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»**

**Выполнил(а):** студент(ка) группы 3822Б1ФИ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Шпынов Н.А. /

Подпись

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

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

Подпись

Нижний Новгород  
2023

**Оглавление**

[Введение 1](#_Toc422)

[1 Постановка задачи 2](#_Toc25487)

[2 Руководство пользователя 3](#_Toc19014)

[2.1 Приложение для демонстрации работы битовых полей 3](#_Toc23871)

[2.2 Приложение для демонстрации работы множеств 4](#_Toc18857)

[2.3 Приложение «решето Эратосфена» 6](#_Toc25697)

[3 Руководство программиста 7](#_Toc17542)

[3.1 Использованнные алгоритмы 7](#_Toc25248)

[3.2 Описание классов 10](#_Toc7783)

[Заключение 16](#_Toc29631)

[Литература 17](#_Toc14557)

[Приложения 18](#_Toc15490)

[Приложение А. Реализация класса TBitField 18](#_Toc4425)

[Приложение Б. Реализация класса TSet 20](#_Toc10693)

# Введе**н**ие

Битовое поле — это структура данных, позволяющая компактно хранить и работать с набором битов, устанавливать их значения и проверять их. Каждое поле представляет собой группу битов в памяти компьютера, каждому из которых присваивается определённая позиция (0 или 1).

Использование битовых полей даёт большую гибкость, обеспечивает оптимизацию памяти и удобство работы с различными флагами и настройками в программе. Также они позволяют управлять информацией и хранить её в компактном формате. А простота этой структуры обеспечивает ее кросс-платформенность.

Примеры использования битовых полей:

1. Флаги состояний. Битовые поля можно использовать для представления и управления флагами состояния объектов или системы. Каждому состоянию "включено/выключено" или "открыто/закрыто" ставится в соответствие свой бит, значение которого и будет указывать на текущее положение.
2. Настройки конфигурации. Битовые поля позволяют упаковать различные настройки или флаги конфигурации в одно поле, что делает управление, использование и применение настроек какого то объекта более удобными.
3. Сериализация данных. Битовые поля могут быть использованы для сериализации или десериализации данных (перевод структуры данных в битовую последовательность и наоборот), когда требуется компактно передать или сохранить данные.
4. *Реализация множеств.* Битовые поля являются одним из простейших способов реализации алгебры логики в программе. Каждому элементу в универсе ставится в соответствие свой бит, обозначающий его наличие или отсутствие в данном множестве.

