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
3822Б1ФИ1
_____ / Созонов И.С. /
Подпись

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

Нижний Новгород
2023

Содержание

Введение

Под множеством математики понимают соединение каких-либо объектов в одно целое. Создатель теории множеств немецкий математик Георг Кантор (1845-1918) определил множество как «объединение в одно целое объектов, хорошо различаемых нашей интуицией или нашей мыслью». Он же сформулировал это короче: «множество – это многое, мыслимое нами как единое». На самом деле ни одна из этих фраз не является определением в строгом математическом понимании. Понятие множества вообще не определяется, это одно из первичных понятий математики. Его можно пояснить, приводя более или менее близкие по смыслу слова: коллекция, класс, совокупность, ансамбль, собрание, или примеры: экипаж корабля – множество людей, стая – множество птиц, созвездие – множество звезд. Множества, рассматриваемые в математике, состоят из математических объектов (чисел, функций, точек, линий и т.д.). Объекты, из которых состоит множество, называют его элементами. Важно отметить, что в множестве все элементы отличаются друг от друга, одинаковых элементов быть не может.

Влияние теории множеств на развитие современной математики очень велико. Прежде всего, теория множеств явилась фундаментом ряда новых математических дисциплин (теории функций действительного переменного, общей топологии, общей алгебры, функционального анализа и др.). Постепенно теоретико-множественные методы находят всё большее применение и в классических частях математики. Например, в области математического анализа они широко применяются в качественной теории дифференциальных уравнений, вариационном исчислении, теории вероятностей и др.

Активное применение аппарата теории множеств в современной науке приводит к необходимости создания соответствующих программных решений. Вместе с тем лишь в отдельных языках программирования предусмотрены встроенные средства для работы с множествами. Так, в стандартной библиотеке C++ есть класс std::set, а в C++11 добавлен класс std::unordered_set.

Чтобы определить множество, достаточно указать характеристическое свойство его элементов, т.е. такое свойство, которым обладают все элементы этого множества и только они. Наиболее эффективный способ представления характеристического вектора – битовое поле (битовая строка). Реализацию битового поля целесообразно вынести в отдельный класс, скрывающий детали, не существенные для представления и работы с множествами.

Данная лабораторная работа посвящена изучению реализации множеств на основе битовых полей.

1 Постановка задачи

Цель – реализовать классы для представления битовых полей TBitField и множеств TSet.

Задачи:

1. Разработать класс TBitField для работы с битовыми полями. Написать следующие операции для работы с битовыми полями: добавить элемент в битовое поле (установить бит в 1), очистить элемент (установить бит в 0), получить значение бита, сравнить два битовых поля, логические «ИЛИ», «И», «НЕ» (| , & , ~ соответственно), ввести битовое поле и вывести битовое поле требуемого формата. Добавить вспомогательные операции получения бита, битовой маски и длины битового поля.
2. Разработать класс TSet для работы с множествами. Написать следующие операции для работы с множествами: включение элемента в множество, исключение элемента из множества, проверка наличия элемента в множестве, сравнение множеств, объединение множеств, пересечение множеств, разность множеств, копирование множества, вычисление максимальной мощности множества, ввод элементов множества и вывод элементов множества требуемого формата.

2 Руководство пользователя

2.1 Приложение для демонстрации работы битовых полей

1. Запустить sample_tbitfield.exe. В результате появится окно для ввода длины битового поля (рис. 1).

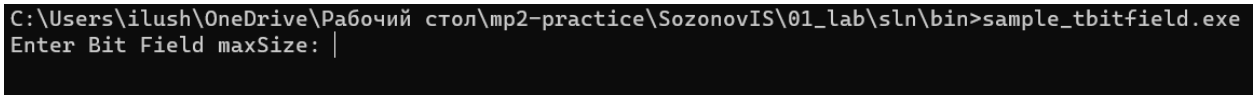

Рис. 1. Основное окно приложения

2. Ввести длину битового поля. В результате появится окно для ввода первой битовой строки (Рис. 2).

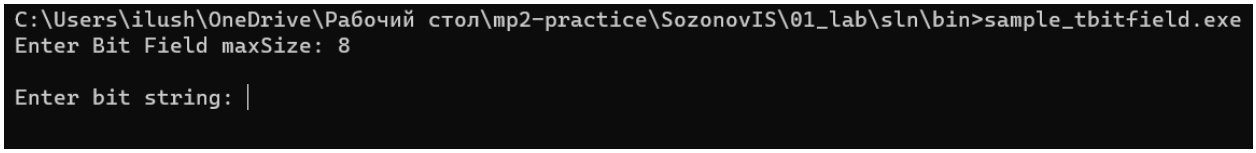

Рис. 2. Ввод длины битового поля

3. Ввести первую битовую строку. В результате появится окно для ввода второй битовой строки (Рис. 3).

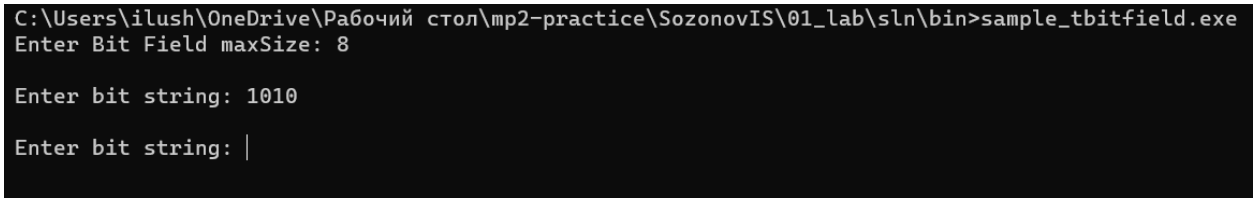

Рис. 3. Ввод первой битовой строки

4. Ввести вторую битовую строку. В результате будут выведены битовые поля, результаты их сравнения, объединения и пересечения, а также дополнение к первому битовому полю. Появится окно для ввода добавляемого бита (Рис. 4).

