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(а): студент(ка) группы 3822Б1ФИ2
_____ / Чижов М.А./
Подпись

Проверил: к.т.н, доцент каф. ВВиСП
_____ / Кустикова В.Д./
Подпись

Нижний Новгород
2023

Содержание

Введение

Битовые поля – это удобный и эффективный способ работы с битами в программировании. Они позволяют компактно хранить множество флагов или опции в одной переменной, что повышает производительность и экономит память.

Термин "битовые поля" относится к технике использования битов в переменных для хранения и управления набором флагов или опций. Обычно каждый флаг или опция представляются отдельным битом в переменной, чтобы обозначить, включено значение или нет.

Использование битовых полей позволяет сократить использование памяти, особенно если нам необходимо хранить большое количество опций или флагов. Например, если мы используем отдельную переменную типа `bool` для каждой опции, то каждая переменная будет занимать 1 байт (или 8 бит), в то время как битовые поля позволяют нам хранить все эти опции в одной переменной, занимающей всего 1 байт (или 8 бит).

Битовые поля также активно применяются в алгоритмах сжатия данных, где они позволяют эффективно хранить большие объемы информации в компактной форме. Например, при работе с изображениями или звуком, где каждый пиксель или сэмпл может быть представлен как набор битовых флагов.

Одной из областей применения битовых полей является работа с битовыми операциями. Битовые операции позволяют выполнять различные операции над битами в битовом поле, такие как логическое И, ИЛИ, отрицание и др. Это может быть полезно при работе с множествами, фильтрации данных или выполнении различных булевых операций.

Наконец, битовые поля находят свое применение в работе с железом, таким как микроконтроллеры или драйвера устройств. Коммуникация с железом обычно требует точного контроля над битами данных, и использование битовых полей позволяет легко манипулировать отдельными битами и управлять железными устройствами.

1 Постановка задачи

Цель – реализовать структуру хранения битового поля и множества

Задачи:

1. Реализовать классы для работы с множествами и битовыми полями.
2. Написать следующие операции для работы с битовыми полями: установить бит в 1, установить бит в 0, получить значение бита, сравнить два битовых поля, сложить и инвертировать, вывести битовое поле требуемого формата и ввести битовое поле.
3. Добавить вспомогательные операции получения бита, маски, бита, длины битового поля.
4. Написать следующие операции для работы с множествами: вставка элемента, удаление, проверка наличия, сравнение множеств, объединение множеств, объединение множества с элементом, пересечение, разность, копирование, вычисление мощности множества, вывод элементов множества требуемого формата и ввод.
5. Добавить вспомогательные операции для получения мощности множества.

2 Руководство пользователя

2.1 Приложение для демонстрации работы битовых полей

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

Рис. 1. Основное окно программы

2. Введите целое число для длины. В результате появится окно, показанное ниже (рис. 2).

Рис. 2. Ввод длины

3. Введите битовое поле, указанной выше длины (рис. 3).

Рис. 3. Ввод битового поля

4. Введите второе битовое поле. В результате появится окно с выполненными операциями над полями (рис. 4).

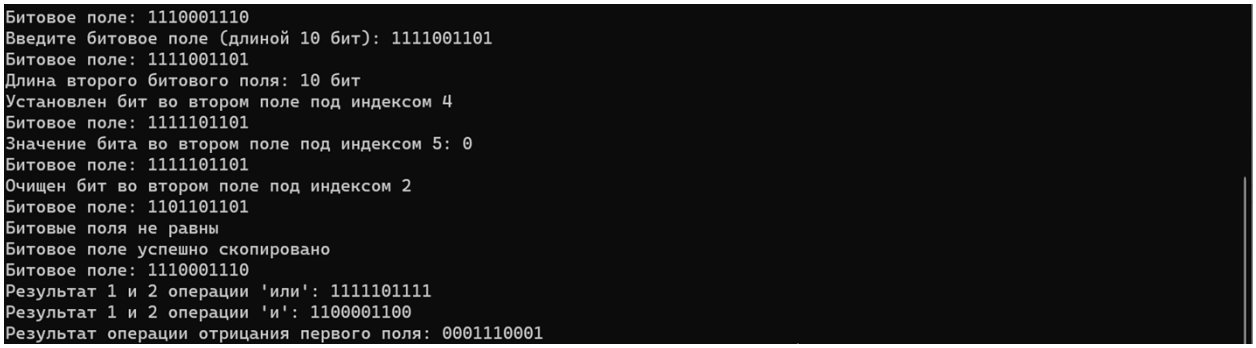

Рис. 4. Результат операций

2.2 Приложение для демонстрации работы множеств

1. Запустите приложение с названием sample_tset.exe. В результате появится окно, показанное ниже (рис. 5).

