《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软件工程 | 姓名 | 梅子羽 | 学号 | 2327406107 |
| 实验布置日期 | | 2024.9.23 | | 提交  日期 | 2024.10.12 | | 成绩 |  |

课程实践实验2-2：大整数的加法和乘法

## 一、问题描述及要求

在C++中，实现大整数的加法和乘法通常使用字符串或数组来模拟，因为内置数据类型如int或long long的范围有限，不能处理非常大的数字。尝试运用所学顺序表的知识来实现大整数加法和乘法。

## 二、概要设计

### (1) 实验内容理解

该实验的目标是实现一个能够处理大整数的类，并通过运算符重载来支持大整数的加法和乘法操作。由于普通的基本数据类型如 int 或 long 无法处理超过其存储范围的大数，因此需要将大整数以字符串的形式存储，并在实现加法和乘法时模拟手工计算的过程。

### (2) 系统功能列表

#### 功能列表：

1. **输入大整数**：用户可以输入任意长度的两个大整数。
2. **大整数的加法**：对两个大整数进行加法运算，并输出结果。
3. **大整数的乘法**：对两个大整数进行乘法运算，并输出结果。
4. **大整数的输出**：通过重载 << 运算符实现大整数的输出。
5. **去除前导零**：确保在每次运算后，大整数结果中没有前导零。

### (3) 程序运行的界面设计

用户界面是一个简单的命令行交互。用户通过键盘输入两个大整数，程序将计算并输出它们的加法和乘法结果。示例如下：

请输入第一个大整数：123456789123456789

请输入第二个大整数：987654321987654321

The sum is: 1111111111111111110

The product is: 121932631112635269121932631112635269

### (4) 总体设计思路

#### 采用的数据结构：

* **大整数的存储**：使用 string 存储大整数，以便处理任意长度的数字。
* **辅助容器**：使用vector 存储中间计算结果，例如在乘法过程中，保存每一位数的结果并进行累加。

#### 类的设计：

* **BigNumbers 类**：
  + **数据成员**：
    - num：存储大整数的字符串。
  + **方法**：
    - operator+：重载加号运算符，实现大整数的加法。
    - operator\*：重载乘号运算符，实现大整数的乘法。
    - operator<<：重载输出流运算符，实现输出大整数。
* **类之间的关系**：
  + BigNumbers 类是主要处理大整数的类。
  + 通过友元函数 operator<< 与 ostream 类进行交互，实现对大整数的输出。

### (5) 程序结构设计

#### 文件结构：

1. **BigNumbers.h**：声明 BigNumbers 类，包括其构造函数和重载的运算符。
2. **BigNumbers.cpp**：实现 BigNumbers 类的所有方法。
3. **main.cpp**：包含主程序逻辑，进行输入输出，调用 BigNumbers 类实现大整数的运算。

#### 数据结构和算法：

* **加法算法：**加法操作是通过模拟手动的进位加法完成的。两个大整数从最低位开始逐位相加，若有进位则保留到下一位继续处理，最终得到结果。
* **乘法算法：**乘法操作通过逐位相乘并加和实现，类似于手工计算的乘法步骤。每一位的乘积结果按照位数补零后累加到最终结果中。

#### 设计理由：

* **字符串存储大整数**：可以处理任意长度的数字，且易于进行逐位的运算。
* **逐位运算的算法**：加法和乘法模拟了手工计算过程，确保了精度，且易于实现运算符重载。

## 三、详细设计

### 大整数类中加法和乘法实现的分析

#### 加法实现分析

在大整数类的加法操作中，关键点是逐位处理两个数的相加，并模拟手工计算中的进位操作。考虑到两个数的位数可能不相同，需要从最右边开始逐位相加，处理进位，并逐步构建结果字符串。以下是对具体难点的分析：

##### 1. ****逐位相加****：

* 加法从两个字符串（代表大整数）的末尾开始，逐位相加。这是因为我们通常从最低位开始进行加法运算。
* 对于较短的数字，在长数字未处理完之前，把短数字的高位当作 0 处理。