# Постановка задачи

Цель – реализовать представление множеств на языке C++ при помощи битовых полей

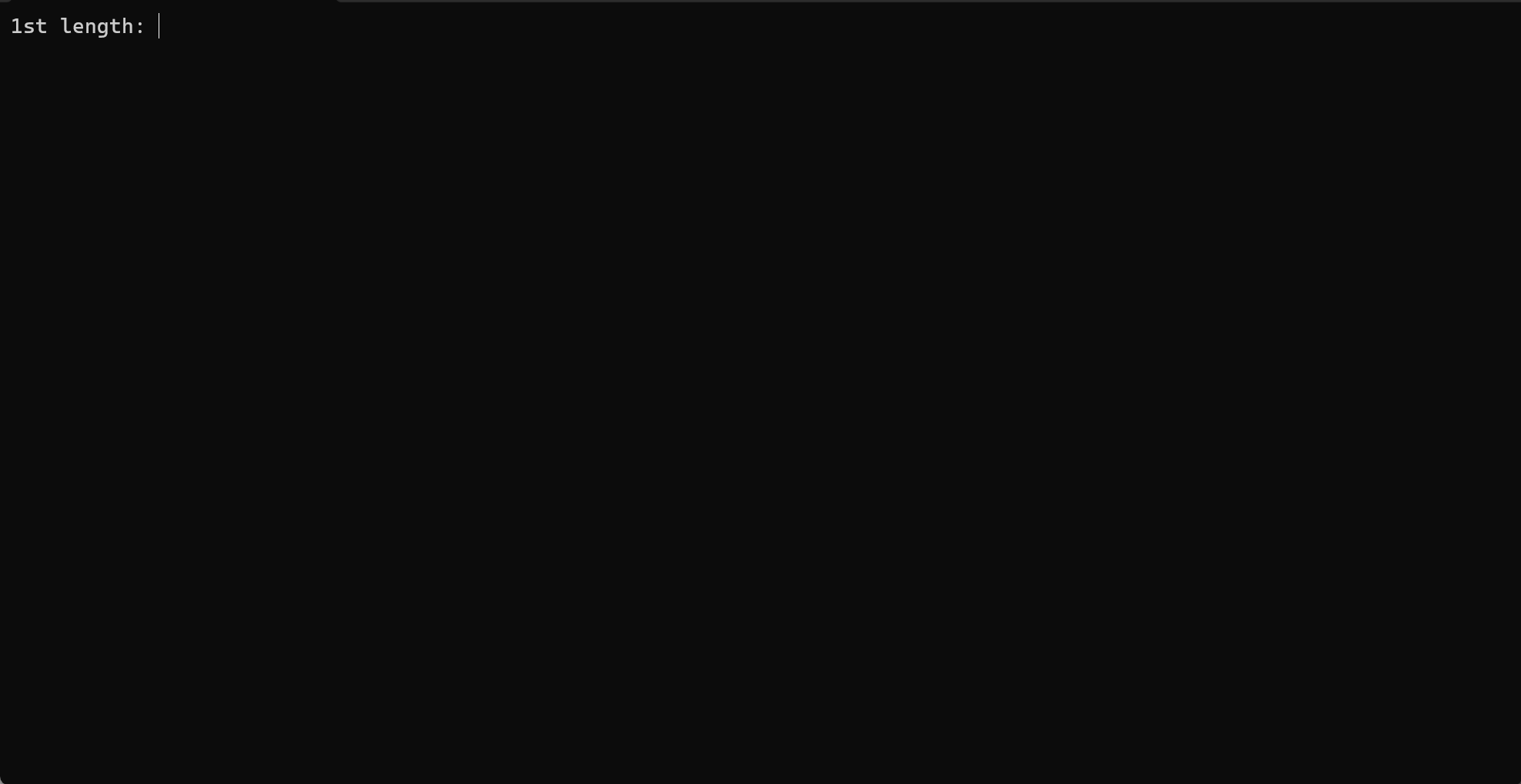
Задачи:

1. Разработка представления битовых полей. Для существования системы множеств на основе битовых полей, следует создать реализацию их самих.
2. Определить способ хранения битовых полей.
3. Создать структуру данных с битовыми полями.
4. Реализовать необходимые для работы поля и методы.
5. Разработка структуры данных представления множества. Определить, каким образом будут храниться элементы множества с использованием битовых полей. Потребуется создать специальную структуру данных, в которой каждый бит будет соответствовать наличию или отсутствию определённого элемента.
6. Определение операций над множествами. Разработать алгоритмы и функции для выполнения операций над множествами, таких как объединение, пересечение, разность, проверка на принадлежность элемента, добавление/удаление элементов из множества и другие.
7. Тестирование и отладка. Как и при любой разработке программного обеспечения, в конце необходимо провести тестирование и отладку кода. Следует обнаружить и исправить возможные ошибки и проблемы, а также убедиться в правильности функционирования.

# Руководство пользователя

## Приложение для демонстрации работы битовых полей

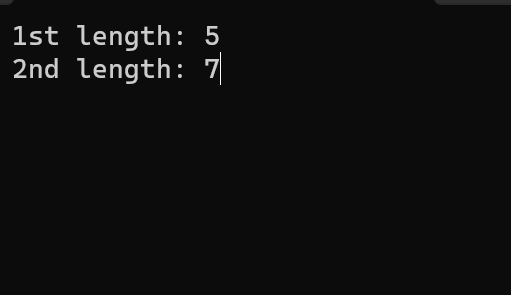
1. Запустить sample\_tbitfield.exe. В результате появится следующее окно (рис. 1):



1. Основное окно приложения

На этом шаге потребуется ввести два числа соответственно: длина первого битового поля, длина второго битового поля

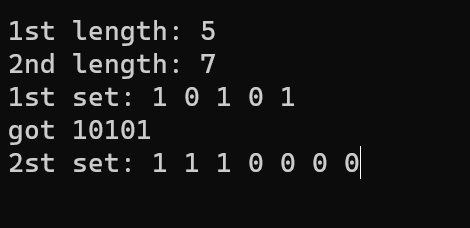
1. После ввода длин полей, следует их заполнить (рис. 2):