Рис. 5. Основное окно программы

2. Введите мощность множества (рис. 6).

Рис. 6. Ввод мощности

3. Введите количество элементов, которое хотите ввести, а потом введите элементы множества (рис. 7).

Рис. 7. Ввод множества

4. Проделайте те же действия для второго множества. В результате появится окно с выполненными операциями (рис. 8).

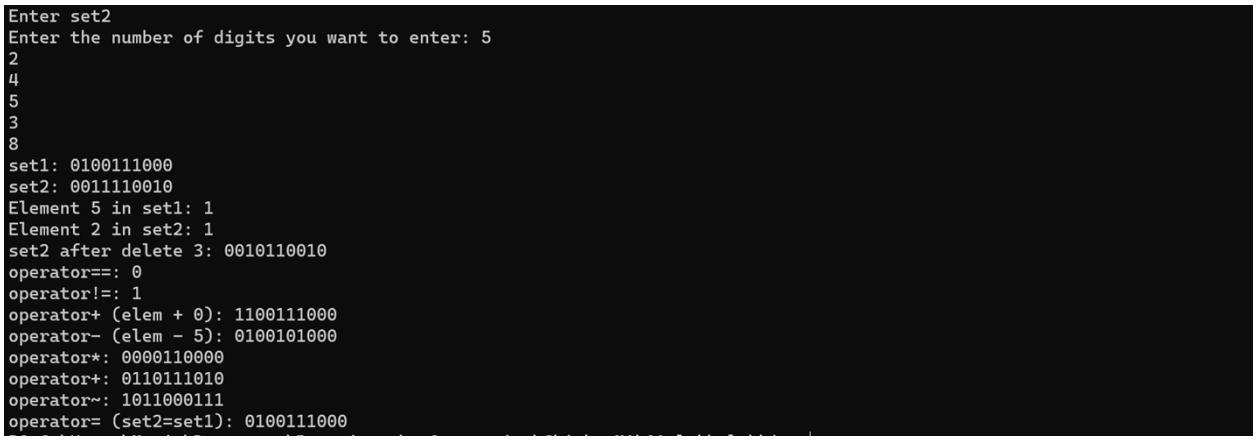

Рис. 8. Результат операций

2.3 «Решето Эратосфено»

1. Запустите приложение с названием sample_primenumbers.exe. В результате появится окно, показанное ниже (рис. 9).

Рис. 9. Основное окно программы

2. Введите число, до которого вы хотите найти все простые числа. В результате появится окно, показанное ниже (рис. 10).

Рис. 10. Вывод простых чисел

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Битовые поля

Битовые поля представляют из себя характеристические массивы,где индексы каждого элемента – это элементы множества. Каждое битовое поле задаётся длиной (унивёрс битов),количеством единиц памяти(кол-во характеристических массивов) и памятью для их хранения. Элемент битового поля может находиться в двух состояниях: 1 и 0. 1- элемент содержится в множестве, а 0 – элемент не содержится в множестве. Данный алгоритм позволяет реализовать интерфейс для работы с множествами.

Операция установки бита

Исходное поле:

1	1	0	0	0	1	0	1

В результате установки бита на позицию 3 получается поле:

1	1	0	1	0	1	0	1

Операция очищения бита

Исходное поле:

1	1	0	1	1	0	1	0

В результате очищения бита на позиции 4 получится поле:

1	1	0	1	0	0	1	0

Операция получения длины битового поля

1	0	1	1	1	0	1	0	0	0

Результат:

10 – длина битового поля.

Операция получения значения

1	0	1	1	1	0	1	0

Результат получения значения бита на позиции 3:

1 – значение бита на позиции 3.

Результат получения значения бита на позиции 1:

0 – значение бита на позиции 1.

