МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механик**

ЛАБОРАТОРНАЯ РАБОТА

на тему:

**«БИТОВЫЕ ПОЛЯ И МНОЖЕСТВА»**

**Выполнил(а):** студент(ка) группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Суворов Д.И. /

Подпись

**Проверил:** к.т.н., доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д. /

Подпись

Нижний Новгород  
2023

**Содержание**

[Введение 3](#_Toc150323732)

[1 Постановка задачи 4](#_Toc150323733)

[2 Руководство пользователя 5](#_Toc150323734)

[2.1 Приложение для демонстрации работы битовых полей 5](#_Toc150323735)

[2.2 Приложение для демонстрации работы множеств 5](#_Toc150323736)

[2.3 Приложение «решето Эратосфена» 8](#_Toc150323737)

[3 Руководство программиста 9](#_Toc150323738)

[3.1 Алгоритмы 9](#_Toc150323739)

[3.1.1 Битовые поля 9](#_Toc150323740)

[3.1.2 Множества 11](#_Toc150323741)

[3.1.3 Алгоритм «решето Эратосфена» 12](#_Toc150323742)

[3.2 Описание классов 12](#_Toc150323743)

[3.2.1 Класс TBitField 12](#_Toc150323744)

[3.2.2 Класс TSet 16](#_Toc150323745)

[Заключение 19](#_Toc150323746)

[Литературы 20](#_Toc150323747)

[Приложения 21](#_Toc150323748)

[Приложение А. Реализация класса TBitField 21](#_Toc150323749)

[Приложение Б. Реализация класса TSet 23](#_Toc150323750)

[Приложение B. Пример использования битовых полей 24](#_Toc150323751)

[Приложение Г. Пример использования множеств 25](#_Toc150323752)

[Приложение Д. Алгоритм решета Эратосфена 26](#_Toc150323753)

# Введение

Множество – это математическая модель, являющаяся совокупностью объектов, которые называются элементами множества, обладающих определёнными признаками, едиными для всех элементов одного множества. Теория множеств – это математическая дисциплина, которая изучает свойства, отношения и операции над множествами. Область применения теории множеств – огромна. Эта теория используется во многих разделах математики, например, в алгебре, топологии, геометрии, математической логике, теории вероятности и т.д. Она также широко применяется в компьютерных науках, физике и экономике. Так, в компьютерных науках теория множеств используется для описания структур данных, для моделирования и решения различных задач; в экономике - для моделирования социальных и экономических процессов. Таким образом, множество – это универсальный математический инструмент для моделирования, описания, изучения различных процессов, а также для решения множества алгоритмических и математических задач.

Одной из самых эффективных структур данных для хранения множества несомненно является битовое поле. Эта структура данных представляет собой непрерывную последовательность битов, каждый из которых может быть либо 0, либо 1. Так как данные хранятся в битах – битовое поле позволяет очень компактно хранить и быстро производить манипуляцию над данными. Благодаря таким сильным сторонам, битовые поля применяются не только в отношении с множествами, но также применяются во множестве компьютерных областей, которые связаны с обработкой или хранением данных: базы данных, сетевые протоколы, криптография и т.п.

Всё описанное выше доказывает высокую значимость и активную применимость множеств и битовых полей.

# Постановка задачи

Цель – изучить теоретические основы и получить практические навыки применения структуры данных битовое поле.

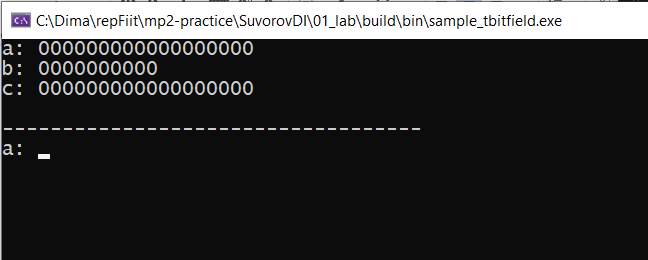
Задачи:

1. Изучить предметную область: узнать, что такое битовое поле.
2. Разработать класс, представляющий битовое поле.
3. Реализовать все необходимые операции с битовыми полями.
4. Разработать класс множеств.
5. Реализовать все необходимые операции с множествами.
6. Протестировать корректность работы разработанного функционала.
7. Применить разработанную программу для решения задачи о простых числах (решето Эратосфена).
8. Сделать вывод о проделанной работе.

