Правительство Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Кафедра «Компьютерной безопасности»

Отчет к лабораторной работе №4

по дисциплине

«Языки программирования»

Работу выполнил студент группы СКБ242 _____ П.В. Жучков

подпись, дата

Работу проверил _____ С.А. Булгаков

подпись, дата

Москва, 2024

Содержание

1. Постановка задачи

Доработать класс «целое произвольной длины» из Лабораторной работы №3. Для класса разработать конструктор копирования. Используя перегрузку операторов (operator) разработать арифметику объектов, включающую действия умножения над объектами и стандартными целыми типами. Организовать операции в виде конвейера значений, с результатом (новым объектом) и сохранением значений входных операндов.

2. Алгоритм решения

Для реализации умножения созданного класса, нужно перегрузить операцию умножения. Для этого напишем вспомогательную функцию, перемножающую абсолютные значения объектов в виде строки. Будем пользоваться ранее написанным сложением. Также напишем методы (которые должны были быть реализованы в предыдущей лабораторной работе) доступа к отдельным частям класса и конструктор копирования.

3. Реализация задачи

Перегрузка оператора умножения реализован довольно просто. В зависимости от знаков множителей определяется знак результата. Произведение определяется вспомогательной функцией `stringMultiply`.

```
BigInt operator*(const BigInt& left, const BigInt& right) {
    if (left.sign == right.sign)
        return BigInt(true, stringMultiply(left.value,
right.value));
    else
        return BigInt(false, stringMultiply(left.value,
right.value));
}
```

Также реализована перегрузка операторов с другими типами данных

```
BigInt operator*(int left, const BigInt& right)  {return
BigInt(left) * right;}
BigInt operator*(std::string left, const BigInt& right)
{return BigInt(left) * right;}
BigInt operator*(short left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(long left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(char left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(const BigInt& left, int right) {return left
* BigInt(right);}
BigInt operator*(const BigInt& left, std::string
right){return left * BigInt(right);}
BigInt operator*(const BigInt& left, long right){return left
* BigInt(right);}
BigInt operator*(const BigInt& left, short right){return
left * BigInt(right);}
BigInt operator*(const BigInt& left, char right){return left
* BigInt(right);}
```

Функция `stringMultiply` реализует умножение в столбик.

```
std::string stringMultiply(const std::string& adin, const
std::string& dva){
    int a, d = dva.size() - 1;
    int dop = 0;
    std::string result = "0", tmp;
    while (d >= 0){
```

```
            tmp = "";
            a = adin.size() - 1;
            for (int i = 0; i < (int)dva.size() - d - 1; ++i){
                tmp.push_back('0');
            }
            while(a >= 0 || dop){
                int prod;
                prod = (a >= 0 ? adin[a--] - '0' : 1) * (dva[d]
- '0') + dop;
                dop = prod / 10;
                tmp.push_back((prod % 10) + '0');
            }
            d--;
            std::reverse(tmp.begin(), tmp.end());
            result = stringSum(result, tmp);
        }
        return result;
}
```

Конструктор копирования (вывод copy используется для тестирования):

```
 BigInt::BigInt(BigInt const &obj){
    value = obj.value;
    sign = obj.sign;
    std::cout << "copy" << "\n";
}
```

Методы для доступа к составным частям объекта:

```
const std::string& BigInt::get_value() const{
    return value;
}
void BigInt::set_value(const std::string& str) {
    value = str;
}
const bool BigInt::get_sign() const{
    return sign;
}
void BigInt::set_sign(const bool s) {
    sign = s;
}
```

Также стоит заметить, что программа принимает только целые числа в стандартном виде. То есть, программа не будет корректно работать с буквами, дробными числами, числами ч ведущими нулями

4. Тестирование

Используя ввод из файла main.cpp получим:

bigint: -2000000 -2000000

int: 2000000 2000000 -2000000 -2000000

int: 2000000 2000000 -2000000 -2000000

using of copy constructor: copy

Заметим, что при изменении мест множителей, произведение осталось неизменным. Также заметим, что конструктор копирования сработал. Тесты выполнены корректно.

