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 механики**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

**«Битовые поля и множества»**

**Выполнил:** студент группы 3822Б1ФИ2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Ляпин Н.А./

Подпись

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

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

Подпись

Нижний Новгород  
2023

**Содержание**

[Введение 3](#_Toc147915966)

[1 Постановка задачи 4](#_Toc147915967)

[2 Руководство пользователя 5](#_Toc147915968)

[2.1 Приложение для демонстрации работы битовых полей 5](#_Toc147915969)

[2.2 Приложение для демонстрации работы множеств 5](#_Toc147915970)

[2.3 «Решето Эратосфено» 5](#_Toc147915971)

[3 Руководство программиста 6](#_Toc147915972)

[3.1 Описание алгоритмов 6](#_Toc147915973)

[3.1.1 Битовые поля 6](#_Toc147915974)

[3.1.2 Множества 6](#_Toc147915975)

[3.1.3 «Решето Эратосфена» 6](#_Toc147915976)

[3.2 Описание программной реализации 6](#_Toc147915977)

[3.2.1 Описание класса TBitField 6](#_Toc147915978)

[3.2.2 Описание класса TSet 7](#_Toc147915979)

[Заключение 8](#_Toc147915980)

[Литература 9](#_Toc147915981)

[Приложения 10](#_Toc147915982)

[Приложение А. Реализация класса TBitField 10](#_Toc147915983)

[Приложение Б. Реализация класса TSet 10](#_Toc147915984)

# Введение

Одним из способов хранить большие данные является битовое поле.

Одно из основных преимуществ использования битовых полей заключается в экономии памяти. Вместо того чтобы хранить несколько переменных, каждая из которых использует все биты своего типа данных, мы можем объединить их в одну переменную, эффективно используя каждый бит для определенного состояния или флага. Таким образом, мы можем существенно сократить объем памяти, занимаемый такими переменными.

Кроме экономии памяти, битовые поля также могут улучшить производительность программы, особенно в ситуациях, где требуется быстрое и эффективное чтение и запись флагов или состояний. Битовые операции, такие как логическое И (&), логическое ИЛИ (|) и сдвиги, позволяют нам манипулировать отдельными битами в битовом поле, что может быть очень полезным в определенных сценариях программирования.

Множества, в свою очередь, являются наборами элементов, где каждый элемент может быть включен или исключен из множества. Множества можно представить с помощью битовых полей, где каждый бит соответствует наличию или отсутствию элемента в множестве. Такое представление позволяет быстро проверять принадлежность элемента к множеству, выполнять операции объединения, пересечения и разности множеств, используя битовые операции.

# Постановка задачи

Цель: Целью данной лабораторной работы является разработка своих Битового поля и Множества битого поля с помощью языка программирования C++. Основными целями, которые мы реализуем, являются операции с битовыми полями, включающие установку, сброс и проверку наличия определенного бита. В то же время, класс Множества будет предоставлять операции с множествами, такие как объединение, пересечение и разность.

Для достижения поставленных целей мы разработаем класс, который будет предоставлять функциональность для работы с отдельными битами внутри переменной. Мы реализуем основные операции с битовыми полями, чтобы можно было устанавливать, сбрасывать и проверять наличие битов.

Задачи:

- Спроектировать класс TBitField, который будет иметь возможность работы с битовым полем.

- Реализовать класс TBitField, реализовать его поля, конструкторы, методы и перегрузки операторов.

- Спроектировать класс TSet, который будет предоставлять функциональность для работы с множествами

- Реализовать класс TSet, его поля, конструкторы, методы и перегрузки операторов.

- Проверка работоспособности этих классов с помощью тестов.

# Руководство пользователя

## Приложение для демонстрации работы битовых полей

1. Запустите приложение с названием sample\_tbitfield.exe. В результате появится окно, показанное ниже (Рисунок 1. Основное окно программы.).

