Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики

ОТЧЕТ  
по лабораторной работе

**Множества**

Выполнила студентка группы 381903-3

Клименко Ксения Евгеньевна

Проверил Лебедев И.Г.

Нижний Новгород

2020 г.

**Оглавление**

[**Введение** 4](#_Toc59118851)

[**1. Постановка учебно-практической задачи** 5](#_Toc59118852)

[**2. Руководство пользователя** 6](#_Toc59118853)

[**3. Руководство программиста** 7](#_Toc59118854)

[**3.1 Описание структур данных** 7](#_Toc59118855)

[**3.2 Описание алгоритмов** 7](#_Toc59118856)

[**3.2.1. Битовое поле** 7](#_Toc59118857)

[**3.2.2. Множество** 7](#_Toc59118858)

[**3.3 Описание структуры программы BitField** 8](#_Toc59118859)

[**3.3.1 GetLength** 8](#_Toc59118860)

[**3.3.2 SetBit** 8](#_Toc59118861)

[**3.3.3 ClrBit** 9](#_Toc59118862)

[**3.3.4 GetBit** 9](#_Toc59118863)

[**3.3.5 operator =** 9](#_Toc59118864)

[**3.3.6 operator = =** 9](#_Toc59118865)

[**3.3.7 operator** ! **=** 9](#_Toc59118866)

[**3.3.8 operator |** 10](#_Toc59118867)

[**3.3.9 operator &** 10](#_Toc59118868)

[**3.3.10 operator ~** 10](#_Toc59118869)

[**3.3.11 istream** 10](#_Toc59118870)

[**3.3.12 ostream** 10](#_Toc59118871)

[**3.4 Описание структуры программы Set** 11](#_Toc59118872)

[**3.4.1 GetMaxPower** 11](#_Toc59118873)

[**3.4.2 IsMember** 11](#_Toc59118874)

[**3.4.3 InsElem** 11](#_Toc59118875)

[**3.4.4 DelElem** 11](#_Toc59118876)

[**3.4.5 operator =** 12](#_Toc59118877)

[**3.4.6 operator = =** 12](#_Toc59118878)

[**3.4.7 operator** ! **=** 12](#_Toc59118879)

[**3.4.8 operator +** 12](#_Toc59118880)

[**3.4.9 operator +** 12](#_Toc59118881)

[**3.4.10 operator -** 12](#_Toc59118882)

[**3.4.11 operator \*** 13](#_Toc59118883)

[**3.4.12 operator ~** 13](#_Toc59118884)

[**3.4.13 istream** 13](#_Toc59118885)

[**3.4.14 ostream** 13](#_Toc59118886)

[**Заключение** 14](#_Toc59118887)

[**Список литературы** 15](#_Toc59118888)

[**Приложение A** 16](#_Toc59118889)

[**Приложение B** 17](#_Toc59118890)

[**Приложение C** 21](#_Toc59118891)

[**Приложение D** 22](#_Toc59118892)

# **Введение**

*«Множество — это большое количество, которое позволяет воспринимать себя как одно»*

*Георг Кантор*

Можно ли дать строгое математическое определение понятию множества? Оказывается, что сделать этого нельзя. Понятие множества относится к числу неопределяемых понятий. Определения математических объектов начинаются с указания рода, в который в качестве вида входят определяемые понятия.  
Раздел математики, занимающийся множествами, называется теорией множеств.

Когда-то давным-давно во всех академических дисциплинах было заложено фундаментальное убеждение — существуетединственнаябесконечность*.* Но в 1874 году довольно малоизвестный математик провёл серию революционных наблюдений, подвергавших сомнению это всеми принятое убеждение. Георг Кантор в своей публикации On a Property of the Collection of the All Real Algeraic Numbers доказал, что множество вещественных чисел «более многочисленно», чем множество натуральных чисел. Так он впервые показал, что существуют бесконечные множества разных размеров.  
С 1874 по 1897 год Кантор неистово публиковал статью за статьёй, разворачивая свою теорию абстрактных множеств в расцветающую дисциплину. Однако она была встречена упорным сопротивлением и критикой. Однако, когда начали находиться практические применения математического анализа, отношение к теории изменилось, а идеи и результаты Кантора начали получать признание. К первому десятилетию 20-го века его наблюдения, теории и публикации достигли своей кульминации — признания современной теории множеств новой, совершенно уникальной областью математики.  
  