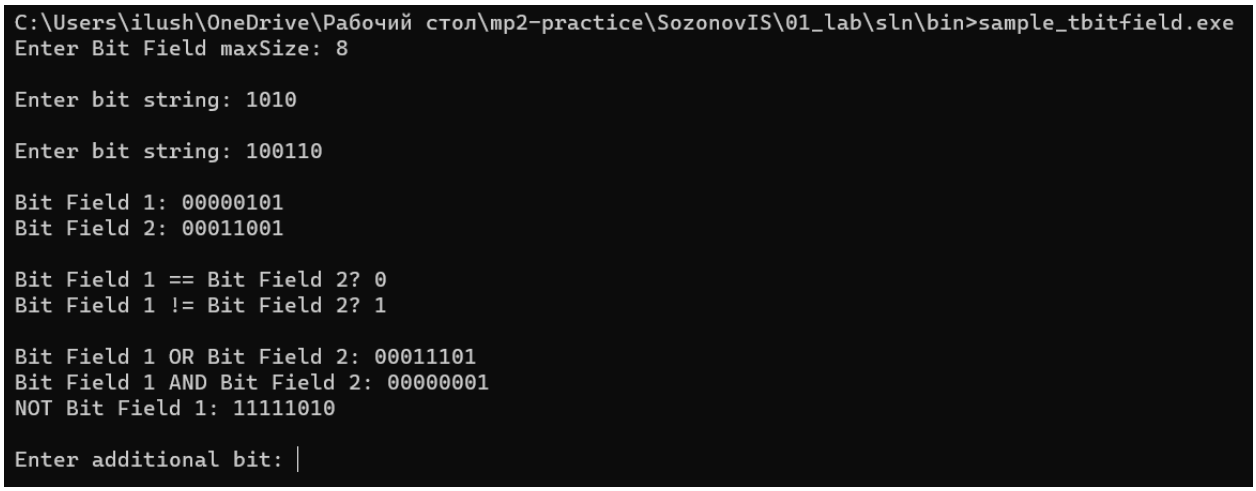

Рис. 4. Ввод второй битовой строки

5. Ввести добавляемый бит. В результате будет выведено первое битовое поле с добавленным битом. Появится окно для ввода исключаемого бита (Рис. 5).

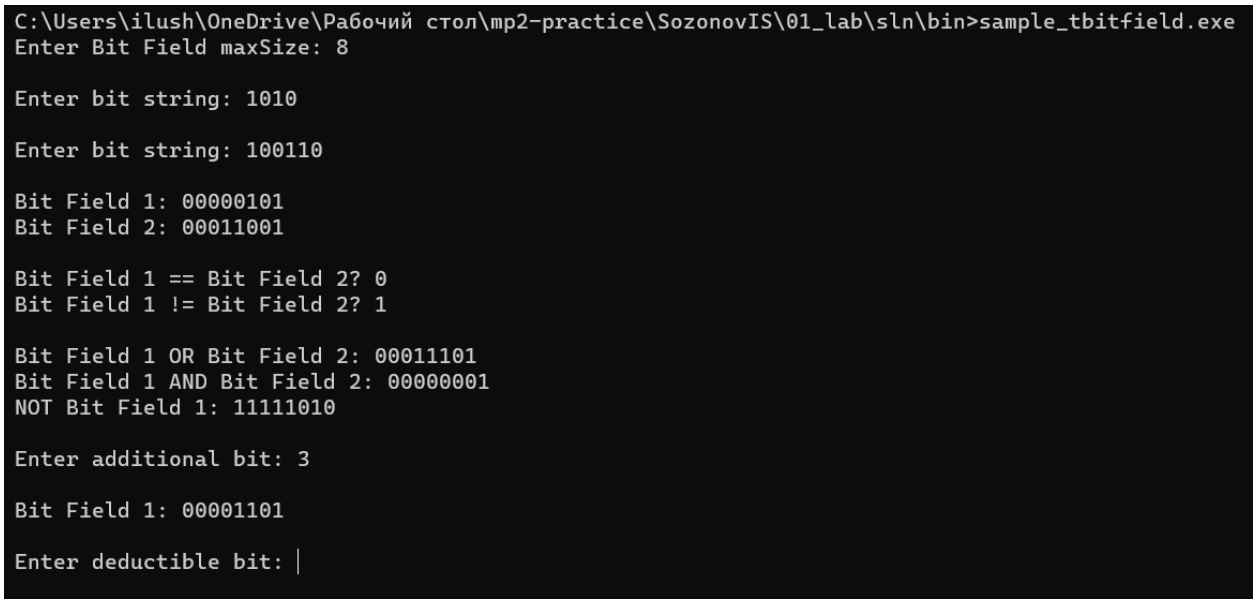

Рис. 5. Ввод добавляемого бита

6. Ввести исключаемый бит. В результате будет выведено второе битовое поле с исключенным битом (Рис. 6).