Операция объединения

1	1	0	1	0	0	0	1
0	1	0	1	1	1	0	0

Результат:

1	1	0	1	1	1	0	1

Операция пересечения

1	1	0	1	0	1	0	1
0	1	0	1	1	1	0	0

Результат:

0	1	0	1	0	1	0	0

Операция дополнения

1	0	1	1	0	1	0	0

Результат:

0	1	0	0	1	0	1	1

Операция сравнения на равенство

0	1	1	1	0	1	1	0
0	1	1	0	1	0	1	1

Результат:

0 – поля не равны.

0	1	1	0	1	1	0	0
0	1	1	0	1	1	0	0

Результат:

1 – поля равны.

Операция сравнения на неравенство

1	0	1	1	0	1	1	0
1	0	1	1	0	1	0	1

Результат:

1 – поля не равны.

1	1	0	1	1	0	0	1
1	1	0	1	1	0	0	1

Результат:

0 – поля равны.

Операция присваивания

1	1	1	0	1	0	1	1
1	0	0	1	0	0	1	1

В результате присваивания первого (верхнего) поля второму (нижнему) полю, получится следующее битовое поле:

1	1	1	0	1	0	1	1

3.1.2 Множества

Множество - это класс TSet, который реализован на основе класса TBitField. Класс TSet используется для создания множеств и выполнения теоретико-множественных операций с помощью класса TBitField. Максимальная мощность множества определяется длиной битового поля.

3.1.3 «Решето Эратосфена»

Решето Эратосфена — алгоритм нахождения всех простых чисел до некоторого целого числа n, который приписывают древнегреческому математику Эратосфену Киренскому. Как и во многих случаях, здесь название алгоритма говорит о принципе его работы, то есть решето подразумевает фильтрацию, в данном случае фильтрацию всех чисел за исключением простых. По мере прохождения списка нужные числа остаются, а ненужные (они называются составными) исключаются.

Для нахождения всех простых чисел не больше заданного числа n, следуя методу Эратосфена, нужно выполнить следующие шаги:

1. Выписать подряд все целые числа от двух до n (2, 3, 4, …, n).
2. Пусть переменная p изначально равна двум — первому простому числу.
3. Зачеркнуть в списке числа от 2p до n, считая шагами по p (это будут числа, кратные p: 2p, 3p, 4p, …).

4. Найти первое не зачёркнутое число в списке, большее чем p, и присвоить значению переменной p это число.
5. Повторять шаги 3 и 4, пока возможно.

Теперь все не зачёркнутые числа в списке — это все простые числа от 2 до n.

3.2 Описание программной реализации

3.2.1 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: **n** – номер бита.

Выходные параметры: номер элемента в памяти.

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

Назначение: получение битовой маски для бита.

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

Выходные параметры: битовое поле.

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

Назначение: создание объекта класса **TBitField**.

Входные параметры: **len** – длина битового поля.

Выходные параметры: битовое поле.

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

Назначение: копирование битового поля.

Входные параметры: **bf** – ссылка на объект **TBitField**, который нужно скопировать.

Выходные параметры: отсутствуют.

```
~TBitField();
```

Назначение: освобождение памяти из под битового поля.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

```
int GetLength(void) const;
```

Назначение: получение длины битового поля.

Входные параметры: отсутствуют.

Выходные параметры: длина битового поля.

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

Назначение: установление бита в значение '1'.

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

Выходные параметры: отсутствуют.

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

Назначение: установление бита в значение '0'.

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

Выходные параметры: отсутствуют.

```
int  GetBit(const int n) const;
```

Назначение: получение значения бита.

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

Выходные параметры: значение бита.

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

Назначение: проверяет, равны ли два объекта **TBitField**.

Входные параметры: **bf** – ссылка на объект **TBitField**, с которым нужно сравнить.

Выходные параметры: 1 – равны, 0 – не равны.

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

Назначение: проверяет, не равны ли два объекта **TBitField**.

Входные параметры: **bf** – ссылка на объект **TBitField**, с которым нужно сравнить.

Выходные параметры: 0 – равны, 1 – не равны.

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

Назначение: присваивание битового поля.

Входные параметры: **bf** – ссылка на объект **TBitField**, который нужно присвоить.

Выходные параметры: ссылка на текущий объект класса **TBitField**.

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

Назначение: выполнение операции "или" для двух битовых полей.

Входные параметры: **bf**– ссылка на объект **TBitField**.

Выходные параметры: новый объект класса **TBitField**.