Теория множеств — это математическая теория о точно определённых наборах (множествах) отдельных объектов, называемых членами или элементами множества.

Влияние теории множеств на развитие современной математики очень велико.

Активное применение теории множеств в современной науке приводит к необходимости создания соответствующих программных решений. Программная реализация множества может выполняться различными способами и обычно тесно связана с использованием битовых операций в выбранном языке программирования.

Данная работа посвящена изучению одного из возможных подходов к хранению и обработке множеств.

# **1. Постановка учебно-практической задачи**

Понятие множества принадлежит к числу простейших математических понятий. Чтобы определить множество нужно указать характеристическое свойство элементов, т. е. такое свойство, которым обладают все элементы этого множества.

Может случиться, что данным свойством не обладает вообще ни один предмет. Тогда говорят, что это свойство определяет пустое множество. То, что предмет X есть элемент множества М, записывают так: X ∈ М. Если каждый элемент множества А является в то же время элементом множества В, то множество А называется подмножеством, или частью, множества В. Это записывают так: A ⊂ В или В ⊃ А. Таким образом, подмножеством данного множества В является и само множество В. Пустое множество считают подмножеством всякого множества. Одной из определяющих характеристик множества является его мощность. Понятие мощности определяется как количество элементов множества. Для таких множеств в математике принята следующая форма записи: A = {a1, a2, …, an}, где А – множество, аi – элементы множества, n – мощность множества. Множество всех возможных элементов называется Универс и обычно обозначается U.

Пусть заданы два множества A = {a1, a2, …, an} и B = {b1, b2, …, bm}. Рассмотрим следующие основные операции над множествами:

1. Включение элемента в множество: A ∪{b} = {a1, a2, …, an, b}
2. Исключение элемента из множества: A \ {aj} = {a1, a2, …, aj-1, aj+1, …, an}
3. Сумма (объединение) множеств:

суммой множеств А и В называется множество С, каждый элемент которого есть элемент хотя бы одного из множеств А и В.

1. Пересечение множеств:

пересечением множеств А и В называется множество С, каждый элемент которого принадлежит обоим множествам А и В.

1. Дополнение множеств:

дополнение множества B называется множество С, каждый элемент которого является элементом А и не является элементом В.

Ставится задача создания программы, поддерживающих эффективное хранение множеств, удовлетворяющих указанным допущениям, и выполнения основных операций над множествами при помощи битового поля:

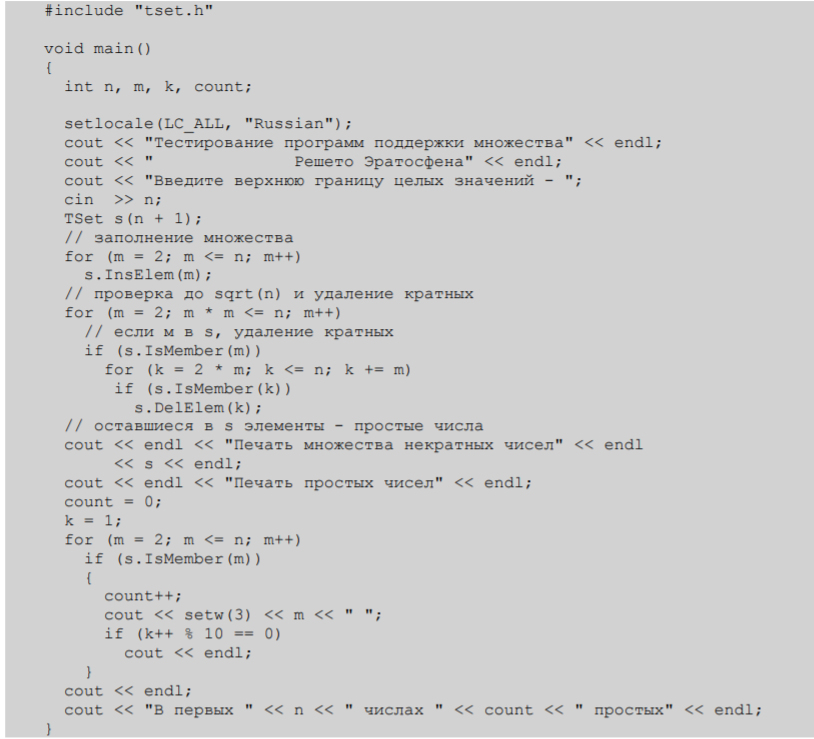
1. включение элемента в множество;
2. исключение элемента из множества;
3. проверка наличия элемента в множестве;
4. объединение множеств;
5. пересечение множеств;
6. дополнение множеств;