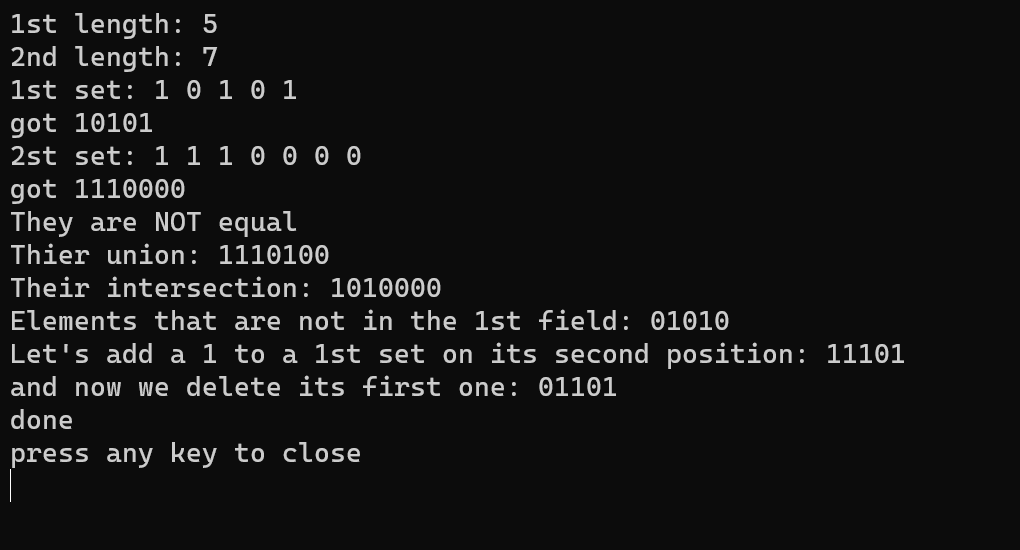
1. Ввод длин

Вводить значения (0 или 1 для каждого элемента в поле) надо по порядку через пробел или Enter, сначала для первого поля, а затем для второго.

1. Для проверки правильности ввода программа выводит полученные ею поля (рис. 3):



1. Результат ввода полей
2. После ввода второго поля программа покажет примеры результатов операций над полями. После чего попросит ввести любое значение для выхода (рис. 4).



1. Итог

## Приложение для демонстрации работы множеств

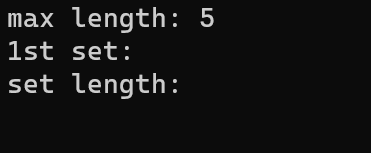
1. Запустить sample\_tset.exe. В результате появится следующее окно   
   (рис. 5):



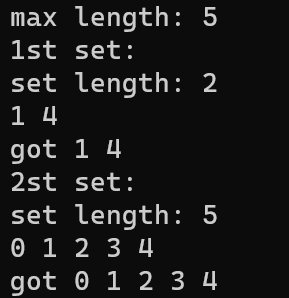
1. Основное окно приложения

Требуется ввести максимально возможный размер множества (то есть его универс).

1. После ввода длины универса (рис. 6):



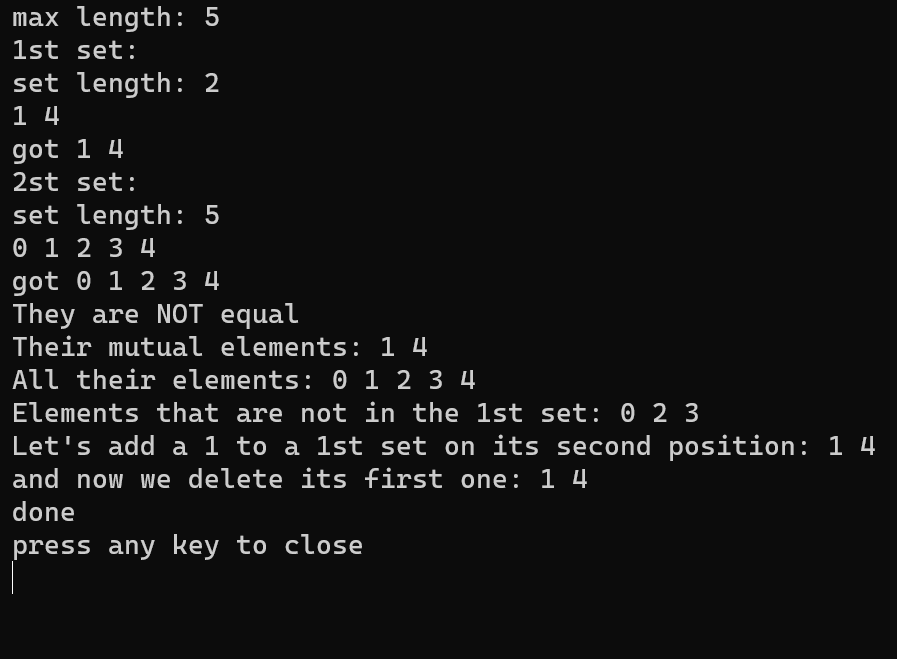
1. Ввод длины
2. Далее потребуется ввести множества соответственно (рис. 7):