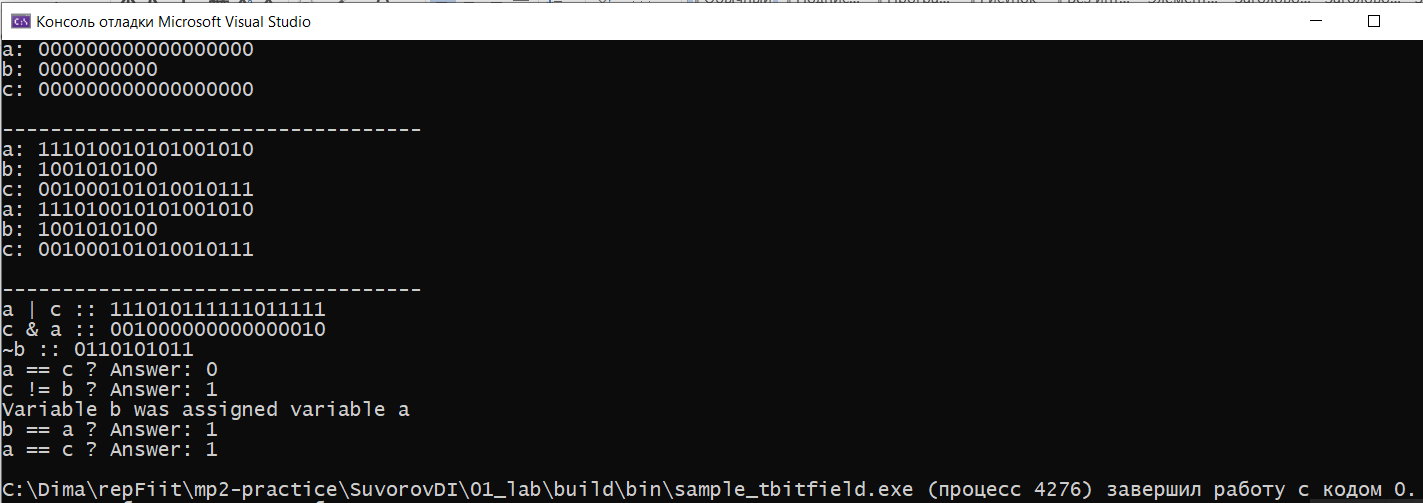
# Руководство пользователя

## Приложение для демонстрации работы битовых полей

1. Запустить sample\_tbitfield.exe. В результате появится следующее окно (рис. 1).



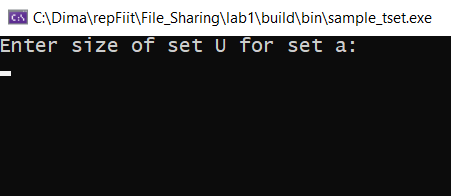
1. Стартовое окно приложения
2. В первых трёх строчках выведены заранее заготовленные пустые битовые поля с длинами 18, 10, 18. После разделительной полосы необходимо ввести значение каждого битового поля.
3. После ввода соответственных битовых полей будут выведены результаты применения основных операций между ними. Каждый вывод сопровождается соответствующими пометками, которые описывают, результат какой операции выведен (рис. 2).



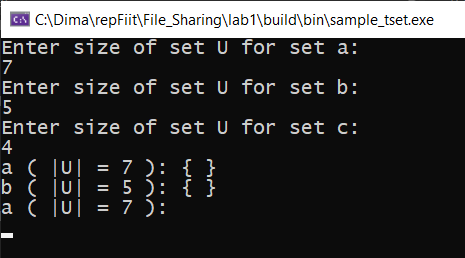
1. Итог работы программы

## Приложение для демонстрации работы множеств

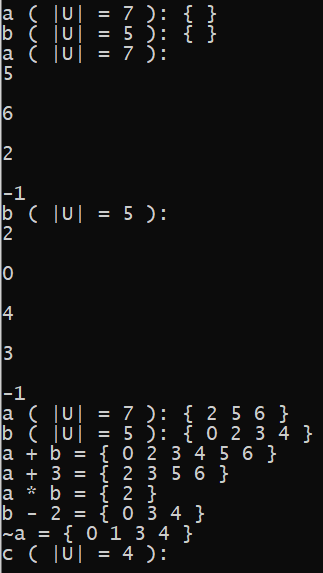
1. Запустить sample\_tset.exe. В результате появится следующее окно (рис. 3).