Для выполнения задачи нужно реализовать структуру битовые поля BitField и вспомогательный класс Set, который будет производить всю работу с множествами.

**2. Руководство пользователя**

1. Создайте переменную типа TSet
2. Чтобы использовать какую-либо доступную функцию для определенной переменной, записывайте ее через точку после вашей переменной.
3. Пользуясь операторами и функциями, вы можете написать программу для осуществления операций с множествами.

Для тестирования созданных классов можно использовать следующую программу (в выбранной среде программирования необходимо создать консольное приложение).



**3. Руководство программиста**

**3.1 Описание структур данных**

Структура данных - модель данных в виде математической структуры S = (M1, …, Mk, p1,…,pn), где M1, …, Mk – базисные множества, p1,…,pn – отношения между элементами базисных множеств. Поскольку построение математической структуры основано на понятии множества, само множество не может быть определено, как структура данных. Каждому множеству A = {a1, a2, …, an} ⊂ U = {u1, …, uk} поставим в соответствие характеристический вектор α= {α1, α2, …, αk}, где

a) k – мощность U;

b) αi = 1, если ui ∈ A;

0, если ui ∉ A.

c) ∑ αi = n.

Все операции над множествами могут быть заменены в таком случае на операции над характеристическими векторами. Таким образом, в работе будет решаться задача хранения и обработки характеристических векторов. Введение характеристического вектора для представления множества позволяет организовать хранение лишь Универса U и избежать многократного дублирования данных, содержащихся в конкретных множествах вида A. Поскольку каждый элемент характеристического вектора принимает значения из множества {0, 1}, наиболее эффективной является его реализация через битовое поле – непрерывный участок памяти, где каждый бит соответствует одному элементу вектора α. Реализацию битового поля целесообразно вынести в отдельный класс, скрывающий детали, не существенные для представления и работы с множествами.

**3.2 Описание алгоритмов**

**3.2.1. Битовое поле**

Для работы с Битовым полем предлагается реализовать следующие операции:

* установить бит;
* очистить бит;
* получить значение бита;
* сравнить два битовых поля;
* выполнить операцию “логическое или” для двух битовых полей;
* выполнить операцию “логическое и” для двух битовых полей;
* выполнить операцию “логическое отрицание” для битового поля.

**3.2.2. Множество**

Для работы с Множеством предлагается реализовать следующие операции:

* включение элемента в множество;
* исключение элемента из множества;
* проверка наличия элемента в множестве;
* сравнение множеств
* объединение множеств;
* пересечение множеств;
* дополнение множеств.

Необходимо отметить, что, операции объединения, пересечения и дополнения должны создавать новые экземпляры структуры данных Множество.

Для реализации алгоритмов будет использовано 2 класса: класс «Битовое поле» (BitField), реализованный с использованием массива и класс «Множество» (Set), реализованный с использованием класса BitField.

**3.3 Описание структуры программы BitField**

Маска- последовательность из нулей и одной единицы в позиции k.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | . . . | 1 | . . . | 0 | 0 | 0 |
| 31 | 30 |  | k |  | 2 | 1 | 0 |

2k задает маску

**Побитовые операции:**

1. Сдвиг

1<<3 (влево)

Сдвиг влево эквивалентен умножению на 2правый операнд = 23=8 (для данного примера)

Результат = 8

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| . . . | 1 | 0 | 0 | 1 |
| . . . | 3 | 2 | 1 | 0 |

1>>3(вправо)

Результат = 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  |  |  |  |
| 0 |  |  |  |  |

1. Побитовое «и»

&-логическое умножение

Для логического умножения результат должен иметь размер меньшего из полей.