Приложение А (main.cpp)

```
#include <iostream>
#include <string>
#include <typeinfo>
#include "class.h"
void f(BigInt n){}
int main(){
    std::string strp = "1000", strm = "-1000";
    int ip = 1000, im = -1000;
    BigInt a = 2000;
    BigInt b = -1000;

    std::cout << "bigint: " << a * b << ' ' << b * a <<
'\n';
    std::cout << "string: " << strp * a << ' ' << a * strp
<< ' ' << a * strm << ' ' << strm * a <<'\n';
    std::cout << "int: " << ip * a << ' ' << a * ip << ' '
<< a * im << ' ' << im * a <<'\n';
    std::cout << "using of copy constructor: ";
    f(a);
    return 0;
}
```

Приложение Б (class.cpp)

```
#include "class.h"
#include <string>
#include <iostream>
#include <algorithm>
#include <stdexcept>
#include <cstdlib>
#include <sstream>

//constructors
BigInt::BigInt() {value = "0"; sign =
true;}           //defolt constructor
BigInt::BigInt(bool insign, std::string str) {     //bool +
string
    sign = insign;
    value = str;
    while (value.size() > 1 && value[0] == '0') {
            value.erase(0, 1);
    }
    if (value == "0") sign = true;
}
BigInt::BigInt(std::string& str) {                  //string
constructor
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
BigInt::BigInt(const char* str)
{                   //cstring constructor
    sign = str[0] == '-' ? false : true;
    value = sign ? std::string(str) : std::string(str + 1);
}
BigInt::BigInt(int num){                            //int
constructor
    std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);

}
BigInt::BigInt(short num){                          //short
constructor
     std::stringstream ss;
```

```
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
BigInt::BigInt(long
num){                              //long  constructor
    std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
BigInt::BigInt(char
num){                              //char  constructor
     std::stringstream ss;
    ss << num;
    std::string str = ss.str();
    sign = str[0] == '-' ? false : true;
    value = sign ? str : str.substr(1);
}
BigInt::BigInt(BigInt const &obj){
    value = obj.value;
    sign = obj.sign;
    std::cout << "copy" << "\n";
}
BigInt::~BigInt(){}

//dop functions
std::string stringDif(const std::string& big, const
std::string& small){     //substract abs values
    std::string result;
    int b = big.size() - 1; int s = small.size() - 1;
    int dop = 0;
    while (b >= 0 || s >= 0){
        int dif, right = (s >= 0 ? small[s--] - '0' : 0),
left = big[b--] - '0';
        dif = left - right - dop;
        if (dif < 0){
            dif += 10;
            dop = 1;
        } else {
            dop = 0;
        }
```

```
        result.push_back(dif + '0');
    }
    while(result.size() > 1 && result[result.size() - 1] ==
'0') {
            result.erase(result.size() - 1, 1);
    }

    std::reverse(result.begin(), result.end());
    return result.empty() ? "0" : result;
}
std::string stringSum(const std::string& adin, const
std::string& dva){     //sumarize abs values
    std::string result;
    int i = adin.size() - 1, j = dva.size() - 1;
    int dop = 0;
    while (i >= 0 || j >= 0 || dop == 1){
        int sum = 0;
        sum = (i >= 0 ? adin[i--] - '0' : 0) + (j >= 0 ?
dva[j--] - '0' : 0) + dop;
        dop = sum / 10;
        result.push_back((sum % 10) + '0');
    }
    std::reverse(result.begin(), result.end());
    return result;
}
std::string stringMultiply(const std::string& adin, const
std::string& dva){
    int a, d = dva.size() - 1;
    int dop = 0;
    std::string result = "0", tmp;
    while (d >= 0){
        tmp = "";
        a = adin.size() - 1;
        for (int i = 0; i < (int)dva.size() - d - 1; ++i){
            tmp.push_back('0');
        }
        while(a >= 0 || dop){
            int prod;
            prod = (a >= 0 ? adin[a--] - '0' : 1) * (dva[d]
- '0') + dop;
            dop = prod / 10;
            tmp.push_back((prod % 10) + '0');
        }
        d--;
```

```
        std::reverse(tmp.begin(), tmp.end());
        result = stringSum(result, tmp);
    }
    return result;
}
bool firstBigger(const std::string& adin, const std::string&
dva){     //compare abs values
    if(adin.length() > dva.length()) return true;
    if(adin.length() < dva.length()) return false;
    for (int i = 0; i < int(adin.length()); ++i){
        if (adin[i] > dva[i]) return true;
        if (adin[i] < dva[i]) return false;
    }
    return true;
}

//get set functions
const std::string& BigInt::get_value() const{
    return value;
}
void BigInt::set_value(const std::string& str) {
    value = str;
}
const bool BigInt::get_sign() const{
    return sign;
}
void BigInt::set_sign(const bool s) {
    sign = s;
}

//operators
BigInt BigInt::operator=(const BigInt& other) {