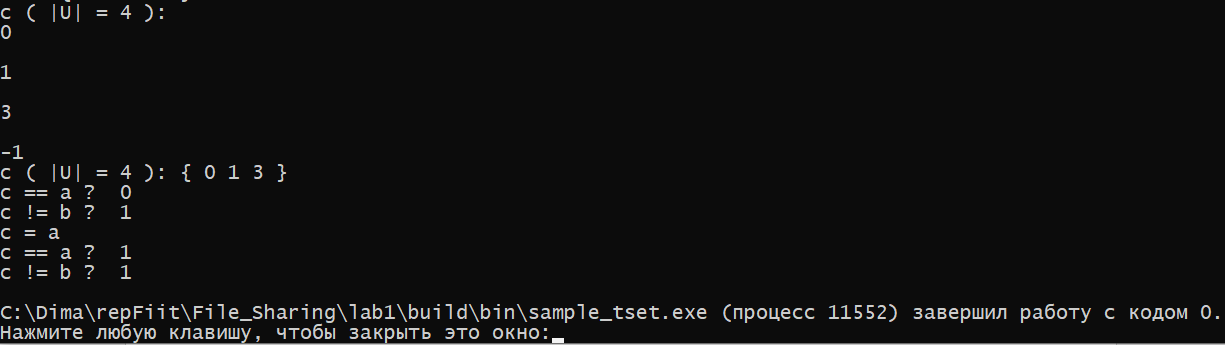
1. Стартовое окно приложения
2. Необходимо последовательно ввести 3 натуральных числа – мощности универсального множества для множества «a», «b», «c» (рис. 4).



1. Ввод мощностей универсальных множеств
2. После ввода на экране выведутся изначально пустые множества «a» и «b». После вывода этих множеств программа ожидает ввода элементов множеств «a» и «b». Элементами могут являться все целые неотрицательные числа меньшие значения мощности универсального множества для конкретного множества. Элементы вводятся последовательно через Enter. По окончании ввода элементов необходимо ввести -1 (значение -1 завершит ввод элементов текущего множества). Таким образом, необходимо ввести элементы двух множеств: «a» и «b». После ввода появятся результаты заранее подготовленных теоретико-множественных операций (рис. 5).



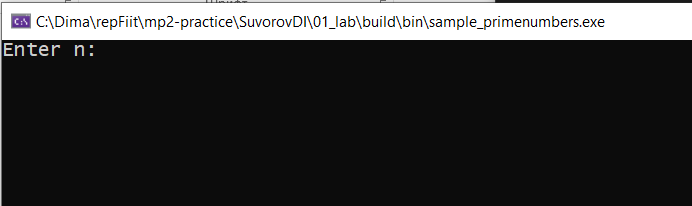
1. Ввод элементов в множества «a» и «b». Вывод результатов выполнения теоретико-множественных операций над ними
2. После вывода результатов выполнения операций программа ожидает ввода элементов множества «c». Правила ввода элементов остаются такими же. После ввода элементов на экран выведутся результаты выполнения операций сравнения (рис. 6).



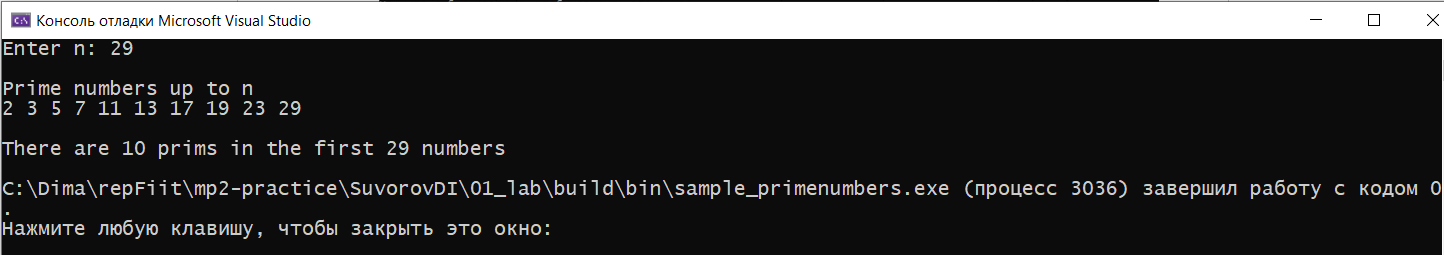
1. Ввод элементов множества «c» и вывод результатов выполнения операций сравнения. Конец программы

## Приложение «решето Эратосфена»

1. Запустить sample\_primenumbers.exe. В результате появится следующее окно   
   (рис. 7).



1. Стартовое окно приложения
2. Необходимо ввести натуральное число. После ввода натурального числа нажать Enter и программа выведет все простые числа от 2 до введённого числа включительно (рис. 8).



1. Итоговое окно приложения

# Руководство программиста

## Алгоритмы

### Битовые поля

Битовые поля представляют собой непрерывное хранение 0 и 1 (битов) в памяти. В языке программирования C++ нет типа данных, который может напрямую хранить значение 0 или 1 в одном бите. Для того чтобы реализовать эту структуру данных мы будем использовать доступный тип данных unsigned int. Как любой беззнаковый целочисленный тип данных он хранится в памяти в виде двоичного представления, то есть из непрерывной последовательности 0 и 1 длинной 32 бита (4 байта). Значит любое битовое поле длины n можно представить как массив из n // 32 + 1 (где // – целочисленное деление) элементов типа unsigned int.

