МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №1**

**«Структура хранения множества»**

**Выполнил:** студент группы 381706-2

Жбанова Надежда Сергеевна

\_\_\_\_\_\_\_\_\_\_\_ Подпись

**Руководитель:**

Ассистент кафедры МОСТ

Лебедев Илья Геннадьевич

\_\_\_\_\_\_\_\_\_\_\_ Подпись

Нижний Новгород

2018

Содержание

[1. Введение 3](#_Toc531791906)

[2. Постановка задачи 4](#_Toc531791907)

[2.1. Используемые инструменты 4](#_Toc531791908)

[3. Руководство пользователя 6](#_Toc531791909)

[4. Руководство программиста 8](#_Toc531791910)

[4.1. Описание структуры программы 8](#_Toc531791911)

[4.2. Описание функций и процедур, их алгоритмов 8](#_Toc531791912)

[5. Эксперименты 13](#_Toc531791913)

[6. Заключение 14](#_Toc531791914)

[7. Литература 15](#_Toc531791915)

[8. Приложения 16](#_Toc531791916)

[8.1. Приложение 1:Класс TBitField 16](#_Toc531791917)

[8.2. Приложение 2:Класс TSet 20](#_Toc531791918)

[8.3. Приложение 3:Код программы тестирования и экспериментов 22](#_Toc531791919)

[8.4. Приложение 4:Тесты для классов 25](#_Toc531791920)

# 1. Введение

Множество – это фундаментальное понятие не только математики, но и всего окружающего мира. Возьмите прямо сейчас в руку любой предмет. Вот вам и множество, состоящее из одного элемента.

В широком смысле, множество – это совокупность объектов (элементов), которые понимаются как единое целое (по тем или иным признакам, критериям или обстоятельствам). Причём, это не только материальные объекты, но и буквы, цифры, теоремы, мысли, эмоции и т.д.

Множество может быть полным и пустым, упорядоченным и неупорядоченным, счётным и несчётным, конечным и бесконечным. Множества используются во многих математических дисциплинах, отсюда вытекает необходимость представления множества на ЭВМ. А в некоторых случаях работать с множествами проще и удобнее чем с другими типами данных.

# 2. Постановка задачи

**Цель данной работы** — разработка структуры данных для хранения множеств с использованием битовых полей, а также освоение таких инструментов разработки программного обеспечения, как система контроля версий [Git](https://git-scm.com/book/ru/v2) и фрэймворк для разработки автоматических тестов [Google Test](https://github.com/google/googletest).

Предполагается, что перед выполнением работы студенты получают данный проект-шаблон, содержащий следующее:

* Интерфейсы классов битового поля и множества (h-файлы)
* Готовый набор тестов для каждого из указанных классов
* Пример использования класса битового поля и множества для решения задачи поиска простых чисел с помощью алгоритма ["Решето Эратосфена"](http://habrahabr.ru/post/91112)

Выполнение работы предполагает решение следующих задач:

1. Реализация класса битового поля TBitField согласно заданному интерфейсу.
2. Реализация класса множества TSet согласно заданному интерфейсу.
3. Обеспечение работоспособности тестов и примера использования.
4. Реализация нескольких простых тестов на базе Google Test.
5. Публикация исходных кодов в личном репозитории на GitHub.

## 2.1. Используемые инструменты

* Система контроля версий [Git](https://git-scm.com/book/ru/v2). Рекомендуется использовать один из следующих клиентов на выбор студента:
  + [Git](https://git-scm.com/downloads)
  + [GitHub Desktop](https://desktop.github.com/)
* Фреймворк для написания автоматических тестов [Google Test](https://github.com/google/googletest). Не требует установки, идет вместе с проектом-шаблоном.
* Среда разработки Microsoft Visual Studio (2008 или старше).
* Опционально. Утилита [CMake](http://www.cmake.org/) для генерации проектов по сборке исходных кодов. Она может быть использована для генерации решения для среды разработки, отличной от Microsoft Visual Studio 2008 или 2010, предоставленных в данном проекте-шаблоне.

# 3. Руководство пользователя

Реализовано решето Эратосфена - поиск подмножества, состоящего из простых чисел.

Запускаем программу (Рис.1):

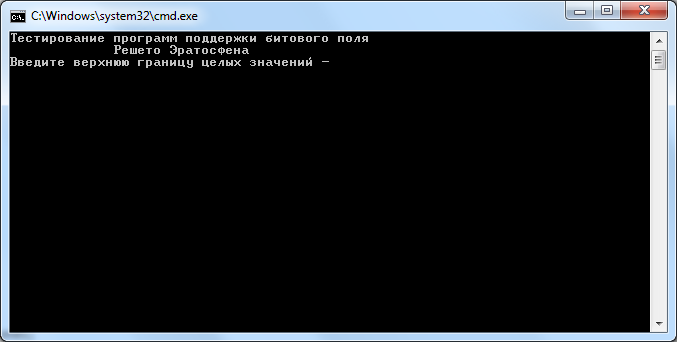


Рис.1. Начало

В появившемся окне консоли пользователю необходимо ввести верхнюю границу целых значений множества (Рис.2):

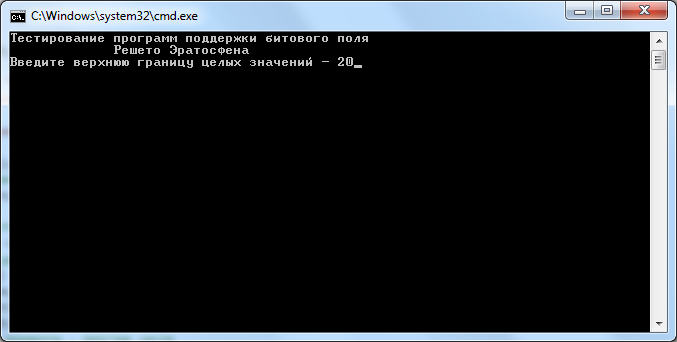


Рис.2. Выбор границы значений

Программа выведет множество некратных чисел и простые числа из заданного промежутка (Рис.3):