1. Побитовое «или»

|-логическое сложение

Для логического сложения результат должен иметь размер большего из полей.

1. Побитовое «не»

~-логическое отрицание

**3.3.1 GetLength**

int TBitField :: GetLength (void) const

{

return BitLen;

}

Для функции с возвращающим значением типа int по получению длины поля в качестве параметров приходит пустой список параметров. Функция возвращает BitLen (длину битового поля).

**3.3.2 SetBit**

void TBitField :: SetBit (const int n)

{

if ((n >= 0) && (n < BitLen))

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

else throw -1;

}

Для ничего не возвращающей функции по установке бита в качестве параметров приходит позиция, в которой будет установлен бит. Создается маска с единицей на нужной позиции. К элементу массива, содержащему данную позицию, и маске вызывается логическое «или».

**3.3.3 ClrBit**

void TBitField :: ClrBit (const int n)

{

if ((n >= 0) && (n < BitLen))   
 pMem[GetMemIndex(n)] &= ~GetMemMask(n);  
 else throw -1;

}

Для ничего не возвращающей функции по очистке бита в качестве параметров приходит позиция, в которой будет очищен бит. Создается маска с нулем на нужной позиции. К элементу массива, содержащему данную позицию, и маске вызывается логическое «и».

**3.3.4 GetBit**

int TBitField :: GetBit (const int n) const

{

if ((n >= 0) && (n < BitLen))  
 return ((bool)(pMem[GetMemIndex(n)] & GetMemMask(n)));

else throw -1;

}

Для функции с возвращающим значением типа int по получению значения бита в качестве параметров приходит позиция, в которой необходимо узнать информацию о наличии бита. Создается маска с единицей на нужной позиции. К элементу массива, содержащему данную позицию, и маске вызывается логическое «и». В результате данная функция возвращает true, если на месте бита стоит «1» или false, если на месте бита стоит «0».

**3.3.5 operator =**

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

{

//тело функции представлено в приложении

}

Для функции типа класса по присваиванию в качестве параметров приходит постоянная ссылка на объект.

**3.3.6 operator = =**

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

{

//тело функции представлено в приложении

}

Для функции по сравнению двух битовых полей в качестве параметров приходит постоянная ссылка на объект.

**3.3.7 operator** ! **=**

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

{

//тело функции представлено в приложении

}

Для функции по сравнению двух битовых полей в качестве параметров приходит постоянная ссылка на объект.

**3.3.8 operator |**

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

{

//тело функции представлено в приложении

}

Для функции типа класса по операции логического сложения в качестве параметров приходит постоянная ссылка на объект. В результате выполнения данной функции будет возвращен результат типа класса.

**3.3.9 operator &**

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

{

//тело функции представлено в приложении

}

Для функции типа класса по операции логического умножения в качестве параметров приходит постоянная ссылка на объект. В результате выполнения данной функции будет возвращен результат типа класса.

**3.3.10 operator ~**

TBitField TBitField :: operator ~ (void)

{

//тело функции представлено в приложении

}

Для функции типа класса по операции логического отрицания не приходит никакой параметр. В результате выполнения данной функции будет возвращен результат типа класса.

**3.3.11 istream**

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

{

//тело функции представлено в приложении

}

Для функции ввода битового поля приходят 2 параметра. Ввод продолжается до тех пор, пока не появится посторонний символ (отличный от «0» и «1»).

**3.3.12 ostream**

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

{

//тело функции представлено в приложении

}

Для функции вывода битового поля приходят 2 параметра.

**3.4 Описание структуры программы Set**

Множество- подмножество U.

} одинаковые множества, т.к. не важен порядок

{a,b,c}

{b,c,a}

{a,b,a} - не является множеством, т.к. не должно быть повторов

Имеется Универс U заданной мощности. Пронумеруем элементы множества. Тогда Универс U-множество номеров заданных элементов (диапазон целых значений).