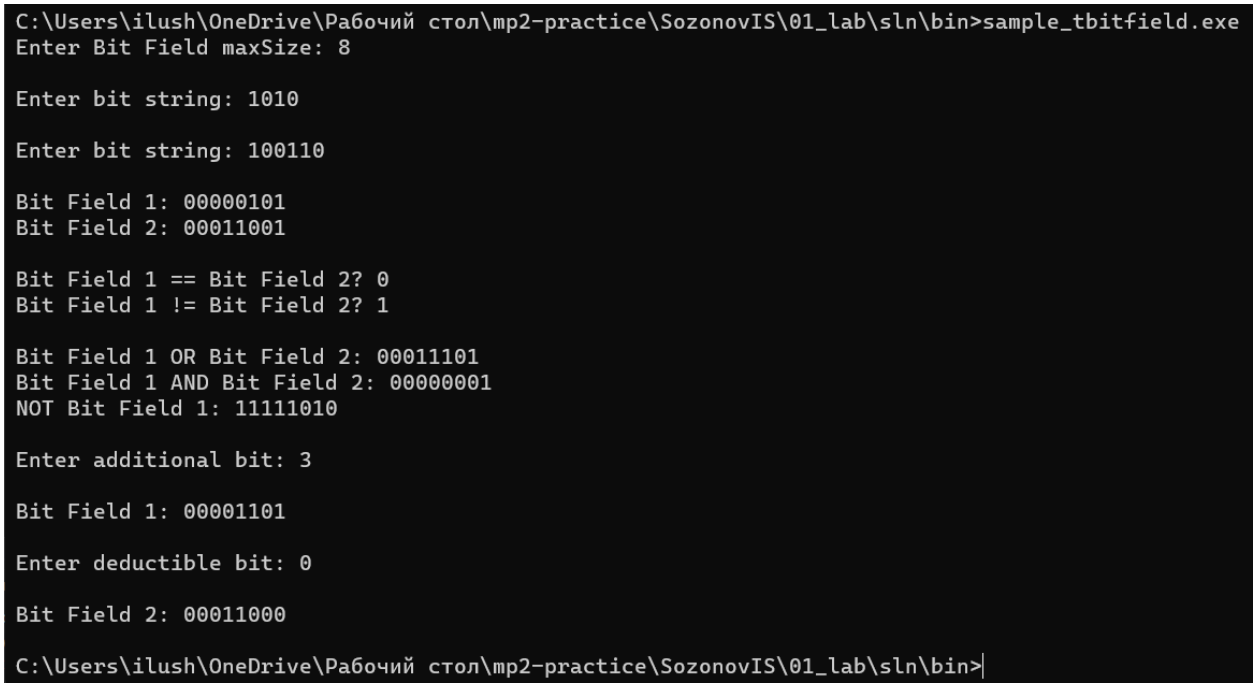

Рис. 6. Ввод исключаемого бита

2.2 Приложение для демонстрации работы множеств

1. Запустить sample_tset.exe. В результате появится окно для ввода максимальной мощности множества (Рис. 7).

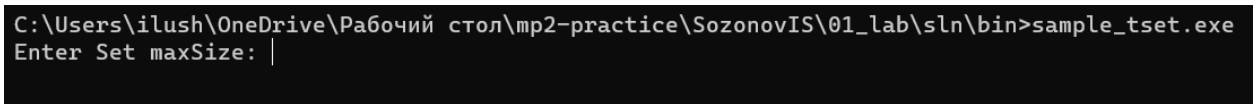

Рис. 7. Основное окно приложения

2. Ввести максимальную мощность множества. В результате появится окно для ввода мощности первого множества (**Ошибка! Источник ссылки не найден.**).

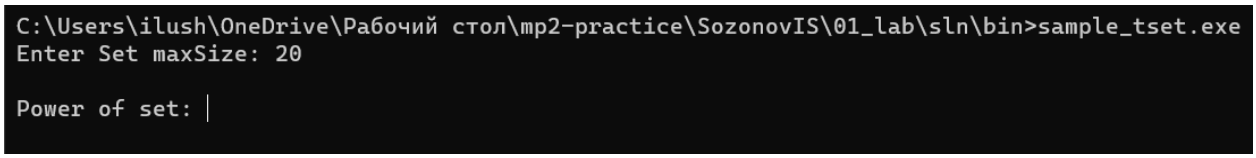

Рис. 8. Ввод максимальной мощности множества

3. Ввести мощность первого множества. В результате появится окно для ввода элементов первого множества (**Ошибка! Источник ссылки не найден.**).

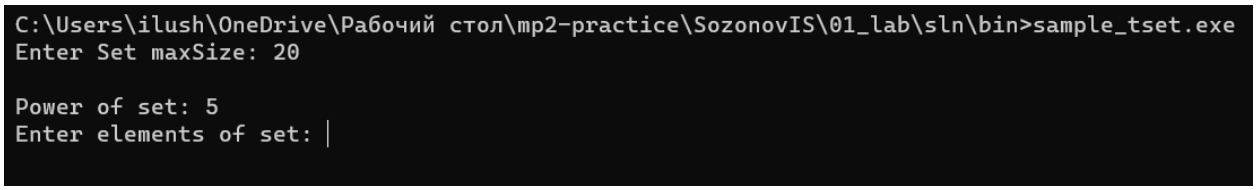

Рис. 9. Ввод мощности первого множества

4. Ввести элементы первого множества. В результате появится окно для ввода мощности второго множества (**Ошибка! Источник ссылки не найден.**).

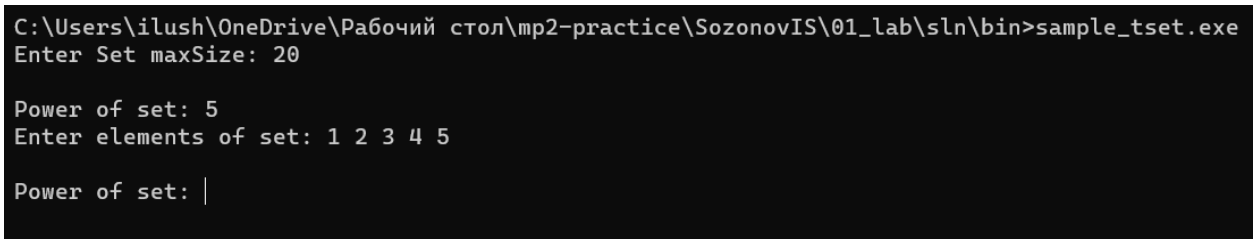

Рис. 10. Ввод элементов первого множества

5. Ввести мощность второго множества. В результате появится окно для ввода элементов второго множества (**Ошибка! Источник ссылки не найден.**).

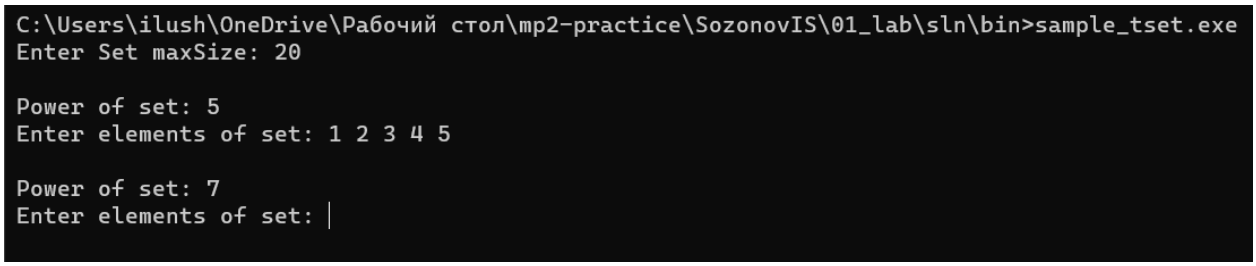

Рис. 11. Ввод мощности второго множества

6. Ввести элементы второго множества. В результате будут выведены множества и результаты их сравнения. Появится окно для ввода случайного элемента (Рис. 12).