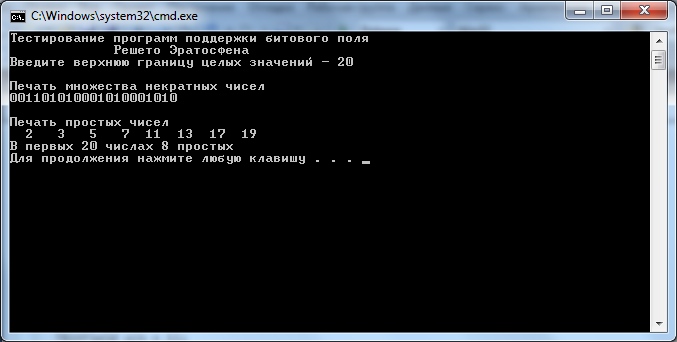


Рис.3. Завершение

Алгоритм можно повторить несколько раз.

# 4. Руководство программиста

Разработка системы вычисления проводились в среде “Microsoft Visual Studio 2010”.

Для реализации алгоритмов будет использовано 2 класса:

* Класс «Битовое поле» (TBitField), реализованный с использованием массива
* Класс «Множество» (TSet), реализованный с использованием класса битового поля.

## 4.1. Описание структуры программы

Модульная структура программы:

1. tbitfield.h, tbitfield.cpp – модуль с классом, реализующим операции над *Битовыми полями*.
2. tset.h, tset.cpp – модуль с классом, реализующим обработку *Множеств*.

* sample\_prime\_numbers.cpp , sample\_performance\_check.cpp, sample\_standart\_input\_output.cpp – модуль программы тестирования, с которым работает пользователь ( решето Эратосфена ), в котором проводятся эксперименты и приводятся работы стандартных потоков ввода и вывода.

1. test\_main.cpp, test\_tbitfield.cpp, test\_tset.cpp – модуль с функциями тестирования для созданных классов. Содержат 25 тестов для класса TBitField и 23 теста для класса TSet.

## 4.2. Описание функций и процедур, их алгоритмов

Рассмотрим реализацию методов базового класса TBitField:

class TBitField

1. TBitField(int len) – конструктор класса, принимающий длину битового поля.

Сначала в нем проверяется не равна ли длина нулю, если равна – бросается исключение. Дальше вычисляется размер массива для записи битового поля данного размера. Выделяется память под динамический массив типа unsigned int, который заполняется нулями.

1. TBitField(const TBitField &bf) – конструктор копирования.

Длина поля равна длине поля копируемого объекта класса. Динамический массив равен динамическому массиву копируемого объекта класса.

1. ~TBitField()– деструктор класса.

Удаляется динамический массив.

1. int GetMemIndex(const int n) const – функция, возвращающая индекс в динамическом массиве для бита n.
2. TELEM GetMemMask (const int n) const – функция, возвращающая бит (0 или 1) бита n. Зависит от самого бита.
3. int GetLength(void) const – функция, возвращающая длину битового поля.
4. void SetBit(const int n) – функция, устанавливающая бит (устанавливает 1) в бит n. Если бит n не принадлежит нашему битовому полю - бросается исключение.
5. int GetBit(const int n) const – функция, возвращающая значение бита n.

Если бит n = 0, то возвращает 0. Если бит n = 1, то возвращает 1. Если бит n не принадлежит нашему битовому полю - бросается исключение.

1. void ClrBit(const int n) – функция, устанавливающая бит (устанавливает 0) в бит n.

Если бит n не принадлежит нашему битовому полю - бросается исключение.

1. int operator==(const TBitField &bf) – функция, сравнивающая два битовых поля.

Если битовые поля двух объектов класса равны, то возвращает true. Иначе возвращает false.

1. int operator!=(const TBitField &bf) – функция, сравнивающая два битовых поля.

Если битовые поля двух объектов класса не равны, то возвращает true. Иначе возвращает false.

1. TBitField& operator=(const TBitField &bf) – функция, присваивающая поля.

Присваивает полю первого объекта класса поле второго объекта класса.

1. TBitField operator|(const TBitField &bf) – функция, возвращающая новый объект класса, основанный на двух других.

Возвращает объект класса, основанный на двух других. Если хотя бы в одном классе на данной позиции бит в битовом поле равен 1, то и в новом объекте класса бит на этой позиции будет равен 1. Иначе бит на этой позиции будет равен 0.

1. TBitField operator&(const TBitField &bf) – функция, возвращающая новый объект класса, основанный на двух других.

Возвращает объект класса, основанный на двух других. Если биты двух классов на одной и той же позиции в битовом поле равны 1, то в битовом поле нового объекта на этой позиции бит будет равен 1. Иначе бит на этой позиции будет равен 0.

1. TBitField operator~(void) – функция, меняющая битовое поле данного объекта.

Если бит на данной позиции битового поля равен 1, то после применения этой функции, бит будет равен 0. Если бит на данной позиции битового поля равен 0, то после применения этой функции, бит будет равен 1.

1. friend istream &operator>>(istream &istr, TBitField &bf) – функция ввода битового поля.

Пользователь с клавиатуры вводит битовое поле заданной ранее длины для данного объекта.

1. friend ostream &operator<<(ostream &ostr, const TBitField &bf) – функция вывода битового поля в консоль.

Выводит битовое поле данного объекта в консоль.

