ГОУ ВПО «ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ

Отчет

по производственной практике

Руководитель:

асс. каф. ПИ

Филипишин Д.А.

\_\_\_\_.\_\_\_\_.2021г.

Выполнили:

ст. гр. ПИ-18а

Данильчук К. М.

\_\_\_\_.\_\_\_\_.2021г.

Донецк – 2021

СОДЕРЖАНИЕ

[Введение 3](#_Toc76384784)

[1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ И ПОСТАНОВКА ЗАДАЧИ 4](#_Toc76384785)

[1.1 Постановка задачи 4](#_Toc76384786)

[1.2 Распределение задач для совместной разработки 4](#_Toc76384787)

[1.3 Анализ предметной области 4](#_Toc76384788)

[2 ОБОСНОВАНИЕ ВЫБОРА ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ 6](#_Toc76384789)

[3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОЙ СИСТЕМЫ 7](#_Toc76384790)

[4 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СИСТЕМЫ И ИНТЕРФЕЙСОВ 8](#_Toc76384791)

[Заключение 19](#_Toc76384792)

# Введение

Точный результат арифметических действий не всегда можно получить из-за ограниченного числа разрядов. Также, мы ограничены размером чисел, с которыми можем работать. Если же нам необходимо выполнить арифметические действия над очень большими числами, то в таких случаях мы сами должны позаботиться о представлении чисел в вычислительной машине и о точном выполнении арифметических операций над ними.

«Длинными» называются числа, для представления которых в базовых типах данных не хватает количества двоичных разрядов. «Длинная» арифметика – это реализация арифметических операций над «длинными» числами.

Для большинства программ, предоставляемых процессором, вполне хватает базовых типов. Тем не менее, встречаются задачи, для которых не хватает значений, предусмотренных базовыми типами. Число из 10000 цифр не поместится ни в один регистр памяти. Поэтому компьютерное представление таких чисел и операции над ними приходится реализовывать самостоятельно. Создавая класс, организующий считывание, вывод, да и сами алгоритмы работы с длинными числами, можно добиться как ускорения времени, затрачиваемого над операциями, так и в разы сократить используемую память.

Возникает вопрос в актуальности использования длинной арифметики. Длинные числа повсеместно используются в бухгалтерии для точного подсчета денежных и других средств. Очень активную роль в последнее время большие числа стали играть в криптографии.

# 1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ И ПОСТАНОВКА ЗАДАЧИ

1.1 Постановка задачи

Итоговым продуктом должен бвть класс, который может работать с десятичными числами произвольной длинны. Реализуемые операции:

* сложение;
* вычитание;
* умножение;
* деление;
* остаток от деления;
* префиксный декремент;
* постфиксный декремент;
* префиксный инкремент;
* постфиксный инкремент;
* сравнение (<, <=, >, >=, ==, !=);
* операции ввода/вывода( <<, >>).

Также необходимо полностью протестировать весь функционал класса. Для тестирования можно использовать google тесты.

1.2 Распределение задач для совместной разработки

Разработка класса, реализующего взаимодействие с «длинными» числами. Обеспечение полного функционала, который описан при постановке задачи (п. 1.1).

1.3 Анализ предметной области

Работа с «длинными» числами может активно использоваться в криптографии. Без чисел произвольного размера максимально можно использовать число с 20 знаками, что ограничивает размер ключей. Для улучшения безопасности данных можно использовать «длинные» числа, которые позволяют создавать ключи потенциально неограниченной длины, одноко чем больше длина ключа, тем дольше будет генерация ключа.

# 2 ОБОСНОВАНИЕ ВЫБОРА ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ

Для разработки класса был выбран язык С++. Он обладает множеством преимуществ, которые выгодно выделяют его на фоне остальных языков программирования.

Во-первых, он очень быстрый. Программа, которая грамотно написана на С++ в подавляющем большинстве случаев окажется быстрее программы с тем же функционалом, написанной на другом языке программирования.