Структура хранения- машинный образ абстрактной структуры данных. Для вектора его структурой хранения является массив. Для множества его структурой хранения является битовое поле (для экономии памяти, а также потому, что элементы в множестве не повторяются, т.е. они либо существуют, либо не существуют). При написании класса «Множество» нужно использовать агрегацию, и в состав должно входить битовое поле.

**3.4.1 GetMaxPower**

int TSet::GetMaxPower(void)const

{

return MaxPower;

}

Для функции с возвращающим значением типа int по получению максимального количества элементов множества в качестве параметров приходит пустой список параметров. Функция возвращает MaxPower (максимальная мощность множества).

**3.4.2 IsMember**

int TSet::IsMember(const int Elem) const

{

return BitField.GetBit(Elem);

}

Для функции с возвращающим значением типа int по проверке наличия элемента в множестве в качестве параметров приходит постоянная позиция. Функция возвращает результат получения значения бита в соответствующей позиции битовой строки, т.е. будет возвращена истина, если в битовой строке на данной позиции стоит единица, иначе будет возвращен ноль.

**3.4.3 InsElem**

void TSet::InsElem(const int Elem)

{

BitField.SetBit(Elem);   
}

Для ничего не возвращающей функции по добавлению элемента в множество в качестве параметров приходит постоянная позиция. Функция устанавливает бит в соответствующую позицию битовой строки.

**3.4.4 DelElem**

void TSet::DelElem(const int Elem)

{

BitField.ClrBit(Elem);   
}

Для ничего не возвращающей функции по удалению элемента из множества в качестве параметров приходит постоянная позиция. Функция очищает бит в соответствующей позиции битовой строки.

**3.4.5 operator =**

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

{

//тело функции представлено в приложении

}

Для функции типа класса по присваиванию в качестве параметров приходит постоянная ссылка на объект.

**3.4.6 operator = =**

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

{

//тело функции представлено в приложении

}

Для функции с возвращающим значением типа int по сравнению двух множеств в качестве параметров приходит постоянная ссылка на объект.

**3.4.7 operator** ! **=**

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

{

//тело функции представлено в приложении

}

Для функции с возвращающим значением типа int по сравнению двух множеств в качестве параметров приходит постоянная ссылка на объект.

**3.4.8 operator +**

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

{

//тело функции представлено в приложении

}

Для функции типа класса по объединению двух множеств в качестве параметров приходит постоянная ссылка на объект. С помощью операции логического «или» складываются две битовые строки. Результат объединения будет соответствовать множеству, которое могло быть получено путем операции объединения двух множеств, каждое из которых соответствует битовой строке.

**3.4.9 operator +**

TSet TSet:: operator+ (const int Elem)

{

//тело функции представлено в приложении

}

Для функции типа класса по объединению множества с элементом в качестве параметров приходит постоянная позиция. С помощью функции по добавлению элемента в множество происходит объединение с элементом.

**3.4.10 operator -**

TSet TSet:: operator- (const int Elem)

{

//тело функции представлено в приложении

}

Для функции типа класса по разности множества с элементом в качестве параметров приходит постоянная позиция. С помощью функции по удалению элемента из множества происходит вычитание элемента из множества.

**3.4.11 operator \***

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

{

//тело функции представлено в приложении

}

Для функции типа класса по пересечении двух множеств в качестве параметров приходит постоянная ссылка на объект. С помощью операции логического «и» умножаются две битовые строки. Результат пересечения будет соответствовать множеству, которое могло быть получено путем операции пересечения двух множеств, каждое из которых соответствует битовой строке.

**3.4.12 operator ~**

TSet TSet:: operator~(void)

{

//тело функции представлено в приложении

}

Для функции типа класса по дополнению одного из множеств в качестве параметров приходит пустой список параметров. С помощью операции побитового «не» отрицается битовая строка. Результат дополнения будет соответствовать множеству, которое могло быть получено путем операции дополнения множества, которое соответствует битовой строке.