Рассмотрим реализацию методов базового класса TSet:

class TSet:

1. TSet(int mp) – конструктор класса. Принимает максимальный возможный элемент множества.

Вызывает конструктор класса TBitField, задавая длину битового поля mp.

1. TSet(const TSet &s) – конструктор копирования.

Вызывает конструктор копирования класса TBitField.

Новому объекту присваиваются все поля копируемого объекта: длина множества, длина битового поля, битовое поле.

1. TSet(const TBitField &bf)-конструктор преобразования типа.
2. operator TBitField() – преобразование типа к битовому полю.
3. int GetMaxPower(void) const – функция, возвращающая максимальную мощность множества (длину битового поля BitLen).
4. void InsElem(const int Elem) – функция, добавляющая элемент n в множество.

Вызывается функция SetBit(Elem), добавляющая элемент в битовое поле.

1. void DelElem(const int Elem) – функция, удаляющая элемент n из множества.

Вызывается функция ClrBit(Elem), удаляющая элемент из битового поля.

1. int IsMember(const int Elem) const – функция, проверяющая наличие элемента в множестве.

Вызывается функция GetBit(Elem), добавляющая элемент в битовое поле.  
Если элемент в множестве присутствует, возвращает значение бита. Иначе бросается исключение.

1. int operator==(const TSet &s) const – функция, сравнивающая два множества.

Если два множества равны между собой, то возвращается true. Иначе возвращается false.

1. int operator!=(const TSet &s) const – функция, сравнивающая два множества.

Если два множества не равны между собой, то возвращается true. Иначе возвращается false.

1. TSet& operator=(const TSet &s) – функция, присваивающая множества.

Присваивает множеству первого объекта класса множество второго объекта класса.

1. TSet operator+(const TSet &s) – функция сложения множеств.

Возвращает новый объект класса, в котором множество – сумма множеств двух других объектов.

1. TSet operator\*(const TSet &s) – функция пересечения множеств.

Возвращает новый объект класса, в котором множество – пересечение множеств двух других объектов.

1. TSet operator~(void) – функция дополнения множества.

Множество элементов становится равно множеству элементов, не принадлежащих данному множеству.

1. TSet operator+ (const int Elem) – функция включения элемента в множество.

Если в данном множестве нет элемента Elem, то после применения этой функции элемент Elem появляется в данном множестве.

1. TSet operator- (const int Elem) – функция удаления элемента из множества.

Если в данном множестве есть элемент Elem, то после применения этой функции элемент Elem удаляется из данного множества.

1. friend istream& operator>> (istream& is, TSet & dt) – функция ввода множества.
2. friend ostream &operator<<(ostream &ostr, const TSet &bf) – функция вывода множества.

# 5. Эксперименты

В качестве примера рассмотрим перегрузку оператора присвоения для класса множества (TSet).

Теоретическая сложность выполнения алгоритма O(1).

Мы провели измерение присваивая множеству значения длиной 10, 100, …, 1000000 элементов. Ниже вы можете увидеть график зависимости времени выполнения операции присвоения от количества элементов множества. (Рис.4) По приведенным данным можно сделать вывод, что практическая сложность выполнения алгоритма равна теоретической.

Рис.4. График зависимости времени выполнения операции присвоения от количества элементов множества

По горизонтали – количество присваиваемых элементов.

По вертикали - время выполнения программы.

# 6. Заключение

В результате лабораторной работы была разработана структура данных для хранения множеств с использованием битовых полей, а также освоены такие инструменты разработки программного обеспечения, как система контроля версий [Git](https://git-scm.com/book/ru/v2) и фрэймворк для разработки автоматических тестов [Google Test](https://github.com/google/googletest).

Созданные классы были протестированы с использованием Google Tests, решена задача поиска простых чисел с помощью алгоритма ["Решето Эратосфена"](http://habrahabr.ru/post/91112), а также проведены эксперименты для сравнения теоретической и практической сложности выполнения операций на методе класса.

# 7. Литература

1. Лабораторный практикум: Учебно-методическое пособие / Мееров И.Б. [и др.] – Нижний Новгород: Нижегородский госуниверситет , 2017. – 105с.
2. Тестирование с использованием Google Test

(http://neerc.ifmo.ru/wiki/index.php?title=Тестирование\_с\_использованием\_Google\_Test#.D0.A4.D1.83.D0.BD.D0.BA.D1.86.D0.B8.D1.8F\_main.28.29)

1. Касперски К. Техника оптимизации программ. Эффективное использование памяти. — СПб.: БХВ-Петербург, 2003. — 464 с.: ил.
2. Основные принципы ООП: : инкапсуляция, наследование, полиморфизм, наследование.

(http://gos-it.wikia.com/wiki/Основные\_принципы\_ООП:\_инкапсуляция,\_наследование,\_полиморфизм)

# 8. Приложения

## 8.1. Приложение 1:Класс TBitField

|  |
| --- |
| **tbitfield.h** |
| #ifndef \_\_BITFIELD\_H\_\_  #define \_\_BITFIELD\_H\_\_  #include <iostream>  using namespace std;  typedef unsigned int TELEM;  class TBitField  {  private:  int bitLen; // длина битового поля - макс. к-во битов  TELEM \*pMem; // память для представления битового поля  int memLen; // к-во эл-тов Мем для представления бит.поля  // методы реализации  int GetMemIndex(const int n) const; // индекс в pМем для бита n  TELEM GetMemMask (const int n) const; // битовая маска для бита n  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);  };  // Структура хранения битового поля  // бит.поле - набор битов с номерами от 0 до bitLen  // массив pМем рассматривается как последовательность memLen элементов  // биты в эл-тах pМем нумеруются справа налево (от младших к старшим)  #endif |

