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
_____ / Чистов А.Д. /
Подпись

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

Подпись

Нижний Новгород
2023

Содержание

Введение

Битовое поле — в программировании число, занимающее некоторый набор битов, напрямую не адресуемый процессором. Битовые поля часто используются для управления флагами или настроек [1].

Активное применение аппарата теории множеств в современной науке приводит к необходимости создания соответствующих программных решений. Они позволяют получать доступ до отдельных битов или групп битов. Доступ до отдельных битов можно осуществлять и с помощью битовых операций, но использование битовых полей часто упрощает понимание программы [2]. Важным преимуществом использования битовых операций является тот факт, что позволяют выполнять различные манипуляции с битами, такие как установка, сброс или инвертирование конкретных битов. Это может быть полезно, например, при работе с масками или фильтрами.

Множества поддерживают базовые операции, такие как объединение, пересечение и разность, что может быть очень удобным при работе с данными. Например, можно объединить два множества, чтобы получить новое множество, содержащее все элементы из обоих исходных множеств.

В целом можно сказать, что битовые поля и множества имеют широкое применение в различных областях, особенно в программировании и компьютерных системах. Они позволяют эффективно использовать память и упрощают работу с большим количеством данных.

1 Постановка задачи

Цель – разработать класс TBitField и класс TSet для работы с битовыми полями и множествами на их основе.

Задачи:

1. Исследовать тематическую литературу.
2. Реализовать класс TBitField.
3. Реализовать класс TSet.
4. Провести тестирование разработанных классов для проверки их корректной работы.
5. Сделать выводы о проделанной работе.

2 Руководство пользователя

2.1 Приложение для демонстрации работы битовых полей

1. Запустить sample_bitfield.exe. В результате появится следующее окно, где вам будет необходимо ввести размерность битового поля (рис.1).

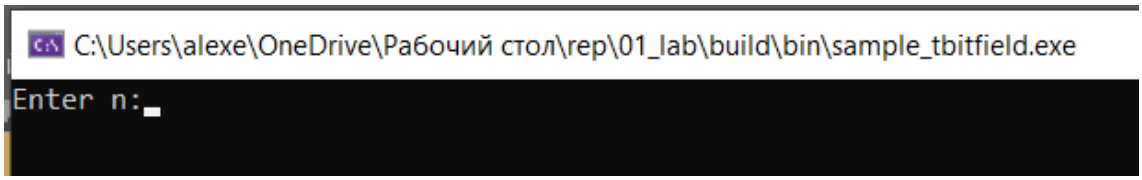

Рис.1. Основное окно приложения битовых полей

2. Дальше вам необходимо ввести 2 битовых поля длины 5 (рис.2).

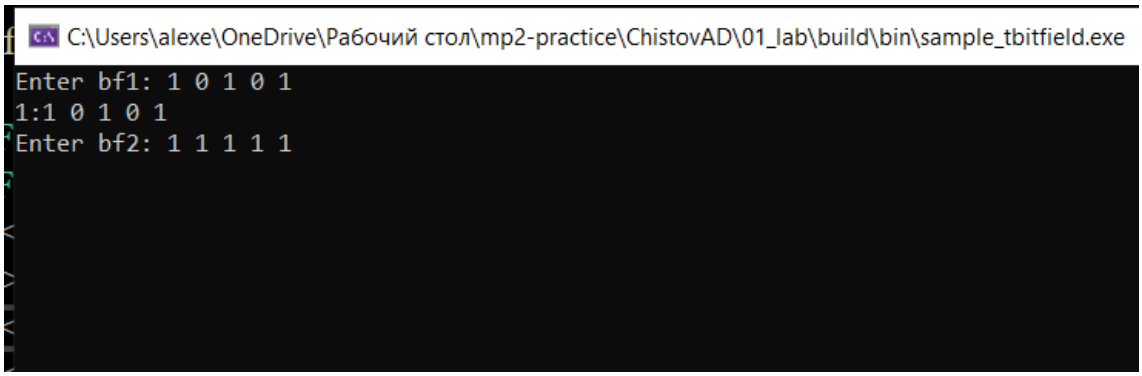

Рис.2. Ввод битовых полей

3. После вы получите результат работы программы с введенными битовыми полями (рис.3).

