，这里采用栈结构存储字符串的每一个字符
    stack<char> stk1;
    stack<char> stk2;
    for (char ch : str1)
    {
        stk1.push(ch);
    }
    for (char ch : str2)
```

```
    {
        stk2.push(ch);
    }
    int carry = 0;
    //在两个栈都非空的情况下，分别取两个栈顶元素进行运算，运算规则与前面类似
    while (!stk1.empty() && !stk2.empty())
    {
        //取栈顶元素作为两个加数
        int num1 = stk1.top() - '0';
        int num2 = stk2.top() - '0';
        stk1.pop();
        stk2.pop();

        int sum = num1 + num2 + carry;
        carry = sum / 10;
        result.push(sum % 10 + '0');
    }
    //检测到是哪一个栈非空就直接对该栈的元素进行对结果栈的导入
    if (stk1.empty())
    {
        while (!stk2.empty())
        {
            int num = stk2.top() - '0';

            stk2.pop();
            int sum = carry + num;
            carry = sum / 10;
            result.push(sum % 10 + '0');
        }
    }
    else
    {
        while (!stk1.empty())
        {
            int num = stk1.top() - '0';

            stk1.pop();
            int sum = carry + num;
            carry = sum / 10;
            result.push(sum % 10 + '0');
        }
    }
    //将结果栈中元素导入结果字符串当中
    while (!result.empty())
    {
        res += result.top();
```