Во-вторых, он обладает завидной универсальностью. Поскольку компилятор С++ есть на подавляющем большинстве операционных системы, появляется возможность написать одну программу, которая сможет скомпилироваться на любой платформе.

В-третьих, до сих пор идет активная поддержка и модификация С++. Разрабатывают и выпускают новые стандарты, что позволяет языку не отставать от своих конкурентов, и даже опережать их как в функциональности, так и в удобстве написания кода.

В-четвертых, он до сих пор остается одним из самых востребованных языков программирования, потому что имеет широкие возможности, позволяющие оптимизировать программируемые системы до такого состояния, до которого невозможно добраться, используя более высокоуровневые языки программирования.

Также были использованы google тесты, которые позволяют сравнить результат выполнения функции с ожидаемым результатом. Это является очень полезной функцией, которая позволяет сразу выявлять ошибки и оперативно с ними справляться.

# 3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОЙ СИСТЕМЫ

При проектировании класса было решено представлять числа в виде массива символов типа «char», и отдельно хранить знак числа. Также для более удобного взаимодействия с классом были переопределены операторы, которые выполняют функции, описанные в задании (п. 1.1).

# 4 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СИСТЕМЫ И ИНТЕРФЕЙСОВ

В результате проектирование и разработки класса был реализован следующий функционал:

**Файл BigInteger.h**

#pragma once

#include <vector>

#include <string>

#include <iostream>

class BigInteger {

friend bool operator<(const BigInteger&, const BigInteger&) noexcept;

friend std::istream& operator>>(std::istream&, BigInteger&);

public:

BigInteger();

BigInteger(int);

BigInteger(const BigInteger&) = default;

BigInteger& operator=(const BigInteger&) = default;

BigInteger(BigInteger&&) noexcept;

BigInteger& operator=(BigInteger&&) noexcept;

std::string toString() const;

BigInteger& operator+=(const BigInteger&);

BigInteger& operator-=(BigInteger);

BigInteger& operator\*=(const BigInteger&);

BigInteger& operator/=(const BigInteger&);

BigInteger& operator%=(const BigInteger&);

BigInteger& operator-() noexcept;

BigInteger& operator++();

BigInteger operator++(int);

BigInteger& operator--();

BigInteger operator--(int);

explicit operator bool() const noexcept;

public:

BigInteger(std::vector<char>&&, int8\_t);

std::vector<BigInteger> fill\_integers(const BigInteger&, const BigInteger&); //private member-function for \*=

std::vector<char> numerals;

mutable char sign;

};

bool operator< (const BigInteger&, const BigInteger&) noexcept;

bool operator> (const BigInteger&, const BigInteger&) noexcept;

bool operator<= (const BigInteger&, const BigInteger&) noexcept;

bool operator>= (const BigInteger&, const BigInteger&) noexcept;

bool operator== (const BigInteger&, const BigInteger&) noexcept;

bool operator!= (const BigInteger&, const BigInteger&) noexcept;

BigInteger operator+(const BigInteger&, const BigInteger&);

BigInteger operator-(const BigInteger&, const BigInteger&);

BigInteger operator\*(const BigInteger&, const BigInteger&);

BigInteger operator/(const BigInteger&, const BigInteger&);

BigInteger operator%(const BigInteger&, const BigInteger&);

std::ostream& operator<< (std::ostream&, const BigInteger&);

**Файл BigInteger.cpp**

#include "BigInteger.h"

#include <iostream>

BigInteger::BigInteger(std::vector<char>&& vector, int8\_t sign)

: numerals(std::move(vector))

, sign(sign)

{}

BigInteger::BigInteger()

: sign(0)

, numerals({ 0 })

{}

BigInteger::BigInteger(int number)

: sign(0)

{

if (number == 0) {

numerals.push\_back(0);

return;

} else if (number > 0) {

sign = 1;

} else {

number \*= -1; //for number % 10

sign = -1;

}

while (number != 0) {

numerals.push\_back(number % 10);

number /= 10;