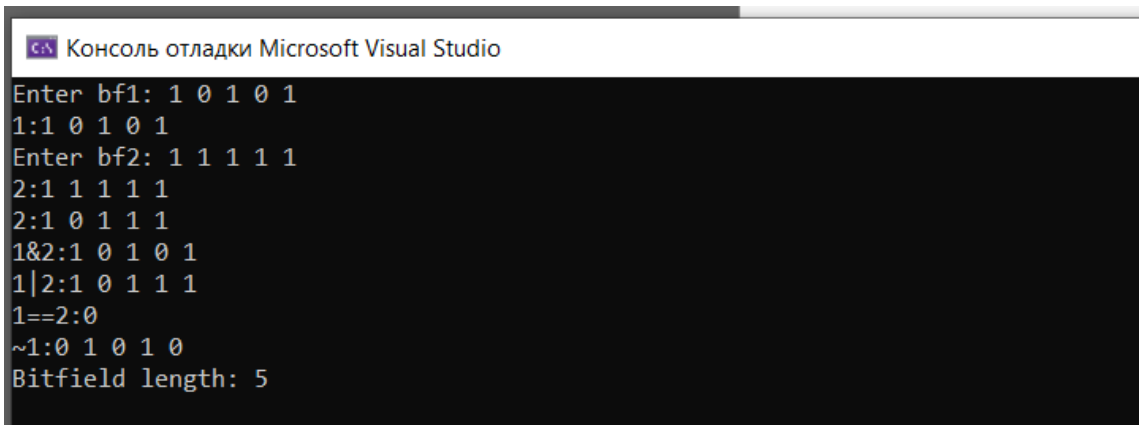

Рис.3. Результат работы программы

2.2 Приложение для демонстрации работы множеств

1. Запустить sample_bitfield.exe. В результате появится следующее окно, где вам будет необходимо ввести размерность множества (рис.4).

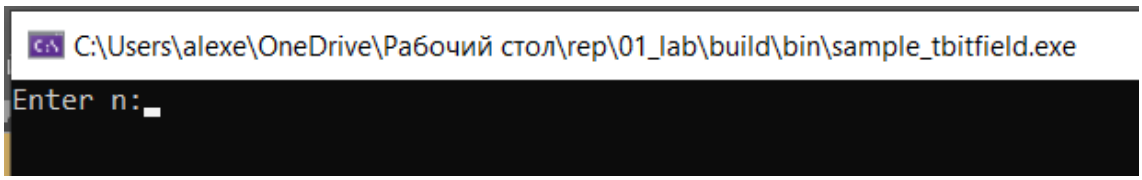

Рис.4. Основное окно работы множеств

2. Дальше вам необходимо ввести 2 множества длины 5 (рис.5).