##### 2. 处理进位：

* 每一位相加时，除了考虑两个对应位的数字外，还需要加上前一位的进位。
* 通过 sum / 10 来判断是否有进位，sum % 10 则为当前位的实际结果。
* 这个操作保证了每一位的运算结果都正确，同时处理了进位。

##### 3. 结果的反转：

* 因为加法是从右往左（从低位到高位）计算的，因此计算出的结果是逆序的。需要在最终返回结果前将字符串反转，得到正确的顺序。

##### 加法实现的难点：

* **处理进位**：确保每一位计算完毕后，进位能正确地保留到下一次计算中。
* **不同长度大整数的处理**：需要处理两个数位数不同的情况，短的数字在高位要自动补零。
* **结束条件**：当两个数都计算完且没有进位时，算法终止。

#### 乘法实现分析

乘法比加法更复杂，因为它需要对每一位进行乘积计算，并将所有的部分结果累加起来，模拟手工乘法中的步骤。该算法的难点和关键点主要集中在逐位相乘和结果的累加上。

##### 1. 逐位相乘：

* 与手工计算乘法类似，对于两个大整数，每一位都要和另一个数的每一位相乘，得到一个部分乘积。
* 每一位的乘积都会有一个相应的“位移”，这是因为乘法中的位数会影响结果的排列方式。

##### 2. 累加中间结果：

* 每次的部分乘积需要加上先前计算的结果。为了实现这一点，需要多次调用大整数的加法运算（即前面实现的 operator+）。
* 每个部分乘积在进行累加时，需要在末尾增加相应数量的零，以表示位移。这一步非常类似于手工乘法中的位移操作。

##### 3. 处理进位：

* 逐位相乘时，单次乘积可能会超出个位，需要进行进位处理。和加法类似，进位通过 product / 10 计算，product % 10 存储当前位的结果。

##### 乘法实现的难点：

* **多次累加**：每次乘积计算后的中间结果都需要通过加法累加，这使得乘法的整体复杂度较高。如何优化累加过程是一个难点。
* **位移的处理**：每次乘积结果都需要相应的位移（即在末尾补零），以确保位数的正确排列。
* **效率问题**：尽管该算法模拟了手工计算过程，但由于乘法涉及到大量的逐位相乘和累加，因此效率较低，对于特别大的数，可能会产生性能瓶颈。