1. результат ввода множеств

Сначала вводится длина первого множества, а затем номера включённых для него элементов в любом порядке через пробел или Enter. После чего программа выводит полученное значение для проверки корректности ввода. Затем аналогично ввод происходит и для второго множества

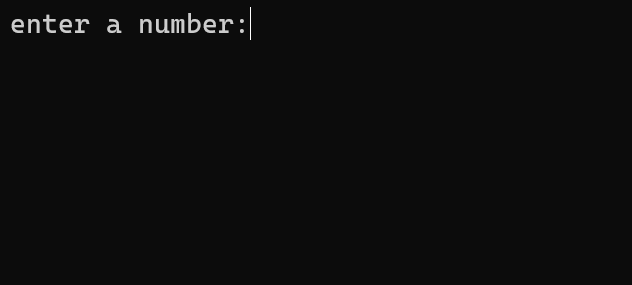
1. Программа покажет примеры результатов операций над множествами. После чего попросит ввести любое значение для выхода (рис. 8).



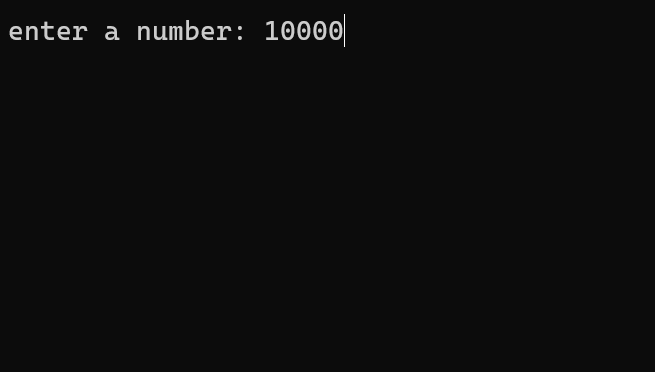
1. Итог

## Приложение «решето Эратосфена»

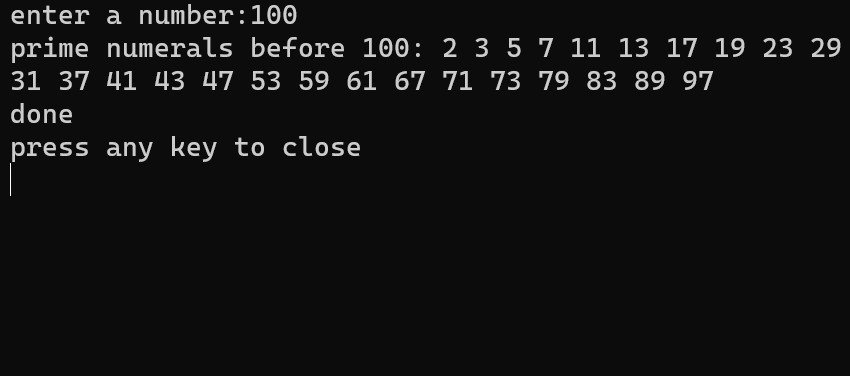
1. Запустить sample\_primenumbers.exe. В результате появится следующее окно (рис. 9):



1. Основное окно приложения
2. Далее необходимо ввести число, в рамках которого будет искать все простые числа (рис. 10):



1. Ввод числа
2. Программа выдаст все простые числа, меньше введённого (рис. 11):



1. Результат

# Руководство программиста

## Использованные алгоритмы

### Битовые поля

Битовое поле – это структура данных, представляющая собой набор битов, с каждым из которых можно работать по отдельности, устанавливать его нулём или единицей, просматривать текущее значение.

1. Получение элемента. Возвращает текущее состояние бита по его номеру.

Входные данные: n – номер желаемого элемента.

