МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

Высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий, математики и механики**

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

**Структура хранения множеств**

**Выполнил:** студент группы 381806-2

Кох Владислав Альбертович

\_\_\_\_\_\_\_\_\_\_\_ Подпись

**Научный руководитель:**

ассистент каф. МОСТ ИИТММ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лебедев И.Г

Нижний Новгород

2019

**Содержание**

[1.](#_30j0zll) Введение 3

[2.](#_1fob9te) Постановка задачи 4

[3.](#_2et92p0) Руководство пользователя 5

[4.](#_tyjcwt) Руководство программиста 7

[4.1 Описание структуры программы 7](#_3dy6vkm)

[4.2](#_1t3h5sf) Описание структур данных 8

[4.3](#_4d34og8) Описание алгоритмов 11

[5.](#_2s8eyo1) Заключение 13

[6. Приложение](#_17dp8vu) 14

# Введение

Вспомним, что любая программа является неким набором данных и команд, обрабатывающих эти данные. Одна из главных задач программиста состоит в выборе наиболее оптимальной модели хранения данных для конкретной задачи и реализации таких алгоритмов, которые будут наилучшим образом использовать доступные ресурсы. Сделать это помогает знание того, какие бывают структуры данных.

**Структура данных** - систематизированный способ организации данных и доступа к ним

В данной лабораторной мы будем говорить о множествах, построенных на битовых полях. Битовое поле ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) bit field) в [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — некоторое количество [бит](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), расположенных последовательно в [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), значение которых [процессор](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) не способен прочитать из-за особенностей аппаратной реализации. Преимущества именно такой структуры для хранения данных:

1. Ограничено место для хранения информации и можно сохранить несколько логических (истина/ложь) переменных в одном байте;
2. Интерфейсы устройств передают информацию, закодировав биты в один байт;
3. Необходим удобный доступ к отдельным битам в байте;
4. Используются объекты с длинной не кратной байту.

Следовательно, нам важен каждый бит. Для записи в битовое поле необходимы логические операции, а именно: операторы OR(|) AND(&) и NOT(-), определенных для отдельных битов, результат выполнения которых возвращает 1(true) и 0(false):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | A&B | A | B | - A | - B |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Для оптимизации так же можно использовать операции сдвига влево(<<) и вправо (>>). Выражение вида (a<<n) означает умножение a на 2n, а (a>>n) деление а на 2n.

# Постановка задачи

**Цель данной работы** — разработка структуры данных для хранения множеств с использованием битовых полей, а также освоение таких инструментов разработки программного обеспечения, как система контроля версий Git и фреймворк для разработки автоматических тестов Google Test.

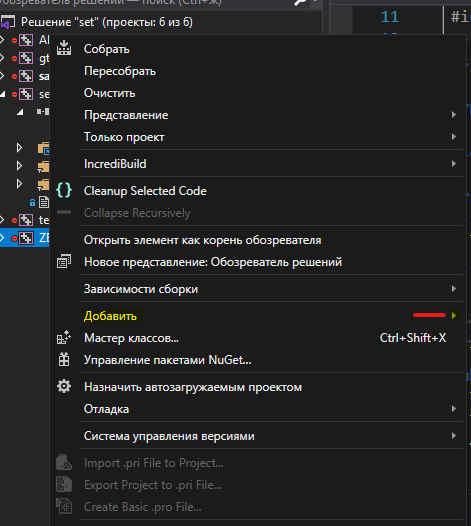
Выполнение работы предполагает решение следующих задач:

1. Реализация класса битового поля TBitField согласно заданному интерфейсу.
2. Реализация класса множества TSet согласно заданному интерфейсу.
3. Обеспечение работоспособности тестов и примера использования.
4. Реализация нескольких простых тестов на базе Google Test.
5. Публикация исходных кодов в личном репозитории на GitHub.

