

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
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»**

**Выполнила:** студентка группы  
3822Б1ФИ2

\_\_\_\_\_/ Ясакова Т.Е./  
Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП  
\_\_\_\_\_/ Кустикова В.Д./  
Подпись

Нижний Новгород  
2023

# Содержание

Введение.....	3
1 Постановка задачи.....	4
2 Руководство пользователя.....	5
2.1 Приложение для демонстрации работы битовых полей.....	5
2.2 Приложение для демонстрации работы множеств.....	5
2.3 «Решето Эратосфена» .....	6
3 Руководство программиста .....	8
3.1 Описание алгоритмов .....	8
3.1.1 Битовые поля .....	8
3.1.2 Множества .....	10
3.1.3 «Решето Эратосфена» .....	10
3.2 Описание программной реализации .....	8
3.2.1 Описание класса TBitField .....	10
3.2.2 Описание класса TSet .....	13
Заключение .....	17
Литература .....	18
Приложения .....	19
Приложение А. Реализация класса TBitField .....	19
Приложение Б. Реализация класса TSet.....	21

# Введение

Битовые поля и множества имеют широкий спектр применений и остаются актуальными в различных областях программирования. Давайте рассмотрим актуальность и применяемость каждой из этих концепций:

## Актуальность битовых полей (Bit Fields):

1. **Управление битовыми масками:** Битовые поля широко используются для управления битовыми масками в структурах данных. Это актуально в разработке операционных систем, устройств и драйверов, где часто требуется управление битовыми состояниями.
2. **Оптимизация памяти:** В системах с ограниченными ресурсами, таких как микроконтроллеры и встроенные системы, оптимизация памяти остается критически важной. Битовые поля могут использоваться для экономии памяти, когда каждый байт ценен.
3. **Сериализация данных:** При передаче данных через сеть или сохранении их на диске можно использовать битовые поля для упаковки и распаковки битовой информации.

## Актуальность множеств (Sets):

1. **Уникальность данных:** Множества используются для хранения уникальных элементов, и это актуально во многих сферах, включая базы данных, управление пользователями и учет дубликатов.
2. **Алгоритмы и структуры данных:** Множества играют важную роль в алгоритмах и структурах данных, таких как хеш-таблицы, сортировка и поиск. Они помогают в решении разнообразных задач, связанных с обработкой данных.
3. **Анализ данных и фильтрация:** Множества часто используются для анализа и фильтрации данных, например, при поиске уникальных элементов в больших наборах данных или при выявлении пересечений множеств.

Оба этих подхода остаются актуальными и востребованными в программировании. Выбор между ними зависит от конкретных задач и контекста разработки.

# 1 Постановка задачи

Цель – реализовать классы: TBitField и TSet

Задачи при реализации класса TBitField:

1. Описать и реализовать конструктор, конструктор копирования, деструктор.
2. Описать и реализовать операции доступа к битам: установить бит в 1, установить бит в 0, получить значение бита, получить количество доступных битов.
3. Описать и реализовать вспомогательные методы: получение индекса элемента, получение маски бита.
4. Перегрузить битовые операции: присваивание ( $=$ ), сравнение ( $==$ ,  $!=$ ), побитовое ИЛИ ( $|$ ), побитовое И ( $\&$ ), побитовое отрицание ( $\sim$ ).
5. Перегрузить операции ввода и вывода.

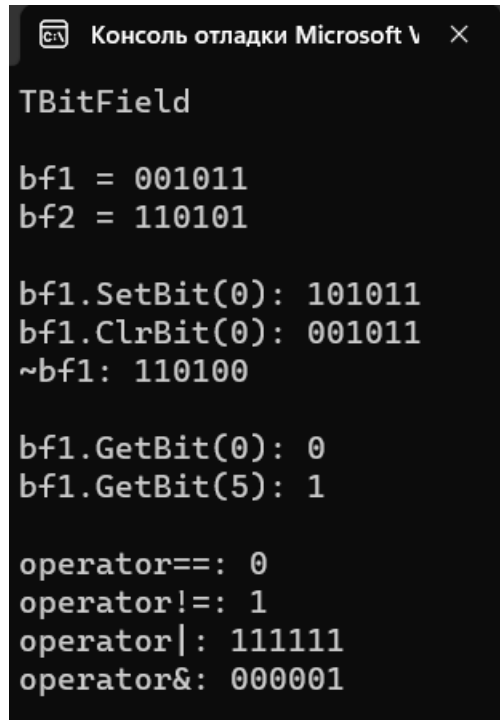
Задачи при реализации класса TSet:

1. Описать и реализовать конструктор, конструктор копирования, конструктор преобразования типа, оператор преобразования типа к битовому полю.
2. Описать и реализовать операции доступа к битам: включить элемент в множество, удалить элемент из множества, проверить наличие элемента в множестве, получить максимальной мощности множества.
3. Перегрузить теоретико-множественные операции: присваивание ( $=$ ), сравнение ( $==$ ,  $!=$ ), объединение ( $+$ ), пересечение ( $*$ ), объединение с элементом из множества ( $+$ ), разность с элементом из множества ( $-$ ), дополнение ( $\sim$ ).
4. Перегрузить операции ввода и вывода.

## 2 Руководство пользователя

### 2.1 Приложение для демонстрации работы битовых полей

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



```
TBitField

bf1 = 001011
bf2 = 110101

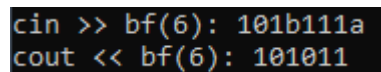
bf1.SetBit(0): 101011
bf1.ClrBit(0): 001011
~bf1: 110100

bf1.GetBit(0): 0
bf1.GetBit(5): 1

operator==: 0
operator!=: 1
operator |: 111111
operator&: 000001
```

Рис. 1. Основное окно программы sample\_tbitfield.exe

2. В появившемся окне вы можете ознакомиться с примером работы реализованных операций. Введите строку с данными битового поля, она должна содержать “0” и “1”, а также быть указанной длины (в данном случае 6). Другие символы будут считаться за “0”, а из строки большей длины будет учитываться только подстрока указанной длины. Нажмите кнопку ввода и выведется результат, пример которого указан на рисунке ниже (рис. 2).



```
cin >> bf(6): 101b111a
cout << bf(6): 101011
```

Рис. 2. Пример функций ввода и вывода класса TBitField

### 2.2 Приложение для демонстрации работы множеств

1. Запустите приложение с названием sample\_tset.exe. В результате появится окно, показанное ниже (рис. 2).

```

TSet

s1 = 001011
s2 = 110101

s1.InsElem(0): 101011
s1.DelElem(0): 001011
~s1: 110100

s1.IsMember(0): 0
s1.IsMember(5): 1

operator==: 0
operator!=: 1
operator+: 111111
operator+ (elem): 101011
operator- (elem): 001010
operator*: 000001

```

Рис. 3. Основное окно программы sample\_tset.exe

1. В появившемся окне вы можете ознакомиться с примером работы реализованных операций. Введите строку с данными битового поля, она должна содержать “0” и “1”, а так же быть указанной длины (в данном случае 6). Другие символы будут считаться за “0”, а из строки большей длинны будет учитываться только подстрока указанной длины. Нажмите кнопку ввода и выведется результат, пример которого указан на рисунке ниже (рис. 4).

```

cin >> s(6): Count of entered values: 3
2
3
7
cout << s(6): 2 3

```

Рис. 4. Пример функций ввода и вывода класса TSet

## 2.3 «Решето Эратосфено»

1. Запустите приложение с названием sample\_primenumbers.exe. В результате появится окно, показанное ниже (рис. 3).

```

Prime numbers

Enter the count of numbers: |

```

Рис. 5. Начало работы программы sample\_primenumbers.exe

2. Введите положительное целое число, чтобы вывести все простые числа до этого числа(включительно). Напечатается результат, показанный на рисунке ниже (рис. 3).

```
Prime numbers  
  
Enter the count of numbers: 40  
Prime numbers under 40:  
2 3 5 7 11 13 17 19 23 29 31 37
```

Рис. 6. Результат работы программы sample\_primenumbers.exe

## 3 Руководство программиста

### 3.1 Описание алгоритмов

#### 3.1.1 Битовые поля

Битовые поля представляют собой наборы характеристических массивов, в которых каждый элемент индексируется элементами множества. Каждое битовое поле характеризуется своей длиной (размером универсума в битах), количеством хранимых массивов и объемом памяти, необходимым для их хранения. Каждый элемент битового поля может принимать одно из двух состояний: 1 (если элемент присутствует в множестве) или 0 (если элемент отсутствует в множестве). Этот алгоритм позволяет создать интерфейс для управления множествами.