Существуют следующие операции для работы с битовыми полями: доступ к конкретному биту (установить бить в 1, в 0, получить текущее значение бита); побитовые операции (побитовые конъюнкция, дизъюнкция между двумя битовыми полями, и побитовое отрицание битового поля). Также два битовых поля можно сравнить.

Битовые операции встроены в C++, но для базовых встроенных типов данных. Так как битовое поле – это динамический массив из элементов встроенного типа данных unsigned int, то применить битовые операции к битовому полю равнозначно последовательному применению битовых операций к каждому элементы массива.

Установление значения отдельного бита в 1:

Для реализации этой операции нам понадобится битовое поле, в котором все биты равны 0, и только один бит равен 1. Этот бит должен стоять на разряде, который в исходном битовом поле необходимо установить в 1, номер такого разряда будем обозначать i. Такой битовой маски можно добиться сдвигом целочисленной 1 на i бит влево.

Вспомогательные битовые поля, которые необходимы для реализации операций над битовыми полями будем называть битовыми масками. Если к исходному битовому полю и подобной битовой маске применить операцию битового «ИЛИ», то в результате получим битовое поле, в котором бит на i-ом разряде установлен в 1, а все оставшиеся биты равны битам в исходном битовом поле.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Изначальное битовое поле |  | Битовая маска |  | Результат |
| 0 | **|** | 0 | **=** | 0 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| **0** | **1** | **1** |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

Установление значения отдельного бита в 0:

Битовая маска для этой операции имеет значение i-ого разряде 0, а значения всех остальных разрядов равны 1. Такую маску можно получить инвертированием маски, которая использовалась при установлении определенного бита в 1.

При применении к исходному битовому полю и полученной маске операции битового «И» в результате будет получено битовое поле, в котором бит на i-ом разряде равен 0, а все оставшиеся биты равны битам исходного битового поля.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Изначальное битовое поле |  | Битовая маска |  | Результат |
| 0 | **&** | 1 | **=** | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |
| **1** | **0** | **0** |
| 0 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |

Получение текущего значения определённого бита:

Для реализации этой операции нам необходимо применить к исходному полю и битовой маске, которая использовалась для операции установления бита в 1, операцию битового «И». Таким образом, в результате мы получим битовое поле, в котором бит на i-ом разряде сохранил такое же значение, какое было на i-ом разряде в исходном битовом поле, а все остальные биты равны 0. Применим сдвиг вправо на i бит для результата и в результате получим значение бита на i-ом разряде в исходном битовом поле.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Изначальное битовое поле |  | Битовая маска |  | Промежуточный результат |  | Результат |
| 0 | **&** | 0 | **=** | 0 | **>>** | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| **1** | **1** | **1** | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | **1** |

Сравниваются битовые поля поэлементно. Сравнение каждого элемента – это всего лишь сравнение двух элементов массива базового типа.

### Множества

Множество основано на битовом поле. Длина битового поля для множества – это максимальная длина для этого множества, то есть мощность универса. Если бит в i-ом разряде битового поля равен 1, то элемент i из универса входит в текущее множество, если равен 0, то соответствующий элемент не входит во множество.

Для множеств реализованы операции добавления и удаления элемента из множества; операция проверки наличия элемента во множестве; теоретико-множественные операции между множествами (объединение, пересечение множеств); взятие дополнения к множеству; операция сравнения.

Добавление элементов в множество и их удаление из него – это установка необходимых битов в битовом поле, которое представляет множество, в 0 или в 1. Алгоритм установки значений битов в битовых полях были описаны в разделе 3.1.1. Операция проверки наличия элемента в множестве – это операция получения значения необходимого бита в битовом поле, которое представляет данное множество. Эта операция над битовым полем также была описана ранее. Теоретико-множественные операции над множествами – это применение к двум битовым полям, представляющим два данных множества, операций битовых «ИЛИ», «И». Дополнение к множеству – это применение операции битового «НЕ» к битовому полю, которое представляет данное множество. Сравнение двух множеств – это сравнение соответствующих битовых полей этих двух множеств.

Примеры множественных операций:

A = {3, 5, 7, 10, 13} – в универсальном множестве из 14 элементов

Это множество характеризуется битовым полем:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

В этом битовом поле биты под индексами 3, 5, 7, 10 и 13 равны 1, остальные 0.