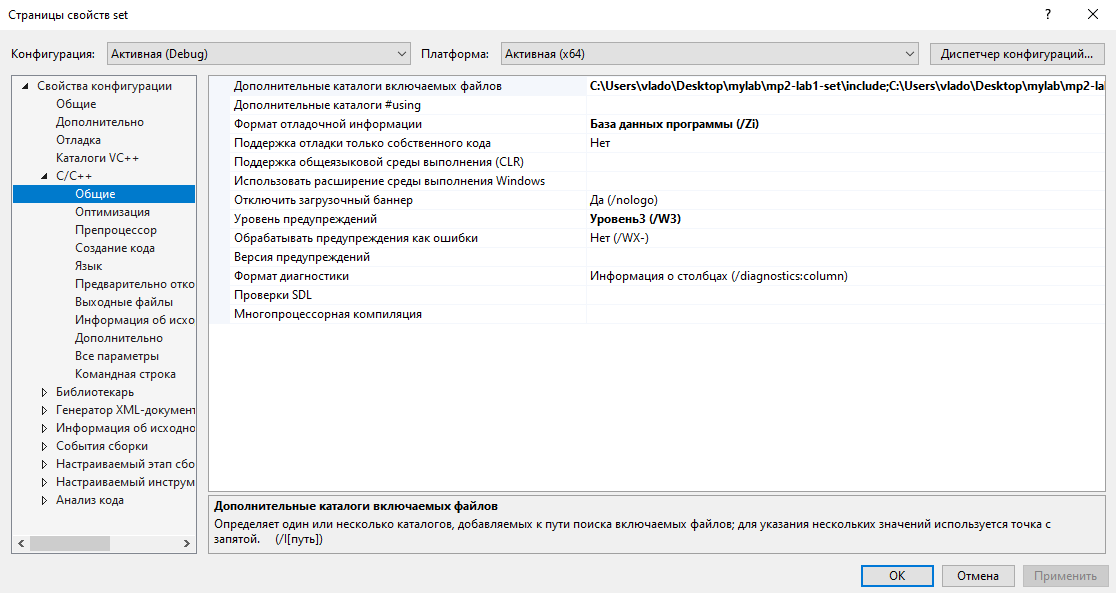
# Руководство пользователя

В ходе лабораторной работы была получена статическая библиотека(staticlib.lib), для того чтобы ее использовать необходимо:

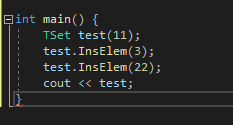
1. Добавить проект статической библиотеки set.vcxproj
2. В свойство проекта указать ссылку на статическую библиотеку



1. Сделать включение заголовочного файла TSet.h, указав в свойствах своего проекта где необходимо искать заголовочный файл



1. В результате можно пользоваться классом Tset и его методами. Например, результат выполнения следующего кода:



Результат:

3, 22

# Руководство программиста

## 4.1 Описание структуры программы

Программа состоит из следующих модулей:

Модуль *set* – статическая библиотека. Содержит заголовочные файлы *tbitfield.h* и *tset.h*, в которых определены интерфейсы классов битового поля *TBitField* и множеств *TSet*. Файлы, содержащие их реализацию – *tbitfield.cpp* и *tset.cpp*.

* Модуль ***test\_set***. Содержит для каждого класса (*TBitField* и *TSet*) набор тестов, реализованный в файлах ***test\_tbitfield.cpp*** и ***test\_tset.cpp*** с помощью использования фреймворка Google Test.

Модуль ***sample\_prime\_numbers***. Содержит файл *sample\_prime\_numbers.cpp*, в котором создан пример использования класса битового поля и множества для поиска простых чисел с использованием алгоритма, называемого "Решетом Эратосфена".

(<https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%88%D0%B5%D1%82%D0%BE_%D0%AD%D1%80%D0%B0%D1%82%D0%BE%D1%81%D1%84%D0%B5%D0%BD%D0%B0>)

## Описание структур данных

Класс *Tset* реализован путем наследования класса *TBitField*

Поля класса:

* *Int MaxPower* – максимально возможный элемент;
* *TBitField BitField* битовое поле, для хранения данных;

Класс битовое поле представлено массивом из unsigned char, каждый элемент битового поля соответствует биту одного из элементов массива.