Рассмотрим представление битового поля, для дальнейшего описания базовых алгоритмов:

ind	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0
sind	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
state	state[1]								state[0]							

В данном примере представлен массив характеристических массивов, состоящий суммарно из 16 битов (bit). В каждом характеристическом массиве хранится по 8 бит (количество битов в элементе зависит от типа памяти элемента). У каждого бита есть свой индекс (ind), как у обычной последовательности битов. Но, из-за способа хранения, у битового поля 2 разных индекса: индекс элемента в массиве элементов (state), индекс бита в элементе (sind).

Рассмотрим побитовые операции, которые будут применяться в битовых полях:

- Побитовый сдвиг вправо на  $i$  (целочисленное деление на  $2^i$ ).

$$1100101_2 \gg 3 = 1100_2 \equiv 101_{10} / 2^3 = 12_{10}$$

- Побитовый сдвиг влево на  $i$  (умножение на  $2^i$ ).

$$1100101_2 \ll 1 = 11001010_2 \equiv 101_{10} * 2^1 = 202_{10}$$

- Побитовое “ИЛИ” ( $|$ ).

	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0
	0	1	0	1	1	0	1	1	0	1	1	0	0	0	0	0
ans	1	1	0	1	1	0	1	1	0	1	1	0	1	0	1	0



- Побитовое “И” ( & ).

	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0
<b>&amp;</b>	0	1	0	1	1	0	1	1	0	1	1	0	0	0	0	0
<b>ans</b>	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0

- Побитовое отрицание ( ~ ).

	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0
<b>~</b>	0	1	1	1	0	1	1	1	1	0	1	1	0	1	0	1

- Битовая маска i-го бита.

$$1 \ll i$$

- Обозначить i-й бит 1.

$$state|(1 \ll i)$$

ind	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	state
	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1<<7
<b>ans</b>	1	0	0	0	1	0	0	1	0	1	0	0	1	0	1	

- Обозначить i-й бит 0.

$$state\&(\sim(1 \ll i))$$

ind	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	1	0	0	0	1	0	0	1	0	1	0	0	1	0	1	state
<b>&amp;</b>	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	~(1<<7)
<b>ans</b>	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	

- Получить состояние i-го бита.

$$(state\&(1 \ll i)) \gg i$$

В реализации битовых полей данные хранятся в массиве характеристических массивов, значит придется получать индекс ячейки памяти и индекс бита в этой ячейке по входящему индексу битового поля. Для удобства введем следующие значения: `bitsInElem` – количество бит в элементе битового поля; `shiftSize = i (2i = bitsInElem)`.

По этому операции над битовым полем приобретут следующий вид:

- Битовая маска i-го бита.

$$1 \ll (i \& (bitsInElem - 1))$$

- Обозначить i-й бит 1.

$$state[i \gg shiftSize] | (1 \ll (i \& (bitsInElem - 1)))$$

- Обозначить i-й бит 0.

$$state[i \gg shiftSize] \& \sim(1 \ll (i \& (bitsInElem - 1)))$$

### 3.1.2 Множества

Множества полностью основаны на битовых полях, представляя собой интерфейс для них. Битовое поле описывает каждый элемент универсума: если бит равен 1, то элемент присутствует в множестве; если бит равен 0, то в множестве его нет.

Каждое множество может иметь свой смысл и применяемость. В данной реализации мы рассматриваем множество натуральных чисел, на основе которого мы формируем подмножества на основе битовых полей.

### 3.1.3 «Решето Эратосфена»

Решето Эратосфена (англ. sieve of Eratosthenes) — алгоритм нахождения всех простых чисел от 1 до n.

Основная идея соответствует названию алгоритма: запишем ряд чисел 1, 2, ..., n, а затем будем вычеркивать сначала

- числа, делящиеся на 2, кроме самого числа 2,
- потом числа, делящиеся на 3, кроме самого числа 3,
- с числами, делящимися на 4, ничего делать не будем — мы их уже вычёркивали,
- потом продолжим вычеркивать числа, делящиеся на 5, кроме самого числа 5,

...и так далее.