}

}

BigInteger::BigInteger(BigInteger&& value) noexcept {

sign = value.sign;

value.sign = 0;

numerals.resize(1);

numerals.front() = 0;

std::swap(numerals, value.numerals);

}

BigInteger& BigInteger::operator=(BigInteger&& value) noexcept {

sign = value.sign;

value.sign = 0;

numerals.resize(1);

numerals.front() = 0;

std::swap(numerals, value.numerals);

return \*this;

}

std::string BigInteger::toString() const {

std::string result;

if (sign < 0) {

result.reserve(numerals.size() + 1);

result += '-';

} else {

result.reserve(numerals.size());

}

for (auto it = numerals.crbegin(), end = numerals.crend(); it != end; ++it) {

result += std::to\_string(\*it);

}

return result;

}

BigInteger& BigInteger::operator+=(const BigInteger& value) {

if (sign == 0) {

sign = value.sign;

}

if (this == &value) {

size\_t count = 0;

for (char & numeral : numerals) {

numeral \*= 2;

numeral += count;

count = numeral / 10;

numeral %= 10;

}

if (count != 0) {

numerals.push\_back(count);

}

return \*this;

}

if (sign == -1 && value.sign == 1) {

sign = 1;

bool abs\_great = \*this > value;

\*this -= value;

sign = abs\_great ? -1 : 1;

if (numerals == std::vector<char>({ 0 })) {

sign = 0;

}

return \*this;

}

if (sign == 1 && value.sign == -1) {

value.sign = 1;

\*this -= value;

value.sign = -1;

return \*this;

}

if (int difference = value.numerals.size() - numerals.size(); difference > 0) { //12345

numerals.insert(numerals.end(), difference, 0); //+

} //00123

size\_t i = 0;

for (size\_t end = value.numerals.size() - 1; i < end; ++i) {

numerals[i] += value.numerals[i];

if (numerals[i] > 9) {

numerals[i] -= 10;

numerals[i + 1] += 1;

}

}

numerals[i] += value.numerals.back();

for (long long end = static\_cast<long long>(numerals.size()) - 1; static\_cast<long long>(i) < end; ++i) {

if (numerals[i] > 9) {

numerals[i] -= 10;

numerals[i + 1] += 1;

}

}

if (numerals[i] > 9) {

numerals[i] -= 10;

numerals.push\_back(1);

}

return \*this;

}

BigInteger& BigInteger::operator-=(BigInteger value) {

if (sign == 0) {

sign = -value.sign;

}

if (sign \* value.sign == -1) {

sign = value.sign;

\*this += value;

sign = -value.sign;

return \*this;

}

int this\_sign = sign;

sign = value.sign = 1;

if (\*this < value) {

std::swap(\*this, value);

this\_sign \*= -1;

}

for (size\_t i = 0; i < value.numerals.size(); ++i) {

if (numerals[i] < value.numerals[i]) {

size\_t j = i + 1;

while (numerals[j] == 0) {

numerals[j] = 9;

++j;

}

--numerals[j];

if (numerals.back() == 0) {

numerals.pop\_back();

}

numerals[i] += 10;

}

numerals[i] -= value.numerals[i];

}

sign = this\_sign;

for (size\_t i = numerals.size() - 1; i > 0; --i) {

if (numerals[i] != 0) {

break;

}

numerals.pop\_back();

}

if (numerals == std::vector<char>{ 0 }) {

sign = 0;

}

return \*this;

}

bool operator<(const BigInteger& x, const BigInteger& y) noexcept {

if (x.sign < y.sign) {

return true;

} else if (x.sign > y.sign) {

return false;

}

if (x.numerals.size() < y.numerals.size()) {

return y.sign > 0 ? true : false;

} else if (x.numerals.size() > y.numerals.size()) {

return x.sign < 0 ? true : false;

}

for (long int i = x.numerals.size() - 1; i > -1; --i) {

if (x.numerals[i] < y.numerals[i]) {

return y.sign > 0 ? true : false;

} else if (x.numerals[i] > y.numerals[i]) {

return x.sign < 0 ? true : false;