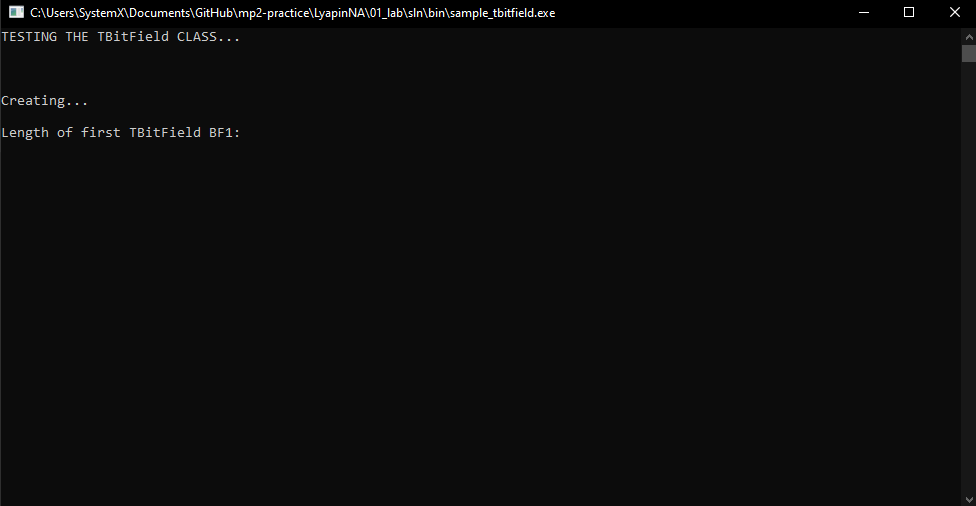


Рисунок . Основное окно программы.

На первом этапе мы проверяем правильность создания битовых полей. Далее нам необходимо ввести длины для двух экземпляров битовых полей BF1 и BF2. Для примера, я введу значения 6 и 10 (Рисунок 2. Создание битовых полей). Использование отрицательных значений гарантирует нам исключение.

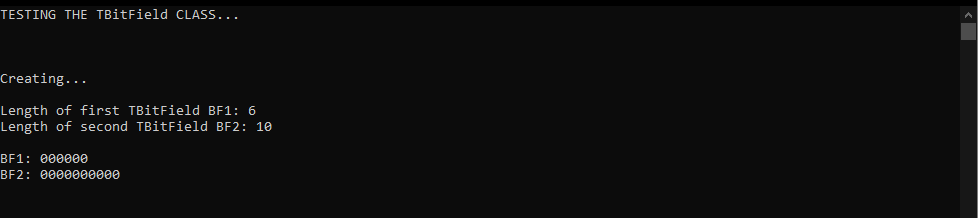


Рисунок . Создание битовых полей

1. На втором этапе мы проверяем работу методов, которые обеспечивают работу с битами поля. Среди этих методов: получение значение бита, установка и очищение бита, получение длины поля (Рисунок 3. Проверка доступа к битам поля.). В этом примере мы в первом битовом поле устанавливаем 4ый бит, а во втором, после предварительной установки всех битов поля, очищаем 2ой бит. Также здесь наглядно показана работа метода получения длины поля и перегрузка вывода для них.

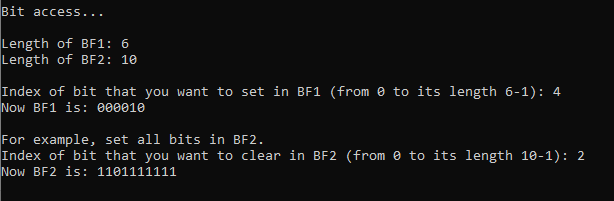


Рисунок . Проверка доступа к битам поля.

1. На заключительном этапе мы проверяем работу операторов битовых полей, таких как: операторы сравнения, отрицания, «или», «и» и присваивания. (Рисунок 4. Работа операций битового поля.).

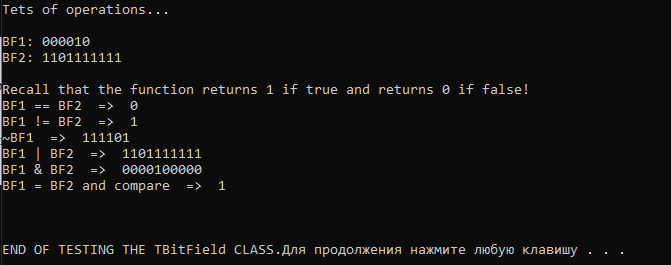


Рисунок . Работа операций битового поля.

## Приложение для демонстрации работы множеств

1. Запустите приложение с названием sample\_tset.exe. В результате появится окно, показанное ниже (Рисунок *5*. Стартовое окно приложения.).