Алгоритм Решето Эратосфена имеет сложность  $O(N \cdot \log(\log(N)))$ , что делает его очень эффективным для нахождения простых чисел в больших диапазонах.

## 3.2 Описание программной реализации

### 3.2.1 Описание класса TBitField

```
typedef unsigned int TELEM;
class TBitField
{
private:
    int BitLen;
    TELEM *pMem;
```

```

int MemLen;

const int bitsInElem = 32;
const int shiftSize = 5;

// методы реализации
int GetMemIndex(const int n) const noexcept;
TELEM GetMemMask (const int n) const noexcept;
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
bool operator==(const TBitField &bf) const;
bool operator!=(const TBitField &bf) const;
const TBitField& operator=(const TBitField &bf);
TBitField operator|(const TBitField &bf);
TBitField operator&(const TBitField &bf);
TBitField operator~(void);

friend istream& operator>>(istream& in, TBitField& bf);
friend ostream& operator<<(ostream& out, 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.

**bitsInElem** – вспомогательное значение, количество битов в элементе памяти.

**shiftSize** – вспомогательное значение для битового целочисленного деления.

### Конструкторы:

```
TBitField(int len);
```

Назначение: выделение и инициализация памяти объекта.

Входные параметры: **len** – количество доступных битов.

```
TBitField(const TBitField &bf);
```

Назначение: выделение памяти и копирование данных.

Входные параметры: **bf** – объект класса **TBitField**.

```
~TBitField();
```

Назначение: очистка выделенной памяти.

### Методы:

**int GetMemIndex(const int n) const noexcept;**

Назначение: получение индекса элемента в памяти.

Входные параметры: **n** – номер бита.

Выходные параметры: индекс элемента в памяти.

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

Назначение: получение маски бита

Входные параметры: **n** – номер бита.

Выходные параметры: маска бита

**int GetLength(void) const;**

Назначение: получение количества доступных битов.

Выходные параметры: **BitLen** - количество доступных битов.

**void SetBit(const int n);**

Назначение: изменить значение бита на 1.

Входные параметры: **n** – номер бита.

**void ClrBit(const int n);**

Назначение: изменить значение бита на 0.

Входные параметры: **n** – номер бита.

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

Назначение: получение значения бита.

Входные параметры: **n** – номер бита.

Выходные параметры: значение бита – 1 или 0.

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

Назначение: перегрузка операции сравнения, сравнение на равенство объектов.

Входные параметры: **bf** – объект класса **TBitField**.

Выходные параметры: **true** или **false**.

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

Назначение: перегрузка операции сравнения, сравнение на неравенство объектов.

Входные параметры: **bf** – объект класса **TBitField**.

Выходные параметры: **true** или **false**.

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

Назначение: присваивание значений объекта **bf**.

Входные параметры: **bf** – объект класса **TBitField**.

Выходные параметры: ссылка на объект класса **TBitField** (себя).

```
TBitField operator|(const TBitField &bf);
```

Назначение: создание объекта с примененной побитовой операцией ИЛИ.

Входные параметры: **bf** – объект класса **TBitField**.

Выходные параметры: объект класса **TBitField**.

```
TBitField operator&(const TBitField &bf);
```

Назначение: создание объекта с примененной побитовой операцией И.

Входные параметры: **bf** – объект класса **TBitField**.

Выходные параметры: объект класса **TBitField**.

```
TBitField operator~(void);
```

Назначение: создание объекта с примененной побитовой операцией отрицания.

Выходные параметры: объект класса **TBitField**.

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

Назначение: ввод данных.

Входные параметры: **in** – поток ввода, **bf** – объект класса **TBitField**.

Выходные параметры: поток ввода.

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

Назначение: вывод данных.

Входные параметры: **out** – поток вывода, **bf** – объект класса **TBitField**.

Выходные параметры: поток вывода.

### 3.2.2 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 noexcept;
    void InsElem(const int Elem);
    void DelElem(const int Elem);
    bool IsMember(const int Elem) const;

    // теоретико-множественные операции
    bool operator== (const TSet& s) const;
    bool operator!= (const TSet& s) const;
    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& in, TSet& bf);
    friend ostream& operator<<(ostream& out, const TSet& bf);
};

```

Назначение: представление множества.