}

}

return false; // if x equals y

}

bool operator>(const BigInteger& x, const BigInteger& y) noexcept {

return y < x;

}

bool operator<= (const BigInteger& x, const BigInteger& y) noexcept {

return !(y < x);

}

bool operator>= (const BigInteger& x, const BigInteger& y) noexcept {

return !(x < y);

}

bool operator==(const BigInteger& x, const BigInteger& y) noexcept {

return !(x > y || x < y);

}

bool operator!=(const BigInteger& x, const BigInteger& y) noexcept {

return !(x == y);

}

BigInteger operator+(const BigInteger& x, const BigInteger& y) {

BigInteger result = x;

result += y;

return result;

}

BigInteger operator-(const BigInteger& x, const BigInteger& y) {

BigInteger result = x;

result -= y;

return result;

}

std::ostream& operator<<(std::ostream& out, const BigInteger& value) {

return out << value.toString();

}

BigInteger::operator bool() const noexcept {

return sign;

}

BigInteger& BigInteger::operator\*=(const BigInteger& value) {

if (sign == 0) {

return \*this;

} else if (value.sign == 0) {

numerals = { 0 };

sign = 0;

return \*this;

}

std::vector<BigInteger> integers;

auto make\_integers = [this, &integers] (const BigInteger& x, const BigInteger& y) {

integers = (numerals.size() > y.numerals.size())

? fill\_integers(\*this, y)

: fill\_integers(y, \*this);

};

size\_t count\_zero = 0;

for (; numerals[count\_zero] == 0; ++count\_zero);

numerals.erase(numerals.begin(), numerals.begin() + count\_zero);

if (value.numerals.front() == 0) {

BigInteger y = value;

{

size\_t i = 0;

for (;y.numerals[i] == 0; ++i);

y.numerals.erase(y.numerals.begin(), y.numerals.begin() + i);

count\_zero += i;

}

make\_integers(\*this, y);

} else {

make\_integers(\*this, value);

}

BigInteger summ = 0;

for (const BigInteger& integer : integers) {

summ += integer;

}

if (count\_zero == 0) {

numerals = std::move(summ.numerals);

} else {

std::vector<char> result\_integer(count\_zero + summ.numerals.size(), 0);

for (size\_t i = 0; i < summ.numerals.size(); ++i) {

result\_integer[i + count\_zero] = summ.numerals[i];

}

numerals = std::move(result\_integer);

}

sign \*= value.sign;

return \*this;

}

std::vector<BigInteger> BigInteger::fill\_integers(const BigInteger& x, const BigInteger& y) {

std::vector<BigInteger> integers(y.numerals.size());

std::vector<char> tmp\_numerals;

char balance = 0;

for (size\_t i = 0; i < y.numerals.size(); ++i) {

if (y.numerals[i] == 0) {

continue;

}

balance = 0;

tmp\_numerals.assign(x.numerals.size() + i, 0);

for (size\_t j = 0; j < x.numerals.size(); ++j) {

tmp\_numerals[j + i] = y.numerals[i] \* x.numerals[j] + balance;

balance = tmp\_numerals[j + i] / 10;

tmp\_numerals[j + i] %= 10;

}

if (balance != 0) {

tmp\_numerals.push\_back(balance);

}

integers.emplace\_back(BigInteger(std::move(tmp\_numerals), 1));

}

return integers;

}

BigInteger operator\*(const BigInteger& x, const BigInteger& y) {

BigInteger result = x;

result \*= y;

return result;

}

BigInteger& BigInteger::operator/=(const BigInteger& value) {

if (value.sign == 0) {

throw std::runtime\_error("division by zero");

}

char old\_value\_sign = value.sign;

{

char old\_this\_sign = sign;

value.sign = sign = 1;

if (\*this < value) {

value.sign = old\_value\_sign;

\*this = 0;

return \*this;

}

sign = old\_this\_sign;

}

std::vector<char> answer(numerals.size() - value.numerals.size() + 1, 0);