|  |
| --- |
| **tbitfield.cpp** |
| // Битовое поле  #include "tbitfield.h"  TBitField::TBitField(int len):bitLen(len)// Подаётся на вход размер битового поля  {  if ( len <= 0 )  throw 1;  memLen = (len + 31 ) >> 5;// Вычисляется размер массива для записи битового поля данного размера  pMem = new TELEM[memLen];// Выделяется память под массив  if ( pMem!= NULL )  for (int i = 0; i < memLen; i++)// Массив заполняется нулями  pMem[i] = 0;  }  TBitField::TBitField(const TBitField &bf) // конструктор копирования  {  bitLen = bf.bitLen;  memLen = bf.memLen;  pMem = new TELEM[memLen];  if ( pMem!= NULL )  for (int i = 0; i < memLen; i++)  pMem[i] = bf.pMem[i];  }  TBitField::~TBitField()  {  delete pMem;  pMem = NULL;  }  int TBitField::GetMemIndex(const int n) const // индекс Мем для бита n  {  return n >> 5;  }  TELEM TBitField::GetMemMask(const int n) const // битовая маска для бита n  {  return 1 << (n & 31);  }  // 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 > -1) && (n < bitLen) )  pMem[GetMemIndex(n)] |= GetMemMask(n);  else  throw 1;  }  void TBitField::ClrBit(const int n) // очистить бит  {  if ( (n > -1) && (n < bitLen) )  pMem[GetMemIndex(n)] &=~GetMemMask(n);  else  throw 1;  }  int TBitField::GetBit(const int n) const // получить значение бита  {  if ( (n > -1) && (n < bitLen) )  return pMem[GetMemIndex(n)] & GetMemMask(n);  else  throw 1;  }  // битовые операции  TBitField& TBitField::operator=(const TBitField &bf) // присваивание  {  bitLen = bf.bitLen;  if ( memLen != bf.memLen )  {  memLen = bf.memLen;  if ( pMem != NULL )  delete pMem;  pMem = new TELEM[memLen];  }  if ( pMem != NULL )  for (int i = 0; i < memLen; i++)  pMem[i] = bf.pMem[i];  return \*this;  }  int TBitField::operator==(const TBitField &bf) const // сравнение  {  int res = 1;  if ( bitLen != bf.bitLen )  res = 0;  else  for (int i = 0; i < memLen; i++)  if ( pMem[i] != bf.pMem[i] )  {  res = 0;  break;  }  return res;  }  int TBitField::operator!=(const TBitField &bf) const // сравнение  {  int res = 0;  if ( bitLen != bf.bitLen )  res = 1;  else  for (int i = 0; i < memLen; i++)  if ( pMem[i] != bf.pMem[i] )  {  res = 1;  break;  }  return res;  }  TBitField TBitField::operator|(const TBitField &bf) // операция "или"  {  int len = bitLen;  if ( bf.bitLen > len )  len = bf.bitLen;  TBitField temp(len);  for (int i = 0; i < memLen; i++)  temp.pMem[i] = pMem[i];  for (int i = 0; i < bf.memLen; i++)  temp.pMem[i] |= bf.pMem[i];  return temp;  }  TBitField TBitField::operator&(const TBitField &bf) // операция "и"  {  int len = bitLen;  if ( bf.bitLen > len )  len = bf.bitLen;  TBitField temp(len);  for (int i = 0; i < memLen; i++)  temp.pMem[i] = pMem[i];  for (int i = 0; i < bf.memLen; i++)  temp.pMem[i] &= bf.pMem[i];  return temp;  }  TBitField TBitField::operator~(void) // отрицание  {  int len = bitLen;  TBitField temp(len);  for (int i = 0; i < memLen; i++)  temp.pMem[i] = ~pMem[i];  TBitField res(len);  for (int i = 0; i < bitLen; i++)  if ( temp.GetBit(i) != 0 )  res.SetBit(i);  return res;  }  // ввод/вывод  istream &operator>>(istream &istr, TBitField &bf) // ввод  {  char ch;  for (int i = 0; i < bf.GetLength(); i++)  {  istr >> ch;  if ( ch == '0' )  bf.ClrBit(i);  else if ( ch == '1' )  bf.SetBit(i);  else  throw(1);  }  return istr;  }  ostream &operator<<(ostream &ostr, const TBitField &bf) // вывод  {  int len = bf.GetLength();  for (int i = 0; i < len; i++)  if( bf.GetBit(i) )  ostr << '1';  else ostr << '0';  return ostr;  } |

## 8.2. Приложение 2:Класс TSet

|  |
| --- |
| **tset.h** |
| // Множество  #ifndef \_\_SET\_H\_\_  #define \_\_SET\_H\_\_  #include "tbitfield.h"  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 &bf);  friend ostream &operator<<(ostream &ostr, const TSet &bf);  };  #endif |