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

Назначение: выполнение операции "и" для двух битовых полей.

Входные параметры: **bf** – ссылка на объект **TBitField**.

Выходные параметры: новый объект класса **TBitField**.

`TBitField  operator~(void);`

Назначение: выполнение операции "отрицание" для битового поля.

Входные параметры: отсутствуют.

Выходные параметры: новый объект класса **`TBitField`**.

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

Назначение: оператор ввода для класса **`TBitField`**.

Входные параметры:

`istr` – ссылка на объект типа **`istream`**, который представляет входной поток.

`bf` – ссылка на объект типа **`TBitField`**, в который будут считываться значения.

Выходные параметры: ссылка на объект типа **`istream`**.

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

Назначение: оператор вывода для класса **`TBitField`**.

Входные параметры:

`ostr` – ссылка на объект типа ostream, который представляет выходной поток.

`bf` – ссылка на объект типа **`TBitField`**, который будет выводиться.

Выходные параметры: ссылка на объект типа **`ostream`**.

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;
  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 istream &operator>>(istream &istr, TSet &bf);
  friend ostream &operator<<(ostream &ostr, const TSet &bf);
};
```

Назначение: представление множества

Поля:

MaxPower – максимальная мощность множества.

BitField - битовое поле для хранения характеристического вектора

Методы:

TSet(int mp);

Назначение: создает объект **TSet** с заданной максимальной мощностью.

Входные параметры: **mp** - максимальная мощность множества.

Выходные параметры: созданный объект **TSet.**

TSet(const TSet &s);

Назначение: создает копию объекта **TSet**.

Входные параметры: **s** - объект **TSet**, который нужно скопировать.

Выходные параметры: отсутствуют.

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

Назначение: создает объект **TSet** на основе заданного битового поля.

Входные параметры: **bf** – ссылка на объект **TBitField**.

Выходные параметры: созданный объект **TSet** с характеристическим вектором, соответствующим битовому полю **bf**.

```
operator TBitField();
```

Назначение: преобразует объект **TSet** в битовое поле.

Входные параметры: отсутствуют.

Выходные параметры: битовое поле, соответствующее характеристическому вектору множества.

```
int GetMaxPower(void) const;
```

Назначение: возвращает максимальную мощность множества.

Входные параметры: отсутствуют.

Выходные параметры: максимальная мощность множества.

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

Назначение: включает элемент в множество.

Входные параметры: **Elem** - элемент, который нужно включить в множество.

Выходные параметры: отсутствуют.

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

Назначение: удаляет элемент из множества.

Входные параметры: **Elem** - элемент, который нужно удалить из множества.

Выходные параметры: отсутствуют.

```
int IsMember(const int Elem) const;
```

Назначение: проверяет наличие элемента в множестве.

Входные параметры: **Elem** - элемент, который нужно проверить на принадлежность множеству.

Выходные параметры: 1, если элемент принадлежит множеству, иначе 0.

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

Назначение: проверяет, равны ли два объекта **TSet**.

Входные параметры: **s** – ссылка на объект **TSet**.

Выходные параметры: 1 – равны, 0 – не равны.

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

Назначение: проверяет, не равны ли два объекта **TSet**.

Входные параметры: **s** – ссылка на объект **TSet**.

Выходные параметры: 0 – равны, 1 – не равны.

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

Назначение: присваивание множества.

Входные параметры: **s** – ссылка на объект **TSet**.

Выходные параметры: ссылка на текущий объект класса **TSet**.

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

Назначение: возвращает объединение множества с заданным элементом.

Входные параметры: **Elem** - элемент, который нужно добавить в множество.

Выходные параметры: новый объект **TSet**, являющийся объединением исходного множества с элементом **Elem**.

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

Назначение: возвращает разность множества с заданным элементом.

Входные параметры: **Elem** - элемент, который нужно удалить из множество.

Выходные параметры: новый объект **TSet**, являющийся разностью исходного множества с элементом **Elem**.

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

Назначение: выполнение операции "или" для двух множеств.

Входные параметры: **s** – ссылка на объект **TSet**.

Выходные параметры: новый объект класса **TSet**.

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

Назначение: выполнение операции "и" для двух множеств.

Входные параметры: **s** – ссылка на объект **TSet**.

Выходные параметры: новый объект класса **TSet**.