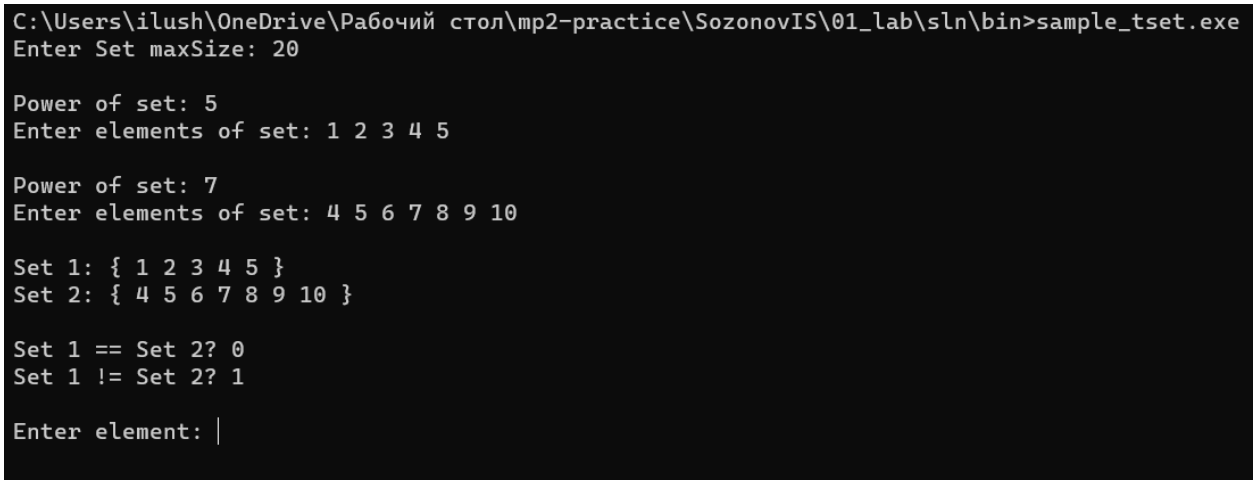

Рис. 12. Ввод элементов второго множества.

7. Ввести случайный элемент. В результате будут выведены результаты сложения первого множества с введенным элементом, разности второго множества с введенным элементом, а также результаты объединения множеств, их пересечения и дополнения к первому множеству. Появится окно для ввода элемента, включаемого в множество (**Ошибка! Источник ссылки не найден.**).

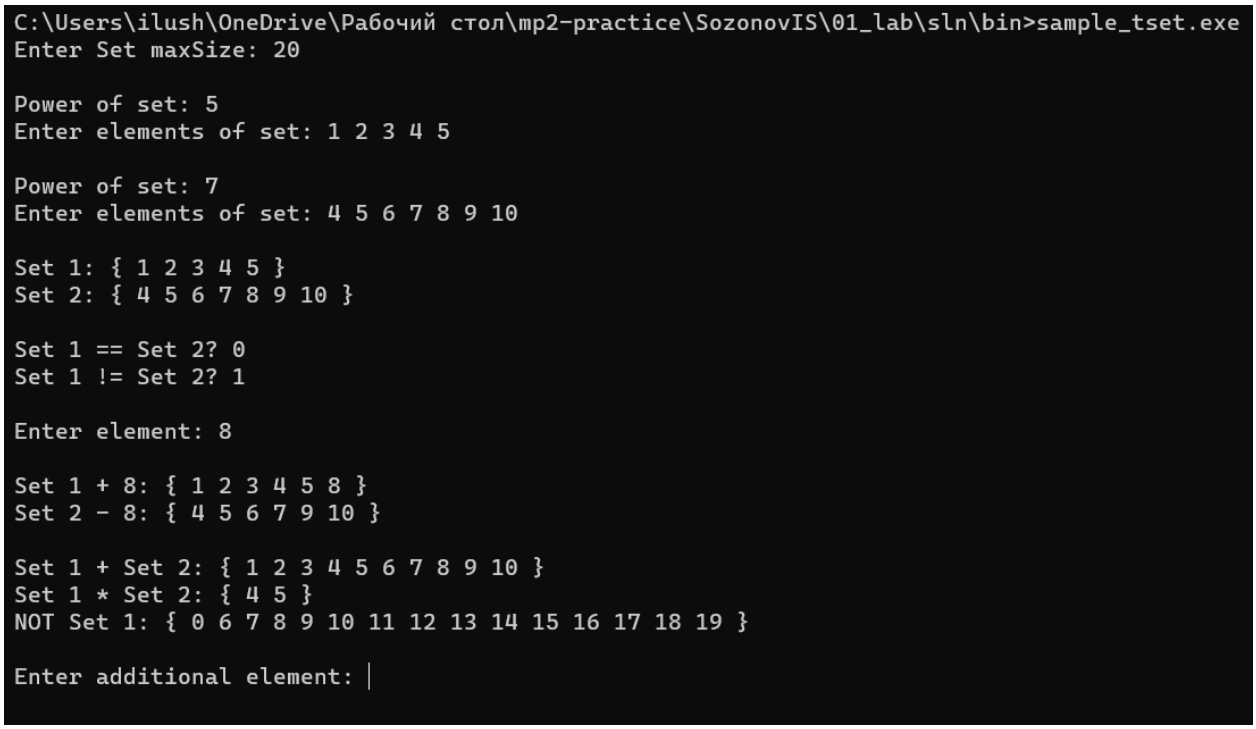

Рис. 13. Ввод случайного элемента

8. Ввести элемент, включаемый в множество. В результате будет выведен результат включения введенного элемента в первое множество. Появится окно для ввода элемента, удаляемого из множества (**Ошибка! Источник ссылки не найден.**).