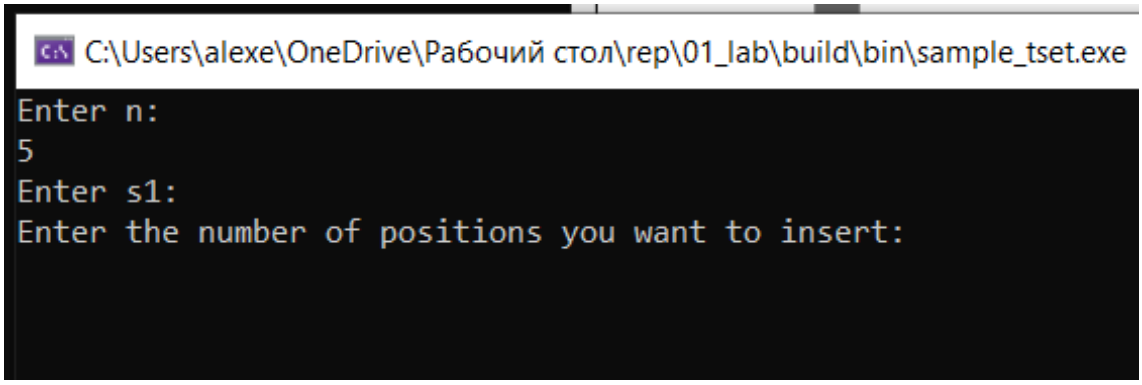

Рис.5. Ввод множеств

3. После вы получите результат работы программы с введенными множествами (рис.6).

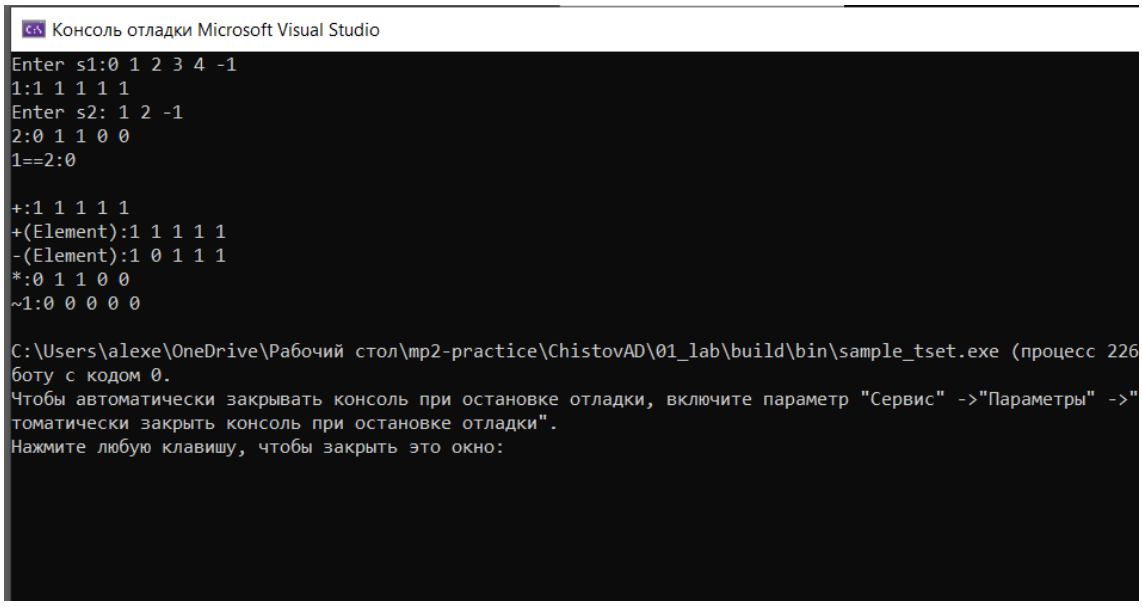

Рис.6. Результат работы программы

2.3 Приложение «решето Эратосфена»

1) Запустите sample_primenumbers.exe. В результате появится следующее окно (рис.7).

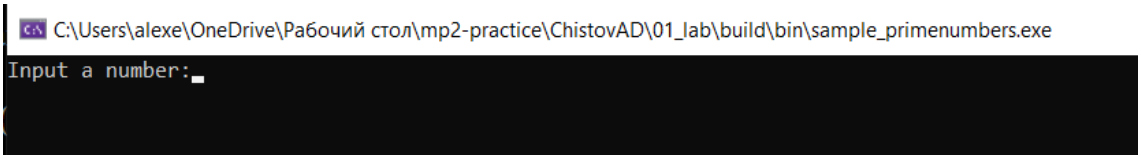

Рис.7. Основное окно приложения

2) Далее необходимо ввести целое положительное число для того, чтобы получить все простые числа до введенного (рис.8).