### Поля:

**MaxPower** – мощность множества.

**BitField** – характеристический массив.

### Конструкторы:

**TSet(int mp);**

Назначение: инициализация битового поля.

Входные параметры: **mp** – количество элементов в универсуме.

**TSet(const TSet& s);**

Назначение: копирование данных из другого множества.

Входные параметры: **s** – объект класса **TSet**.

**TSet(const TBitField& bf);**

Назначение: формирование множества на основе битового поля.

Входные параметры: **bf** – объект класса **TBitField**.

**operator TBitField();**

Назначение: получение поля **BitField**.

Выходные параметры: объект класса **TBitField**.

### Методы:

**int GetMaxPower(void) const noexcept;**

Назначение: получение максимальной мощности множества.

Выходные параметры: **MaxPower** – максимальная мощность множества.

**void InsElem(const int Elem);**

Назначение: добавить элемент в множество.

Входные параметры: **Elem** – индекс элемента.

**void DelElem(const int Elem);**

Назначение: удаление элемента из множества.

Входные параметры: **Elem** – индекс элемента.

**bool IsMember(const int Elem) const;**

Назначение: проверка, состоит ли элемент в множестве.

Входные параметры: **true** или **false**.

**bool operator== (const TSet& s) const;**

Назначение: перегрузка операции сравнения, сравнение на равенство объектов.

Входные параметры: **s** – объект класса **TSet**.

Выходные параметры: **true** или **false**.

**bool operator!= (const TSet& s) const;**

Назначение: перегрузка операции сравнения, сравнение на неравенство объектов.

Входные параметры: **s** – объект класса **TSet**.

Выходные параметры: **true** или **false**.

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

Назначение: присваивание значений объекта **s**.

Входные параметры: **s** – объект класса **TSet**.

Выходные параметры: ссылка на объект класса **TSet** (себя).

**TSet operator+ (const int Elem);**

Назначение: добавление элемента в множество.

Входные параметры: **Elem** – индекс элемента.

Выходные параметры: объект класса **TSet**.

**TSet operator- (const int Elem);**

Назначение: удаление элемента из множества.

Входные параметры: **Elem** – индекс элемента.

Выходные параметры: объект класса **TSet**.

**TSet operator+ (const TSet& s);**

Назначение: объединение двух множеств.

Входные параметры: **s** – объект класса **TSet**.

Выходные параметры: объект класса **TSet**.

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

Назначение: пересечение двух множеств.

Входные параметры: **s** – объект класса **TSet**.

Выходные параметры: объект класса **TSet**.

**TSet operator~ (void);**

Назначение: получение дополнения к множеству.

Выходные параметры: объект класса **TSet**.

**friend istream& operator>>(istream& in, TSet& s);**

Назначение:

Входные параметры: **s** – объект класса **TSet**, **in** – поток ввода.

Выходные параметры: поток ввода.

**friend ostream& operator<<(ostream& out, const TSet& s);**

Назначение:

Входные параметры: **s** – объект класса **TSet**, **out** – поток вывода.

Выходные параметры: поток вывода.



## Заключение

Были реализованы классы: TBitField и TSet

Выполнены задачи при реализации класса TBitField:

1. Описать и реализовать конструктор, конструктор копирования, деструктор.
2. Описать и реализовать операции доступа к битам: установить бит в 1, установить бит в 0, получить значение бита, получить количество доступных битов.
3. Описать и реализовать вспомогательные методы: получение индекса элемента, получение маски бита.
4. Перегрузить битовые операции: присваивание ( $=$ ), сравнение ( $==$ ,  $!=$ ), побитовое ИЛИ ( $|$ ), побитовое И ( $\&$ ), побитовое отрицание ( $\sim$ ).
5. Перегрузить операции ввода и вывода.