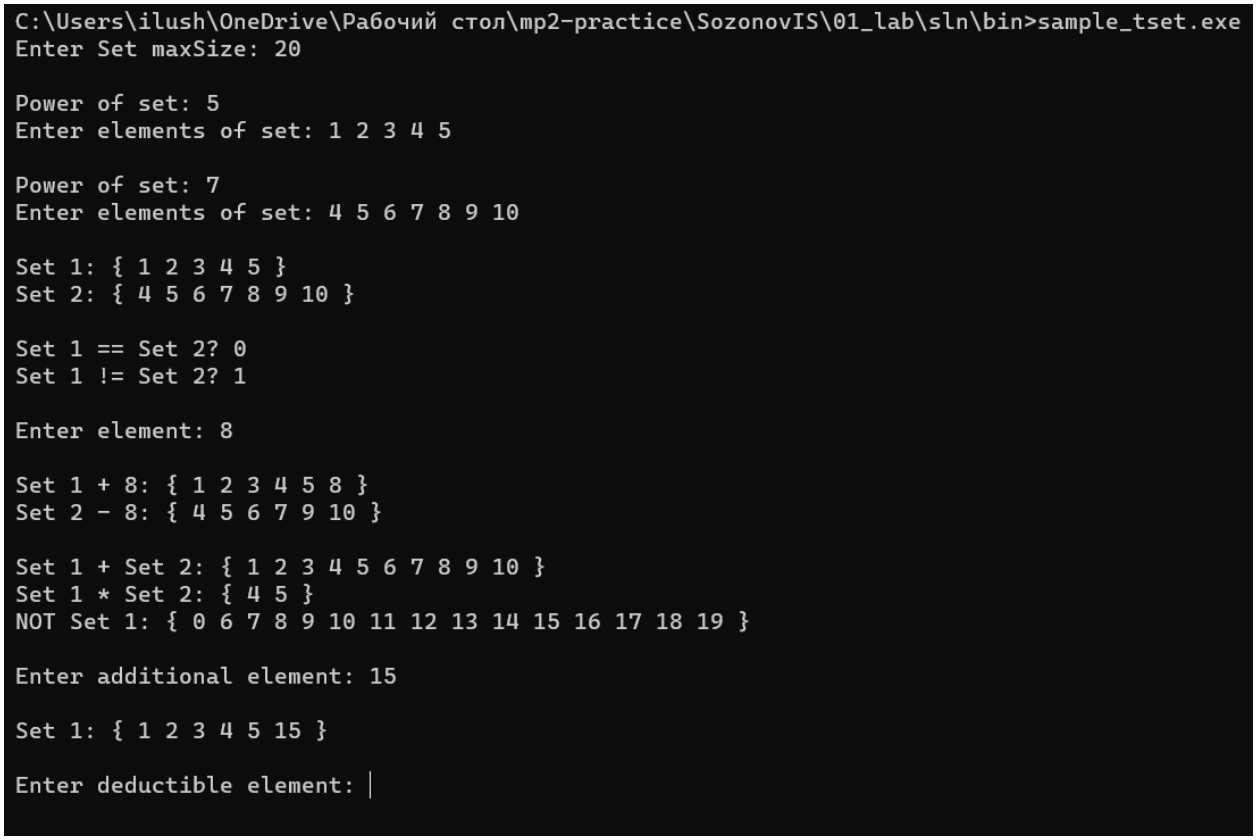

Рис. 14. Ввод элемента, включаемого в множество

9. Ввести элемент, удаляемый из множества. В результате будет выведен результат удаления введенного элемента из второго множества (Рис. 15).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\01_lab\sln\bin>sample_tset.exe
Enter Set maxSize: 20

Power of set: 5
Enter elements of set: 1 2 3 4 5

Power of set: 7
Enter elements of set: 4 5 6 7 8 9 10

Set 1: { 1 2 3 4 5 }
Set 2: { 4 5 6 7 8 9 10 }

Set 1 == Set 2? 0
Set 1 != Set 2? 1

Enter element: 8

Set 1 + 8: { 1 2 3 4 5 8 }
Set 2 - 8: { 4 5 6 7 9 10 }

Set 1 + Set 2: { 1 2 3 4 5 6 7 8 9 10 }
Set 1 * Set 2: { 4 5 }
NOT Set 1: { 0 6 7 8 9 10 11 12 13 14 15 16 17 18 19 }

Enter additional element: 15

Set 1: { 1 2 3 4 5 15 }

Enter deductible element: 6

Set 2: { 4 5 7 8 9 10 }

C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\01_lab\sln\bin>
```

Рис. 15. Основное окно приложения

2.3 Приложение «решето Эратосфена»

1. Запустить sample_tbitfield.exe. В результате появится окно для ввода верхней границы для вывода простых чисел (Рис. 16).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\01_lab\sln\bin>sample_primenumbers.exe
Enter the maximum integer:
```

Рис. 16. Основное окно приложения

2. Ввести верхнюю границу. В результате будут выведены все простые числа до введенного числа (**Ошибка! Источник ссылки не найден.**):

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\01_lab\sln\bin>sample_primenumbers.exe
Enter the maximum integer: 100