Выходные данные: текущее состояние бита.

|  |  |  |
| --- | --- | --- |
| **А** | **№** | **RES** |
| 010 | 2 | 1 |
| 010 | 1 | 0 |

1. Получение маски. Возвращает битовую маску для заданной позиции

Входные данные: n – номер желаемого элемента.

Выходные данные: маска для заданного бита (2(n mod BitsInElem) -1).

|  |  |  |
| --- | --- | --- |
| **А** | **№** | **RES** |
| 010 | 2 | 2 |
| 010 | 3 | 4 |

1. Побитовое **ИЛИ**. Если хотя бы в одной позиции битового поля стоит 1, то после применения операции в результате будет стоять 1.

Входные данные: А и В – два битовых поля.

Для каждого элемента класса, которым реализованы битовые поля применяется операция побитового или.

Выходные данные: результирующее битовое поле.

|  |  |  |
| --- | --- | --- |
| **А** | **В** | **А|В** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1. Побитовое **И. Е**сли в обоих полях в соответствующих позициях стоит 1, то после применения операции в результате будет стоять 1.

Входные данные: А и В – два битовых поля.

Для каждого элемента класса, которым реализованы битовые поля применяется операция побитового и.

Выходные данные: результирующее битовое поле.

|  |  |  |
| --- | --- | --- |
| **А** | **В** | **А&В** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

1. Инверсия каждого разряда.

Входные данные: А – битовое поле.

Проверяется значение каждого бита, если он 1, то к нему применяется удаление бита и наоборот, для 0 применяется включение бита.

Выходные данные: результирующее битовое поле.

|  |  |
| --- | --- |
| **А** | **~А** |
| 0 | 1 |
| 1 | 0 |

1. Включение/удаление бита в поле по его номеру.

Входные данные: А – битовое поле, n – порядковый номер бита.

1. По номеру бита ищется его маска и элемент, в котором он лежит, и путем побитового «или» между ними бит включается в поле.
2. По номеру бита ищется его маска и элемент, в котором он лежит, и путем побитового «и» между элементом и числом, обратным маске бит исключается из поля.

Выходные данные: результирующее битовое поле.

|  |  |  |  |
| --- | --- | --- | --- |
| **А** | **№** | **+** | **-** |
| 010 | 3 | 011 | 010 |
| 010 | 2 | 010 | 000 |

1. Поэлементное сравнение двух полей.

|  |  |  |
| --- | --- | --- |
| **А** | **В** | **==** |
| 010 | 0011 | 0 |
| 010 | 010 | 1 |

### Множества

Множество – структура данных, представляющая множество из алгебры логики.

Для реализации множеств удобно использовать битовое поле, хранящее в каждом бите информацию о включении соответствующего элемента в это множество. Для определенности далее предположим, что универс содержит 5 элементов.

1. Объединение. Если элемент есть хотя бы в одном множестве, то он будет включён и в результат.

Применяется побитовое «или» для их полей, результирующее поля интерпретируется как искомое множество.

|  |  |  |
| --- | --- | --- |
| **А** | **В** | **А|В** |
| 1, 2, 3 | 2, 3, 4 | 1, 2, 3, 4 |

1. Пересечение. Если элемент есть хотя бы в одном множестве, то он будет включён и в результат.

Применяется побитовое «или» для их полей, результирующее поля интерпретируется как искомое множество.

|  |  |  |  |
| --- | --- | --- | --- |
| **А** | **В** | **А&В** |  |
| 1, 2, 3 | 2, 3, 4 | 2, 3 |  |

1. Дополнение. Множество, обратное данному.

Применяется отрицание для его поля, результирующее интерпретируется как искомое множество.

|  |  |
| --- | --- |
| **А** | **~А** |
| 0, 3, 4 | 1, 2 |

1. Включение/исключение элемента из множества.

Применяется включение/удаление для бита с соответствующим индексом для его поля, результирующее интерпретируется как искомое множество.

|  |  |  |  |
| --- | --- | --- | --- |
| **А** | **Elem** | **+** | **-** |
| 0, 3, 4 | 3 | 0, 3, 4 | 0, 4 |
| 0, 3, 4 | 2 | 0, 2, 3, 4 | 0, 3, 4 |

1. Поэлементное сравнение двух множеств.

