**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 1**

Тема: Простые классы на языке С++

Студент: Ватулин В. М.

Преподаватель: Чернышев Л. Н.

Дата:

Оценка:

Москва, 2020

1. **Постановка задачи(вариант 6)**

*Создать класс BitString* для работы с 96-битовыми строками. Битовая строка должна быть представлена двумя полями: старшая часть unsigned long long, младшая часть unsigned int. Должны быть реализованы все традиционные операции для работы с битами: and, or, xor, not. Реализовать сдвиг влево shiftLeft и сдвиг вправо shiftRight на заданное количество битов. Реализовать операцию вычисления количества единичных битов, операции сравнения по количеству единичных битов. Реализовать операцию проверки включения.

1. **Репозиторий github**

<https://github.com/eri412/oop_exercise_01>

1. **Описание программы**

Был реализован класс BitString, содержащий 3 приватных атрибута: \_head, \_tail, \_set\_bits. У класса нет методов для прямого изменения приватных полей, поля изменяются только при присваивании объекту другого объекта, так что можно сказать, что объекты класса иммутабельны. Поле \_set\_bits можно получить с помощью функции-геттера.

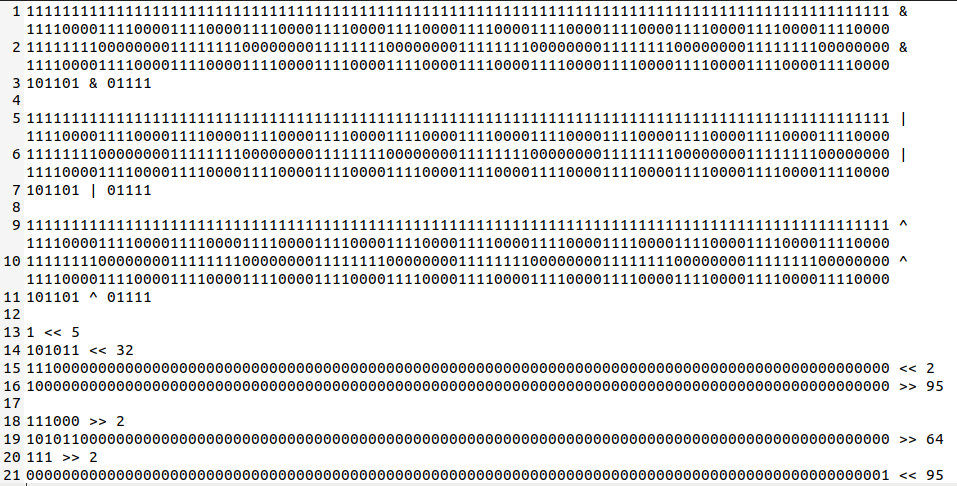
Класс имеет 2 конструктора: один в качестве аргументов принимает два числа, которые присваиваются \_head и \_tail; другой конструктор принимает в качестве аргумента строку и парсит ее.

Операции and(&), or(|), xor(^), not(~), shiftLeft(<<), shiftRight(>>) реализованы через перегрузку соответствующих операторов. Количество единичных битов вычисляется при создании объекта и его можно получить функцией count\_set\_bits(). Операции сравнения по количеству единичных бит реализованы через перегрузку соответствующих операторов сравнения(<, ==, > и т.д.). Функция includes() проверяет включение одной строки в другую. Также присутствует функция to\_string() для представления объекта класса в виде строки.

Программа принимает ввод пока не достигнет EOF.  
Для бинарных операций ввод имеет вид <operand1> <operator> <operand2>  
Для унарных операций ввод имеет вид <operand1> <operator>.  
Пример ввода можно найти в тестовых данных.