Prime numbers: { 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 }

C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\01_lab\sln\bin>
```

Рис. 17. Ввод верхней границы

3 Руководство программиста

3.1 Использованные алгоритмы

3.1.1 Битовые поля

Битовое поле – некоторое количество бит, расположенных последовательно в памяти.

Битовые поля обеспечивают удобный доступ к отдельным битам данных. Они позволяют формировать объекты с длиной, не кратной байту, что в свою очередь позволяет экономить память, более плотно размещая данные.

A	0	0	0	0	0	1	0	1
B	0	0	0	1	1	0	0	1

Для работы с битовыми полями реализуются следующие операции:

- Сравнение двух битовых полей: результат операции равен 1, если каждый бит первого поля равен соответствующему биту второго поля, иначе 0.
- Логическое «ИЛИ» для двух битовых полей: результат операции равен 1, если один из соответствующих битов равен 1, иначе 0.

A	0	0	0	0	0	1	0	1
B	0	0	0	1	1	0	0	1
A | B	0	0	0	1	1	1	0	1

- Логическое «И» для двух битовых полей: результат операции равен 1, если оба из соответствующих битов равен 1, иначе 0.

A	0	0	0	0	0	1	0	1
B	0	0	0	1	1	0	0	1
A & B	0	0	0	0	0	0	0	1

- Логическое «НЕ» для битового поля: изменяет значение соответствующего бита на противоположное.

A	0	0	0	0	0	1	0	1
~ A	1	1	1	1	1	0	1	0

- Добавить элемент в битовое поле (установить бит в 1): нужно создать маску, в которой бит в данной позиции должен быть установлен в единицу путем операции побитового сдвига, а остальные в – нули, и объединить ее с битовым полем с помощью операции логическое «ИЛИ».

A	0	0	0	0	0	1	0	1
mask = 1 << i	0	0	0	0	1	0	0	0
A | mask	0	0	0	0	1	1	0	1

- Очистить элемент (установить бит в 0): нужно создать маску, в которой бит в данной позиции должен быть установлен в единицу путем операции побитового сдвига, а остальные в – нули, получить логическое «НЕ» к полученной маске и объединить ее с битовым полем с помощью операции логическое «И».

mask = 1 << i	0	0	0	0	0	0	0	1
~ mask	1	1	1	1	1	1	1	0

B	0	0	0	1	1	0	0	1
~ mask	1	1	1	1	1	1	1	0
B & (~ mask)	0	0	0	1	1	0	0	0

- Получить значение бита: нужно создать маску, в которой бит в данной позиции должен быть установлен в единицу путем операции побитового сдвига, а остальные в – нули, и объединить ее с битовым полем с помощью операции логическое «И».

A	0	0	0	0	0	x	0	1
mask = 1 << i	0	0	0	0	0	1	0	0
A & mask	0	0	0	0	0	x	0	0

Если результат равен 0, то x также равен 0, иначе x = 1.

3.1.2 Множества

Множество – это совокупность элементов, объединенных на основе общих свойств или признаков. Чтобы определить множество, достаточно указать характеристическое свойство его элементов, т.е. такое свойство, которым обладают все элементы этого множества и только они.

Поскольку не требуется хранить элементы множеств, то любое множество $A \subset U = \{u_1, u_2, \dots, u_n\}$ может быть описано характеристическим вектором:

$$\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n), \text{где } \alpha_i = \begin{cases} 1, \text{если } u_i \in A \\ 0, \text{иначе} \end{cases}$$

Таким образом каждый элемент характеристического вектора принимает значения из множества {0, 1}, поэтому наиболее эффективной (с точки зрения расхода памяти) является его реализация через битовое поле – непрерывный участок памяти (количество бит в котором достаточно для представления универса), где каждый бит соответствует одному элементу вектора.

A = {1, 2, 3, 4, 5}

A	0	0	0	0	0	1	1	1	1	1	0

B ={4, 5, 6, 7, 8, 9, 10}

B	1	1	1	1	1	1	1	0	0	0	0

Для работы с множествами реализуются следующие операции:

- Сравнение множеств: множества равны, если элементы множеств совпадают. Операция основана на сравнении соответствующих битовых полей данных множеств.
- Объединение множеств: состоит из всех элементов исходных множеств. Операция основана на логическом «ИЛИ» соответствующих битовых полей данных множеств.

A	0	0	0	0	0	1	1	1	1	1	0
B	1	1	1	1	1	1	1	0	0	0	0
A | B	1	1	1	1	1	1	1	1	1	1	0

$A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

- Пересечение множеств: состоит из элементов, которые принадлежат обоим исходным множествам. Если нет элементов, принадлежащих обоим множествам, то пересечение множеств будет пустым. Операция основана на логическом «И» соответствующих битовых полей данных множеств.

A	0	0	0	0	0	1	1	1	1	1	0
B	1	1	1	1	1	1	1	0	0	0	0
A & B	0	0	0	0	0	1	1	0	0	0	0

$A \cap B = \{4, 5\}$

- Включение элемента в множество: операция основана на операции включения бита в битовое поле данного множества.

A	0	0	0	0	0	1	1	1	1	1	0
mask = 1 << i	0	1	0	0	0	0	0	0	0	0	0
A | mask	0	1	0	0	0	1	1	1	1	1	0

A + 9 = {1, 2, 3, 4, 5, 9}

- Исключение элемента из множества: операция основана на операции исключения бита из битового поля данного множества.

mask = 1 << i	0	0	0	0	1	0	0	0	0	0	0
~ mask	1	1	1	1	0	1	1	1	1	1	1

B	1	1	1	1	1	1	1	0	0	0	0
~ mask	1	1	1	1	0	1	1	1	1	1	1
B & (~ mask)	1	1	1	1	0	1	1	0	0	0	0

B - 6 = {4, 5, 7, 8, 9, 10}

- Проверка наличия элемента в множестве: операция основана на операции получения бита из битового поля данного множества.

A	0	0	0	0	0	x	1	1	1	1	0
mask = 1 << i	0	0	0	0	0	1	0	0	0	0	0
A & mask	0	0	0	0	0	x	0	0	0	0	0

Если результат равен 0, то x также равен 0, иначе x = 1.

3.1.3 Алгоритм «решето Эратосфена»

«Решето Эратосфена» – это алгоритм, позволяющий найти все простые числа до заданного числа n.

Целое положительное число называется простым, если оно имеет ровно два различных натуральных делителя – единицу и самого себя. Единица простым числом не считается.

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

3.2 Описание классов

3.2.1 Класс TBitField

Объявление класса:

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

  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;      // получить длину (к-во битов)
  void SetBit(const int n);       // установить бит
  void ClrBit(const int n);       // очистить бит
  int GetBit(const int n) const; // получить значение бита

  // битовые операции
  int operator==(const TBitField &bf) const; // сравнение
  int operator!=(const TBitField &bf) const; // сравнение
  TBitField& operator=(const TBitField &bf); // присваивание
  TBitField operator|(const TBitField &bf); // операция "или"
  TBitField operator&(const TBitField &bf); // операция "и"
  TBitField operator~(void);                // отрицание

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

Поля:

BitLen – длина битового поля.

pMem – указатель типа **TELEM** на первый элемент битового поля.

MemLen – количество элементов для представления битового поля.

Конструкторы:

TBitField(int len);

Назначение: конструктор инициализатор, создание битового поля.

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

Выходные данные: отсутствуют.

TBitField(const TBitField &bf);

Назначение: конструктор копирования, создание копии битового поля.

Входные данные: **`const TBitField &bf`** – константная ссылка на битовое поле.

Выходные данные: отсутствуют.

Деструктор:

```
~TBitField();
```

Назначение: освобождение памяти, занимаемой динамическими полями объекта.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

Методы:

```
int GetMemIndex(const int n) const;
```

Назначение: получение индекса элемента, где хранится бит.

Входные данные: **`n`** – номер бита.

Выходные данные: индекс элемента, где хранится бит с номером n.

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

Назначение: получение битовой маски.

Входные данные: **`n`** – номер бита.

Выходные данные: битовая маска для бита с номером n.

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

Назначение: получение длины битового поля.

Входные данные: отсутствуют.

Выходные данные: длина битового поля.

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

Назначение: добавление элемента в битовое поле.

Входные данные: **`n`** – номер бита.

Выходные данные: отсутствуют.

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

Назначение: исключение бита из битового поля.

Входные данные: **n** – номер бита.

Выходные данные: отсутствуют.

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

Назначение: получение значения бита.

Входные данные: **n** – номер бита.

Выходные данные: значение бита.

Операторы:

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

Назначение: проверка на равенство битовых полей.

Входные данные: **const TBitField &bf** – константная ссылка на битовое поле.

Выходные данные: результат сравнения (1 – битовые поля равны, 0 – битовые поля не равны).

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

Назначение: проверка на неравенство битовых полей.

Входные данные: **const TBitField &bf** – константная ссылка на битовое поле.

Выходные данные: результат сравнения (1 – битовые поля не равны, 0 – битовые поля равны).

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

Назначение: присваивание объекту *this константной ссылки на битовое поле **bf**.

Входные данные: **const TBitField &bf** – константная ссылка на битовое поле.

Выходные данные: ссылка на битовое поле.

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

Назначение: логическое «ИЛИ» битовых полей.

Входные данные: **const TBitField &bf** – константная ссылка на битовое поле.

Выходные данные: битовое поле.

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

Назначение: логическое «И» битовых полей.

Входные данные: **`const TBitField &bf`** – константная ссылка на битовое поле.

Выходные данные: битовое поле.

`TBitField operator~(void);`

Назначение: логическое «НЕ» к битовому полю.

Входные данные: отсутствуют.

Выходные данные: битовое поле.

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

Назначение: ввод битового поля.

Входные данные: **`istream &istr`** – ссылка на стандартный поток ввода,

`TBitField &bf` – ссылка на битовое поле.

Выходные данные: ссылка на поток ввода.

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

Назначение: вывод битового поля.

Входные данные: **`ostream &ostr`** – ссылка на стандартный поток вывода,

`const TBitField &bf` – константная ссылка на битовое поле.

Выходные данные: ссылка на поток вывода.

3.2.2 Класс TSet

Объявление класса:

```
class TSet
{
private:
  int MaxPower;       // максимальная мощность множества
  TBitField BitField; // битовое поле для хранения характеристического вектора
public:
  TSet(int mp);
  TSet(const TSet &s);       // конструктор копирования
```

```
  TSet(const TBitField &bf); // конструктор преобразования типа
  operator TBitField();      // преобразование типа к битовому полю
  // доступ к битам
  int GetMaxPower(void) const;     // максимальная мощность множества
  void InsElem(const int Elem);       // включить элемент в множество
  void DelElem(const int Elem);       // удалить элемент из множества
  int IsMember(const int Elem) const; // проверить наличие элемента в множестве
  // теоретико-множественные операции
  int operator==(const TSet &s) const; // сравнение
  int operator!=(const TSet &s) const; // сравнение
  TSet& operator=(const TSet &s);  // присваивание
  TSet operator+(const int Elem); // объединение с элементом
                                   // элемент должен быть из того же универса
  TSet operator-(const int Elem); // разность с элементом
                                   // элемент должен быть из того же универса
  TSet operator+(const TSet &s);  // объединение
  TSet operator*(const TSet &s);  // пересечение
  TSet operator~(void);           // дополнение

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

Поля:

MaxPower – максимальная мощность множества.

BitField – битовое поле для хранения характеристического вектора.

Конструкторы:

TSet(int mp);

Назначение: конструктор инициализатор, создание множества.

Входные данные: **mp** – максимальная мощность множества.

Выходные данные: отсутствуют.

TSet(const TSet &s);

Назначение: конструктор копирования, создание копии множества.

Входные данные: **const TSet &s** – константная ссылка на множество.

Выходные данные: отсутствуют.

TSet(const TBitField &bf);

Назначение: конструктор преобразования типа, формирование объекта класса **`TSet`** из объекта класса **`TBitField`**.

Входные данные: константная ссылка на битовое поле **`bf`**.

Выходные данные: отсутствуют.

Методы:

`int GetMaxPower(void) const;`

Назначение: получение максимальной мощности множества.

Входные данные: отсутствуют.

Выходные данные: максимальная мощность множества.

`void InsElem(const int Elem);`

Назначение: включение элемента в множество.

Входные данные: **`Elem`** – включаемый элемент.

Выходные данные: отсутствуют.

`void DelElem(const int Elem);`

Назначение: удаление элемента из множества.

Входные данные: **`Elem`** – удаляемый элемент.

Выходные данные: отсутствуют.

`int IsMember(const int Elem) const;`

Назначение: проверка наличия элемента в множестве.

Входные данные: **`Elem`** – проверяемый элемент.

Выходные данные: результат проверки (1 – элемент принадлежит множеству, 0 – элемента не принадлежит множеству).

Операторы:

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

Назначение: проверка на равенство множеств.

Входные данные: **`const TSet &s`** – константная ссылка на множество.

Выходные данные: результат сравнения (1 – битовые поля равны, 0 – битовые поля не равны).

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

Назначение: проверка на неравенство множеств.

Входные данные: **`const TSet &s`** – константная ссылка на множество.

Выходные данные: результат сравнения (1 – битовые поля не равны, 0 – битовые поля равны).

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

Назначение: присваивание объекту *this константной ссылки на множество **`s`**.

Входные данные: **`const TSet &s`** – константная ссылка на множество.

Выходные данные: ссылка на множество.

`TSet operator+(const int Elem);`

Назначение: объединение с элементом.

Входные данные: **`ELem`** – элемент, с которым нужно объединить множество.

Выходные данные: множество.

`TSet operator-(const int Elem);`

Назначение: разность с элементом.

Входные данные: **`ELem`** – элемент, разность множества с которым нужно найти.

Выходные данные: множество.

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

Назначение: объединение множеств.

Входные данные: **`const TSet &s`** – константная ссылка на множество.

Выходные данные: множество.

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

Назначение: пересечение множеств.

Входные данные: **`const TSet &s`** – константная ссылка на множество.

Выходные данные: множество.

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

Назначение: дополнение к множеству.

Входные данные: отсутствуют.

Выходные данные: множество.

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

Назначение: ввод множества.

Входные данные: **`istream &istr`** – ссылка на стандартный поток ввода,

`TSet& s` – ссылка на множество.

Выходные данные: ссылка на поток ввода.

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

Назначение: вывод множества.

Входные данные: **`istream &istr`** – ссылка на стандартный поток вывода,

`const TSet& s` – константная ссылка на множество.

Выходные данные: ссылка на поток вывода.

Заключение

В ходе лабораторной работы были изучены основные термины и понятия теории множеств, а также наиболее эффективные способы представления (хранения) множеств, в первую очередь, на основе битовых полей.