#### Класс TBitField – битовое поле

Битовое поле - это набор битов с номерами принадлежащим полуинтервалу [0,BitLen). Массив pMem рассматривается как последовательность MemLen элементов. Биты в элементах pMem номеруются справа налево(от младших к старшим).

**Элементы, объявленные со спецификатором private:**

***int bitLen*** – длина битового поля;

***int memLen*** – количество битов для представления битового поля;

***int \*pMem*** – память для представления битового поля;

***int GetMemIndex(const int n) const*** – метод определяющий индекс бита n в массиве *pMem*;

***int GetMemMask(const int n) const*** – метод возвращающий маску для бита n.

**Доступ public:**

***TBitField(int len)*** – конструктор инициализации. Принимает длину битового поля.

***TBitField(const TBitField &bf)*** – конструктор копирования.

***~TBitField()*** – деструктор.

**Методы доступа к битам:**

*int GetLength() const* – возвращает длину битового.

*void SetBit(const int n)* – устанавливает n-й бит битового поля.

*void ClrBit(const int n)* – очищает n-й бит битового поля.

*int GetBit(const int n) const* – возвращает значение бита n.

**Битовые операторы:**

*int operator==(const TBitField &bf) const* – принимает ссылку на объект класса TBitField, проверяет на равенство два битовых поля. Если они идентичны, то возвращает 1, иначе 0.

*int operator!=(const TBitField &bf) const* – принимает ссылку на объект класса *TBitField*, проверяет на неравенство два битовых поля. Если они различны, то возвращает 1, иначе 0.

*TBitField& operator=(const TBitField &bf)* – принимает ссылку на объект класса *TBitField*, присваивает полученное битовое поле к тому, которым был вызван оператор.

*TBitField operator|(const TBitField &bf)* – принимает ссылку на объект класса *TBitField*, возвращает новый объект класса *TBitField* полученный путем использования операции побитового «ИЛИ» между исходным и пришедшим битовыми полями.

*TBitField operator&(const TBitField &bf) –* принимает ссылку на объект класса *TBitField*, возвращает новый объект класса *TBitField* полученный путем использования операции побитового «И» между исходным и пришедшим битовыми полями.

*TBitField operator~()* – возвращает объект битового поля, полученный путем применения операции побитового отрицания для исходного битового поля.

Дружественные функции:

*friend istream &operator>>(istream &istr, TBitField &bf)* – ввод битового поля с консоли.

*friend ostream &operator<<(ostream &ostr, const TBitField &bf)* – вывод битового поля на консоль.

#### Класс TSet – множество

**Элементы, объявленные со спецификатором private:**

*int maxPower* – максимальная мощность множества.

*TBitField bitField* – битовое поле для хранения характеристического вектора.

**Со спецификатором доступа public:**

*TSet(int mp)* – конструктор-инициализатор. Принимает максимальную мощность множества.

*TSet(const TSet &s)* – конструктор копирования.

*TSet(const TBitField &bf)* – конструктор преобразования типа. Принимает ссылку на объект с типом TBitField.

**Метод преобразования типа битовому полю:**

***operator TBitField()*** – возвращает битовое поле характеристического вектора.

**Методы доступа к битам:**

***int GetMaxPower(void) const*** – возвращает максимальную мощность множества.

***void InsElem(const int Elem)*** – добавляет элемент *Elem* в множество.

***void DelElem(const int Elem)*** – удаляет элемент *Elem* из множества.

***int IsMember(const int Elem) const*** – проверяет наличие элемента Elem в множестве. Если элемент есть, то возвращает 1, иначе 0.

## Описание алгоритмов

Рассмотрим некоторые алгоритмы, работа которых не очевидна на первый взгляд.

**Создание битового поля.**

Необходимо представить в памяти множество натуральных чисел. Для этого будем использовать битовую строку, в которой каждый бит будет принимать либо значение «1», если элемент присутствует во входном множестве, либо «0» в противном случае.

Битовую строку будем хранить в виде массива. Заметим, что в битовой строке индексация бит происходит слева направо, в массиве аналогично, однако биты элемента массива нумеруются справа налево.

