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**

**Отчет по Лабораторной работе**

**Структура хранения множеств**

**Выполнил:** студент группы 381806-2

Пасухин Дмитрий Андреевич

\_\_\_\_\_\_\_\_\_\_\_ Подпись

**Научный руководитель:**

ассистент каф. МОСТ ИИТММ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лебедев И.Г

Нижний Новгород

2019

Содержание

[1. Введение 3](#_Toc531466307)

[2. Постановка задачи 4](#_Toc531466308)

[3. Руководство пользователя 5](#_Toc531466309)

[4. Руководство программиста 7](#_Toc531466310)

[4.1 Описание структуры программы 7](#_Toc531466311)

[4.2 Описание структур данных](#_Toc531466312) 8

[4.3 Описание алгоритмов](#_Toc531466313) 11

[5. Заключение 1](#_Toc531466314)3

[6. Приложение 1](#_Toc531466315)4

# Введение

Вспомним, что любая программа является неким набором данных и команд, обрабатывающих эти данные. Одна из главных задач программиста состоит в выборе наиболее оптимальной модели хранения данных для конкретной задачи и реализации таких алгоритмов, которые будут наилучшим образом использовать доступные ресурсы. Сделать это помогает знание того, какие бывают структуры данных.

**Структура данных** - систематизированный способ организации данных и доступа к ним

В данной лабораторной мы будем говорить о множествах, построенных на битовых полях. Битовое поле ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) bit field) в [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — некоторое количество [бит](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), расположенных последовательно в [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), значение которых [процессор](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) не способен прочитать из-за особенностей аппаратной реализации. Преимущества именно такой структуры для хранения данных:

1. Ограничено место для хранения информации и можно сохранить несколько логических (истина/ложь) переменных в одном байте;
2. Интерфейсы устройств передают информацию, закодировав биты в один байт;
3. Необходим удобный доступ к отдельным битам в байте;
4. Используются объекты с длинной не кратной байту.

Следовательно, нам важен каждый бит. Для записи в битовое поле необходимы логические операции, а именно: операторы OR(|) AND(&) и NOT(-), определенных для отдельных битов, результат выполнения которых возвращает 1(true) и 0(false):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | A&B | A | B | - A | - B |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Для оптимизации так же можно использовать операции сдвига влево(<<) и вправо (>>). Выражение вида (a<<n) означает умножение a на 2n, а (a>>n) деление а на 2n.

# 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 и фрэймворк для разработки автоматических тестов Google Test.

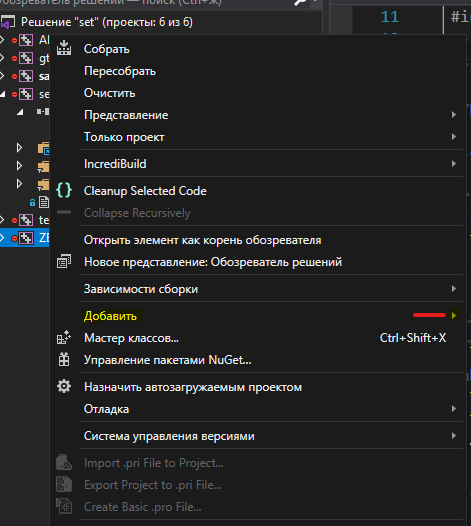
Выполнение работы предполагает решение следующих задач:

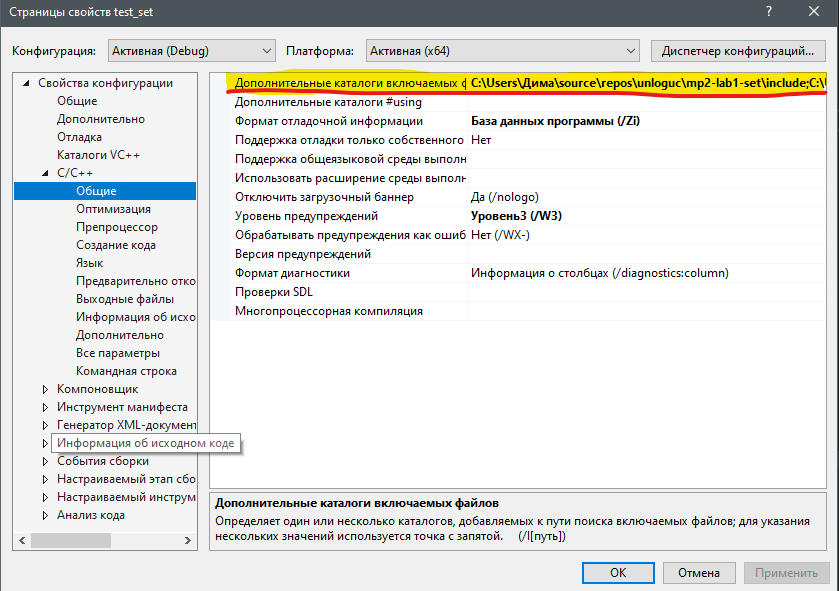
1. Реализация класса битового поля TBitField согласно заданному интерфейсу.
2. Реализация класса множества TSet согласно заданному интерфейсу.
3. Обеспечение работоспособности тестов и примера использования.
4. Реализация нескольких простых тестов на базе Google Test.
5. Публикация исходных кодов в личном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