1. **Тестовые данные**

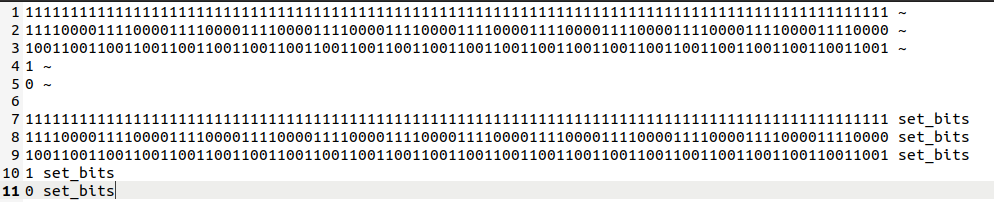
**test\_01.txt:**

****

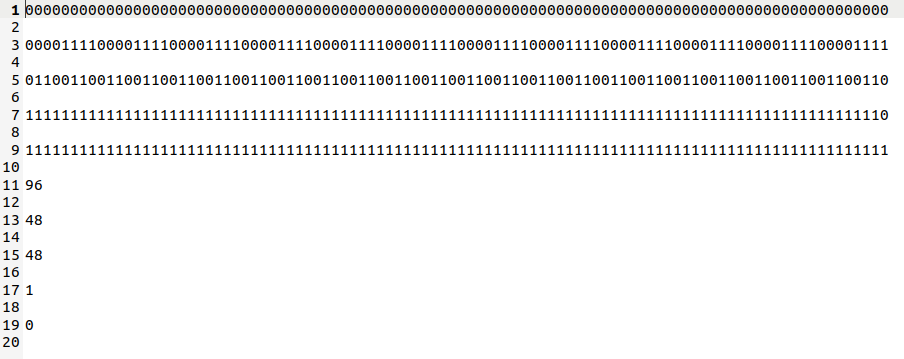
**Результат работы программы:**

****

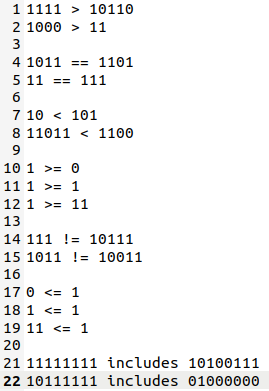
**test\_02.test :**

****

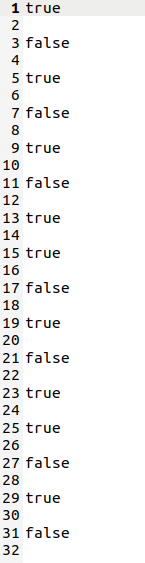
**Результат работы программы**

****

**test\_03.test:**

****

**Результат работы программы**



1. **Листинг программы**

/\*\*

\* Ватулин М8О-206Б-19

\* Создать класс BitString для работы с 96-битовыми строками. Битовая строка должна быть представлена двумя полями:

\* старшая часть unsigned long long, младшая часть unsigned int. Должны быть реализованы все традиционные операции

\* для работы с битами: and, or, xor, not. Реализовать сдвиг влево shiftLeft и сдвиг вправо shiftRight на заданное

\* количество битов. Реализовать операцию вычисления количества единичных битов, операции сравнения по количеству

\* единичных битов. Реализовать операцию проверки включения.

\*/

/\*\*

\* Usage: <operand1> <operator> <operand2>

\* -- if operator is unary, operand2 is ignored

\* -- EOF to end the input

\* -- any symbols other that '1' in operands are considered to be '0'

\* -- one test file can contain any number of tests

\* -- tests can be delimited by blank line

\*/

#include <bitset>

#include <iostream>

#include <string>

#include <limits>

#include <ios>

/\*\*

\* class that stores 96-bit string

\*/

class BitString {

private:

unsigned long long \_head; // most significant

unsigned int \_tail; // least significant

unsigned int \_set\_bits;

unsigned int \_count\_set\_bits(unsigned long long number) const {

unsigned int count = 0;

while (number != 0) {

count += number & 1;

number >>= 1;

}

return count;

}

public:

static const unsigned int bits\_in\_head = sizeof(unsigned long long) \* 8;

static const unsigned int bits\_in\_tail = sizeof(unsigned int) \* 8;

// constructor that takes two numbers as arguments and assigns them 'as is'

BitString(unsigned long long head = 0, unsigned int tail = 0){

\_head = head;

\_tail = tail;

\_set\_bits = \_count\_set\_bits(\_head) + \_count\_set\_bits(\_tail);

}

// constructor that parses string, at that anything except '1' will be considered 0

BitString(const std::string& sample) {

\_head = 0;

\_tail = 0;

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

if (sample[i] == '1') {

if (sample.size() - i - 1 < bits\_in\_tail) {

\_tail |= 1u << (sample.size() - i - 1);

}

else {

\_head |= 1ull << (sample.size() - bits\_in\_tail - i - 1);

}

}

}

\_set\_bits = \_count\_set\_bits(\_head) + \_count\_set\_bits(\_tail);

}

BitString operator&(const BitString& other) const {

BitString temp(\_head & other.\_head, \_tail & other.\_tail);

return temp;

}

BitString operator&(const unsigned int other) const {

return \_tail & other;

}

BitString operator|(const BitString& other) const {

BitString temp(\_head | other.\_head, \_tail | other.\_tail);

return temp;

}

BitString operator|(const unsigned int other) const {

return \_tail | other;

}

BitString operator^(const BitString& other) const {

BitString temp(\_head ^ other.\_head, \_tail ^ other.\_tail);

return temp;

}

BitString operator^(const unsigned int other) const {

return \_tail ^ other;

}

BitString operator~() const {

BitString temp(~\_head, ~\_tail);

return temp;

}

// next two operators may seem as an untidy code, but actually I did all this checks

// to avoid undefined behaviour when bitshifting

BitString operator<<(const unsigned int bits) const {

unsigned long long temp\_head;

if (bits >= bits\_in\_head) {

temp\_head = 0;

}

else {

temp\_head = \_head << bits;

}

unsigned int temp\_tail;

if (bits >= bits\_in\_tail) {

temp\_head = (temp\_head | \_tail) << (bits - bits\_in\_tail);

temp\_tail = 0;

}

else {

temp\_head |= \_tail >> (bits\_in\_tail - bits);

temp\_tail = \_tail << bits;

}

BitString temp(temp\_head, temp\_tail);

return temp;

}

BitString operator>>(const unsigned int bits) const {

unsigned int temp\_tail;

if (bits >= bits\_in\_tail) {

temp\_tail = 0;