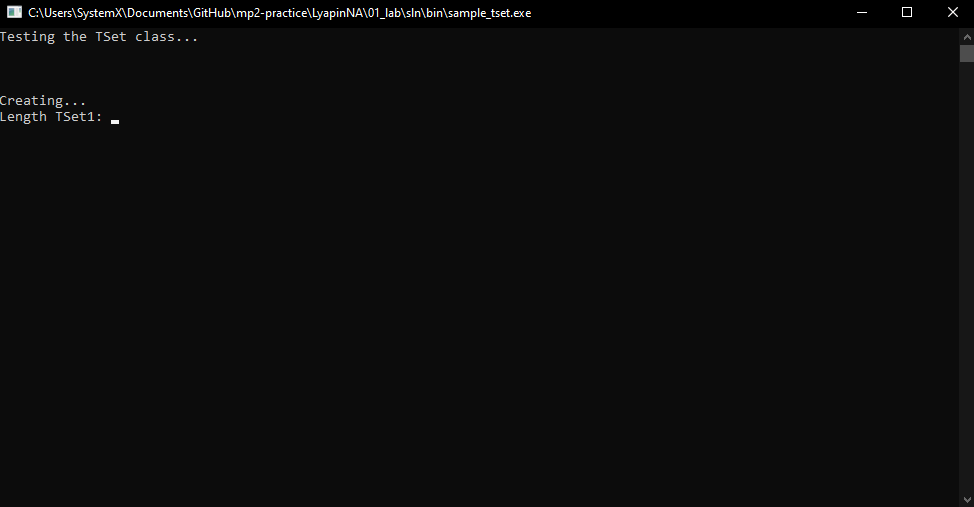


Рисунок 5. Стартовое окно приложения.

1. На первом этапе мы проверяем создание экземпляров множества. Создадим два экземпляра размера 6 и 10 соответственно. Заполним второе множество цифрами 1, 2, 3. (Рисунок *6*. Создавшиеся экземпляры TS1 и TS2.).

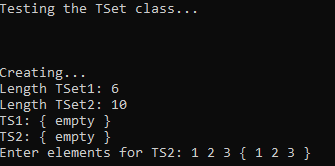


Рисунок 6. Создавшиеся экземпляры TS1 и TS2.

1. На втором этапе мы проверяем работу методов класса, которые позволяют нам получить доступ к битовому полю. На картинке (Рисунок *7*. Демонстрация работы методов множества.) мы видим включение элемента под индексом 4 в экземпляре TS1, удаление элемента под индексом 2 в экземпляре TS2, определение принадлежности элемента множеству и получение мощности множества.

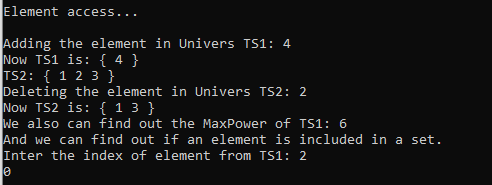


Рисунок 7. Демонстрация работы методов множества.

1. На заключительном этапе мы проверяем работу перегрузок операторов класса

(Рисунок *8*. Демонстрация работы операторов множества.). Мы проверяем работу операторов сравнения, работу дополнения, сложения и умножения двух экземпляров классов, прибавления и вычитания элемента множества.

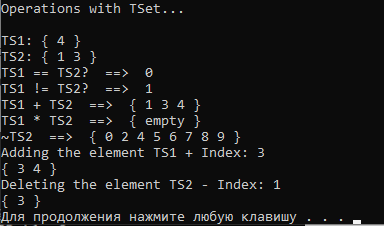


Рисунок 8. Демонстрация работы операторов множества.

## «Решето Эратосфено»

1. Запустите приложение с названием sample\_tbitfield.exe. В результате появится окно, показанное ниже (Рисунок *9*. Основное окно программы.).

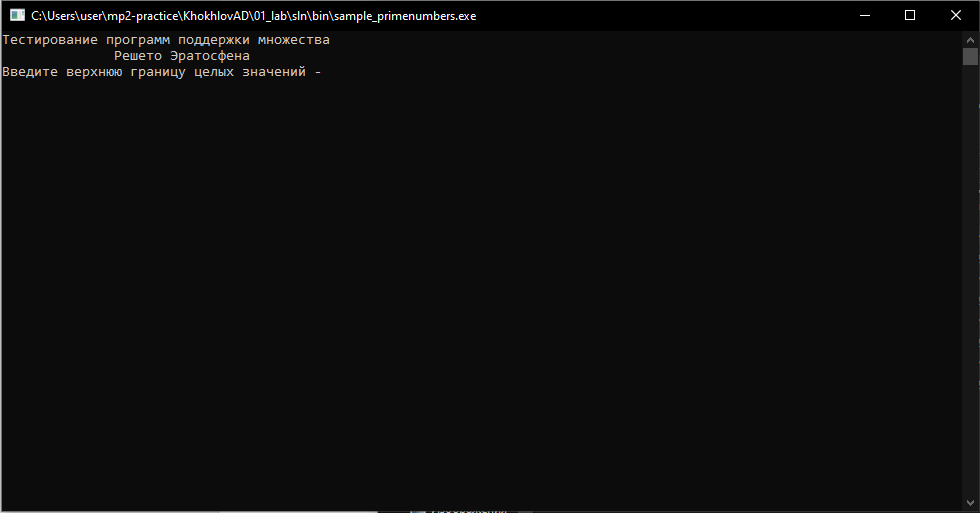


Рисунок 9. Основное окно программы.