|  |
| --- |
| **tset.cpp** |
| // Множество - реализация через битовые поля  #include "tset.h"  TSet::TSet(int mp) : maxPower(mp), bitField(mp)  {  }  // конструктор копирования  TSet::TSet(const TSet &s) : maxPower(s.maxPower), bitField(s.bitField)  {  }  // конструктор преобразования типа  TSet::TSet(const TBitField &bf) : maxPower(bf.GetLength()), bitField(bf)  {  }  //преобразование типа к TBitField  TSet::operator TBitField()  {  TBitField temp( this->bitField );  return temp;  }  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  {  bitField.SetBit(Elem);  }  void TSet::DelElem(const int Elem) // исключение элемента множества  {  bitField.ClrBit(Elem);  }  // теоретико-множественные операции  TSet& TSet::operator=(const TSet &s) // присваивание  {  bitField = s.bitField;  maxPower = s.maxPower;  return \*this;  }  int TSet::operator==(const TSet &s) const // сравнение  {  return bitField == s.bitField;  }  int TSet::operator!=(const TSet &s) const // сравнение  {  return bitField != s.bitField;  }  TSet TSet::operator+(const TSet &s) // объединение  {  TSet temp (bitField | s.bitField);  return temp;  }  TSet TSet::operator+(const int Elem) // объединение с элементом  {  TSet temp (bitField);  temp.bitField.SetBit(Elem);  return temp;  }  TSet TSet::operator-(const int Elem) // разность с элементом  {  TSet temp (bitField);  temp.bitField.ClrBit(Elem);  return temp;  }  TSet TSet::operator\*(const TSet &s) // пересечение  {  TSet temp (bitField & s.bitField);  return temp;  }  TSet TSet::operator~(void) // дополнение  {  TSet temp (~bitField);  return temp;  }  // перегрузка ввода/вывода  istream &operator>>(istream &istr, TSet &s) // ввод  {  // формат ввода 0 1 4...  // для прекращения ввода введите любой символ не являющийся цифрой или пробелом  int temp;  for (int i = 0; i<s.GetMaxPower();i++)  {  istr >> temp;  s.InsElem(temp);  }  return istr;  }  ostream& operator<<(ostream &ostr, const TSet &s) // вывод  {  for (int i = 0; i < s.GetMaxPower(); i++)  if ( s.IsMember(i) )  ostr << i << ' ';  return ostr;  } |

## 8.3. Приложение 3:Код программы тестирования и экспериментов

|  |
| --- |
| **sample\_performance\_check.cpp** |
| // Тестирование битового поля и множества  #include <iomanip>  // #define USE\_SET // Использовать класс TSet,  // закоментировать, чтобы использовать битовое поле  #ifndef USE\_SET // Использовать класс TBitField  #include "tbitfield.h"  int main()  {  int n, m, k, count;  setlocale(LC\_ALL, "Russian");  cout << "Тестирование программ поддержки битового поля" << endl;  cout << " Решето Эратосфена" << endl;  cout << "Введите верхнюю границу целых значений - ";  cin >> n;  TBitField s(n + 1);  // заполнение множества  for (m = 2; m <= n; m++)  s.SetBit(m);  // проверка до sqrt(n) и удаление кратных  for (m = 2; m \* m <= n; m++)  // если m в s, удаление кратных  if (s.GetBit(m))  for (k = 2 \* m; k <= n; k += m)  if (s.GetBit(k))  s.ClrBit(k);  // оставшиеся в s элементы - простые числа  cout << endl << "Печать множества некратных чисел" << endl << s << endl;  cout << endl << "Печать простых чисел" << endl;  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 << "В первых " << n << " числах " << count << " простых" << endl;  }  #else  #include "tset.h"  int main()  {  int n, m, k, count;  setlocale(LC\_ALL, "Russian");  cout << "Тестирование программ поддержки множества" << endl;  cout << " Решето Эратосфена" << endl;  cout << "Введите верхнюю границу целых значений - ";  cin >> n;  TSet s(n + 1);  // заполнение множества  for (m = 2; m <= n; m++)  s.InsElem(m);  // проверка до sqrt(n) и удаление кратных  for (m = 2; m \* m <= n; m++)  // если м в s, удаление кратных  if (s.IsMember(m))  for (k = 2 \* m; k <= n; k += m)  if (s.IsMember(k))  s.DelElem(k);  // оставшиеся в s элементы - простые числа  cout << endl << "Печать множества некратных чисел" << endl << s << endl;  cout << endl << "Печать простых чисел" << endl;  count = 0;  k = 1;  for (m = 2; m <= n; m++)  if (s.IsMember(m))  {  count++;  cout << setw(3) << m << " ";  if (k++ % 10 == 0)  cout << endl;  }  cout << endl;  cout << "В первых " << n << " числах " << count << " простых" << endl;  }  #endif |

|  |
| --- |
| **sample\_performance\_check.cpp** |
| #include "tset.h"  int main()  {  int max\_count = 0;  setlocale(LC\_ALL, "Russian");  cout << "Введите количество измерений - ";  cin >> max\_count;  clock\_t time;  clock\_t average\_time;  for (unsigned size = 10; size < 10000000; size \*= 10)  {  TSet set1(size);  TSet set2(size);  average\_time = 0;  for (unsigned count = 0; count < max\_count; count++)  {  time = clock();  set1 = set2;  average\_time += clock() - time;  }  average\_time /= max\_count;  cout<< "Время: " <<average\_time<<endl;  }  return 0;  } |

|  |
| --- |
| **sample\_standart\_input\_output.cpp** |
| #include <iostream>  #include "tbitfield.h"  #include "tset.h"  using namespace std;  int main()  {  int size;  cout << "Enter Bitfield size: ";  // Введите к примеру: 5  cin >> size;  TBitField bitfield(size);  cout << "Enter Bitfield content:" << endl;  // Введите к примеру: 01010  cin >> bitfield;  cout << "This is Bitfield content:" << endl;  cout << bitfield << endl << endl;  // Вы должны увидеть: 01010  cout << "Enter Set size: ";  // Введите к примеру: 10  cin >> size;  TSet set(size);  cout << "Enter Set content:" << endl;  // Введите к примеру: 1 3 6 ;  cin >> set;  cout << "This is Set content:" << endl;  cout << set << endl;  // Вы должны увидеть: 1 3 6  return 0;  } |