На основе подготовленной теоретической базы, были реализованы классы для представления битовых полей TBitField и множеств TSet со всеми необходимыми операциями. Для проверки работоспособности и эффективности реализации перечисленных выше классов были написаны приложения sample_tbitfield и sample_tset, а также приложение для вывода всех простых чисел из указанного диапазона sample_primenumbers, основанное на алгоритме «решето Эратосфена»

Литература

1. Барышева И.В. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017 – 105 с.

2. Битовая операция [https://ru.wikipedia.org/wiki/Битовая_операция].

3. Как установить, сбросить, проверить нужный бит или битовые операции [https://hubstub.ru/programming/64-kak-ustanovit-sbrosit-proverit-nuzhnyy-bit-ili-bitovyye-operatsii.html].

4. Как пересечение и объединение множеств используются в анализе данных [https://practicum.yandex.ru/blog/peresechenie-i-obedinenie-mnozhestv].

5. Решето Эратосфена [https://ru.wikipedia.org/wiki/Решето_Эратосфена].

Приложения

Приложение А. Реализация класса TBitField

```
#include "tbitfield.h"

using namespace std;

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

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

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

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

TELEM TBitField::GetMemMask(const int n) const // битовая маска для бита n
{
    return 1 << (n & (BitsInMem - 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 < 0 || n >= BitLen) {
        throw exception("Negative length");
    }
    pMem[GetMemIndex(n)] = pMem[GetMemIndex(n)] | GetMemMask(n);
}

void TBitField::ClrBit(const int n) // очистить бит
{
    if (n < 0 || n >= BitLen) {
        throw exception("Negative length");
    }
    pMem[GetMemIndex(n)] = pMem[GetMemIndex(n)] & ~GetMemMask(n);
}

int TBitField::GetBit(const int n) const // получить значение бита
{
    if (n < 0 || n >= BitLen) {
        throw exception("Negative length");
```

```
    }
    return (pMem[GetMemIndex(n)] & GetMemMask(n));
}

// битовые операции

TBitField& TBitField::operator=(const TBitField &bf) // присваивание
{
    if (this == &bf)
        return *this;
    BitLen = bf.BitLen;
    if (MemLen != bf.MemLen) {
        TELEM* p = new TELEM[bf.MemLen];
        MemLen = bf.MemLen;
        delete[] pMem;
        pMem = p;
    }
    memcpy(pMem, bf.pMem, MemLen);
    return *this;
}

int TBitField::operator==(const TBitField &bf) const // сравнение
{
    if (BitLen != bf.BitLen)
        return false;
    for (size_t 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 TBitField* max_field = this;
    int min_size = bf.BitLen;
    if (BitLen < min_size) {
        max_field = &bf;
        min_size = BitLen;
    }
    TBitField tmp(*max_field);
    for (int i = 0; i < min_size; i++)
        if ((GetBit(i) || bf.GetBit(i)) == 1)
            tmp.SetBit(i);
    return tmp;
}

TBitField TBitField::operator&(const TBitField &bf) // операция "и"
{
    int max_size = BitLen;
    int min_size = bf.BitLen;
    if (bf.BitLen > max_size) {
        max_size = bf.BitLen;
        min_size = BitLen;
    }
    TBitField tmp(max_size);
    for (int i = 0; i < min_size; i++)
        if ((GetBit(i) && bf.GetBit(i)) == 1)
            tmp.SetBit(i);
    return tmp;
}

TBitField TBitField::operator~(void) // отрицание
{
    TBitField tmp = (*this);
    for (int i = 0; i < BitLen; i++)
```

```
        if (tmp.GetBit(i))
            tmp.ClrBit(i);
        else
            tmp.SetBit(i);
    return tmp;
}

// ввод/вывод

istream &operator>>(istream &istr, TBitField &bf) // ввод
{
    cout << "Enter bit string: ";
      string BitField;
      istr >> BitField;
      if (BitField.length() > bf.BitLen)
          throw exception("Out of range");
      for (int i = 0; i < BitField.length(); i++)
          if (BitField[i] == '1')
              bf.SetBit(i);
          else
              bf.ClrBit(i);
      return istr;
}

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

Приложение Б. Реализация класса TSet

```
#include "tset.h"
#include "tbitfield.h"

using namespace std;

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

// конструктор копирования
TSet::TSet(const TSet &s) : BitField(s.GetMaxPower())
{
    MaxPower = s.GetMaxPower();
    BitField = s.BitField;
}

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

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

int TSet::GetMaxPower(void) const // получить макс. к-во эл-тов
{
    return MaxPower;
}
```

```
int TSet::IsMember(const int Elem) const // элемент множества?
{
    return BitField.GetBit(Elem);
}

void TSet::InsElem(const int Elem) // включение элемента множества
{
    if (Elem < 0 || Elem >= MaxPower) {
        throw exception("Element is out of univers");
    }
    BitField.SetBit(Elem);
}

void TSet::DelElem(const int Elem) // исключение элемента множества
{
    if (Elem < 0 || Elem >= MaxPower) {
        throw exception("Element is out of univers");
    }
    BitField.ClrBit(Elem);
}

// теоретико-множественные операции

TSet& TSet::operator=(const TSet &s) // присваивание
{
    MaxPower = s.MaxPower;
    BitField = s.BitField;
    return *this;
}

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

int TSet::operator!=(const TSet &s) const // сравнение
{
    return !(*this == s);
}

TSet TSet::operator+(const TSet &s) // объединение
{
    if (*this == s)
        return *this;
    return TSet(BitField | s.BitField);
}

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

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

TSet TSet::operator*(const TSet &s) // пересечение
{
    if (*this == s)
        return *this;
    return TSet(BitField & s.BitField);
}

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

```
{
    return TSet(~BitField);
}

// ввод/вывод

istream &operator>>(istream& istr, TSet& s) // ввод
{
    int a;
      size_t count;
      cout << "Power of set: ";
      cin >> count;
      cout << "Enter elements of set: ";
      for (int i = 0; i < count; i++){
          istr >> a;
          s.InsElem(a);
      }
      return istr;
}

ostream &operator<<(ostream& ostr, const TSet& s) // вывод
{
    ostr << "{";
      for (int i = 0; i < s.GetMaxPower(); i++)
          if (s.IsMember(i))
              ostr << " " << i;
      ostr << " }";
      return ostr;
}
```