1. Введите верхнюю границу целых значений для поиска простых чисел. В результате выведено множество в битовом представлении. Ниже выведен список простых чисел (Рисунок *10*. Результат работы.). Для завершения программы нажмите любую клавишу.

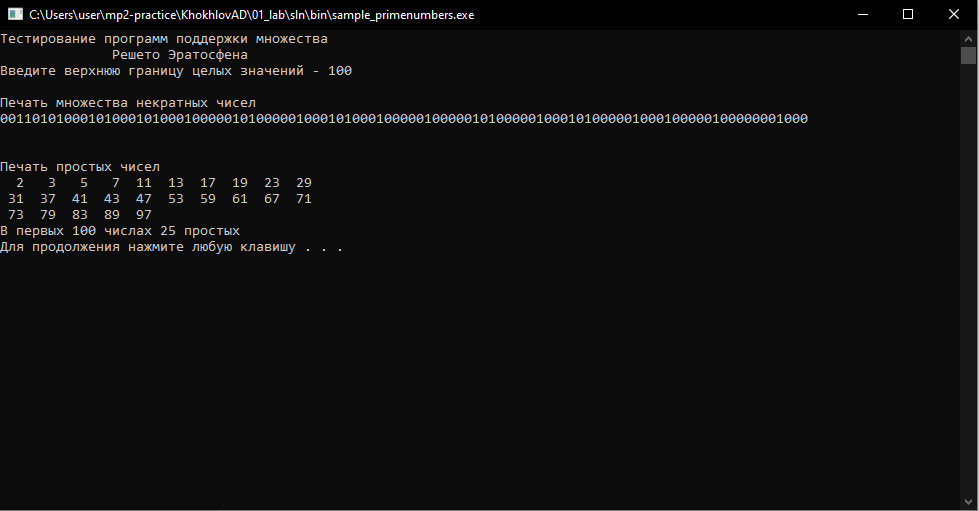


Рисунок 10. Результат работы.

# Руководство программиста

## Описание алгоритмов

### Битовые поля

Битовые поля реализуются на языке программирования C++ с помощью класса с полями, конструкторами, методами и перегрузками операторов, которые необходимы для правильной работы этой структуры данных.

В классе мы храним поля: размер битового поля, размер массива целочисленных элементов, благодаря которым реализуется битовое поле, а также размер этого массива элементов.

Пример битового поля: 0010111 – битовое поле, размера 7.

У битового поля реализованы операции сравнения, пересечения, обьединения, дополнения.

Операция сравнения выполняет поэлементное сравнение каждого бита в одной поле с каждым битом в другом поле, причем эти поля должны быть одного размера, иначе поля предварительно не могут быть равными. Операция возвращает булевое значение – true(1) в случае равенства полей, false(0) в случае неравенства.

Пример: 01001 == 01001 вернет нам правду (true).

Результатом операции пересечения является битовое поле, которое образовано из двух исходных. Алгоритм создания следующий: сравниваются побитово исходные поля, в результирующем поле устанавливается бит, если хотя бы один бит из первоначальных полей был установлен, и очищается, если оба бита на этом индексе в двух исходных полях равны 0, то есть отсутствуют. Операция возвращает экземпляр класса.

Пример: 01001110 & 1101110 вернет нам 01001100.

Результатом операции объединения является битовое поле, которое также, как и в случае операции пересечения, образовано из двух исходных. Алгоритм также схож с алгоритмом пересечения, но при побитовом сравнении полей для того чтобы в результирующем поле был включен бит необходимо, чтобы на этом же индексе оба бита в сходных полях были установлены, в остальных случаях в результирующем поле на этом месте стоит 0. Алгоритм возвращает экземпляр класса.

Пример: 11011 | 001001 вернет нам 111111.

Операция дополнения (или отрицания) возвращает побитовое поле, то есть экземпляр класса, образованное на основе исходного. Алгоритм преобразования следующий – каждый установленный бит в исходном поле – очищается в результирующем, и каждый отсутствующий бит в исходной – устанавливается в возвращаемом.

Пример: ~101010 вернет нам 010101.

В классе реализованы перегрузки ввода и вывода побитового поля в консоли.

Операция ввода побитового поля выполняется следующим образом: вам необходимо ввести единицу 1 в месте, где необходимо чтобы бит был включен, и нуль 0 в месте, где бит должен быть очищен, столько раз, сколько свободного места в поле (какого он размера).

Операция вывода побитового поля выводит 1 в случае включения бита и 0 в случае его отсутствия, проходят таким образом побитово по всему полю.