    if (this != &other) {
        value = other.value;
        sign = other.sign;
    }
    return *this;
}
BigInt BigInt::operator=( short num){
    *this = BigInt(num);
    return *this;
}
BigInt BigInt::operator=( char num){
    *this = BigInt(num);
```

```
    return *this;
}
BigInt BigInt::operator=( int num){
    *this = BigInt(num);
    return *this;
}
BigInt BigInt::operator=( long num){
    *this = BigInt(num);
    return *this;
}

std::ostream& operator<<(std::ostream& out, const BigInt&
num) {
    if (num.sign == false) out << '-';
    out << num.value;
    return out;
}
std::istream& operator>>(std::istream& in, BigInt& num){
    std::string input;
    in >> input;
    if (input[0] == '-') {num.sign = false; num.value =
input.substr(1);}
    else {num.sign = true; num.value = input;}
    return in;
}

BigInt operator+(const BigInt& left, const BigInt& right) {
    if (left.sign == right.sign) {
        return BigInt(left.sign, stringSum(left.value,
right.value));
    }
    if (left.sign){
        if(firstBigger(left.value, right.value)){
            return BigInt(true, stringDif(left.value,
right.value));
        }
        return BigInt(false, stringDif(right.value,
left.value));
    } else {
        if(firstBigger(right.value, left.value)){
            return BigInt(true, stringDif(right.value,
left.value));
        }
```

```
        return BigInt(false, stringDif(left.value,
right.value));
    }
}
BigInt operator-(const BigInt& left, const BigInt& right){
    return left + BigInt(!right.sign, right.value);
}
BigInt operator*(const BigInt& left, const BigInt& right) {
    if (left.sign == right.sign)
        return BigInt(true, stringMultiply(left.value,
right.value));
    else
        return BigInt(false, stringMultiply(left.value,
right.value));
}

//+++++++++++++
BigInt operator+(int left, const BigInt& right)  {return
BigInt(left) + right;}
BigInt operator+(std::string left, const BigInt& right)
{return BigInt(left) + right;}
BigInt operator+(short left, const BigInt& right) {return
BigInt(left) + right;}
BigInt operator+(long left, const BigInt& right) {return
BigInt(left) + right;}
BigInt operator+(char left, const BigInt& right) {return
BigInt(left) + right;}
BigInt operator+(const BigInt& left, int right) {return left
+ BigInt(right);}
BigInt operator+(const BigInt& left, std::string
right){return left + BigInt(right);}
BigInt operator+(const BigInt& left, long right){return left
+ BigInt(right);}
BigInt operator+(const BigInt& left, short right){return
left + BigInt(right);}
BigInt operator+(const BigInt& left, char right){return left
+ BigInt(right);}

//-------------
BigInt operator-(int left, const BigInt& right)  {return
BigInt(left) - right;}
BigInt operator-(std::string left, const BigInt& right)
{return BigInt(left) - right;}
```

```
BigInt operator-(short left, const BigInt& right) {return
BigInt(left) - right;}
BigInt operator-(long left, const BigInt& right) {return
BigInt(left) - right;}
BigInt operator-(char left, const BigInt& right) {return
BigInt(left) - right;}
BigInt operator-(const BigInt& left, int right) {return left
- BigInt(right);}
BigInt operator-(const BigInt& left, std::string
right){return left - BigInt(right);}
BigInt operator-(const BigInt& left, long right){return left
- BigInt(right);}
BigInt operator-(const BigInt& left, short right){return
left - BigInt(right);}
BigInt operator-(const BigInt& left, char right){return left
- BigInt(right);}

//*****************
BigInt operator*(int left, const BigInt& right)  {return
BigInt(left) * right;}
BigInt operator*(std::string left, const BigInt& right)
{return BigInt(left) * right;}
BigInt operator*(short left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(long left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(char left, const BigInt& right) {return
BigInt(left) * right;}
BigInt operator*(const BigInt& left, int right) {return left
* BigInt(right);}
BigInt operator*(const BigInt& left, std::string
right){return left * BigInt(right);}
BigInt operator*(const BigInt& left, long right){return left
* BigInt(right);}
BigInt operator*(const BigInt& left, short right){return
left * BigInt(right);}
BigInt operator*(const BigInt& left, char right){return left
* BigInt(right);}

//to other types
BigInt::operator std::string() const {
    return (sign ? "" : "-") + value;
}
BigInt::operator int() const {
```

```
    return (sign ? 1 : -1) * std::atoi(value.c_str());
}
BigInt::operator short() const {
    return (sign ? 1 : -1) * std::atoi(value.c_str());
}
BigInt::operator char() const {
    return (sign ? 1 : -1) * std::atoi(value.c_str());
}
BigInt::operator long() const {
    return (sign ? 1 : -1) * std::atol(value.c_str());
}
```