Применяется сравнение для их полей.

|  |  |  |  |
| --- | --- | --- | --- |
| **А** | **В** | **==** | **!=** |
| 0, 3, 4 | 2, 3 | 0 | 1 |
| 0, 3, 4 | 0, 3, 4 | 1 | 0 |

### Алгоритм «решето Эратосфена»

С помощью алгоритма «Решето Эратосфена» найти все простые числа, меньшие заданного.

Входные данные: n – число, в пределах которого будет производиться поиск.

Выходные данные: список всех простых чисел, меньших n.

Решето Эратосфена – [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC" \o "Алгоритм) нахождения всех [простых чисел](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE" \o "Простое число) до некоторого целого числа n. Решето подразумевает отфильтровывание всех чисел за исключением простых.

1. Выписать подряд все целые числа от двух до n (2, 3, 4, …, n).
2. Пусть переменная p изначально равна 2– наименьшему простому числу.
3. Зачеркнуть в списке числа от 2p до n, кратные p.
4. Найти первое незачёркнутое число в списке, большее чем p, и присвоить значению переменной p это число.
5. Повторять шаги 3 и 4, пока возможно.
6. Все оставшиеся числа и будут простыми.

## Описание классов

### 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();

int GetLength(void) const;

void SetBit(const int n);

void ClrBit(const int n);

int GetBit(const int n) const;

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

int 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 &istr, TBitField &bf);

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

};

*Поля:*

BitLen – длина битового поля.

pMem – массив элементов заданного типа данных, в котором хранится битовое поле.

MemLen – количество элементов массива pMem.

*Конструкторы:*

TBitField(int len) – конструктор с параметром.

Назначение: создание поля с заданной длиной

Входные данные: len – длина поля.

TBitField(const TBitField &bf) – конструктор копирования.

Назначение: создание поля из другого.

Входные данные: &bf – заданное поле.

*Деструктор*:

~TBitField();

*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 маски элемента, где хранится бит.

Входные данные: n – номер бита.

Выходные данные: число, являющееся маской бита с номером n

int GetLength(void) const;

Назначение: получение длины поля вне метода.

Входные данные: нет.

Выходные данные: длина поля.

void SetBit(const int n);

Назначение: установка бита с индексом n единицей.

Входные данные: n – номер.

Выходные данные: нет.

void ClrBit(const int n);

Назначение: установка бита с индексом n нулем.

Входные данные: n – номер.

Выходные данные: нет.

int GetBit(const int n) const;

Назначение: получение значения бита под индексом n.

Входные данные: n – номер.

Выходные данные: значение бита под индексом n.

*Операторы:*

int operator==(const TBitField &bf) const

Назначение: оператор сравнения.

Сравнить на равенство 2 битовых поля.

Входные данные: bf – битовое поле, с которым идёт сравнение.

Выходные данные: 1 если равны, 0 если нет.

int operator!=(const TBitField &bf) const

Назначение: оператор сравнения на неравенство.

Сравнить на неравенство 2 битовых поля.

Входные данные: bf – битовое поле, с которым идёт сравнение.

Выходные данные: 0 если равны, 1 если нет.

сonst TBitField& operator=(const TBitField &bf)

Назначение: оператор присваивания. Присвоить полю this поле bf.

Входные данные: bf – битовое поле, которое присваевается.

Выходные параметры: ссылка на поле this равное bf.

TBitField operator|(const TBitField &bf)

Назначение: оператор **или**.

Входные данные: bf – битовое поле, с которым проводится операция.

Выходные данные: поле (this | bf).

TBitField operator&(const TBitField &bf)

Назначение: оператор **и**.

Входные данные: bf – битовое поле, с которым проводится операция.

Выходные данные: поле (this & bf).

TBitField operator~(void);

Назначение: оператор **не**.

Входные данные: нет .

Выходные данные: поле ~this.

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

Назначение: оператор **ввода** из терминала.

Входные данные: istr – поток ввода, bf – битовое поле.

Выходные данные: нет.

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

Назначение: оператор **вывода** в терминал.

Входные данные: ostr – поток вывода, bf – битовое поле.

Выходные данные: переданное поле выводится в терминал.

### Класс TSet

class TSet

{

private:

int MaxPower;

TBitField BitField;

public:

TSet(int mp);

TSet(const TSet &s);

TSet(const TBitField &bf);

operator TBitField();

int GetMaxPower(void) const;

void InsElem(const int Elem);

void DelElem(const int Elem);

int IsMember(const int Elem) const;

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

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

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);

};

*Поля*:

int MaxPower;  – мощность универса для этого множества.