В битовом поле есть возможность ручной установки или очищения бита.

Также стоит упомянуть один механизм, позволяющий производить операции пересечения и объединения для полей разного размера. В примерах как раз проводились такие операции. Секрет очень прост – полю, с недостающим количеством битом, приписываются в конец нули (выключенные биты) для уравновешивания размеров полей.

### Множества

Множество реализовано на языке программирования C++ в виде класса, одним из полей которого является поле с экземпляром битового поля, с методами и перегрузками операций, которые должны соответствовать алгебраическому множеству.

Множество – набор целочисленных положительных элементов из юниверса U. Их наличие или отсутствие в множестве определяется битовым полем. Множество также содержит поле с его длиной.

Пример представления множества: множество A {2, 3, 4, 5, 7}

Множество поддерживает операции объединения, пересечения, дополнения, сравнения, включения и исключение элемента, а также ввода и вывода.

Операция пересечения с другим множеством возвращает экземпляр класса (множества), который содержит все уникальные элементы из обоих исходных множеств. Включение этих элементов в множество регулируется битовым полем.

Пример: A{6, 8, 9} + B{1, 2, 5, 8} = (A+B){1, 2, 5, 6, 8, 9}.

0000001011 | 011001001 = 0110011011.

Операция объединения с другим множеством возвращает экземпляр класса (множества), который содержит только уникальные элементы из обоих исходных множеств, которые принадлежат двум множествам одновременно. Включение этих элементов в множество регулируется битовым полем.

Пример: А{1, 2, 5, 6} \* B{0, 2, 3, 6, 7} = (A\*B){2, 6}.

0110011 & 10110011 = 00100010.

Операция включения и исключения в классе (множестве) возвращает множество с включенным или исключенным в него или из него числом.

Пример: A{0, 2, 4} + {7} = A`{0, 2, 4, 7}.

B{1, 2, 3} – {1} = B`{2, 3}.

Операция дополнения возвращает множество с исключенными всеми элементами и включенным в него недостающих до полного множества.

Пример: ~A{1, 3, 5} = A{0, 2, 4, 6, 7, 8} при максимальной мощности равной 9.

Операция сравнения возвращает булевое значение true или false (иначе 1 или 0) в зависимости от того, совпадают ли два сравниваемых множества. Реализовано также отрицательное сравнение через !=.

Пример: A{0, 2, 3, 4} == B{0, 2, 3, 4, 5} вернет нам false.

Операции ввода и вывода.

Операция ввода дает возможность через консоль ввести элементы множества. Для этого вам необходимо ввести поэлементно каждый элемент множества по принципу “цифра => enter” или по “e => enter”, который дает программе понять о завершении ввода (e (exit) на английской раскладке) .

Операция вывода позволяет вывести поэлементно все множество в консоль.

### «Решето Эратосфена»

Данный алгоритм реализован двумя способами, при помощи классов TSet.

Сначала необходимо ввести целое положительное число, до которого будет . После чего алгоритм заполняет все элементы классов равными 1.

После чего алгоритм , начинает перебирать все числа от 2 до N.

1. Если это число есть в нашем множестве, то мы переходим к шагу 2, иначе к шагу 3.

2. Это число, и дальше все кратные ему числа удаляются из нашего множества.

3. Выбирается следующее число. Если это число больше N, то алгоритм заканчивается, иначе – переход к шагу 1.

После чего выводятся все числа от 1 до N с идентификатором (0, если оно не простое, 1 если простое), затем выводятся все простые числа и их количество.

## Описание программной реализации

### Описание класса TBitField

class TBitField

{

private:

int BitLen;

TELEM \*pMem;

int MemLen;

// методы реализации

int GetMemIndex(const int n) const;

TELEM GetMemMask (const int n) const;

public:

TBitField(int len);

TBitField(const TBitField &bf);

~TBitField();

// доступ к битам

int GetLength(void) const;

void SetBit(const int n);

void ClrBit(const int n);

int GetBit(const int n) const;

// битовые операции

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

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

TBitField& operator=(const TBitField &bf);

TBitField operator|(const TBitField &bf);

TBitField operator&(const TBitField &bf);

TBitField operator~(void);

friend istream &operator>>(istream &istr, TBitField &bf);

friend ostream &operator<<(ostream &ostr, const TBitField &bf);

};

Назначение: представление битового поля.

Поля:

BitLen – длина битового поля – максимальное количество битов.

pMem – память для представления битового поля.

MemLen – количество элементов для представления битового поля.

Методы:

int GetMemIndex(const int n) const;

Назначение: получение индекса элемента в памяти.

Входные параметры: const int n – номер бита.