**3.4.13 istream**

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

{

//тело функции представлено в приложении

}

Для функции ввода множества приходят 2 параметра. Ввод продолжается до тех пор, пока не появится элемент «}».

**3.4.14 ostream**

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

{

//тело функции представлено в приложении

}

Для функции вывода множества приходят 2 параметра.

# **Заключение**

Реализована структура битового поля BitField и вспомогательный класс Set.

Написанная программа, поддерживающая эффективное хранение множеств и удовлетворяющая указанным допущениям, верно выполняет необходимые операции.

# **Список литературы**

1. Практическая работа 1: Структуры хранения множества

http://www.itmm.unn.ru/files/2018/10/Primer-1.1.-Struktury-hraneniya-mnozhestva.pdf

2. Рабочие материалы студента по общему курсу «Методы программирования»

http://www.itmm.unn.ru/files/2019/02/Metody-programmirovaniya-1-chast-2015.pdf

3. Лабораторный практикум

http://www.unn.ru/books/met\_files/Pract\_ADS.pdf

# **Приложение A**

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 (#О2)

TELEM GetMemMask (const int n) const; // битовая маска для бита n (#О3)

public:

TBitField(int len); // (#О1)

TBitField(const TBitField &bf); // (#П1)

~TBitField(); // (#С)

// доступ к битам

int GetLength(void) const; // получить длину (к-во битов) (#О)

void SetBit(const int n); // установить бит (#О4)

void ClrBit(const int n); // очистить бит (#П2)

int GetBit(const int n) const; // получить значение бита (#Л1)

// битовые операции

int operator==(const TBitField &bf) const; // сравнение (#О5)

int operator!=(const TBitField &bf) const; // сравнение

TBitField& operator=(const TBitField &bf); // присваивание (#П3)

TBitField operator|(const TBitField &bf); // операция "или" (#О6)

TBitField operator&(const TBitField &bf); // операция "и" (#Л2)

TBitField operator~(void); // отрицание (#С)

friend istream &operator>>(istream &istr, TBitField &bf); // (#О7)

friend ostream &operator<<(ostream &ostr, const TBitField &bf); // (#П4)

};

#endif

# **Приложение B**

tbitfield.cpp

#include "tbitfield.h"

#include <cmath>

TBitField::TBitField(int len)

{

if (len > 0)

{

BitLen = len;

MemLen = int(ceil((double)BitLen / 8 \* sizeof(TELEM)));

pMem = nullptr;

pMem = new TELEM[MemLen];

if (pMem != nullptr)

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

pMem[i] = 0;

else

throw - 1;

}

else

throw - 1;

}

TBitField::TBitField(const TBitField& bf) // конструктор копирования

{

BitLen = bf.BitLen;

MemLen = bf.MemLen;

pMem = nullptr;

pMem = new TELEM[MemLen];

if (pMem != nullptr)

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

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

else

throw - 1;

}

TBitField::~TBitField()

{

if (pMem!=nullptr)

{

delete[] pMem;

pMem = nullptr;

}

}

int TBitField::GetMemIndex(const int n) const // индекс Мем для бита n

{

if (n >= 0 && n < BitLen)

return (floor((double)n / (8 \* sizeof(TELEM))));

else

throw - 1;

}

TELEM TBitField::GetMemMask(const int n) const // битовая маска для бита n

{

if (n >= 0 && n < BitLen)

return 1 << (n % (8 \* sizeof(TELEM)));

else

throw - 1;

}

// доступ к битам битового поля

int TBitField::GetLength(void) const // получить длину (к-во битов)

{

return BitLen;

}

void TBitField::SetBit(const int n) // установить бит

{

if (n >= 0 && n < BitLen)

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

else

throw - 1;

}

void TBitField::ClrBit(const int n) // очистить бит

{

if (n >= 0 && n < BitLen)

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

else

throw - 1;

}

int TBitField::GetBit(const int n) const // получить значение бита

{

if (n >= 0 && n < BitLen)

return ((bool)(pMem[GetMemIndex(n)] & GetMemMask(n)));

else

throw - 1;

}

// битовые операции

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

{

if (this != &bf)