TBitField BitField;– поле, характеризующее множество.

*Конструкторы*:

TSet(int mp) – конструктор с параметром.

Назначение: создание множества с заданной длиной.

Входные данные: mp – длина поля.

TSet(const TSet &s) – конструктор копирования.

Назначение: создание множества из другого.

Входные данные: s – заданное множество.

TSet(const TBitField &bf) – конструктор присваивания.

Назначение: создание множества из поля.

Входные данные: bf – заданное поле.

*Методы*:

int GetMaxPower() const;

Назначение: получение максимальной мощности множества, то есть размера универса.

Входные данные: нет.

Выходные данные: максимальная мощность множества.

void InsElem(const int Elem);

Назначение: включение элемента в множество.

Входные данные: Elem – номер включаемого элемента.

Выходные данные: нет.

void DelElem(const int Elem);

Назначение: удаление элемента из множества.

Входные данные: Elem – номер удаляемого элемента.

Выходные данные: нет.

int IsMember(const int Elem) const;

Назначение: проверка элемента на наличие в множестве.

Входные данные: Elem – номер проверяемого элемента.

Выходные данные:

1 – элемент присутствует в множестве.

0 – элемента в множестве нет.

*Операторы:*

int operator==(const TSet &s) const

Назначение: оператор **сравнения**.

Сравнить на равенство 2 множеств.

Входные данные: s – множество, с которым идёт сравнение.

Выходные данные: 1 если равны, 0 если нет.

int operator!=(const TSet &s) const

Назначение: оператор **сравнения на неравенство**.

Сравнить на неравенство 2 множеств.

Входные данные: s – множество, с которым идёт сравнение.

Выходные данные: 0 если равны, 1 если нет.

сonst TSet& operator=(const TSet &s)

Назначение: оператор **присваивания**.

Входные данные: s – множество, которое присваивается.

Выходные параметры: ссылка на множество this, равное s.

TSet operator\*(const TSet &s)

Назначение: оператор **пересечения множеств**.

Входные данные: s – множество, с которым проводится операция.

Выходные данные: множество (this \* s).

TSet operator+(const TSet &s)

Назначение: оператор **объединения множеств**.

Входные данные: s – множество, с которым проводится операция.

Выходные данные: множество (this + s).

TSet operator-(const TSet &s)

Назначение: оператор **вычитания множества**.

Входные данные: s – множество, с которым проводится операция.

Выходные данные: множество (this - s).

TSet operator-(const int Elem)

Назначение: оператор **исключения элемента**.

Входные данные: s – номер элемента, с которым проводится операция.

Выходные данные: множество (this без Elem).

TSet operator+(const int Elem)

Назначение: оператор **включения элемента**.

Входные данные: s – номер элемента, с которым проводится операция.

Выходные данные: множество (this c Elem).

TSet operator~(void);

Назначение: оператор **обратного множества**.

Входные данные: нет.

Выходные данные: множество ~this.

friend istream &operator>>(istream &istr, const TSet &s)

Назначение: оператор **ввода** из терминала.

Входные данные: istr – поток ввода, s – множество.

Выходные данные: нет.

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

Назначение: оператор **вывода** в терминал.

Входные данные: ostr – поток вывода, s – множество.

Выходные данные: переданное множество выводится в терминал.

# Заключение

Реализация множеств при помощи битовых полей оказалась эффективным подходом, позволяющим компактно хранить и оперировать наборами элементов. Этот метод позволяет сократить затраты на память, обеспечивает быстрые операции с множествами и может быть применён в решении различных задач, связанных с манипуляцией данными и их анализом.

# Литература

1. Битовые поля [<https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-bit-fields?view=msvc-170>].
2. Справочник по С++ [[https://prog-cpp.ru/cpp](https://prog-cpp.ru/cpp/)].
3. Справочник по С++ [[https://en.cppreference.com/w](https://en.cppreference.com/w/)].