Выходные параметры: номер элемента в памяти.

**TELEM GetMemMask(const int n) const;**

Назначение: получение битовой маски.

Входные параметры: const int n – номер бита.  
 Выходные параметры: битовая маска.

**~TBitField()**

Назначение: освобождение памяти.

Входные параметры: нет.  
 Выходные параметры: нет.

**void ClrBit(const int n)**

Назначение: очищение бита.

Входные параметры: номер бита.  
 Выходные параметры: нет.

**int GetBit(const int n) const**

Назначение: получение значения бита.

Входные параметры: номер бита.  
 Выходные параметры: значение бита true (1) или false (0).

**int GetLength() const**

Назначение: получение размера битового поля.

Входные параметры: нет.  
 Выходные параметры: размер битового поля.

**int operator!=(const TBtField& tmp) const**

Назначение: отрицательное сравнение битовых полей.

Входные параметры: сравниваемое битовое поле.  
 Выходные параметры: результат сравнения true (1) или false (0).

**TBitField operator&(const TBitField& tmp)**

Назначение: проведение операции побитового «и».

Входные параметры: второе побитовое поле.  
 Выходные параметры: результирующее побитовое поле.

**TBitField operator|(const TBitField& tmp)**

Назначение: проведение операции побитового «или».

Входные параметры: второе побитовое поле.   
 Выходные параметры: результирующее побитовое поле.

**TBitField operator~()**

Назначение: проведение операции дополнения (отрицания).

Входные параметры: нет.  
 Выходные параметры: результирующее побитовое поле.

**TBitField& operator=(const TBitField& tmp)**

Назначение: операция присваивания полю значений другого поля.

Входные параметры: присваиваемое битовое поле.  
 Выходные параметры: ссылка на результирующее поле.

**int operator==(const TBitField& tmp) const**

Назначение: сравнение побитовых полей.

Входные параметры: сравниваемое битовое поле.  
 Выходные параметры: результат сравнения true (1) или false (0).

**void SetBit(const int n)**

Назначение: установка бита.

Входные параметры: номер бита.  
 Выходные параметры: нет.

**TBitField(const TBitField & tmp)**

Назначение: конструктор копирования.

Входные параметры: побитовое поле.  
 Выходные параметры: нет.

**TBitField(int n)**

Назначение: конструктор с параметром.

Входные параметры: длина битовое поля.  
 Выходные параметры: нет.

### Описание класса TSet

class TSet

{

private:

int MaxPower;

TBitField BitField;

public:

TSet(int mp);

TSet(const TSet &s);

TSet(const TBitField &bf);

operator TBitField();

// доступ к битам

int GetMaxPower(void) const;

void InsElem(const int Elem);

void DelElem(const int Elem);

int IsMember(const int Elem) const;

// теоретико-множественные операции

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 istream &operator>>(istream &istr, TSet &bf);

friend ostream &operator<<(ostream &ostr, const TSet &bf);

};

Назначение: представление множества через битовое поле.

Поля:

**BitField – битовое поле для хранения характеристического вектора.**

**MaxPower – максимальная мощность множества.**

Методы:

**void DelElem(const int n)**

Назначение: удаление элемента из множества.

Входные параметры: номер элемента.  
 Выходные параметры: нет.

**void InsElem(const int n)**

Назначение: вставка элемента в множество.

Входные параметры: элемент для вставки.  
 Выходные параметры: нет.

**int isMember(const int n)**

Назначение: проверить наличие элемента в множества.

Входные параметры: номер бита.  
 Выходные параметры: результат поиска true (1) или false (0).

**operator TBitField()**

Назначение: преобразование типа к битовому полю.

Входные параметры: нет.  
 Выходные параметры: нет.

**int operator!=(const TSet& tmp) const**

Назначение: отрицательное сравнение.

Входные параметры: сравниваемое множество.  
 Выходные параметры: результат сравнения true (1) или false (0).

**TSet operator-(const int elem)**

Назначение: разность с элементом.

Входные параметры: вычитаемый элемент.  
 Выходные параметры: результирующее множество.

**TSet operator\*(const TSet& tmp)**

Назначение: проведение операции объединения для множеств.

Входные параметры: второе множество.  
 Выходные параметры: результирующее множество.

**TSet operator~()**

Назначение: проведение операции дополнения к множеству.

Входные параметры: нет.  
 Выходные параметры: результирующее множество.

**TSet operator+(const int elen)**

Назначение: объединение с элементом.

Входные параметры: объединяемый элемент.  
 Выходные параметры: результирующее множество.

**TSet operator+(const TSet& tmp)**

Назначение: объединение с множеством.

Входные параметры: объединяемое множество.  
 Выходные параметры: результирующее множество.