```
        result.pop();
    }

    return res;
}
```

Part 3 - Result and Verification

Test case #1(基本样例)

解释：

第一种情况是通过命令行参数进行两个数字的传入，此时直接显示运算结果

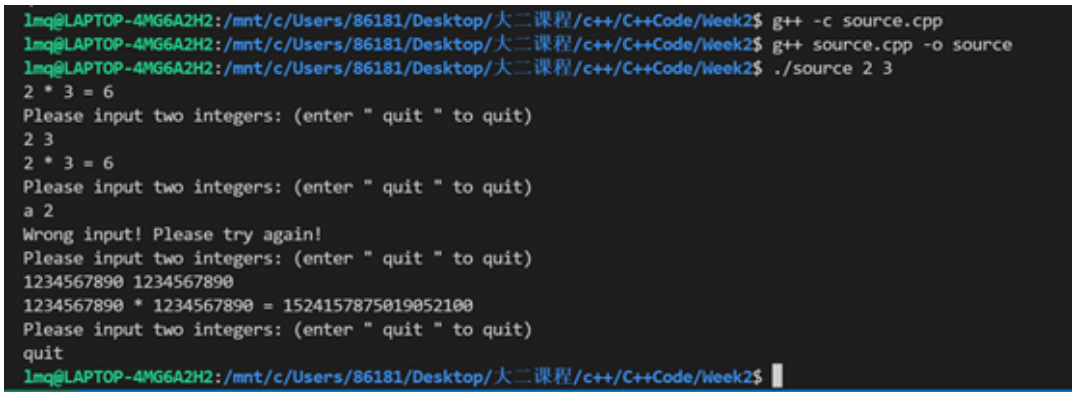
`2 * 3 = 6`

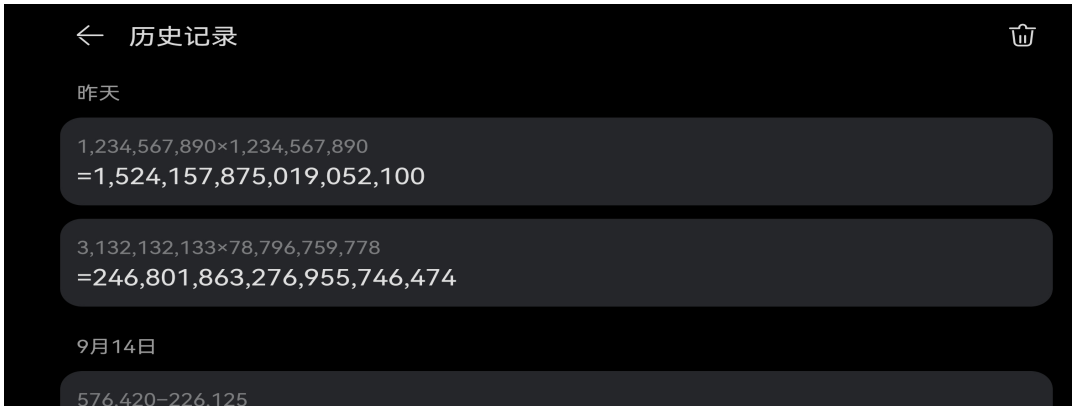
第二种是在程序运行当中进行数字的输入，回车之后显示运算结果，

第三种情况则是在考验程序的鲁棒性，输入 a 和 2，最后程序会提示用户这是一个不合法的输入，并请求用户重新输入一遍两个数字，

第四种情况则是考验程序的高精度乘法，实现两个大数相乘，最终经过计算器检验，计算结果正确。

测试以及验证如下两图所示：





Test case #2(测试高精度正整数乘法准确性)

测试结果如下图所示，经验证，结果均为正确。

第一组：

input: 111111111111111111111 222222222233333333333

output: 24691358025925925925864197530862962962963

第二组：

input:1234567890987654321 1234567890987654321

output:1524157877457704723228166437789971041

第三组：

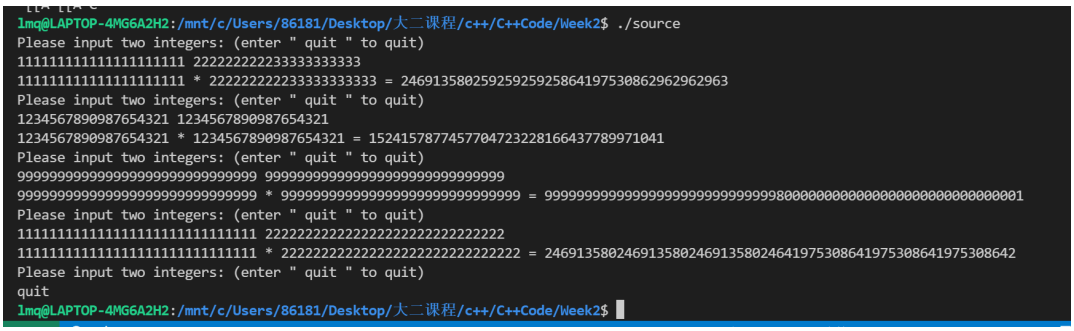
input: 999999999999999999999999999999 999999999999999999999999999999

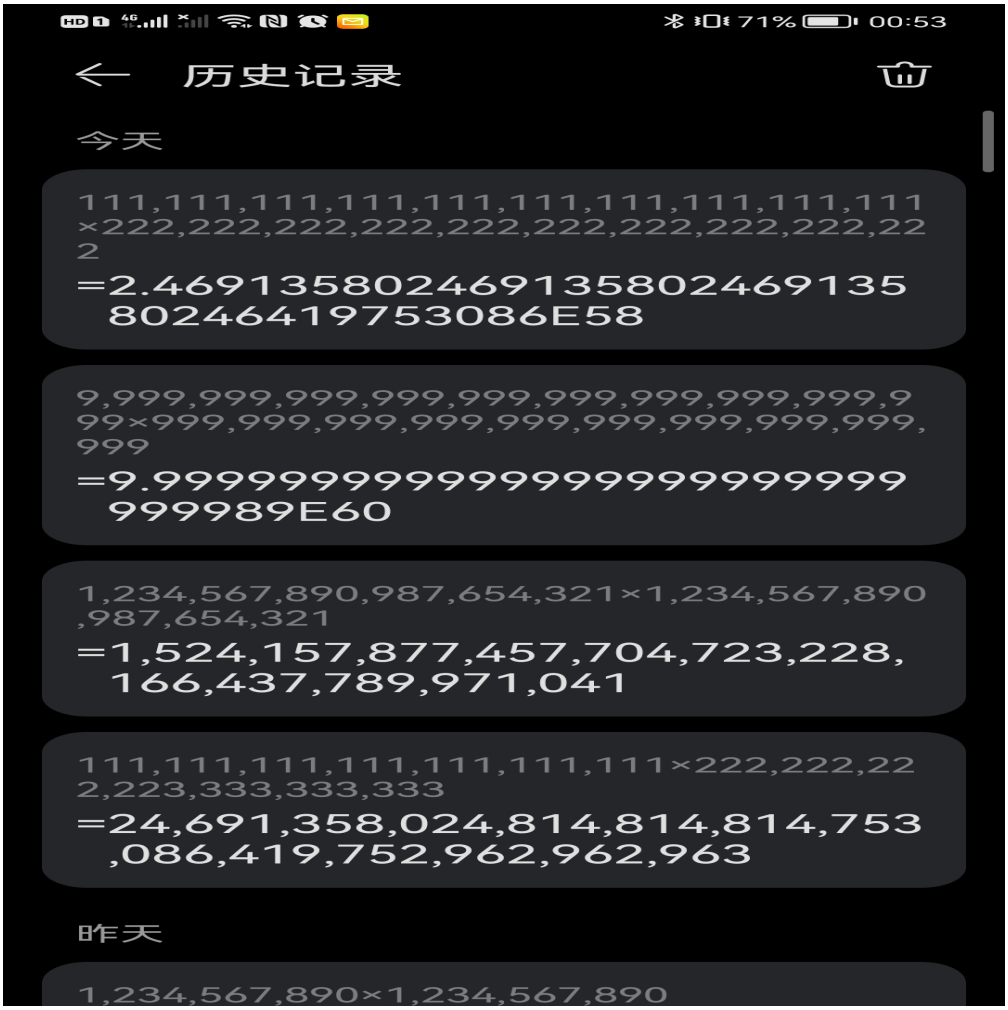
output
999999999999999999999999999999800000000000000000000000000000001

第四组：

input:11111111111111111111111111111 22222222222222222222222222222

output: 24691358024691358024691358024619753086419753086319753-8641975308642



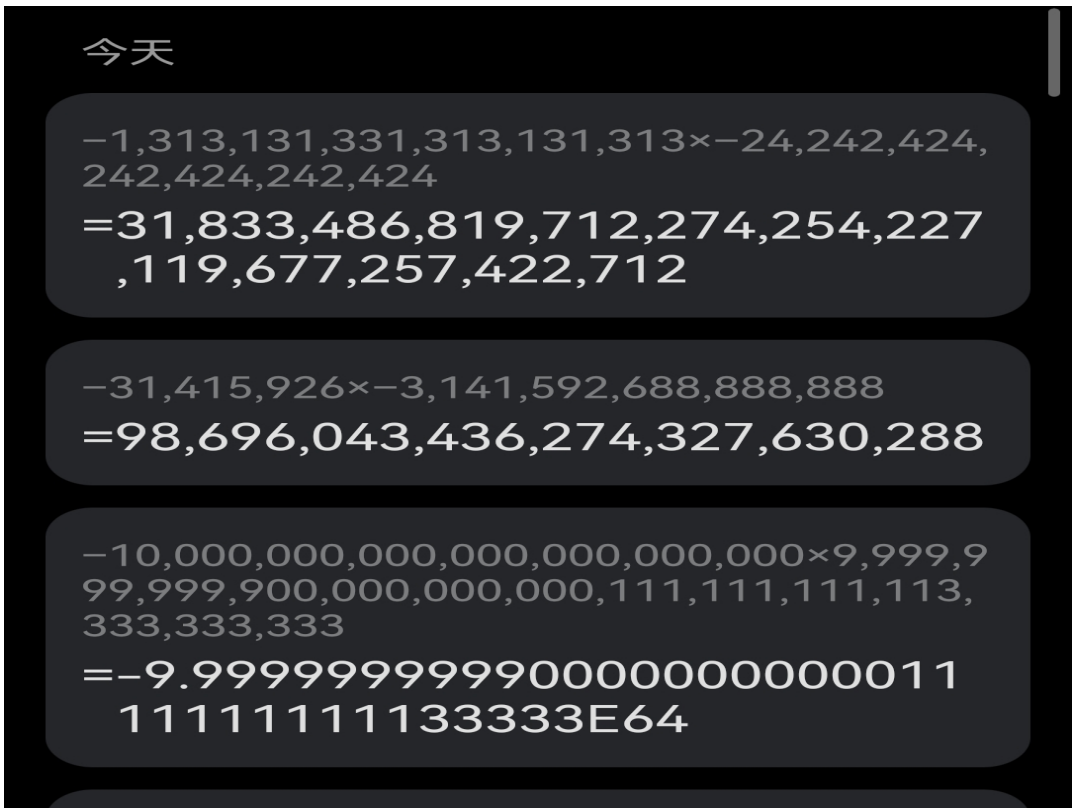


Test case #3(测试高精度负整数乘法准确性)

即使是负整数，本程序依然计算地十分准确

具体测试实验如下图所示：