BigInteger tmp = value;

for (size\_t i = 0; i < value.numerals.size(); ++i) {

tmp.numerals[i] = numerals[i + numerals.size() - value.numerals.size()];

}

tmp.sign = (tmp.numerals.empty()) ? 0 : 1;

for (size\_t i = 0; i < numerals.size() - value.numerals.size() + 1; ++i) {

while (tmp >= value) {

tmp -= value;

++answer[answer.size() - 1 - i];

}

if (char next\_numerals = numerals[numerals.size() - value.numerals.size() - 1 - i]; tmp == 0) {

tmp.numerals.front() = next\_numerals;

tmp.sign = (next\_numerals == 0) ? 0 : 1;

} else {

tmp.numerals.insert(tmp.numerals.begin(), next\_numerals);

}

}

numerals = std::move(answer);

if (numerals.back() == 0) {

numerals.pop\_back();

}

sign \*= old\_value\_sign;

value.sign = old\_value\_sign;

return \*this;

}

BigInteger operator/(const BigInteger& x, const BigInteger& y) {

BigInteger result = x;

result /= y;

return result;

}

BigInteger& BigInteger::operator%=(const BigInteger& value) {

if (value.sign == 0) {

throw std::runtime\_error("division on module by zero");

}

char old\_this\_sign = sign;

char old\_value\_sign = value.sign;

value.sign = sign = 1;

\*this -= value \* ((\*this)/value);

sign = (sign == 0) ? 0 : old\_this\_sign;

value.sign = old\_value\_sign;

return \*this;

}

BigInteger operator%(const BigInteger& x, const BigInteger& y) {

BigInteger result = x;

result %= y;

return result;

}

BigInteger& BigInteger::operator-() noexcept {

sign \*= -1;

return \*this;

}

BigInteger& BigInteger::operator++() {

return \*this += 1;

}

BigInteger BigInteger::operator++(int) {

BigInteger result = \*this;

++(\*this);

return result;

}

BigInteger& BigInteger::operator--() {

return \*this -= 1;

}

BigInteger BigInteger::operator--(int) {

BigInteger result = \*this;

--(\*this);

return result;

}

std::istream& operator>>(std::istream& in, BigInteger& value) {

std::string string\_view\_value;

char tmp = '\0';

tmp = in.get();

switch (tmp) {

case '-': value.sign = -1; break;

case '+': value.sign = 1; break;

default:

if (tmp > '9' || tmp < '0') {

in.putback(tmp);

in.setstate(std::ios\_base::failbit);

return in;

} else {

string\_view\_value += tmp;

value.sign = 1;

}

}

tmp = in.get();

if (tmp == '\n') {

if (string\_view\_value.empty()) {

in.putback(value.sign == 1 ? '+' : '-');

in.putback(tmp);

in.setstate(std::ios\_base::failbit);

return in;

} else {

goto end;

}

}

do {

if ('0' <= tmp && tmp <= '9') {

string\_view\_value += tmp;

} else {

in.putback(tmp);

break;

}

} while ((tmp = in.get()) != '\n');

end:

if (string\_view\_value.front() == '0') {

string\_view\_value.erase(string\_view\_value.find\_first\_of('0'),

string\_view\_value.find\_first\_not\_of('0'));

}

if (string\_view\_value.empty()) {

value = 0;

return in;

}

value.numerals.resize(string\_view\_value.size(), 0);

{

std::vector<char>::reverse\_iterator rbegin = value.numerals.rbegin();

for (char number : string\_view\_value) {

\*(rbegin++) = number - 48;

}

}

return in;

}

# Заключение

Результат работы – класс, который позволяет работать с числами произволдьной длины реализованный при помощи языка программирования C++ и протестированная с помощью google тестов.

К преимуществам класса относятся: быстрая работа, удобство при использовании класса за счет перегрузки базовых операторов, оттестированный функционал, что обеспечивает правильность работы всех реализованных функций.

При дальнейшей работе над классом можно реализовать более сложные математические операции, а также добавить возможность работы нецелыми числами.