```
TSet operator~ (void);
```

Назначение: выполнение операции "отрицание" для множества.

Входные параметры: отсутствуют.

Выходные параметры: новый объект класса **TSet**.

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

Назначение: оператор ввода для класса **`TSet`**.

Входные параметры:

`istr` – ссылка на объект типа istream, который представляет входной поток.

`s`– ссылка на объект типа **`TSet`**, в который будут считываться значения.

Выходные параметры: ссылка на объект типа **`istream`**.

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

Назначение: оператор вывода для класса **`TSet`**.

Входные параметры:

`ostr` – ссылка на объект типа ostream, который представляет выходной поток.

`s` – ссылка на объект типа **`TSet`**, который будет выводиться.

Выходные параметры: ссылка на объект типа **`ostream`**.

Заключение

В результате разработки классов TSet и TBitField была создана структура данных, позволяющая эффективно работать с множествами и битовыми полями.

Класс TSet предоставляет возможность работать с множествами, заданными максимальной мощностью и характеристическим вектором в виде битового поля. В рамках этого класса были реализованы операции, такие как добавление элемента в множество, удаление элемента из множества, проверка наличия элемента в множестве, сравнение множеств, присваивание множеств, объединение множеств, разность множеств, пересечение множеств и операция дополнения множества. Класс TSet также предоставляет методы для получения максимальной мощности множества и преобразование типа к битовому полю.

Класс TBitField служит для работы с битовыми полями и предоставляет методы для доступа, установки и очистки битов, а также операции сравнения, присваивания, логического "или", логического "и" и операцию отрицания. Класс TBitField также имеет методы для получения длины битового поля и индексации элементов в массиве памяти.

В целом, создание и реализация этих классов позволяет эффективно работать с множествами и битовыми полями, выполнять различные операции с ними и получать нужные результаты.

Литература

1. Битовые поля [https://qaa-engineer.ru/bitovye-polya].
2. Решето Эратосфена [https://ru.wikipedia.org/wiki/Решето_Эратосфена].

Приложения

Приложение А. Реализация класса TBitField

```
TBitField::TBitField(int len)
{
    if (len > 0)
    {
        BitLen = len;
        MemLen = (len + sizeof(TELEM) * 8 - 1) / (sizeof(TELEM) * 8);
        pMem = new TELEM[MemLen];
        for (int i = 0; i < MemLen; i++)
            pMem[i] = 0;
    }
    else if (len == 0)
    {
        BitLen = 0;
        MemLen = 0;
        pMem = nullptr;
    }
    else
        throw "Size error";
}

TBitField::TBitField(const TBitField &bf)
{
    BitLen = bf.BitLen;
    MemLen = bf.MemLen;
    if (MemLen)
    {
        pMem = new TELEM[MemLen];
        for (int i = 0; i < MemLen; i++)
            pMem[i] = bf.pMem[i];
    }
    else
        pMem = nullptr;
}

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

int TBitField::GetMemIndex(const int n) const
{
    return n / (sizeof(TELEM) * 8);}

TELEM TBitField::GetMemMask(const int n) const
{
    return 1 << (n % (sizeof(TELEM) * 8));
}

// доступ к битам битового поля

int TBitField::GetLength(void) const
{
    return BitLen;
}
```

```
void TBitField::SetBit(const int n)
{
    if (n >= 0 && n < BitLen)
        pMem[GetMemIndex(n)] |= GetMemMask(n);
    else
        throw "Error";
}

void TBitField::ClrBit(const int n)
{
    if (n >= 0 && n < BitLen)
        pMem[GetMemIndex(n)] &= ~GetMemMask(n);
    else
        throw "Error";
}

int TBitField::GetBit(const int n) const
{
    if (n >= 0 && n < BitLen)
        return (pMem[GetMemIndex(n)] & GetMemMask(n)) != 0;
    else
        throw "Error";
}

TBitField& TBitField::operator=(const TBitField &bf)
{
    if (this != &bf)
    {
        if (MemLen != bf.MemLen)
        {
            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;
}

int TBitField::operator==(const TBitField &bf) const
{
    if (BitLen != bf.BitLen)
        return 0;
    for (int i = 0; i < MemLen; i++)
    {
        if (pMem[i] != bf.pMem[i])
            return 0;
    }
    return 1;
}

int TBitField::operator!=(const TBitField &bf) const
{
    return !(*this == bf);
}
```