## 8.4. Приложение 4:Тесты для классов

|  |
| --- |
| **test\_main.cpp** |
| #include <gtest.h>  int main(int argc, char \*\*argv) {  ::testing::InitGoogleTest(&argc, argv);  return RUN\_ALL\_TESTS();  } |

|  |
| --- |
| **test\_tbitfield.cpp** |
| #include "tbitfield.h"  #include <gtest.h>  TEST(TBitField, can\_create\_bitfield\_with\_positive\_length)  {  ASSERT\_NO\_THROW(TBitField bf(3));  }  TEST(TBitField, can\_get\_length)  {  TBitField bf(3);  EXPECT\_EQ(3, bf.GetLength());  }  TEST(TBitField, new\_bitfield\_is\_set\_to\_zero)  {  TBitField bf(100);  int sum = 0;  for (int i = 0; i < bf.GetLength(); i++)  {  sum += bf.GetBit(i);  }  EXPECT\_EQ(0, sum);  }  TEST(TBitField, can\_set\_bit)  {  TBitField bf(10);  EXPECT\_EQ(0, bf.GetBit(3));  bf.SetBit(3);  EXPECT\_NE(0, bf.GetBit(3));  }  TEST(TBitField, can\_clear\_bit)  {  TBitField bf(10);  int bitIdx = 3;  bf.SetBit(bitIdx);  EXPECT\_NE(0, bf.GetBit(bitIdx));  bf.ClrBit(bitIdx);  EXPECT\_EQ(0, bf.GetBit(bitIdx));  }  TEST(TBitField, throws\_when\_create\_bitfield\_with\_negative\_length)  {  ASSERT\_ANY\_THROW(TBitField bf(-3));  }  TEST(TBitField, throws\_when\_set\_bit\_with\_negative\_index)  {  TBitField bf(10);  ASSERT\_ANY\_THROW(bf.SetBit(-3));  }  TEST(TBitField, throws\_when\_set\_bit\_with\_too\_large\_index)  {  TBitField bf(10);  ASSERT\_ANY\_THROW(bf.SetBit(11));  }  TEST(TBitField, throws\_when\_get\_bit\_with\_negative\_index)  {  TBitField bf(10);  ASSERT\_ANY\_THROW(bf.GetBit(-3));  }  TEST(TBitField, throws\_when\_get\_bit\_with\_too\_large\_index)  {  TBitField bf(10);  ASSERT\_ANY\_THROW(bf.GetBit(11));  }  TEST(TBitField, throws\_when\_clear\_bit\_with\_negative\_index)  {  TBitField bf(10);  ASSERT\_ANY\_THROW(bf.ClrBit(-3));  }  TEST(TBitField, throws\_when\_clear\_bit\_with\_too\_large\_index)  {  TBitField bf(10);  ASSERT\_ANY\_THROW(bf.ClrBit(11));  }  TEST(TBitField, can\_assign\_bitfields\_of\_equal\_size)  {  const int size = 2;  TBitField bf1(size), bf2(size);  for (int i = 0; i < size; i++)  {  bf1.SetBit(i);  }  bf2 = bf1;  EXPECT\_NE(0, bf2.GetBit(0));  EXPECT\_NE(0, bf2.GetBit(1));  }  TEST(TBitField, assign\_operator\_changes\_bitfield\_size)  {  const int size1 = 2, size2 = 5;  TBitField bf1(size1), bf2(size2);  for (int i = 0; i < size1; i++)  {  bf1.SetBit(i);  }  bf2 = bf1;  EXPECT\_EQ(2, bf2.GetLength());  }  TEST(TBitField, can\_assign\_bitfields\_of\_non\_equal\_size)  {  const int size1 = 2, size2 = 5;  TBitField bf1(size1), bf2(size2);  for (int i = 0; i < size1; i++)  {  bf1.SetBit(i);  }  bf2 = bf1;  EXPECT\_NE(0, bf2.GetBit(0));  EXPECT\_NE(0, bf2.GetBit(1));  }  TEST(TBitField, compare\_equal\_bitfields\_of\_equal\_size)  {  const int size = 2;  TBitField bf1(size), bf2(size);  for (int i = 0; i < size; i++)  {  bf1.SetBit(i);  }  bf2 = bf1;  EXPECT\_EQ(bf1, bf2);  }  TEST(TBitField, or\_operator\_applied\_to\_bitfields\_of\_equal\_size)  {  const int size = 4;  TBitField bf1(size), bf2(size), expBf(size);  // bf1 = 0011  bf1.SetBit(2);  bf1.SetBit(3);  // bf2 = 0101  bf2.SetBit(1);  bf2.SetBit(3);  // expBf = 0111  expBf.SetBit(1);  expBf.SetBit(2);  expBf.SetBit(3);  EXPECT\_EQ(expBf, bf1 | bf2);  }  TEST(TBitField, or\_operator\_applied\_to\_bitfields\_of\_non\_equal\_size)  {  const int size1 = 4, size2 = 5;  TBitField bf1(size1), bf2(size2), expBf(size2);  // bf1 = 0011  bf1.SetBit(2);  bf1.SetBit(3);  // bf2 = 01010  bf2.SetBit(1);  bf2.SetBit(3);  // expBf = 01110  expBf.SetBit(1);  expBf.SetBit(2);  expBf.SetBit(3);  EXPECT\_EQ(expBf, bf1 | bf2);  }  TEST(TBitField, and\_operator\_applied\_to\_bitfields\_of\_equal\_size)  {  const int size = 4;  TBitField bf1(size), bf2(size), expBf(size);  // bf1 = 0011  bf1.SetBit(2);  bf1.SetBit(3);  // bf2 = 0101  bf2.SetBit(1);  bf2.SetBit(3);  // expBf = 0001  expBf.SetBit(3);  EXPECT\_EQ(expBf, bf1 & bf2);  }  TEST(TBitField, and\_operator\_applied\_to\_bitfields\_of\_non\_equal\_size)  {  const int size1 = 4, size2 = 5;  TBitField bf1(size1), bf2(size2), expBf(size2);  // bf1 = 0011  bf1.SetBit(2);  bf1.SetBit(3);  // bf2 = 01010  bf2.SetBit(1);  bf2.SetBit(3);  // expBf = 00010  expBf.SetBit(3);  EXPECT\_EQ(expBf, bf1 & bf2);  }  TEST(TBitField, can\_invert\_bitfield)  {  const int size = 2;  TBitField bf(size), negBf(size), expNegBf(size);  // bf = 01  bf.SetBit(1);  negBf = ~bf;  // expNegBf = 10  expNegBf.SetBit(0);  EXPECT\_EQ(expNegBf, negBf);  }  TEST(TBitField, can\_invert\_large\_bitfield)  {  const int size = 38;  TBitField bf(size), negBf(size), expNegBf(size);  bf.SetBit(35);  negBf = ~bf;  for(int i = 0; i < size; i++)  expNegBf.SetBit(i);  expNegBf.ClrBit(35);  EXPECT\_EQ(expNegBf, negBf);  }  TEST(TBitField, invert\_plus\_and\_operator\_on\_different\_size\_bitfield)  {  const int firstSze = 4, secondSize = 8;  TBitField firstBf(firstSze), negFirstBf(firstSze), secondBf(secondSize), testBf(secondSize);  // firstBf = 0001  firstBf.SetBit(0);  negFirstBf = ~firstBf;  // negFirstBf = 1110  // secondBf = 00011000  secondBf.SetBit(3);  secondBf.SetBit(4);  // testBf = 00001000  testBf.SetBit(3);  EXPECT\_EQ(secondBf & negFirstBf, testBf);  }  TEST(TBitField, can\_invert\_many\_random\_bits\_bitfield)  {  const int size = 38;  TBitField bf(size), negBf(size), expNegBf(size);  std::vector<int> bits;  bits.push\_back(0);  bits.push\_back(1);  bits.push\_back(14);  bits.push\_back(16);  bits.push\_back(33);  bits.push\_back(37);  for (unsigned int i = 0; i < bits.size(); i++)  bf.SetBit(bits[i]);  negBf = ~bf;  for(int i = 0; i < size; i++)  expNegBf.SetBit(i);  for (unsigned int i = 0; i < bits.size(); i++)  expNegBf.ClrBit(bits[i]);  EXPECT\_EQ(expNegBf, negBf);  }  TEST(TBitField, bitfields\_with\_different\_bits\_are\_not\_equal)  {  const int size = 4;  TBitField bf1(size), bf2(size);  bf1.SetBit(1);  bf1.SetBit(3);  bf2.SetBit(1);  bf2.SetBit(2);  EXPECT\_NE(bf1, bf2);  } |