# Приложения

## Приложение А. Реализация класса TBitField

TBitField::TBitField(int len)

{

    if (len < 0)

        throw "negative length caught";

    BitLen = len;

    MemLen = BitLen / bitsInElem + 1;

    pMem = new TELEM[MemLen];

    memset(pMem, 0, Amount);

}

TBitField::TBitField(const TBitField &bf)

{

    BitLen = bf.BitLen;

    MemLen = bf.MemLen;

    pMem = new TELEM[MemLen];

    memcpy(pMem, bf.pMem, Amount);

}

TBitField::~TBitField()

{

    delete[] pMem;

}

int TBitField::GetMemIndex(const int n) const

{

    if (n < BitLen)

        return n >> shiftSize;

    else

        throw "no such bit";

}

TELEM TBitField::GetMemMask(const int n) const

{

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

}

int TBitField::GetLength(void) const

{

    return BitLen;

}

void TBitField::SetBit(const int n)

{

     if ((n >= BitLen) || (n < 0))

        throw "out of range";

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

}

void TBitField::ClrBit(const int n)

{

    if ((n >= BitLen) || (n < 0))

        throw "out of range";

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

}

int TBitField::GetBit(const int n) const

{

    if (n >= BitLen || n < 0)

        throw "out of range";

    if ((pMem[GetMemIndex(n)] & GetMemMask(n)) == 0)

        return 0 ;

    else

        return 1;

}

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

{

    if (\*this == bf)

        return \*this;

    if (BitLen != bf.BitLen)

    {

        delete[] pMem;

        BitLen = bf.BitLen;

        MemLen = bf.MemLen;

        pMem = new TELEM[MemLen];

    }

    memcpy(pMem, bf.pMem, Amount);

    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 maxlen = max(BitLen, bf.BitLen);

    TBitField Ftmp(maxlen);

    for (int i = 0; i < Ftmp.MemLen; i++)

        Ftmp.pMem[i] = pMem[i] | bf.pMem[i];

    return Ftmp;

}

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

{

    int maxlen = max(BitLen, bf.BitLen);

    TBitField Ftmp(maxlen);

    for (int i = 0; i < Ftmp.MemLen; i++)

        Ftmp.pMem[i] = pMem[i] & bf.pMem[i];

    return Ftmp;

}

TBitField TBitField::operator~(void)

{

    TBitField Ftmp(BitLen);

    for (int i = 0; i < BitLen; i++)

        if (GetBit(i))

            Ftmp.ClrBit(i);

        else Ftmp.SetBit(i);

    return Ftmp;

}

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

{

    int bt;

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

            istr >> bt;

            if (!bt) {

                bf.ClrBit(i);

            }

            else if (bt == 1) {

                bf.SetBit(i);

            }

            else{

                cout << "not 0 or 1, setting to 1";

                bf.SetBit(i);

                throw "not 0 or 1";

            }

        }

        return istr;

}

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

{

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

        if (bf.GetBit(i)) ostr << "1";

        else ostr << "0";

    }

    return ostr;

}

## Приложение Б. Реализация класса TSet

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

TSet::TSet(const TSet &s) :

BitField(s.BitField), MaxPower(s.GetMaxPower()) {}

TSet::TSet(const TBitField &bf) :

BitField(bf), MaxPower(bf.GetLength()){}

int TSet::GetMaxPower(void) const

{

    return MaxPower;

}

int TSet::IsMember(const int Elem) const

{

    return BitField.GetBit(Elem);

}

void TSet::InsElem(const int Elem)

{

    BitField.SetBit(Elem);

}

void TSet::DelElem(const int Elem)

{

    BitField.ClrBit(Elem);

}

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

{

    if (\*this == s)

        return \*this;

    BitField = s.BitField;

    MaxPower = s.MaxPower;

    return \*this;

}

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

{

    if (MaxPower != s.MaxPower)

        return 0;

    return (BitField == s.BitField);

}

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

    return !(\*this == s);

}

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

{

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

    TStmp.BitField = BitField | s.BitField;

    return TStmp;

}

TSet TSet::operator+(const int Elem)

{

    TSet TStmp(\*this);

    TStmp.InsElem(Elem);

    return TStmp;

}

TSet TSet::operator-(const int Elem)

{

    TSet TStmp(\*this);

    TStmp.DelElem(Elem);

    return TStmp;

}

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

{

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

    TStmp.BitField = BitField & s.BitField;

    return TStmp;

}

TSet TSet::operator~(void)

{

    TSet TStmp(MaxPower);

    TStmp.BitField = ~BitField;

    return TStmp;

}

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

{

   int elem;

   int n;

   cout << "set length: ";

   istr >> n;

   for (int i = 0; i < n; i++) {

       istr >> elem;

       s.InsElem(elem);

   }

    return istr;

}

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

{

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

        if (s.IsMember(i))

            ostr << i << " ";

    return ostr;

}