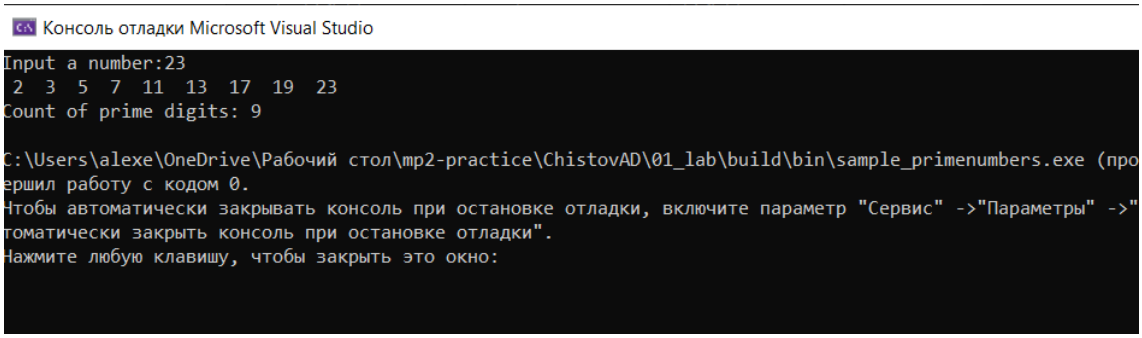

Рис.8. Результат работы приложения решето Эратосфена

3 Руководство программиста

3.1 Использованные алгоритмы

3.1.1 Битовые поля

Битовое поле — это структура данных, состоящая из одного или нескольких соседних битов, выделенных для определенных целей, так что любой отдельный бит или группа битов в структуре может быть установлен или проверен. Битовые поля обеспечивают удобный доступ к отдельным битам данных.

Пример:

A	0	1	1	0	0	1

С битовыми полями можно реализовать операции: взятие маски, получение значения бита, очистка бита, установка бит 1.

На основе битовых полей можно реализовать следующие побитовые операции:

- **Операция побитовое «ИЛИ»:**

Входные данные: 2 битовых поля.

Выходные данные: битовое поле, каждый бит которого равен 1, если он есть хотя бы в 1 битовом поле, которые объединяем, и 0 в противном случае.

Пример:

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

- **Операция побитовое «И»:**

Входные данные: 2 битовых поля.

Выходные данные: битовое поле, каждый бит которого равен 1, если он есть в обоих в 1 битовом поле, которые объединяем, и 0 в противном случае.

Пример:

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

- **Операция побитовое «Отрицание»:**

Входные данные: битовое поле.

Выходные данные: битовое поле, каждый бит которого равен 0, если он равен 1 в исходном поле, и 1, в противном случае.

Пример:

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

- **Добавление и удаление бита:**

Входные данные: битовое поле, индекс (номер) бита.

Выходные данные: битовое поле, с добавленным (удаленным) битом. Добавление и удаление бита ставит 1 и 0 соответственно на указанную позицию.

Пример:

A	0	1	0	0	0	1

a) Установление отдельного бита в 1

Для реализации этой операции нам понадобится битовое поле, в котором все биты равны 0, и один бит равен 1. Этот бит должен стоять на разряде, который в исходном битовом поле необходимо установить в 1, обозначим за i. Такой битовой маски можно добиться сдвигом целочисленной 1 на i бит влево. Такое поле будем называть битовой маской. Если к исходному битовому полю и битовой маске применить операцию битового «ИЛИ»:

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

В результате получим битовое поле, в котором бит на i-ом разряде установлен в 1, а все оставшиеся биты равны битам в исходном битовом поле:

A	0	1	1	0	0	1

b) Установление отдельного бита в 0

Битовая маска для этой операции имеет значение i-ого разряда 0, а значения всех остальных разрядов равны 1. Такую маску можно получить инвертированием маски, которая использовалась при установлении определенного бита в 1. При применении к исходному битовому полю и инвертированной маске операции битового «И»:

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

В результате получим битовое поле, в котором бит на i-ом разряде равен 0, а все оставшиеся биты равны битам исходного битового поля.