{

BitLen = bf.BitLen;

MemLen = bf.MemLen;

delete[] pMem;

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 // сравнение

{

bool res = 1;

if (BitLen != bf.BitLen)

res = 0;

else

for (int i = 0; i < BitLen; i++)

if (GetBit(i) != bf.GetBit(i))

{

res = 0;

break;

}

return res;

}

int TBitField::operator!=(const TBitField& bf) const // сравнение

{

bool res = 1;

if (BitLen == bf.BitLen)

for (int i = 0; i < BitLen; i++)

if (GetBit(i) == bf.GetBit(i))

res = 0;

else

{

res = 1;

break;

}

return res;

}

TBitField TBitField::operator|(const TBitField& bf) // операция "или"

{

int len = BitLen;

if (bf.BitLen > len)

len = bf.BitLen;

TBitField tmp(len);

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

tmp.pMem[i] |= pMem[i];

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

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

return tmp;

}

TBitField TBitField::operator&(const TBitField& bf) // операция "и"

{

int len = BitLen;

if (bf.BitLen > len)

len = bf.BitLen;

TBitField tmp(len);

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

tmp.pMem[i] = pMem[i];

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

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

return tmp;

}

TBitField TBitField::operator~(void) // отрицание

{

int len = BitLen;

TBitField tmp(len);

for (int i = 0; i < len; i++)

if (this->GetBit(i) == 0)

tmp.SetBit(i);

else

tmp.ClrBit(i);

return tmp;

}

// ввод/вывод

//Формат данных - последовательность из 0 и 1 без пробелов

//Начальные пробелы игнорируются

//Если не получены 0 или 1, то ввод завершается

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

{

char bit;

int i = 0;

//поиск

do { istr >> bit; } while (bit != ' ');

//ввод элементов и включение в множество

while (true)

{

istr >> bit;

if (bit == '0')

bf.ClrBit(i++);

else if (bit == '1')

bf.SetBit(i++);

else

break;

}

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;

}

# **Приложение C**

tset.h

#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

# **Приложение D**

tset.cpp

#include "tset.h"

TSet::TSet(int mp) : BitField(mp)

{

MaxPower = mp;

}

// конструктор копирования

TSet::TSet(const TSet& s) : BitField(s.BitField)

{

MaxPower = s.MaxPower;

}

// конструктор преобразования типа

TSet::TSet(const TBitField& bf) : BitField(bf)

{

MaxPower = bf.GetLength();

}

TSet::operator TBitField()

{

TBitField tmp(this->BitField);

return tmp;

}

int TSet::GetMaxPower(void) const // получить макс. к-во эл-тов

{

return MaxPower;

}

int TSet::IsMember(const int Elem) const // элемент множества?

{

return BitField.GetBit(Elem);

}

void TSet::InsElem(const int Elem) // включение элемента множества

{

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

{

if (this != &s)

{

MaxPower = s.MaxPower;

BitField = s.BitField;

}

return \*this;

}

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

{

return BitField == s.BitField;

}

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

{

return BitField != s.BitField;

}

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

{

TSet tmp(BitField | s.BitField);

return tmp;

}

TSet TSet::operator+(const int Elem) // объединение с элементом

{

TSet tmp(\*this);

tmp.InsElem(Elem);

return tmp;

}

TSet TSet::operator-(const int Elem) // разность с элементом

{

TSet tmp(\*this);

tmp.DelElem(Elem);

return tmp;

}

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

{

TSet tmp(BitField & s.BitField);

return tmp;

}

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

{

TSet tmp(~BitField);

return tmp;

}

// перегрузка ввода/вывода

// Формат данных { i1, i2, ..., in}

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

{

int tmp;

char bit;

do

{

istr >> bit;

} while (bit != '{');

do

{

istr >> tmp;

s.InsElem(tmp);

do

{

istr >> bit;

} while ((bit != ',') && (bit != '}'));

} while (bit != '}');

return istr;

}

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

{

char bit = ' ';

ostr << "{";

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

if (s.IsMember(i))

{

ostr << bit << ' ' << i;

bit = ',';

}

ostr << "}";

return ostr;

}