B = {1, 3, 6, 7, 11, 13} – в универсальном множестве из 14 элементов

Это множество характеризуется битовым полем:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

В этом битовом поле биты под индексами 1, 3, 6, 7, 11 и 13 равны 1, остальные 0.

A | B = {1, 3, 5, 6, 7, 10, 11, 13}

Множество, являющееся результатом объединения двух других, получается в результате применения к битовым представлениям двух исходных множеств операции побитового «ИЛИ». Как работает эта битовая операция было описано выше.

A & B = {3, 7, 13}

Множество, являющееся результатом пересечения двух других, получается в результате применения к битовым представлениям двух исходных множеств операции побитового «И». Как работает эта битовая операция было описано выше.

¬A = {1, 2, 4, 6, 8, 9, 11, 12}

Множество, являющееся дополнением к исходному, получается в результате применения к битовому представлению исходного множества операции побитового отрицания. Как работает эта битовая операция было описано выше.

### Алгоритм «решето Эратосфена»

Решето Эратосфена — алгоритм нахождения всех простых чисел до некоторого целого числа n включительно.

Входные данные: Натуральное число n – число, до которого включительно пользователь хочет вывести простые числа.

Выходные данные: Все простые числа в диапазоне от 1 до n включительно.

Алгоритм выполнения:

1) У пользователя запрашивается целое положительное число N, до которого необходимо найти все простые числа.

2) Создается множество из положительных целых чисел от 2 до N включительно.

3) Далее берутся подряд элементы, которые принадлежат множеству. На каждом шаге берется следующий такой элемент и из множества удаляются все последующие элементы, которые кратны ему.

4) Эти шаги продолжаются до квадратного корня от N.

5) Таким образом в множестве остаются только простые числа.

## Описание классов

### Класс TBitField

Объявление класса:

class TBitField {

private:

elem\_t\* bField;

int fieldSize;

int memSize;

int bitsInElem = sizeof(elem\_t)\*8;

int shiftSize = round(log2(bitsInElem));

int GetMemIndex(const int n) const;

elem\_t GetMemMask(const int n) const;

public:

TBitField(int len);

TBitField(const TBitField& bf);

~TBitField();

int GetLength(void) const;

void SetBit(const int n);

void ClrBit(const int n);

int GetBit(const int n) const;

bool operator==(const TBitField& bf) const;

bool operator!=(const TBitField& bf) const;

TBitField& operator=(const TBitField& bf);

TBitField operator|(const TBitField& bf);

TBitField operator&(const TBitField& bf);

TBitField operator~(void);

friend std::istream& operator>>(std::istream& in, TBitField& bf);

friend std::ostream& operator<<(std::ostream& out, const TBitField& bf);

};

Поля:

fieldSize – длина битового поля.

bField – указатель типа elem\_t на область памяти, где хранится представление битового поля.

memSize – длина выделенной для битового поля памяти.

bitsInElem – количество битов в типе elem\_t.

shiftSize – число на которое надо сдвинуть значение, чтобы взять остаток от деления на bitsInElem.

Методы:

int GetMemIndex(const int n) const;

Назначение: получение индекса элемента, где хранится бит.

Входные данные:

n – номер бита.

Выходные данные:

Индекс элемента, в котором хранится бит с номером n.

elem\_t GetMemMask(const int n) const;

Назначение: получение битовой маски.

Входные данные: n – номер бита.

Выходные данные: число типа elem\_t у которого все биты, кроме n-ого равны 0, а n-ый равен 1.

int GetLength(void) const;

Назначение: получение длины битового поля.

Выходные данные: целое число – количество бит в битовом поле.

void SetBit(const int n);

Назначение: установление бита под номером n в 1.

Входные данные: n – номер бита.

void ClrBit(const int n);

Назначение: установление бита под номером n в 0.

Входные данные: n – номер бита.

int GetBit(const int n) const;

Назначение: получение бита под номером n.

Входные данные: n – номер бита.

Выходные данные: целое число – значение бита.

bool operator==(const TBitField& bf) const;

Назначение: получение результата сравнения двух битовых полей.

Входные данные: bf – ссылка на константный объект типа Битовое поле – битовое поле, с которым будет сравниваться текущее.

Выходные данные: логическое значение – true, если поля равны; false – в противном случае.

bool operator!=(const TBitField& bf) const;

Назначение: получение результата сравнения двух битовых полей.

Входные данные: bf – ссылка на константный объект типа Битовое поле – битовое поле, с которым будет сравниваться текущее.

Выходные данные: логическое значение – false, если поля равны; true – в противном случае.

TBitField& operator=(const TBitField& bf);

Назначение: копирование значения входного битового поля в текущий.