A	0	0	0	0	0	1

- **Операции сравнения:**

Операция равенства выведет 1, если два битовых поля равны, или каждые их биты совпадают, 0 в противном случае. Операция, обратная операции равенства, выведет 0, если хотя бы два бита совпадают, 1 в противном случае.

3.1.2 Множества

Множества представляют собой набор целых положительных чисел и реализованы при помощи битового поля, соответственно каждый бит которых интерпретируется элементом, равным индексом бита. Битовые поля обеспечивают удобный доступ к отдельным битам данных и позволяют формировать объекты с длиной, не кратной байту, что позволяет экономить память и более плотно размещать данные.

Множество поддерживает следующие операции:

- **Операция объединения множеств:**

Входные данные: 2 множества.

Выходные данные: множество, равное объединению множеств, содержащее все элементы из двух множеств.

Пример:

A = {0, 1, 2, 3}

A	1	1	1	1	0	0

B = {5}

B	0	0	0	0	0	1

Результат объединения множеств A∪B:

A∪B = {0, 1, 2, 3, 5}

A∪B	1	1	1	1	0	1

- **Операция пересечения множеств:**

Операция пересечения множеств, содержащее все элементы, содержащиеся в обоих множествах.

Входные данные: множество.

Выходные данные: множество, равное пересечению множеств, содержащее элементы из двух множеств.

A = {1, 2, 3}

A	0	1	1	1	0	0

B = {1,3,5}

B	0	1	0	1	0	1

Результат пересечения множеств A∩B:

A∩B = {1, 3}

A∩B	0	1	0	1	0	0

- **Операция дополнения множества:**

  Входные данные: отсутствуют.

  Выходные данные: множество, равное дополнению исходного множества.

  Пример:

A = {1, 2, 3}

A	0	1	1	1	0	0

  Результат дополнения множества ~A:

~A = {0, 4, 5}

A	1	0	0	0	1	1

- **Операция сравнения множеств:**

Операция равенства выведет 1, если два множества равны, или каждые их биты совпадают, 0 в противном случае. Операция, обратная операции равенства, выведет 0, если хотя бы два бита совпадают, 1 в противном случае.

3.1.3 Алгоритм «решето Эратосфена»

Алгоритм «решето Эратосфена» предназначен для поиска всех простые числа в отрезке от 2 до целого положительного числа.

Входные данные: целое положительное число (далее N).

Выходные данные: множество простых чисел.

Алгоритм выполнения состоит в следующем:

1) У пользователя запрашивается целое положительное число.
2) Заполнение множества.
3) Проверка до квадратного корня и удаление кратных членов (данный шаг повторяется несколько раз, пока остаются кратные числа).
4) Полученные элементы будут простыми числами.

Ниже представлен код этого алгоритма на основе битовых полей:

```
TBitField s(n + 1);
for (m = 2; m <= n; m++)
    s.SetBit(m);
for (m = 2; m * m <= n; m++)
```

```
        if (s.GetBit(m))
            for (k = 2 * m; k <= n; k += m)
                if (s.GetBit(k))
                    s.ClrBit(k);
    count = 0;
    k = 1;
    for (m = 2; m <= n; m++)
        if (s.GetBit(m))
        {
            count++;
            cout << setw(3) << m << " ";
            if (k++ % 10 == 0)
                cout << endl;
        }
    cout << endl;
    cout << "Count of prime digits: " << count << endl;
}
```

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 – память для представления битового поля.

MemLen – количество элементов Мем для представления битового поля.

Конструкторы:

- **TBitField(int len);**

Назначение: конструктор с параметром, выделение памяти.

Входные параметры:

len – длина битового поля.

Выходные параметры:

Отсутствуют.

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

Назначение: конструктор копирования. Создание экземпляра класса на основе другого экземпляра.

Входные параметры:

`bf` – ссылка, адрес экземпляра класса, на основе которого будет создан другой.

Выходные параметры:

Отсутствуют.