```
Please input two integers: (enter " quit " to quit)
-10000000000000000000000000000000 99999999990000000001111111113333333333
-10000000000000000000000000000000 * 99999999990000000001111111113333333333 = -999999999900000000011111111133333333330000000000000000000000000000
Please input two integers: (enter " quit " to quit)
-31415926 -3141592688888888
-31415926 * -3141592688888888 = 986960434362743276302 88
Please input two integers: (enter " quit " to quit)
-13131313131313131313 -24242424242424242424
-13131313131313131313 * -24242424242424242424 = 318334863789409243948331802877257422712
Please input two integers: (enter " quit " to quit)
```



Test case #4 (程序鲁棒性)

下面继续进阶地测试几种测试样例，检验程序的鲁棒性：

第一组：当有一个数字是0时，另一个数字正常时，输出 0；

第二组：当两个数都是-0时，输出正常的计算结果0；

第三组以及之后的组：当两个输入的数当中存在不符合规范的数（即中间含有非数字元素时），提示用户这是一个不合法的输入并要求重新输入；

实验结果如下图所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/Week2$ ./source
Please input two integers: (enter " quit " to quit)
0 1009909
0 * 1009909 = 0
Please input two integers: (enter " quit " to quit)
-0 -0
-0 * -0 = 0
Please input two integers: (enter " quit " to quit)
-0 1-122321
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
-2-13141414 31427878
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
0 -131488jj
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
-313131 b
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
quit
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/Week2$
```