|  |
| --- |
| **test\_tset.cpp** |
| #include "tset.h"  #include <gtest.h>  TEST(TSet, can\_get\_max\_power\_set)  {  const int size = 5;  TSet set(size);  EXPECT\_EQ(size, set.GetMaxPower());  }  TEST(TSet, can\_insert\_non\_existing\_element)  {  const int size = 5, k = 3;  TSet set(size);  set.InsElem(k);  EXPECT\_NE(set.IsMember(k), 0);  }  TEST(TSet, can\_insert\_existing\_element)  {  const int size = 5;  const int k = 3;  TSet set(size);  set.InsElem(k);  set.InsElem(k);  EXPECT\_NE(set.IsMember(k), 0);  }  TEST(TSet, can\_delete\_non\_existing\_element)  {  const int size = 5, k = 3;  TSet set(size);  set.DelElem(k);  EXPECT\_EQ(set.IsMember(k), 0);  }  TEST(TSet, can\_delete\_existing\_element)  {  const int size = 5, k = 3;  TSet set(size);  set.InsElem(k);  EXPECT\_GT(set.IsMember(k), 0);  set.DelElem(k);  EXPECT\_EQ(set.IsMember(k), 0);  }  TEST(TSet, compare\_two\_sets\_of\_non\_equal\_sizes)  {  const int size1 = 4, size2 = 6;  TSet set1(size1), set2(size2);  EXPECT\_EQ(1, set1 != set2);  }  TEST(TSet, compare\_two\_equal\_sets)  {  const int size = 4;  TSet set1(size), set2(size);  // set1 = set2 = {1, 3}  set1.InsElem(1);  set1.InsElem(3);  set2.InsElem(1);  set2.InsElem(3);  EXPECT\_EQ(set1, set2);  }  TEST(TSet, compare\_two\_non\_equal\_sets)  {  const int size = 4;  TSet set1(size), set2(size);  // set1 = {1, 3}  set1.InsElem(1);  set1.InsElem(3);  // set2 = {1, 2}  set2.InsElem(1);  set2.InsElem(2);  EXPECT\_EQ(1, set1 != set2);  }  TEST(TSet, can\_assign\_set\_of\_equal\_size)  {  const int size = 4;  TSet set1(size), set2(size);  // set1 = {1, 3}  set1.InsElem(1);  set1.InsElem(3);  set2 = set1;  EXPECT\_EQ(set1, set2);  }  TEST(TSet, can\_assign\_set\_of\_greater\_size)  {  const int size1 = 4, size2 = 6;  TSet set1(size1), set2(size2);  // set1 = {1, 3}  set1.InsElem(1);  set1.InsElem(3);  set2 = set1;  EXPECT\_EQ(set1, set2);  }  TEST(TSet, can\_assign\_set\_of\_less\_size)  {  const int size1 = 6, size2 = 4;  TSet set1(size1), set2(size2);  // set1 = {1, 3, 5}  set1.InsElem(1);  set1.InsElem(3);  set1.InsElem(5);  set2 = set1;  EXPECT\_EQ(set1, set2);  }  TEST(TSet, can\_insert\_non\_existing\_element\_using\_plus\_operator)  {  const int size = 4;  const int k = 3;  TSet set(size), updatedSet(size);  set.InsElem(0);  set.InsElem(2);  updatedSet = set + k;  EXPECT\_NE(0, updatedSet.IsMember(k));  }  TEST(TSet, throws\_when\_insert\_non\_existing\_element\_out\_of\_range\_using\_plus\_operator)  {  const int size = 4;  const int k = 6;  TSet set(size), updatedSet(size);  set.InsElem(0);  set.InsElem(2);  ASSERT\_ANY\_THROW(updatedSet = set + k);  }  TEST(TSet, can\_insert\_existing\_element\_using\_plus\_operator)  {  const int size = 4;  const int k = 3;  TSet set(size), updatedSet(size);  set.InsElem(0);  set.InsElem(k);  updatedSet = set + k;  EXPECT\_NE(0, set.IsMember(k));  }  TEST(TSet, check\_size\_of\_the\_combination\_of\_two\_sets\_of\_equal\_size)  {  const int size = 5;  TSet set1(size), set2(size), set3(size);  // set1 = {1, 2, 4}  set1.InsElem(1);  set1.InsElem(2);  set1.InsElem(4);  // set2 = {0, 1, 2}  