Входные данные: bf – ссылка на константный объект типа Битовое поле – битовое поле, значение из которого будут копироваться.

Выходные данные: ссылка на изменённый текущий объект.

TBitField operator|(const TBitField& bf);

Назначение: получение результата операции битового ИЛИ между текущим и переданным битовыми полями.

Входные данные: bf – ссылка на константный объект типа Битовое поле – битовое поле, с которым у текущего будет производиться операция.

Выходные данные: результирующий объект, полученный в результате операции.

TBitField operator&(const TBitField& bf);

Назначение: получение результата операции битового И между текущим и переданным битовыми полями.

Входные данные: bf – ссылка на константный объект типа Битовое поле – битовое поле, с которым у текущего будет производиться операция.

Выходные данные: битовое поле, полученное в результате битовой операции.

TBitField operator~(void);

Назначение: получение результата операции битового отрицания текущего битового поля.

Выходные данные: битовое поле, полученное в результате битовой операции.

### Класс TSet

class TSet

{

private:

int MaxPower; // максимальная мощность множества

TBitField BitField; // битовое поле для хранения характеристического вектора

public:

TSet(int mp) : MaxPower(mp), BitField(mp) {};

TSet(const TSet& s) : MaxPower(s.MaxPower), BitField(s.MaxPower) {};

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

operator TBitField();

int GetMaxPower(void) const;

void InsElem(const int Elem);

void DelElem(const int Elem);

int IsMember(const int Elem) const;

int operator== (const TSet& s) const;

int operator!= (const TSet& s) const;

TSet& operator=(const TSet& s);

TSet operator+ (const int Elem);

TSet operator- (const int Elem);

TSet operator+ (const TSet& s);

TSet operator\* (const TSet& s);

TSet operator~ (void);

friend std::istream& operator>>(std::istream& istr, TSet& s);

friend std::ostream& operator<<(std::ostream& ostr, const TSet& s);

};

Поля:

MaxPower – максимальная мощность множества (мощность универса)

BitField – битовое поле, описывающее множество

Методы:

int GetMaxPower(void) const;

Назначение: получение мощности универса для текущего множества.

Выходные данные: целое число – мощность универса.

void InsElem(const int Elem);

Назначение: добавление элемента в множество.

Входные данные: целочисленная константа Elem – элемент, который будет добавлен.

void DelElem(const int Elem);

Назначение: удаление элемента из множества.

Входные данные: целочисленная константа Elem – элемент, который будет удален.

int IsMember(const int Elem) const;

Назначение: проверка наличия переданного элемента в множестве.

Входные данные: целочисленная константа Elem – элемент, наличие которого проверяется.

Выходные данные: целое число – 1, если входит в множество; 0, если не входит.

int operator== (const TSet& s) const;

Назначение: получение результата сравнения двух множеств.

Входные данные: s – ссылка на константный объект типа множество – множество, с которым будет сравниваться текущее.

Выходные данные: целое значение – 1, если поля равны; 0 – в противном случае.

int operator!= (const TSet& s) const;

Назначение: получение результата сравнения двух множеств.

Входные данные: s – ссылка на константный объект типа множество – множество, с которым будет сравниваться текущее.

Выходные данные: целое значение – 0, если поля равны; 1 – в противном случае.

TSet& operator=(const TSet& s);

Назначение: копирование значения входного множества в текущее.

Входные данные: s – ссылка на константный объект типа множество – множество, значение из которого будут копироваться.

Выходные данные: ссылка на изменённый текущий объект.

TSet operator+ (const int Elem);

Назначение: добавление элемента в множество.

Входные данные: элемент, который будет добавлен в множество.

Выходные данные: результирующее множество.

TSet operator- (const int Elem);

Назначение: удаление элемента из множества.

Входные данные: элемент, который будет удален из множества.

Выходные данные: результирующее множество.

TSet operator+ (const TSet& s);

Назначение: объединение двух множеств.

Входные данные: множество, с которым будет объединяться текущее.

Выходные данные: результирующее множество.

TSet operator\* (const TSet& s);

Назначение: пересечение двух множеств.

Входные данные: множество, с которым будет пересечено текущее.

Выходные данные: результирующее множество.

TSet operator~ (void);

Назначение: взятие дополнения к множеству.

Выходные данные: результирующее множество.

# Заключение

В результате данной лабораторной работы были изучены теоретические основы и практические навыки применения битовых полей и множеств в программировании. На основе полученных знаний в этой предметной области была разработана программа, которая реализует хранение битовых полей и множеств, а также операции над ними. Была произведена проверка корректной работы реализованного функционала с различными наборами данных. В целом, лабораторная работа помогла понять основные принципы работы с битовыми полями и множествами, их преимущества и ограничения.