}

else {

temp\_tail = \_tail >> bits;

}

unsigned long long temp\_head;

if (bits >= bits\_in\_head) {

temp\_tail = \_head >> (bits\_in\_head/2) >> (bits - bits\_in\_head);

temp\_head = 0;

}

else {

temp\_tail |= \_head << (bits\_in\_head - bits) >> bits\_in\_tail;

temp\_head = \_head >> bits;

}

BitString temp(temp\_head, temp\_tail);

return temp;

}

BitString& operator|=(const BitString &other) {

return (\*this = \*this | other);

}

BitString& operator&=(const BitString &other) {

return (\*this = \*this & other);

}

BitString& operator^=(const BitString &other) {

return (\*this = \*this ^ other);

}

BitString& operator<<=(const unsigned int bits) {

return (\*this = \*this << bits);

}

BitString& operator>>=(const unsigned int bits) {

return (\*this = \*this >> bits);

}

// set bits getter

unsigned int count\_set\_bits() const {

return \_set\_bits;

}

bool operator>(const BitString& other) const {

return this->count\_set\_bits() > other.count\_set\_bits();

}

bool operator<(const BitString& other) const {

return this->count\_set\_bits() < other.count\_set\_bits();

}

// this operator DOESN'T mean bitstrings are the same, it means bitstrings have the same number of set bits

bool operator==(const BitString& other) const {

return this->count\_set\_bits() == other.count\_set\_bits();

}

bool operator<=(const BitString& other) const {

return !(\*this > other);

}

bool operator>=(const BitString& other) const {

return !(\*this < other);

}

bool operator!=(const BitString& other) const {

return !(\*this == other);

}

bool includes(const BitString& other) const {

return ((this->\_head & other.\_head) == other.\_head

&& (this->\_tail & other.\_tail) == other.\_tail);

}

std::string to\_string() const {

return std::bitset<bits\_in\_head>(\_head).to\_string() + std::bitset<bits\_in\_tail>(\_tail).to\_string();

}

};

/\*\*

\* execute functions take operands and operator and print the result of the operation

\*/

void execute(const BitString& lhs, const std::string& op, const BitString& rhs) {

if (op == "|") {

std::cout << (lhs | rhs).to\_string() << "\n" << std::endl;

}

else if (op == "&") {

std::cout << (lhs & rhs).to\_string() << "\n" << std::endl;

}

else if (op == "^") {

std::cout << (lhs ^ rhs).to\_string() << "\n" << std::endl;

}

else if (op == "<") {

std::cout << (lhs < rhs) << "\n" << std::endl;

}

else if (op == "==") {

std::cout << (lhs == rhs) << "\n" << std::endl;

}

else if (op == ">") {

std::cout << (lhs > rhs) << "\n" << std::endl;

}

else if (op == "<=") {

std::cout << (lhs <= rhs) << "\n" << std::endl;

}

else if (op == "!=") {

std::cout << (lhs != rhs) << "\n" << std::endl;

}

else if (op == ">=") {

std::cout << (lhs >= rhs) << "\n" << std::endl;

}

else if (op == "includes") {

std::cout << lhs.includes(rhs) << "\n" << std::endl;

}

else if (op == "~") {

std::cout << (~lhs).to\_string() << "\n" << std::endl;

}

else if (op == "set\_bits") {

std::cout << lhs.count\_set\_bits() << "\n" << std::endl;

}

else {

std::cout << "Bad input!" << "\n" << std::endl;

}

}

void execute(const BitString& lhs, const std::string& op, const unsigned int rhs) {

if (op == "<<") {

std::cout << (lhs << rhs).to\_string() << "\n" << std::endl;

}

else if (op == ">>") {

std::cout << (lhs >> rhs).to\_string() << "\n" << std::endl;

}

else {

std::cout << "Bad input!" << "\n" << std::endl;

}

}

/\*\*

\* parse function takes input from stdin and calls execute function with corresponding arguments

\*/

void parse() {

std::cout << std::boolalpha;

std::string lhs;

std::string op;

std::string rhs;

while (true) {

std::cin >> lhs >> op;

// break out of loop in case input has ended

if (std::cin.eof()) {

break;

}

// if op is unary operation

if (op == "~" || op == "set\_bits") {

execute(BitString(lhs), op, BitString(lhs));

}

// if i need to call overloaded function

else if (op == ">>" || op == "<<") {

std::cin >> rhs;

execute(BitString(lhs), op, std::stoul(rhs));

}

// binary operation

else {

std::cin >> rhs;

execute(BitString(lhs), op, BitString(rhs));

}

// clearing the buffer

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

}

int main() {

parse();

return 0;

}

1. **Вывод**

Реализована программа, включающая в себя простой класс с методами и атрибутами. Получены знания о перегрузках операторов в C++, о конструкторах, о модификаторах доступа. Получены базовые умения работы с cmake.

**Список литературы**

1. Синтаксис основных конструкций — https://metanit.com/cpp/tutorial/
2. Про перегрузку операторов в C++ — https://habr.com/ru/post/132014/