Деструктор:

`~TBitField();`

Назначение: деструктор. Отчистка выделенной памяти.

Входные и выходные параметры 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 n) const;`**

Назначение: получение битовой маски.

Входные данные:

`n` – номер бита.

Выходные данные:

Элемент под номером n.

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

Назначение: получение длинны битового поля.

Входные параметры отсутствуют.

Выходные параметры: длинна битового поля.

- **`void SetBit(const int n)`**

Назначение: установить бит=1.

Входные параметры:

`n` - номер бита, который нужно установить.

Выходные параметры отсутствуют.

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

Назначение: отчистить бит (установить бит = 0).

Входные параметры:

`n` - номер бита, который нужно отчистить.

Выходные параметры отсутствуют

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

Назначение: вывести бит (узнать бит).

Входные параметры:

`n` - номер бита, который нужно вывести (узнать).

Выходные параметры: бит (1 или 0, в зависимости есть установлен он, или нет).

Операторы:

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

Назначение: оператор сравнения. Сравнить на равенство 2 битовых поля.

Входные параметры:

`bf` – битовое поле, с которым мы сравниваем.

Выходные параметры: 1 или 0, в зависимости равны они, или нет соответственно.

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

Назначение: оператор сравнения. Сравнить на равенство 2 битовых поля.

Входные параметры:

`bf` – битовое поле, с которым мы сравниваем.

Выходные параметры: 1 или 0, в зависимости равны они, или нет соответственно.

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

Назначение: оператор присваивания. Присвоить экземпляру *this экземпляр &bf

Входные параметры:

`bf` – битовое поле, которое мы присваиваем.

Выходные параметры: ссылка на экземпляр класса **`TBitField`**, *this.

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

Назначение: оператор побитового «ИЛИ».

Входные параметры:

`bf` – битовое поле.

Выходные параметры: экземпляр класса , который равен { *this | bf }.

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

Назначение: оператор побитового «И».

Входные параметры:

bf – битовое поле, с которым мы сравниваем.

Выходные параметры: Экземпляр класса, который равен { *this & bf }.

- **TBitField operator~(void);**

Назначение: оператор инверсии.

Входные параметры отсутствуют.

Выходные параметры: Экземпляр класса, каждый элемент которого равен{~*this}.

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

Назначение: оператор ввода из консоли.

Входные параметры:

istr – буфер консоли

bf – класс, который нужно ввести из консоли.

Выходные параметры: Ссылка на буфер (поток) istr.

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

Назначение: оператор вывода из консоли.

Входные параметры:

istr – буфер консоли.

bf – класс, который нужно вывести в консоль.

Выходные параметры: Ссылка на буфер (поток) istr.

3.2.2 Класс TSet

Объявление класса:

```
class TSet
{
private:
int MaxPower; // максимальная мощность множества
TBitField BitField; // битовое поле для хранения хар-го 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 n); // включить элемент в множество
void DelElem(const int n); // удалить элемент из множества
int IsMember(const int n) const; // проверить наличие элемента в
// множестве
// теоретико-множественные операции
```

```
int operator== (const TSet &s); // сравнение
TSet& operator=(const TSet &s); // присваивание
TSet operator+ (const int n); // включение элемента в множество
TSet operator- (const int n); // удаление элемента из множества
TSet operator+ (const TSet &s); // объединение
TSet operator* (const TSet &s); // пересечение
TSet operator~ (void); // дополнение
friend istream &operator>>(istream &istr, TSet &bf);
friend ostream &operator<<(ostream &ostr, const TSet &bf);
```

Назначение: представление множества чисел.

Поля:

MaxPower – максимальный элемент множества.

BitField – экземпляр битового поля, на котором реализуется множество.

Конструкторы:

- **TSet(int mp);**

Назначение: конструктор с параметром, выделение памяти.

Входные параметры:

mp – максимальный элемент множества.

Выходные параметры: Отсутствуют

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

Назначение: конструктор копирования. Создание экземпляра класса на основе другого экземпляра.