Выполнены задачи при реализации класса TSet:

1. Описать и реализовать конструктор, конструктор копирования, конструктор преобразования типа, оператор преобразования типа к битовому полю.
2. Описать и реализовать операции доступа к битам: включить элемент в множество, удалить элемент из множества, проверить наличие элемента в множестве, получить максимальной мощности множества.
3. Перегрузить теоретико-множественные операции: присваивание ( $=$ ), сравнение ( $==$ ,  $!=$ ), объединение ( $+$ ), пересечение ( $*$ ), объединение с элементом из множества ( $+$ ), разность с элементом из множества ( $-$ ), дополнение ( $\sim$ ).
4. Перегрузить операции ввода и вывода.

## Литература

1. Лекция «Множества и битовые поля» Сысоева А.В.  
<https://cloud.unn.ru/s/DLRHnt54ircG2WL>
2. Битовые поля [https://www.youtube.com/watch?v=\\_XJAeR7obBk](https://www.youtube.com/watch?v=_XJAeR7obBk)
3. Решето Эратосфена <https://yandex.ru/video/preview/2908856360907561981>

# Приложения

## Приложение А. Реализация класса TBitField

```
#include "tbitfield.h"

TBitField::TBitField(int len)
{
    if (len > 0) {
        BitLen = len;
        MemLen = ((len + bitsInElem - 1) >> shiftSize);
        pMem = new TELEM[MemLen];
        memset(pMem, 0, MemLen * sizeof(TELEM));
    }
    else if (len == 0) {
        BitLen = 0;
        MemLen = 0;
        pMem = nullptr;
    }
    else {
        throw "error: BitFields size < 0";
    }
}

TBitField::TBitField(const TBitField &bf) // конструктор копирования
{
    BitLen = bf.BitLen;
    MemLen = bf.MemLen;
    if (MemLen) {
        pMem = new TELEM[MemLen];
        memcpy(pMem, bf.pMem, MemLen * sizeof(TELEM));
    }
    else {
        pMem = nullptr;
    }
}

TBitField::~TBitField()
{
    if (MemLen > 0)
        delete[] pMem;
}

int TBitField::GetMemIndex(const int n) const noexcept // индекс Мем для бита
n
{
    return n >> shiftSize;
}

TELEM TBitField::GetMemMask(const int n) const noexcept // битовая маска для
бита n
{
    return 1 << (n & (bitsInElem - 1));
}

// 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 >= BitLen || n < 0) throw "error: index out of range";
    pMem[GetMemIndex(n)] |= GetMemMask(n);
}
```

```

void TBitField::ClrBit(const int n) // очистить бит
{
    if (n >= BitLen || n < 0) throw "error: index out of range";
    pMem[GetMemIndex(n)] &= ~GetMemMask(n);
}
int TBitField::GetBit(const int n) const // получить значение бита
{
    if (n >= BitLen || n < 0) throw "error: index out of range";
    return ((pMem[GetMemIndex(n)] & GetMemMask(n)) >> (n & (bitsInElem - 1)));
}

// битовые операции
const TBitField& TBitField::operator=(const TBitField &bf) // присваивание
{
    if (MemLen != bf.MemLen)
        if (MemLen > 0) {
            delete[] pMem;
            MemLen = bf.MemLen;
            pMem = new TELEM[MemLen];
        }

    BitLen = bf.BitLen;
    for (int i = 0; i < MemLen; i++)
        pMem[i] = bf.pMem[i];
    return (*this);
}
bool TBitField::operator==(const TBitField &bf) const // сравнение
{
    if (BitLen != bf.BitLen) return false;
    bool flag = true;
    for (int i = 0; i < MemLen; i++)
        if (pMem[i] != bf.pMem[i]) {
            flag = false;
            break;
        }
    return flag;
}
bool TBitField::operator!=(const TBitField &bf) const // сравнение
{
    return !((*this) == bf);
}
TBitField TBitField::operator|(const TBitField &bf) // операция "или"
{
    int len = (BitLen > bf.BitLen) ? BitLen : bf.BitLen;

    TBitField res(len);
    for (int i = 0; i < BitLen; i++) {
        if (GetBit(i)) res.SetBit(i);
    }
    for (int i = 0; i < bf.BitLen; i++) {
        if (bf.GetBit(i)) res.SetBit(i);
    }
    return res;
}
TBitField TBitField::operator&(const TBitField &bf) // операция "и"
{
    int len = (BitLen > bf.BitLen) ? BitLen : bf.BitLen;
    int mlen = (BitLen < bf.BitLen) ? BitLen : bf.BitLen;

    TBitField res(len);
    for (int i = 0; i < BitLen; i++) {
        if (GetBit(i) && bf.GetBit(i)) res.SetBit(i);
    }
}