#### 总结

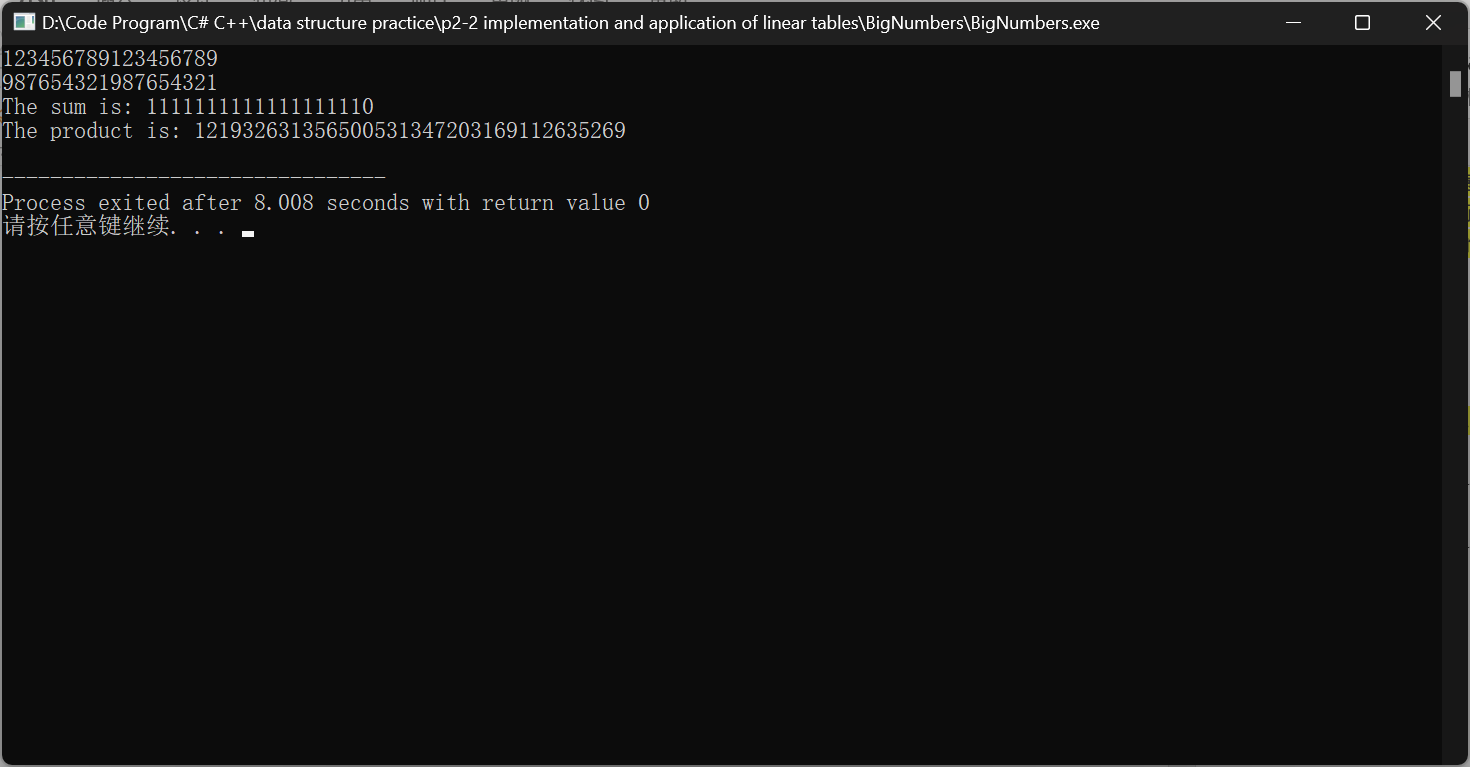
* **加法**：核心在于逐位相加和处理进位，重点是确保不同长度的大整数能正确计算，结果字符串需要反转。
* **乘法**：核心是逐位相乘和多次累加，每次乘积结果都要处理进位并加到最后的结果中，特别要注意位移的处理。