**TSet& operator=(const TSet & tmp)**

Назначение: операция присваивания.

Входные параметры: присваиваемое множество.  
 Выходные параметры: результирующее множество.

**int operator==(const TSet & tmp) const**

Назначение: сравнение двух множеств.

Входные параметры: сравниваемое множество.  
 Выходные параметры: результат сравнения true (1) или false (0).

**TSet(const TBitField &tmp)**

Назначение: конструктор преобразования.

Входные параметры: преобразуемое битовое поле.  
 Выходные параметры: нет.

**Tset{const TSet &tmp)**

Назначение: конструктор копирования.

Входные параметры: копируемое множество.  
 Выходные параметры: нет.

**TSet(int n)**

Назначение: конструктор с параметром.

Входные параметры: размер множества.  
 Выходные параметры: нет.

# Заключение

Что сделано?

В ходе выполнения первой лабораторной работы «Битовые поля и множества» были сделаны следующие пункты.

1. Изучена концепция работы битовых полей.
2. Изучена теория выполнения битовых полей.
3. Была реализована программа на языке программирования C++, которая использует в своей работе класс битового поля с соответствующими битовому полю полями, методами и операциями.
4. Была продумана концепция работы множеств, основанных на работе битовых полей.
5. Была реализована программа на языке программирования С++, которая включает в себя реализованный класс множеств, работающий на основе битового поля.
6. Были проанализированы полученные результаты и сделаны выводы о преимуществах и ограничениях использования битовых полей и множеств. Оказалось, что эти структуры данных особенно полезны при работе с большими объемами данных, где компактность представления и эффективность операций являются ключевыми факторами.

# Литература

1. Сысоев А.В., Алгоритмы и структуры данных, лекция 03, 19 сентября.
2. Сайт <https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-bit-fields?view=msvc-170>, официальная документация Microsoft по битовым полям.
3. Ресурсы, свободно размещенные на территории интернета.

# Приложения

## Приложение А. Реализация класса TBitField

#include "tbitfield.h"

TBitField::TBitField(int len)

{

if (len < 0) throw "the len of massive couldn't be less then 0";

else if (len == 0)

{

BitLen = 0;

MemLen = 0;

pMem = nullptr;

}

else

{

BitLen = len;

MemLen = ((BitLen - 1) / (sizeof(TELEM) \* 8)) + 1;

pMem = new TELEM[MemLen];

for (int i = 0; i < MemLen; ++i) pMem[i] = 0;

}

}

TBitField::TBitField(const TBitField &bf) // конструктор копирования

{

BitLen = bf.BitLen;

MemLen = bf.MemLen;

pMem = new TELEM[MemLen];

for (int i = 0; i < MemLen; ++i) pMem[i] = bf.pMem[i];

}

TBitField::~TBitField() { delete[] this->pMem; }

int TBitField::GetMemIndex(const int n) const // индекс Мем для бита n

{

if(n<0 || n>BitLen-1) throw "Method is not implemented";

return n/(sizeof(TELEM)\*8);

}

TELEM TBitField::GetMemMask(const int n) const // битовая маска для бита n

{

if ((n < 0) || (n > BitLen - 1)) throw "Method is not implemented";

return 1u << (sizeof(TELEM) \* 8) - (n % (sizeof(TELEM) \* 8));

}

// доступ к битам битового поля

int TBitField::GetLength(void) const // получить длину (к-во битов)

{

return BitLen;

}

void TBitField::SetBit(const int n) // установить бит

{

if (n < 0 || n > BitLen - 1) throw "Method is not implemented";

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

}

void TBitField::ClrBit(const int n) // очистить бит

{

if (n < 0 || n > BitLen - 1) throw "Method is not implemented";

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

}

int TBitField::GetBit(const int n) const // получить значение бита

{

if (n < 0 || n > BitLen - 1) throw "Method is not implemented";

if ((pMem[GetMemIndex(n)] | GetMemMask(n)) == pMem[GetMemIndex(n)]) return 1;

else return 0;

}

// битовые операции

TBitField& TBitField::operator=(const TBitField &bf)// присваивание

{

//throw "Method is not implemented";

BitLen = bf.BitLen;

MemLen = bf.MemLen;

for (int i(0); i < MemLen; ++i) pMem[i] = bf.pMem[i];

return \*this;

}

int TBitField::operator==(const TBitField &bf) const // сравнение

{

if ((BitLen != bf.BitLen) || (MemLen != bf.MemLen)) return false;

for (int i(0); i < MemLen; ++i) if (pMem[i] != bf.pMem[i]) return false;

return true;

}

int TBitField::operator!=(const TBitField &bf) const // сравнение

{

return !(\*this == bf);