倒数第二组，当出现前导零的时候，将忽略前导零再进行计算，结果依然正常。

input: 0000313 2

output: 626

```
Please input two integers: (enter " quit " to quit)
0000313 2
0000313 * 2 = 626
```

最后一组，考虑到中文字符在ASCII码当中占两个字节的空间，和一般表示数字的字符不同，所以测试几组中文输入的数据，测试一下程序会不会崩溃：

```
Please input two integers: (enter " quit " to quit)
把？ 3
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
quit
PS C:\Users\86181\Desktop\大二课程\c++\C++Code\Week2> ./source
Please input two integers: (enter " quit " to quit)
3 3344加31
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
我喜欢学习 12315
Wrong input! Please try again!
Please input two integers: (enter " quit " to quit)
```

发现代码依旧运行正常，还是可以提示用户信息出错。

综合上面多组试验，可以证明代码的完备性以及精确性。

Part 4 - Difficulties & Solutions

在编写这次的乘法器代码时，遇到的几个问题以及解决方法如下：

困难1与解决方案

起初在实现字符串相加方法的时候是通过使用字符串直接相加，后来发现由于字符串长度不同，无法很便捷地同时取到字符串的末位数字进行相加，造成了方法编写的瓶颈，后来发现可以通过巧妙地使用两个栈去存储两个字符串的值，之后如果要进行运算就可以直接将栈顶元素弹出实现字符串末位数字的相加，使用栈也大大简化了该方法的编写以及运行效率。

困难2与解决方案

关于去掉前导零的问题: 在一次偶然的测试当中，发现有时候会输出前导零，这显然不是想要的结果，于是在最后通过结果栈逐个弹出栈顶元素组成结果字符串的过程中，就维护一个布尔变量`flag`，直到出现第一个非零元素`flag`才会变为真，直到`flag`变为真才将栈顶元素输出到字符串当中，这样就解决了前导零的问题。

困难3与解决方案

关于 char 类型 与 int 类型转换的问题：有多处地方原本是希望通过 字符类型 char 去存储的，后来却发现经常忽视掉 字符类型与整数类型的转换，此类型的转换大致如下公式所示：

```
char ch_num;//假设为意义0-9的字符，如'1', '2'等等
int num;
//则满足下列关系：
num = ch_num - '0';
```

经常导致存储的字符并不是目标要存储的字符，后来通过`debug`才找出来几处这样的错误，整段程序才得以正常运行。

困难4与解决方案

字符倒序输出的问题：使用栈或者字符串存储运算的结果经常是倒序存储的，在编写程序的过程中经常遇到结果并非是我想要的目标结果，往往是顺序恰好相反，解决方案就是在需要使用正序结果的时候需要将栈或者存储有目标结果倒序的字符串进行翻转，翻转程序如下所示:

```
//翻转字符串
        string temp = sum1;//sum1是存储有倒序的目标结果的字符串
        for (long long k = 0; k < sum1.size(); k++)
        {
            temp[k] = sum1[sum1.size() - k - 1];
        }
        sum1 = temp;
```

对每一次需要用到的字符串先进行一次判定，判断是否是我们想要的字符串还是仅仅是它的倒序形式，如果是后者，则需要在使用前进行字符串的翻转或者是栈倒序输出到字符串当中，这样可以很大程度减少程序的BUG

Part 5 - Thinking and Summarize

通过编写这段乘法器代码，我更加深刻地了解了乘法的运算机制，实现了错位相加的方法，也更深刻地了解了c++数据类型的范围，因为起初我只使用了int类型来实现这个乘法器，发现精度很低，当输入的数字位数稍微大一点程序就近乎崩溃，所以通过这次写乘法器我也更加深刻了解了c++数据类型的魅力，同时在编写程序的过程当中也更加注意到要完善自己的编码风格，也更加深入地了解了c++代码编写的规范性，例如一个好的代码需要有足够的注释让别人看懂，变量名的起名也要易于理解。在鲁棒性方面，也要防止一些不合法的输入等等。也更加为用户着想，例如用户可能不小心输入了不合法的数值，就要及时提醒他重新输入数值，用户可能并不想一次执行只能运算一次乘法，所以程序也设置了循环运行的机制，综上所述，这次小项目带给我不小的收获。

以上就是我对此次项目的思考，感谢阅读！