Приложение В (class.h)

```
#ifndef CLASS_H
#define CLASS_H

#include <string>
#include <iostream>
#include <algorithm>

class BigInt{
    private:
        std::string value;
        bool sign;

    public:
    //constructors
        BigInt();
        ~BigInt();
        BigInt(bool sign, std::string str);
        BigInt(std::string& str);
        BigInt(const char* str);
        BigInt(int num);
        BigInt(short num);
        BigInt(long num);
        BigInt(char num);
        BigInt(const BigInt& obj);

    //set and get methods

    const std::string& get_value() const;
    void set_value(const std::string& str);
    const bool get_sign() const;
    void set_sign(const bool s);

    //operators
        friend BigInt operator+(const BigInt& left, const
BigInt& right);
        friend BigInt operator-(const BigInt& left, const
BigInt& right);
        friend BigInt operator*(const BigInt& left, const
BigInt& right);
        friend std::ostream& operator<<(std::ostream& out,
const BigInt& num);
```

```
        friend std::istream& operator>>(std::istream& in,
BigInt& num);

        BigInt operator=(const BigInt& other);
        BigInt operator=( short num);
        BigInt operator=( char num);
        BigInt operator=( int num);
        BigInt operator=( long num);

        operator std::string() const;
        operator int() const;
        operator short() const;
        operator long() const;
        operator char() const;
};
//++++++++++++++++++++
BigInt operator+(int left, const BigInt& right);
BigInt operator+(std::string left, const BigInt& right);
BigInt operator+(short left, const BigInt& right);
BigInt operator+(long left, const BigInt& right);
BigInt operator+(char left, const BigInt& right);
BigInt operator+(const BigInt& left, int right);
BigInt operator+(const BigInt& left, std::string right);
BigInt operator+(const BigInt& left, long right);
BigInt operator+(const BigInt& left, short right);
BigInt operator+(const BigInt& left, char right);

//---------------------
BigInt operator-(int left, const BigInt& right);
BigInt operator-(std::string left, const BigInt& right);
BigInt operator-(short left, const BigInt& right);
BigInt operator-(long left, const BigInt& right);
BigInt operator-(char left, const BigInt& right);
BigInt operator-(const BigInt& left, int right);
BigInt operator-(const BigInt& left, std::string right);
BigInt operator-(const BigInt& left, long right);
BigInt operator-(const BigInt& left, short right);
BigInt operator-(const BigInt& left, char right);

//************/
BigInt operator*(const BigInt& left, const BigInt& right);

BigInt operator*(const BigInt& left, int right);
BigInt operator*(const BigInt& left, std::string right);
```

```
BigInt operator*(const BigInt& left, short right);
BigInt operator*(const BigInt& left, long right);
BigInt operator*(const BigInt& left, char right);
BigInt operator*(char left, const BigInt& right);
BigInt operator*(long left, const BigInt& right);
BigInt operator*(short left, const BigInt& right);
BigInt operator*(std::string left, const BigInt& right);
BigInt operator*(int left, const BigInt& right);

std::string stringDif(const std::string& big, const
std::string& small);
std::string stringSum(const std::string& adin, const
std::string& dva);
std::string stringMultiply(const std::string& adin, const
std::string& dva);
bool firstBigger(const std::string& adin, const std::string&
dva);

#endif
```