整体上，这两种算法都是手工计算的模拟，结合了字符串的处理，确保支持任意长度的大整数计算。

## 四、实验分析与探讨

样例1：普通测试，数A：123456789123456789 数B：987654321987654321，主要测试

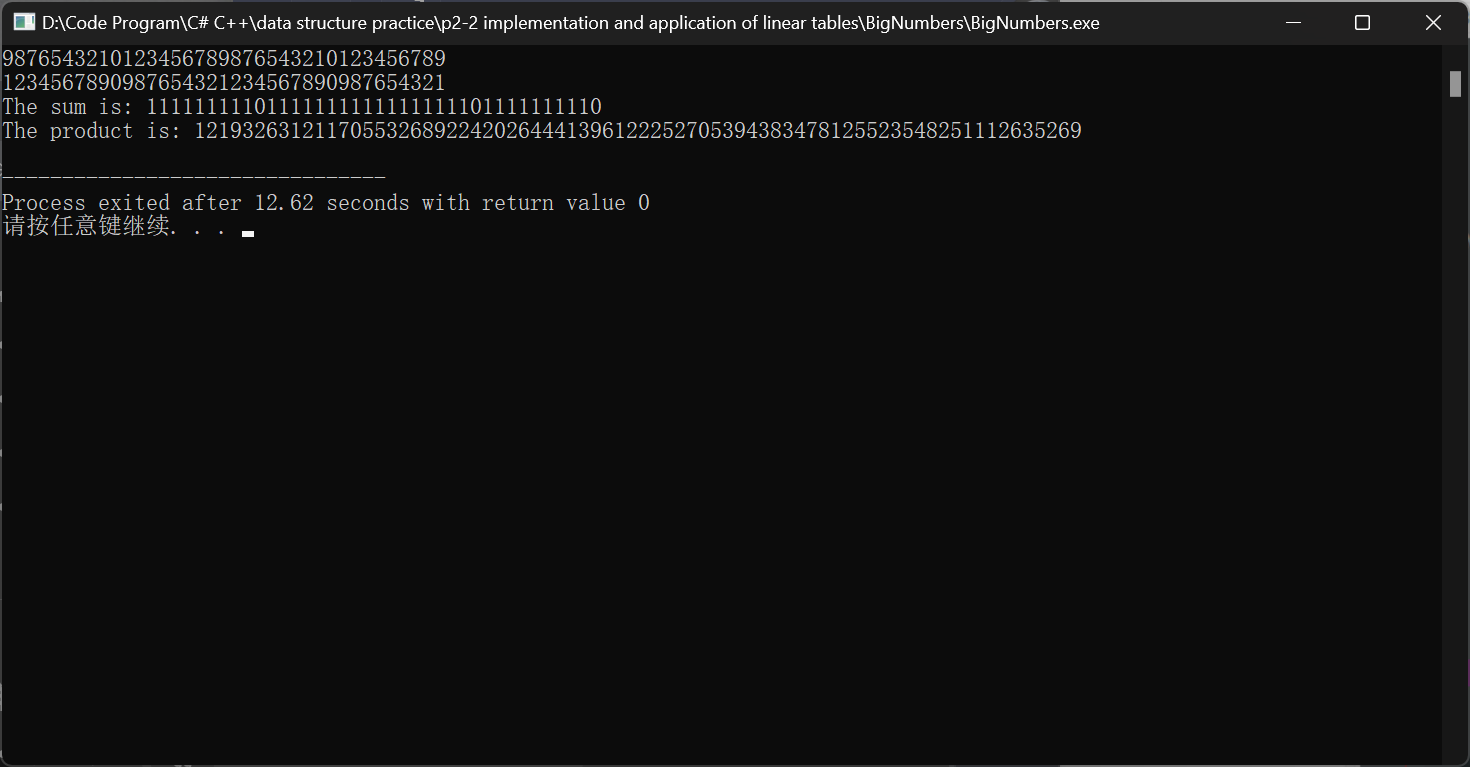
大整数的加法和乘法函数基本功能，输出正确。



样例2：普通测试，数A：9876543210123456789876543210123456789

数B：1234567890987654321234567890987654321

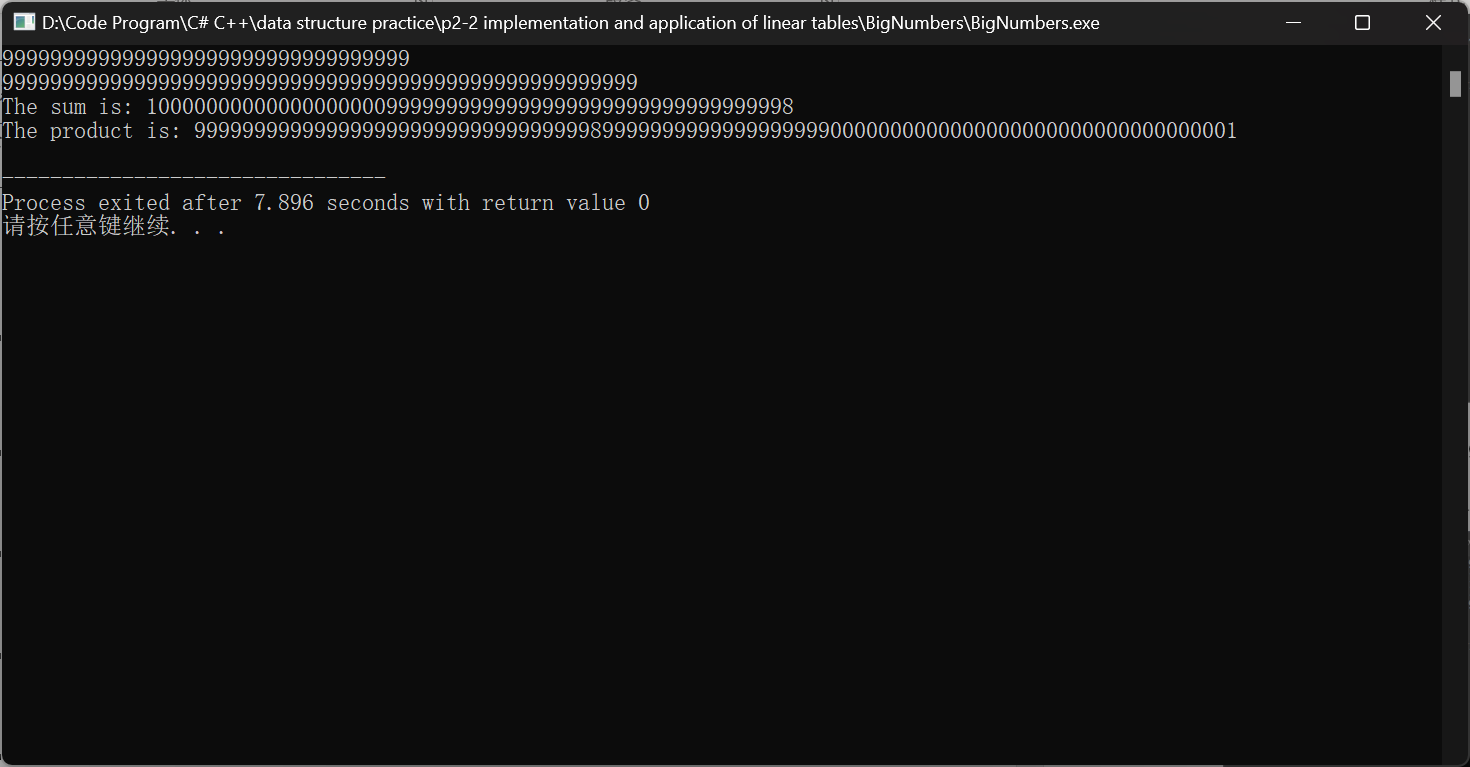
其余同样例1，输出正确。



样例3：极端测试，数A：9999999999999999999999999999999999

数B：99999999999999999999999999999999999999999999999999999

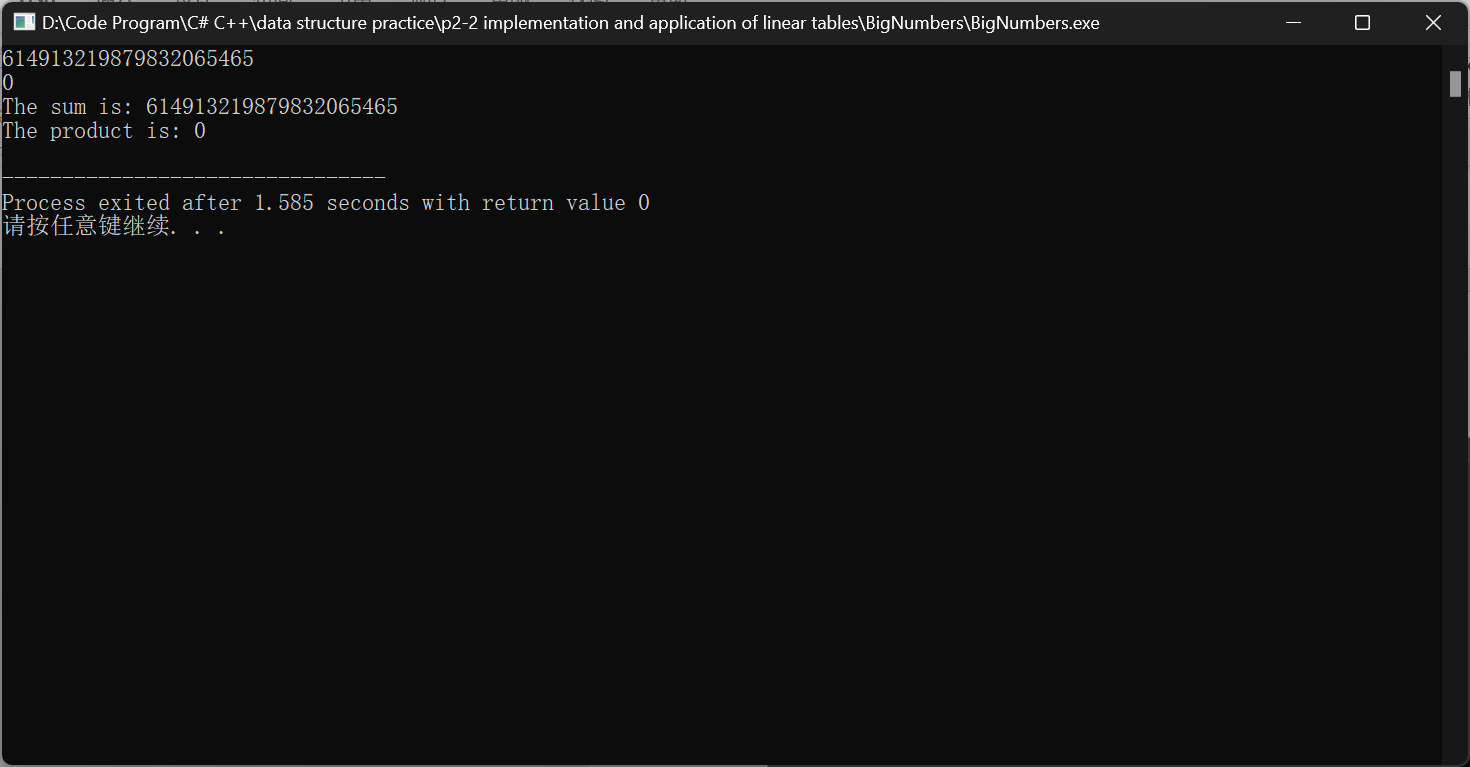
目的是测试加乘法函数中的进位处理，输出正确：



样例4：极端测试，数A：614913219879832065465