set2.InsElem(0);  set2.InsElem(1);  set2.InsElem(2);  set3 = set1 + set2;  EXPECT\_EQ(size, set3.GetMaxPower());  }  TEST(TSet, can\_combine\_two\_sets\_of\_equal\_size)  {  const int size = 5;  TSet set1(size), set2(size), set3(size), expSet(size);  // set1 = {1, 2, 4}  set1.InsElem(1);  set1.InsElem(2);  set1.InsElem(4);  // set2 = {0, 1, 2}  set2.InsElem(0);  set2.InsElem(1);  set2.InsElem(2);  set3 = set1 + set2;  // expSet = {0, 1, 2, 4}  expSet.InsElem(0);  expSet.InsElem(1);  expSet.InsElem(2);  expSet.InsElem(4);  EXPECT\_EQ(expSet, set3);  }  TEST(TSet, check\_size\_changes\_of\_the\_combination\_of\_two\_sets\_of\_non\_equal\_size)  {  const int size1 = 5, size2 = 7;  TSet set1(size1), set2(size2), set3(size1);  // set1 = {1, 2, 4}  set1.InsElem(1);  set1.InsElem(2);  set1.InsElem(4);  // set2 = {0, 1, 2}  set2.InsElem(0);  set2.InsElem(1);  set2.InsElem(2);  set3 = set1 + set2;  EXPECT\_EQ(size2, set3.GetMaxPower());  }  TEST(TSet, can\_combine\_two\_sets\_of\_non\_equal\_size)  {  const int size1 = 5, size2 = 7;  TSet set1(size1), set2(size2), set3(size1), expSet(size2);  // set1 = {1, 2, 4}  set1.InsElem(1);  set1.InsElem(2);  set1.InsElem(4);  // set2 = {0, 1, 2, 6}  set2.InsElem(0);  set2.InsElem(1);  set2.InsElem(2);  set2.InsElem(6);  set3 = set1 + set2;  // expSet = {0, 1, 2, 4, 6}  expSet.InsElem(0);  expSet.InsElem(1);  expSet.InsElem(2);  expSet.InsElem(4);  expSet.InsElem(6);  EXPECT\_EQ(expSet, set3);  }  TEST(TSet, can\_intersect\_two\_sets\_of\_equal\_size)  {  const int size = 5;  TSet set1(size), set2(size), set3(size), expSet(size);  // set1 = {1, 2, 4}  set1.InsElem(1);  set1.InsElem(2);  set1.InsElem(4);  // set2 = {0, 1, 2}  set2.InsElem(0);  set2.InsElem(1);  set2.InsElem(2);  set3 = set1 \* set2;  // expSet = {1, 2}  expSet.InsElem(1);  expSet.InsElem(2);  EXPECT\_EQ(expSet, set3);  }  TEST(TSet, can\_intersect\_two\_sets\_of\_non\_equal\_size)  {  const int size1 = 5, size2 = 7;  TSet set1(size1), set2(size2), set3(size1), expSet(size2);  // set1 = {1, 2, 4}  set1.InsElem(1);  set1.InsElem(2);  set1.InsElem(4);  // set2 = {0, 1, 2, 4, 6}  set2.InsElem(0);  set2.InsElem(1);  set2.InsElem(2);  set2.InsElem(4);  set2.InsElem(6);  set3 = set1 \* set2;  // expSet = {1, 2, 4}  expSet.InsElem(1);  expSet.InsElem(2);  expSet.InsElem(4);  EXPECT\_EQ(expSet, set3);  }  TEST(TSet, check\_negation\_operator)  {  const int size = 4;  TSet set(size), set1(size), expSet(size);  // set1 = {1, 3}  set.InsElem(1);  set.InsElem(3);  set1 = ~set;  // expSet = {0, 2}  expSet.InsElem(0);  expSet.InsElem(2);  EXPECT\_EQ(expSet, set1);  }  TEST(TSet, check\_existing\_element\_substruction)  {  const int size = 4;  TSet set(size), expSet(size);  // set = {1, 3}  set.InsElem(1);  set.InsElem(3);  set = set - 1;  // expSet = {3}  expSet.InsElem(3);  EXPECT\_EQ(expSet, set);  }  TEST(TSet, check\_non\_existing\_element\_substruction)  {  const int size = 4;  TSet set(size), expSet(size);  // set = {1, 3}  set.InsElem(1);  set.InsElem(3);  set = set - 0;  // expSet = {1, 3}  expSet.InsElem(1);  expSet.InsElem(3);  EXPECT\_EQ(expSet, set);  } |