Входные параметры:

s – ссылка, адрес экземпляра класса, на основе которого будет создан другой.

Выходные параметры: Отсутствуют.

Деструктор:

- **~TSet();**

Назначение: деструктор. Отчистка выделенной памяти.

Входные и выходные параметры отсутствуют.

Методы:

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

Назначение: получение максимального элемента множества.

Входные параметры отсутствуют.

Выходные параметры: максимальный элемент множества.

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

Назначение: добавить элемент в множество.

Входные параметры: **Elem** - добавляемый элемент.

Выходные параметры отсутствуют.

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

Назначение: удалить элемент из множества.

Входные параметры:**Elem** - удаляемый элемент.

Выходные параметры отсутствуют.

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

Назначение: узнать, есть ли элемент в множестве.

Входные параметры:

Elem - элемент, который нужно проверить на наличие.

Выходные параметры: 1 или 0, в зависимости есть элемент в множестве, или нет.

Операторы:

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

Назначение: оператор сравнения. Сравнить на равенство 2 множества.

Входные параметры: **s** – битовое поле, с которым мы сравниваем.

Выходные параметры: 1 или 0, в зависимости равны они, или нет соответственно.

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

Назначение: оператор сравнения. Сравнить на равенство 2 множества.

Входные параметры: **s** – битовое поле, с которым мы сравниваем .

Выходные параметры: 0 или 1, в зависимости равны они, или нет соответственно.

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

Назначение: оператор присваивания. Присвоить экземпляру ***this** экземпляр **s**.

Входные параметры:

s – множество , которое мы присваиваем .

Выходные параметры: ссылка на экземпляр класса **TSet, *this**.

- **TSet operator+(const TSet &bf);**

Назначение: оператор объединения множеств.

Входные параметры: **s** - множество;

Выходные параметры: Экземпляр класса , который равен **{ *this | s }**.

- **TSet operator*(const TSet &bf);**

Назначение: оператор пересечения множеств.

Входные параметры: **s** - множество;
Выходные параметры: Экземпляр класса , который равен **{ *this & s }**.

- **TSet operator~(void);**

Назначение: оператор дополнение до полного множества.

Входные параметры: отсутствуют.

Выходные параметры: Экземпляр класса, каждый элемент которого равен **{~*this}**, т.е. если i элемент исходного экземпляра будет равен будет находиться в множестве, то на выходе его не будет, и наоборот.

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

Назначение: оператор ввода из консоли.

Входные параметры:

istr – буфер консоли.

s – класс, который нужно ввести из консоли.

Выходные параметры: ссылка на буфер (поток) &istr

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

Назначение: оператор вывода из консоли.

Входные параметры:

istr – буфер консоли .

s – класс, который нужно вывести в консоль.

Выходные параметры: Ссылка на буфер (поток) &istr

- **operator TBitField();**

Назначение: вывод поля **BitField.**

Входные параметры отсутствуют.

Выходные данные: поле **BitField.**

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

Назначение: оператор объединения множества и элемента. Данный оператор аналогичен метод добавления элемента в множество.

Входные параметры:

Elem - число

Выходные параметры: исходный экземпляр класса, содержащий Elem.

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

Назначение: оператор объединения множества и элемента. Данный оператор аналогичен методу удаления элемента из множества.

Входные параметры: **Elem** – число.

Выходные параметры: исходный экземпляр класса, не содержащий **Elem**.

Заключение

В результате данной лабораторной работы были изучены основы битовых полей и множеств, а также принципы их использования в программировании. Были проведены эксперименты с различными наборами данных, чтобы проверить работу программы и изучить ее производительность. Проведенный анализ результатов показал, что использование битовых полей и множеств может быть очень полезным в решении определенных задач. В целом, лабораторная работа помогла понять основные принципы работы с битовыми полями и множествами.

Литература

1. Битовое поле [https://metanit.com/c/tutorial/6.7.php].
2. Битовое поле [https://en.wikipedia.org/wiki/Bit_field].
3. Битовые поля [https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-bit-fields?view=msvc-170].