```
TBitField TBitField::operator|(const TBitField &bf)
{
    int len = BitLen;
    if (bf.BitLen > len)
        len = bf.BitLen;
    TBitField result(len);
    for (int i = 0; i < MemLen; i++)
        result.pMem[i] = pMem[i] | bf.pMem[i];
    return result;
}

TBitField TBitField::operator&(const TBitField &bf)
{
    int len = BitLen;
    if (bf.BitLen > len)
        len = bf.BitLen;
    TBitField result(len);
    for (int i = 0; i < MemLen; i++)
        result.pMem[i] = pMem[i] & bf.pMem[i];
    return result;
}

TBitField TBitField::operator~(void) /
{
    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)
{
    string str;
    istr >> str;
    for (int i = 0; i < bf.BitLen; i++)
    {
        int bit = str[i] - '0';
        if (bit)
            bf.SetBit(i);
        else
            bf.ClrBit(i);
    }
    return istr;
}

ostream &operator<<(ostream &ostr, const TBitField &bf)
{
    //for (int i = bf.BitLen - 1; i >= 0; i--)
    for (int i = 0; i < bf.BitLen; i++)
        ostr << bf.GetBit(i);
    return ostr;
}
```

Приложение Б. Реализация класса TSet

```
TSet::TSet(int mp) : BitField(mp)
{
    if (mp >= 0)
        MaxPower = mp;
    else
        throw "Size error";
}

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

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

TSet::operator TBitField()
{
    return BitField;
}

int TSet::GetMaxPower(void) const
{
    return MaxPower;
}

int TSet::IsMember(const int Elem) const
{
    if (Elem >= MaxPower || Elem < 0)
        throw "Error";
    return BitField.GetBit(Elem);
}

void TSet::InsElem(const int Elem)
{
    if (Elem >= MaxPower || Elem < 0)
        throw "Error";
    return BitField.SetBit(Elem);
}

void TSet::DelElem(const int Elem)
{
    if (Elem >= MaxPower || Elem < 0)
        throw "Error";
    return BitField.ClrBit(Elem);
}


TSet& TSet::operator=(const TSet &s)
{
    if (this != &s)
    {
        MaxPower = s.MaxPower;
        BitField = s.BitField;
    }
    return *this;
}
```

```
int TSet::operator==(const TSet &s) const
{
    return BitField == s.BitField;;
}

int TSet::operator!=(const TSet &s) const
{
    return BitField != s.BitField;
}

TSet TSet::operator+(const TSet &s)
{
    size_t newMaxPower;
    if (MaxPower > s.MaxPower)
        newMaxPower = MaxPower;
    else
        newMaxPower = s.MaxPower;
    TSet result(newMaxPower);
    result.BitField = BitField | s.BitField;
    return result;
}

TSet TSet::operator+(const int Elem)
{
    if (Elem >= MaxPower || Elem < 0)
        throw "Error";
    TSet temp(*this);
    temp.BitField.SetBit(Elem);
    return temp;
}

TSet TSet::operator-(const int Elem)
{
    if (Elem >= MaxPower || Elem < 0)
        throw "Error";
    TSet tmp(*this);
    tmp.BitField.ClrBit(Elem);
    return tmp;
}

TSet TSet::operator*(const TSet &s)
{
    size_t newMaxPower;
    if (MaxPower > s.MaxPower)
        newMaxPower = MaxPower;
    else
        newMaxPower = s.MaxPower;
    TSet result(newMaxPower);
    result.BitField = BitField & s.BitField;
    return result;
}

TSet TSet::operator~(void)
{
    TSet result(MaxPower);
    result.BitField = ~BitField;
    return result;
}
```

```
istream& operator>>(istream& istr, TSet& s)
{
    int elem, n = 0;
    do
    {
        cout << "Enter the number of digits you want to enter: ";
        istr >> n;
    } while (n <= 0 || n > s.MaxPower);
    for (int i = 0; i < n; i++)
    {
        istr >> elem;
        if (elem > 0 || elem <= s.MaxPower)
            s.InsElem(elem);
        else
            throw "error";
    }
    return istr;
}
ostream& operator<<(ostream &ostr, const TSet &s)
{
    ostr << s.BitField;
    return ostr;
}
```