数B：0

目的是测试加乘函数对于输入有0的处理，也是一种多位大整数与一位整数的特殊情况，输出正确



总结：本次实验的关键函数加法与乘法主要是利用计算机的性能来模拟手写计算的过程，从而实现大整数的计算。加法的时间复杂度是O(n)，而乘法是O(n^2)，这是手写计算方法所决定的。乘法实际上就是重复了n遍的加法。整体下来最麻烦的就是处理进位问题，其中尤其是头部的进位问题，这对循环体内的语句要求较高，我也是改了好几遍才做到正确

## 五、小结

#### 一、完成的内容

1. **大整数类的基本功能实现**：
   * 成功实现了大整数类 BigNumbers，包括构造函数、加法和乘法运算符重载。
   * 实现了加法和乘法的逻辑，支持任意长度的大整数计算。
   * **输入输出的功能**：
   * 通过运算符重载实现了输出流的重载，使得大整数的输出更加直观。
   * 使用标准输入读取用户输入的字符串形式的大整数。
2. **设计文档**：
   * 编写了系统功能列表和程序结构设计，清晰描述了类的功能及其关系。
   * 提供了算法实现分析，着重分析了加法和乘法实现的关键点和难点。

#### 二、未完成的内容

1. **异常处理**：
   * 当前实现未考虑对输入的有效性进行校验，如非数字字符的输入或输入过大的数字导致内存问题。
2. **性能优化**：
   * 尽管实现了基本的加法和乘法功能，但在处理非常大的数字时，算法的效率仍有待提高。例如，乘法可以考虑使用更高效的算法（如 Karatsuba 算法）。
3. **更全面的测试**：
   * 当前实现没有进行详细的单元测试，缺少对边界情况（如零、负数、非常大的数字等）的测试用例。
4. **输入输出文件的支持**：
   * 现有实现仅支持从标准输入读取数据，尚未支持从文件读取大整数。

#### 三、选做部分的完成情况

* 在当前实现中，选择性地完成了对运算符重载的实现，使得大整数的使用更加便捷。但未实现其他额外的功能（如模运算、除法等），这些功能可以在未来的扩展中加入。

#### 四、程序的局限性

* **内存限制**：由于使用字符串表示大整数，程序的内存使用量将取决于字符串的长度。对于极大的数字，可能会导致内存不足的问题。
* **性能**：当前的加法和乘法实现是线性的，虽然能够正确处理大整数，但对于处理非常大的数字时，性能较低，可能无法满足实时计算的需求。