Приложения

Приложение А. Реализация класса TBitField

```
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 ("Bitfield size less than 0");
    }
}

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

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

int TBitField::GetMemIndex(const int n) const
{
    return n >> shiftSize;
}

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("bit position is out of range");
    pMem[GetMemIndex(n)] |= GetMemMask(n);
}

void TBitField::ClrBit(const int n)
{
    if (n >= BitLen || n < 0)
        throw("bit position is out of range");
    pMem[GetMemIndex(n)] &= ~GetMemMask(n);
```

```
}

int TBitField::GetBit(const int n) const {
    if (n >= BitLen || n < 0)
        throw("bit position is out of range");
    return (pMem[GetMemIndex(n)] & GetMemMask(n)) ? 1 : 0;
}

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];
    }
    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 = max(BitLen, bf.BitLen);
    TBitField tmp(len);
    for (int i = 0; i < tmp.MemLen; i++)
        tmp.pMem[i] = pMem[i] | bf.pMem[i];
    return tmp;
}

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

TBitField TBitField::operator~(void)
{
    TBitField tmp(BitLen);
    for (int i = 0; i < BitLen; i++)
        if (GetBit(i)==0)
            tmp.SetBit(i);
    return tmp;
}
```

```
 istream& operator>>(std::istream& in, TBitField& bf) {
    string answer;
    in >> answer;
    if (answer.length() > bf.BitLen)  throw "out of range";
    for (int i = 0; i < bf.BitLen; i++) {
        if (answer[bf.BitLen - 1 - i] == '1') {
            bf.SetBit(i);
        }
    }
    return in;
}

ostream& operator<<(ostream& ostr, const TBitField& bf)
{
    for (int i = 0; i < bf.GetLength(); ++i)
    {
        ostr << bf.GetBit(i) << " ";
    }
    return ostr;
}
```

Приложение Б. Реализация класса TSet

```
#include "tset.h"

TSet::TSet(int mp) :MaxPower(mp), BitField(mp) {}
TSet::TSet(const TSet& s) : BitField(s.BitField),MaxPower(s.GetMaxPower()) {}
TSet::TSet(const TBitField& bf) :MaxPower(bf.GetLength()), BitField(bf) {}
int TSet::GetMaxPower(void) const { return MaxPower;}
TSet::operator TBitField(){ return BitField;}

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

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

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

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

int TSet::operator==(const TSet& s) const
{
    if (MaxPower != s.GetMaxPower())
    {  return 0;}
    return (BitField == s.BitField);
}

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

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

```
    TSet tmp(max(MaxPower, s.GetMaxPower()));
    tmp.BitField = BitField | s.BitField;
    return tmp;
}

TSet TSet::operator+(const int Elem)
{
    if (Elem >= MaxPower || Elem < 0)  throw ("Elemet is out of universe");
    TSet tmp(*this);
    tmp.InsElem(Elem);
    return tmp;
}

TSet TSet::operator-(const int Elem)
{
    if (Elem >= MaxPower || Elem < 0)  throw ("Elemet is out of universe");
    TSet tmp(*this);
    tmp.DelElem(Elem);
    return tmp;
}

TSet TSet::operator*(const TSet& s)
{
    TSet tmp(max(MaxPower, s.GetMaxPower()));
    tmp.BitField = BitField & s.BitField;
    return tmp;
}

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

 ostream& operator<<(ostream& ostr, const TSet& s)
 {
     const int x = s.MaxPower - 1;
     for (int i = 0; i <= x; ++i)
     {
         ostr << s.IsMember(i) << " ";
     }
     return ostr;
 }

 istream& operator>>(std::istream& istr, TSet& s) {
     int elemt;
     int n;
     cout << "Enter the number of positions you want to insert:";
     cin >> n;
     for (int i = 0; i < n; i++)
     {
    istr>>elemt;
    s.InsElem(elemt);
     }
     return istr;
}
```