```

```

        return res;
    }
    TBitField TBitField::operator~(void) // отрицание
    {
        TBitField res(BitLen);
        for (int i = 0; i < BitLen; i++)
            if (!GetBit(i)) res.SetBit(i);
        return res;
    }

    // ВВОД/ВЫВОД
    istream& operator>>(istream& in, TBitField& bf) // ввод
    {
        string ans;
        in >> ans;

        int blen = bf.BitLen;
        int len = (ans.size() < blen) ? ans.size() : blen;

        for (int i = 0; i < blen; i++) {
            bf.ClrBit(i);
        }
        for (int i = 0; i < len; i++) {
            if (ans[i] == '1') bf.SetBit(i);
        }

        return in;
    }
    ostream& operator<<(ostream& out, const TBitField& bf) // вывод
    {
        int len = bf.BitLen;
        for (int i = 0; i < len; i++) {
            if (bf.GetBit(i))
                out << '1';
            else
                out << '0';
        }
        return out;
    }
}

```

## Приложение Б. Реализация класса TSet

```

#include "tset.h"

TSet::TSet(int mp) : BitField(mp)
{
    if (mp >= 0) {
        MaxPower = mp;
    }
    else {
        throw "error: Set size < 0";
    }
}

// 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()
{
    return BitField;
}

// доступ к битам
int TSet::GetMaxPower(void) const noexcept // получить макс. к-во эл-тов
{
    return MaxPower;
}
bool TSet::IsMember(const int Elem) const // элемент множества?
{
    if (Elem >= MaxPower || Elem < 0) throw "error: index out of range";

    if (BitField.GetBit(Elem))
        return true;
    return false;
}
void TSet::InsElem(const int Elem) // включение элемента множества
{
    if (Elem >= MaxPower || Elem < 0) throw "error: index out of range";
    BitField.SetBit(Elem);
}
void TSet::DelElem(const int Elem) // исключение элемента множества
{
    if (Elem >= MaxPower || Elem < 0) throw "error: index out of range";
    BitField.ClrBit(Elem);
}

// теоретико-множественные операции
const TSet& TSet::operator=(const TSet& s) // присваивание
{
    MaxPower = s.MaxPower;
    BitField = s.BitField;
    return *this;
}
bool TSet::operator==(const TSet& s) const // сравнение
{
    if (MaxPower != s.MaxPower) return false;
    return (BitField == s.BitField);
}
bool TSet::operator!=(const TSet& s) const // сравнение
{
    return !(*this == s);
}
TSet TSet::operator+(const TSet& s) // объединение
{
    int len = (MaxPower > s.MaxPower) ? MaxPower : s.MaxPower;
    TSet res(len);

    res.BitField = BitField | s.BitField;
    return res;
}
TSet TSet::operator+(const int Elem) // объединение с элементом
{
    if (Elem >= MaxPower || Elem < 0) throw "error: index out of range";

    TSet res(*this);
    res.InsElem(Elem);
    return res;
}
TSet TSet::operator-(const int Elem) // разность с элементом
{

```

```

        if (Elem >= MaxPower || Elem < 0) throw "error: index out of range";

        TSet res(*this);
        res.DelElem(Elem);
        return res;
    }
    TSet TSet::operator*(const TSet& s) // пересечение
    {
        int len = (MaxPower > s.MaxPower) ? MaxPower : s.MaxPower;
        TSet res(len);

        res.BitField = BitField & s.BitField;
        return res;
    }
    TSet TSet::operator~(void) // дополнение
    {
        TSet res(MaxPower);
        res.BitField = ~BitField;
        return res;
    }

    // перегрузка ввода/вывода
    istream& operator>>(istream& in, TSet& s) // ввод
    {
        in >> s.BitField;
        return in;
    }
    ostream& operator<<(ostream& out, const TSet& s) // вывод
    {
        out << s.BitField;
        return out;
    }
}

```