# Литературы

1. Битовое поле (C++) — Википедия (wikipedia.org) [https://en.wikipedia.org/wiki/Bit\_field].
2. Битовые поля в C++ [<https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-bit-fields?view=msvc-170> ].

# Приложения

## Приложение А. Реализация класса TBitField

TBitField::TBitField(int len) {

if (len < 0)

throw std::exception("Negative size does not support");

fieldSize = len;

memSize = len / bitsInElem + 1;

bField = new elem\_t[memSize];

for (int i = 0; i < fieldSize; i++) {

ClrBit(i);

}

}

TBitField::TBitField(const TBitField& bf) {

fieldSize = bf.fieldSize;

memSize = bf.memSize;

bField = new elem\_t[memSize];

for (int i = 0; i < memSize; i++)

bField[i] = bf.bField[i];

}

TBitField::~TBitField() {

delete[] bField;

}

int TBitField::GetMemIndex(const int pos) const {

return pos >> shiftSize;

}

elem\_t TBitField::GetMemMask(const int pos) const {

return 1 << (pos & (bitsInElem - 1));

}

int TBitField::GetLength(void) const {

return fieldSize;

}

void TBitField::SetBit(const int n) {

if (n >= fieldSize || n < 0)

throw std::exception("out of range");

bField[GetMemIndex(n)] |= GetMemMask(n);

}

void TBitField::ClrBit(const int n) {

if (n >= fieldSize || n < 0)

throw std::exception("out of range");

bField[GetMemIndex(n)] &= ~(GetMemMask(n));

}

int TBitField::GetBit(const int n) const {

if (n >= fieldSize || n < 0)

throw std::exception("out of range");

return (bField[GetMemIndex(n)] & GetMemMask(n)) >> (n & (bitsInElem - 1));

}

bool TBitField::operator==(const TBitField& bf) const {

if (fieldSize != bf.fieldSize)

return false;

for (int i = 0; i < memSize - 1; i++)

if (bField[i] != bf.bField[i])

return false;

for (int i = (memSize - 1) \* bitsInElem; i < fieldSize; i++)

if (GetBit(i) != bf.GetBit(i))

return false;

return true;

}

bool TBitField::operator!=(const TBitField& bf) const {

return !(\*this == bf);

}

TBitField& TBitField::operator=(const TBitField& bf) {

if (this == &bf)

return \*this;

fieldSize = bf.fieldSize;

if (memSize != bf.memSize) {

memSize = bf.memSize;

delete[] bField;

bField = new elem\_t[memSize];

}

for (int i = 0; i < memSize; i++) {

bField[i] = bf.bField[i];

}

return \*this;

}

TBitField TBitField::operator|(const TBitField& bf) {

const TBitField\* max\_field = this;

int min\_size = bf.fieldSize;

if (fieldSize < min\_size) {

max\_field = &bf;

min\_size = fieldSize;

}

TBitField res(\*max\_field);

for (int i = 0; i < min\_size; i++)

if ((GetBit(i) || bf.GetBit(i)) == 1)

res.SetBit(i);

return res;

}

TBitField TBitField::operator&(const TBitField& bf) {

int max\_size = fieldSize;

int min\_size = bf.fieldSize;

if (bf.fieldSize > max\_size) {

max\_size = bf.fieldSize;

min\_size = fieldSize;

}

TBitField res(max\_size);

for (int i = 0; i < min\_size; i++)

if ((GetBit(i) && bf.GetBit(i)) == 1)

res.SetBit(i);

return res;

}

TBitField TBitField::operator~(void) {

TBitField res(fieldSize);

for (int i = 0; i < memSize; i++) {

res.bField[i] = ~bField[i];

}

return res;

}

## Приложение Б. Реализация класса TSet

TSet::operator TBitField() {

TBitField tmp(BitField);

return tmp;

}

int TSet::GetMaxPower(void) const {

return MaxPower;

}

void TSet::InsElem(const int Elem) {

if (Elem >= MaxPower)

throw std::exception("Elem is out of univers");

BitField.SetBit(Elem);

}

void TSet::DelElem(const int Elem) {

if (Elem >= MaxPower)

throw std::exception("Elem is out of univers");

BitField.ClrBit(Elem);

}

int TSet::IsMember(const int Elem) const {

return BitField.GetBit(Elem);

}

int TSet::operator==(const TSet& s) const {

return BitField == s.BitField;

}

int TSet::operator!=(const TSet& s) const {

return !(\*this == s);

}

TSet& TSet::operator=(const TSet& s) {

MaxPower = s.MaxPower;