}

TBitField TBitField::operator|(const TBitField &bf) // операция "или"

{

const int newBitLen = max(BitLen, bf.BitLen);

TBitField result(newBitLen);

for (int i(0); i < newBitLen; i++)

{

if (i >= BitLen)

{

//if (bf.GetBit(i) == 1) result.SetBit(i);

(bf.GetBit(i) == 1) ? (result.SetBit(i)) : false;

}

else if (i >= bf.BitLen)

{

(GetBit(i) == 1) ? (result.SetBit(i)) : false;

}

else

{

if ((GetBit(i) == 0) && (bf.GetBit(i) == 0)) true;

else result.SetBit(i);

}

}

return result;

}

TBitField TBitField::operator&(const TBitField &bf) // операция "и"

{

//throw "Method is not implemented";

const int newBitLen = max(BitLen, bf.BitLen);

TBitField result(newBitLen);

for (int i(0); i < newBitLen; i++)

{

if ((i >= BitLen) || (i >= bf.BitLen))

{

false;

}

else

{

if ((GetBit(i) == 1) && (bf.GetBit(i) == 1)) result.SetBit(i);

}

}

return result;

}

TBitField TBitField::operator~(void) // отрицание

{

//throw "Method is not implemented";

TBitField result(BitLen);

for (int i(0); i < BitLen; ++i)

{

if (!GetBit(i)) result.SetBit(i);

}

return result;

}

// ввод/вывод

istream &operator>>(istream &istr, TBitField &bf) // ввод

{

//throw "Method is not implemented";

for (int i = 0; i < bf.GetLength(); ++i)

{

int tmp;

istr >> tmp;

if (tmp) bf.SetBit(i);

else bf.ClrBit(i);

}

return istr;

}

ostream &operator<<(ostream &ostr, const TBitField &bf) // вывод

{

//throw "Method is not implemented";

for (int i = 0; i < bf.BitLen; ++i) ostr << bf.GetBit(i);

ostr << endl;

return ostr;

}

## Приложение Б. Реализация класса TSet

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

{

}

//copy

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

{

}

//converting

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

{

}

//return maxpower

int TSet::GetMaxPower(void) const

{

return MaxPower;

}

//return bit

int TSet::IsMember(const int Elem) const

{

if (Elem < 0 || Elem >= MaxPower)

throw "Method is not implemented";

return BitField.GetBit(Elem);

}

//set bit in bitfield

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

{

if (Elem < 0 || Elem >= MaxPower)

throw "Method is not implemented";

BitField.SetBit(Elem);

}

//

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

{

if (Elem < 0 || Elem >= MaxPower)

throw "Method is not implemented";

BitField.ClrBit(Elem);

}

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//OPERATORS

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

TSet::operator TBitField()

{

return BitField;

}

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

{

if (this == &s)

return \*this;

MaxPower = s.MaxPower;

BitField = s.BitField;

return \*this;

}

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

{

if (MaxPower != s.MaxPower)

return 0;

if (BitField != s.BitField)

return 0;

return 1;

}

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

{

return !(\*this == s);

}

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

{

TSet tmp(max(MaxPower, s.GetMaxPower()));

tmp.BitField = BitField | s.BitField;

return tmp;

}

TSet TSet::operator+(const int Elem)

{

if (Elem < 0 || Elem >= MaxPower)

throw "Method is not implemented";

TSet tmp(\*this);

tmp.BitField.SetBit(Elem);

return tmp;

}

TSet TSet::operator-(const int Elem)

{

if (Elem < 0 || Elem >= MaxPower)

throw "Method is not implemented";

TSet tmp(\*this);

tmp.BitField.ClrBit(Elem);

return tmp;

}

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

{

TSet tmp(max(MaxPower, s.GetMaxPower()));

tmp.BitField = BitField & s.BitField;

return tmp;

}

TSet TSet::operator~(void)

{

TSet tmp(\*this);

tmp.BitField = ~tmp.BitField;

return tmp;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//OVERFLOWS

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//in

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

{

TELEM tmp;

for (int i = 0; i < s.GetMaxPower(); ++i)

{

tmp = \_getch();

\_getch();

if (tmp == 101) break;

cout << tmp - 48 << " ";

s.InsElem(tmp - 48);

}

return istr;

}

//out

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

{

ostr << "{";

bool haveMembers = false;

for (int i = 0; i < s.GetMaxPower(); ++i)

{

if (s.IsMember(i) == 1)

{

ostr << " " << i;

haveMembers = true;

}

}

if (haveMembers == false) ostr << " empty";

ostr << " }";

return ostr;

}

///////////////////////////////////////////////////////////////