Для удобства работы с методами получения, очищения, добавления бита и подобным в классе *TBitField* используются вспомогательные методы:

1. *GetMemIndex* – для n-го бита определяется индекс элемента m в массиве pMem, в котором хранится битовое поле. Заметим, что m является целой частью от деления номера бита n на количество битов выделяемых на один элемент массива.
2. *GetMemMask* – определяет битовую маску для n-го бита. Ставим значение 1 в позицию k, определяемую по формуле 2. Битовую маску рассматриваем как двоичную запись числа.

Рассмотрим непосредственно методы работы с битами.

Чтобы установить бит на позицию можем использовать побитовое «ИЛИ» между битами элемента массива *pMem*, в котором находится бит n, и между его битовой маской:

|  |
| --- |
|  |

Чтобы очистить бит n используем побитовое «И» между битами элемента массива *pMem*, в котором находится бит n, и между дополнением его битовой маски:

|  |
| --- |
|  |

Чтобы получить значение бита n используем побитовое «И» между битами элемента массива *pMem*, в котором находится бит n, и между его битовой маской:

|  |
| --- |
|  |

# Заключение

Реализован класс битового поля TBitField согласно заданному интерфейсу.

Реализован классмножества TSet согласно заданному интерфейсу.

Обеспечена работоспособность тестов.

Реализовано несколько простых тестов на базе Google Test.

Код опубликован в личном репозитории на GitHub.

# Приложение: Фрагменты исходного кода

## TSet.cpp

#include "tset.h"

TSet::TSet(int mp) : BitField(mp),MaxPower(mp)

{

}

// конструктор копирования

TSet::TSet(const TSet &s) : BitField(s.BitField),MaxPower(s.MaxPower)

{

}

// конструктор преобразования типа

TSet::TSet(const TBitField &bf) : BitField(bf),MaxPower(bf.GetLength())

{

}

TSet::operator TBitField()

{

TBitField temp(this->BitField);

return temp;

}

int TSet::GetMaxPower(void) const // получить макс. к-во эл-тов

{

return MaxPower;

}

int TSet::IsMember(const int Elem) const // элемент множества?

{

return BitField.GetBit(Elem);

}

void TSet::InsElem(const int Elem) // включение элемента множества

{

if (Elem <= MaxPower)

BitField.SetBit(Elem);

}

void TSet::DelElem(const int Elem) // исключение элемента множества

{

if (Elem <= MaxPower)

BitField.ClrBit(Elem);

}

// теоретико-множественные операции

TSet& TSet::operator=(const TSet &s) // присваивание

{

BitField = s.BitField;

MaxPower = s.GetMaxPower();

return \*this;

}

int TSet::operator==(const TSet &s) const // сравнение

{

return BitField == s.BitField;

}

int TSet::operator!=(const TSet &s) const // сравнение

{

return BitField != s.BitField;

}

TSet TSet::operator+(const TSet &s) // объединение

{

TSet temp\_var(BitField|s.BitField);

return temp\_var;

}

TSet TSet::operator+(const int Elem) // объединение с элементом

{

TSet temp\_var(BitField);

temp\_var.BitField.SetBit(Elem);

return temp\_var;

}

TSet TSet::operator-(const int Elem) // разность с элементом

{

TSet temp\_var(BitField);

temp\_var.BitField.ClrBit(Elem);

return temp\_var;

}

TSet TSet::operator\*(const TSet &s) // пересечение

{

TSet temp\_var(BitField);

temp\_var.BitField = temp\_var.BitField & s.BitField;

return temp\_var;

}

TSet TSet::operator~(void) // дополнение

{

TSet temp\_var(BitField);

temp\_var.BitField = ~BitField;

return temp\_var;

}

// перегрузка ввода/вывода

istream &operator>>(istream &istr, TSet &s) // ввод

{

istr>>s.BitField;

return istr;

}

ostream& operator<<(ostream &ostr, const TSet &s) // вывод

{

ostr << s.BitField << ' ';

return ostr;

}