BitField = s.BitField;

return \*this;

}

TSet TSet::operator+(const int Elem) {

TSet res(\*this);

res.InsElem(Elem);

return res;

}

TSet TSet::operator- (const int Elem) {

TSet res(\*this);

res.DelElem(Elem);

return res;

}

TSet TSet::operator+(const TSet& s) {

return TSet(BitField | s.BitField);

}

TSet TSet::operator\*(const TSet& s) {

return TSet(BitField & s.BitField);

}

TSet TSet::operator~(void) {

return TSet(~BitField);

}

## Приложение B. Пример использования битовых полей

#include "tbitfield.h"

#include <iostream>

int main() {

TBitField a(18);

TBitField b(10);

TBitField c(18);

try {

std::cout << "a: " << a << std::endl;

std::cout << "b: " << b << std::endl;

std::cout << "c: " << c << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

std::cout << "\n-----------------------------------\n";

try {

std::cout << "a: ";

std::cin >> a;

std::cout << "b: ";

std::cin >> b;

std::cout << "c: ";

std::cin >> c;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "a: " << a << std::endl;

std::cout << "b: " << b << std::endl;

std::cout << "c: " << c << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

std::cout << "\n-----------------------------------\n";

try {

std::cout << "a | c :: " << (a | c) << std::endl;

std::cout << "c & a :: " << (c & a) << std::endl;

std::cout << "~b :: " << (~b) << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "a == c ? Answer: " << (a == c) << std::endl;

std::cout << "c != b ? Answer: " << (c != b) << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

b = a;

std::cout << "Variable b was assigned variable a\n";

std::cout << "b == a ? Answer: " << (b == a) << std::endl;

std::cout << "c != b ? Answer: " << (c != b) << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

return 0;

}

## Приложение Г. Пример использования множеств

#include "tset.h"

#include <iostream>

int main() {

int size\_U;

std::cout << "Enter size of set U for set a:" << std::endl;

std::cin >> size\_U;

TSet a(size\_U);

std::cout << "Enter size of set U for set b:" << std::endl;

std::cin >> size\_U;

TSet b(size\_U);

std::cout << "Enter size of set U for set c:" << std::endl;

std::cin >> size\_U;

TSet c(size\_U);

try {

std::cout << "a ( |U| = " << a.GetMaxPower() << " ): " << a << std::endl;

std::cout << "b ( |U| = " << b.GetMaxPower() << " ): " << b << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "a ( |U| = " << a.GetMaxPower() << " ): ";

std::cin >> a;

std::cout << "b ( |U| = " << b.GetMaxPower() << " ): ";

std::cin >> b;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "a ( |U| = " << a.GetMaxPower() << " ): " << a << std::endl;

std::cout << "b ( |U| = " << b.GetMaxPower() << " ): " << b << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "a + b = " << (a + b) << std::endl;

std::cout << "a + 3 = " << (a + 3) << std::endl;

std::cout << "a \* b = " << (a \* b) << std::endl;

std::cout << "b - 2 = " << (b - 2) << std::endl;

std::cout << "~a = " << (~a) << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "c ( |U| = " << c.GetMaxPower() << " ): ";

std::cin >> c;

std::cout << "c ( |U| = " << c.GetMaxPower() << " ): " << c << std::endl;

std::cout << "c == a ? " << (c == a) << std::endl;

std::cout << "c != b ? " << (c != b) << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

try {

std::cout << "c = a" << std::endl;

c = a;

std::cout << "c == a ? " << (c == a) << std::endl;

std::cout << "c != b ? " << (c != b) << std::endl;

}

catch (std::exception exp) {

std::cout << exp.what();

}

return 0;

}

## Приложение Д. Алгоритм решета Эратосфена

#include <iostream>

#include "tbitfield.h"

using namespace std;

int main()

{

int n, m, k, count;

setlocale(LC\_ALL, "Russian");

cout << "Enter n: ";

cin >> n;

TBitField s(n + 1);

// заполнение множества

for (m = 2; m <= n; m++)

s.SetBit(m);

// проверка до sqrt(n) и удаление кратных

for (m = 2; m \* m <= n; m++)

// если m в s, удаление кратных

if (s.GetBit(m))

for (k = 2 \* m; k <= n; k += m)

s.ClrBit(k);

cout << endl << "Prime numbers up to n" << endl;

count = 0;

k = 1;

for (m = 2; m <= n; m++)

if (s.GetBit(m)) {

count++;

cout << m << " ";

if (k++ % 10 == 0)

cout << endl;

}

cout << endl;

cout << "There are " << count << " prims in the first " << n << " numbers" << endl;

return 0;

}