#### 五、展望与建议

1. **进一步学习**：
   * 深入学习数值计算中的高级算法，例如快速乘法和FFT（快速傅里叶变换）算法，以提高大整数运算的效率。
   * 掌握更复杂的数据结构，研究如何优化大整数的存储和运算，例如使用链表或树结构。
2. **完善程序**：
   * 加入更多的异常处理，以确保程序对用户输入的鲁棒性。
   * 增加对输入文件的支持，使得用户可以从文件中批量读取大整数进行运算。
3. **优化性能**：
   * 研究和实现更高效的大整数乘法算法，例如 Karatsuba 算法或分治法。
   * 在乘法中尝试使用位运算、查表等优化策略。
4. **详细的文档和注释**：
   * 在代码中添加更多的注释，并撰写更详细的使用说明，便于其他开发者理解和维护。

#### 六、必要的补充说明

* **输入格式要求**：程序目前要求用户从标准输入提供两个大整数，格式应为两个非负整数字符串，之间用空格分隔，例如：

12345678901234567890 98765432109876543210

* **输入长度限制**：尽管程序可以处理任意长度的大整数，但在实践中，过大的输入可能导致性能问题，因此建议输入的数字应合理控制在数千位以内，以确保程序能高效运行。

通过本次项目的设计与实现，不仅提升了对大整数运算的理解，也增强了在实际应用中对数据结构与算法的应用能力。未来将继续努力，探索更多的可能性与实现方向。

## 附录：源代码

1.实验环境：Dev-C++ 5.11集成开发环境，C++11标准。

2、

（1）//BigNumbers.h

#ifndef BIGNUMBERS\_H

#define BIGNUMBERS\_H

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

using namespace std;

class BigNumbers

{

private:

string num;

public:

BigNumbers(const string&);

BigNumbers operator+(const BigNumbers&)const;

BigNumbers operator\*(const BigNumbers&)const;

friend ostream& operator<<(ostream&, const BigNumbers&);

};

#endif

（2）//BigNumbers.cpp

#include "BigNumbers.h"

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

using namespace std;

BigNumbers::BigNumbers(const string &val) : num(val)

{

while(num.size() > 1 && num[0] == '0')

num.erase(0, 1);

}

BigNumbers BigNumbers::operator+(const BigNumbers& other)const

{

string num1 = num;

string num2 = other.num;

string result;

int carry = 0, sum = 0;

int i = num1.size() - 1;

int j = num2.size() - 1;

while(i >= 0 || j >= 0 || carry)

{

int digit1 = (i >= 0) ? num1[i --] - '0' : 0;

int digit2 = (j >= 0) ? num2[j --] - '0' : 0;

sum = digit1 + digit2 + carry;

carry = sum / 10;

result += sum % 10 + '0';

}

reverse(result.begin(), result.end());

return BigNumbers(result);

}

BigNumbers BigNumbers::operator\*(const BigNumbers& other)const

{

string num1 = num;

string num2 = other.num;

vector<int> result(num1.size() + num2.size(), 0);

for(int i = num2.size() - 1; i >= 0; i --)

{

for(int j = num1.size() - 1; j >= 0; j --)

{

int digit1 = num1[j] - '0';

int digit2 = num2[i] - '0';

int sum = digit1 \* digit2 + result[i + j + 1];

result[i + j + 1] = sum % 10;

result[i + j] += sum / 10;

}

}

string product = "";

for(int i : result)

product += i + '0';

return BigNumbers(product);

}

ostream& operator<<(ostream &os, const BigNumbers &bignum)

{

os << bignum.num;

return os;

}

（2）//main.cpp

#include "BigNumbers.h"

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

/\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/

void testFunc(string &bigint1, string &bigint2)

{

BigNumbers num1(bigint1);

BigNumbers num2(bigint2);

cout << "The sum is: " << num1 + num2 << endl;

cout << "The product is: " << num1 \* num2 << endl;

return;

}

int main(int argc, char\*\* argv)

{

string bigint1, bigint2;

cin >> bigint1 >> bigint2;

testFunc(bigint1, bigint2);

return 0;

}