# Руководство пользователя

В ходе лабораторной работы была получена статическая библиотека(staticlib.lib), для того чтобы ее использовать необходимо:

1. В рабочий проект добавить проект статической библиотеки set.vcxproj
2. В свойство вашего проекта указать ссылку на статическую библиотеку



1. Сделать включение заголовочного фала TSet.h, указав в свойствах своего проекта где необходимо искать заголовочный файл
2. В результате можно пользоваться классом Tset и его методами. Например, результат выполнения следующего кода:

#include "tset.h"

int main()

{

*setlocale*(*LC\_ALL*, "Russian");

TSet test(11);

test.InsElem(3);

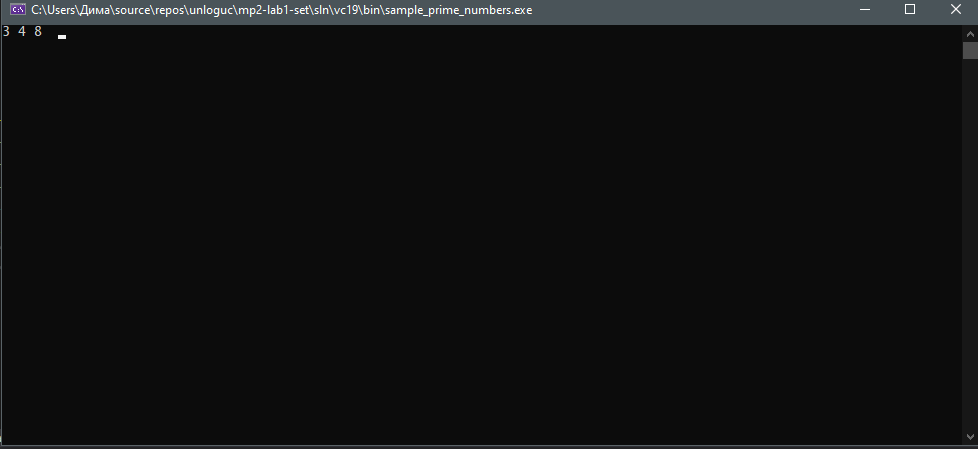
test.InsElem(8);

test.InsElem(4);

*cout* << test;

*system*("pause>nul");

}

Результат:

# Руководство программиста

## 4.1 Описание структуры программы

Программа состоит из следующих модулей:

Модуль *set* – статическая библиотека. Содержит заголовочные файлы *tbitfield.h* и *tset.h*, в которых определены соответственно интерфейсы классов битового поля *TBitField* и множеств *TSet*. Файлы, содержащие их реализацию – *tbitfield.cpp* и *tset.cpp*.

* Модуль *test\_set*. Содержит для каждого класса (*TBitField* и *TSet*) набор тестов, реализованный в файлах *test\_tbitfield.cpp* (29 тестов) и *test\_tset.cpp* (25 тестов) с помощью использования фреймворка Google Test

Модуль *sample\_prime\_numbers*. Содержит файл *sample\_prime\_numbers.cpp*, в котором реализован пример использования класса битового поля и множества для поиска простых чисел с использованием алгоритма, называемого "Решетом Эратосфена".

## Описание структур данных

Класс *Tset* реализован путем наследования класса *TBitField*

Поля класса:

* *Int MaxPower* – максимально возможный элимент;
* *TBitField BitField* битовое поле, для хранения данных;

Класс битовое поле представлено массивом из unsigned char, каждый элемент битового поля соответствует определённому биту одного из элементов массива:

**pMem[0]**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**pMem[1]**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

Т.е. каждый бит представляет один элемент множества, и элемент находится в этом множестве, если соответствующий бит равен 1. В данном случае целые значения 1 7 8 9 10 принадлежат этому битовому полю (остальные не принадлежат)

#### Класс TBitField – битовое поле

Битовое поле - это набор битов с номерами принадлежащим полуинтервалу [0,BitLen). Массив pMem рассматривается как последовательность MemLen элементов. Биты в элементах pMem номеруются справа налево(от младших к старшим).

**Элементы, объявленные со спецификатором private:**

*int bitLen* – длина битового поля;

*int memLen* – количество битов для представления битового поля;

*int \*pMem* – память для представления битового поля;

*int GetMemIndex(const int n) const* – метод определяющий индекс бита n в массиве *pMem*;

*int GetMemMask(const int n) const* – метод возвращающий маску для бита n.

**Со спецификатором доступа public:**

*TBitField(int len)* – конструктор инициализации. Принимает длину битового поля.

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

*~TBitField()* – деструктор.

**Методы доступа к битам:**

*int GetLength() const* – возвращает длину битового.

*void SetBit(const int n)* – устанавливает n-й бит битового поля.

*void ClrBit(const int n)* – очищает n-й бит битового поля.

*int GetBit(const int n) const* – возвращает значение бита n.

**Битовые операторы:**

*int operator==(const TBitField &bf) const* – принимает ссылку на объект класса TBitField, проверяет на равенство два битовых поля. Если они идентичны, то возвращает 1, иначе 0.

*int operator!=(const TBitField &bf) const* – принимает ссылку на объект класса *TBitField*, проверяет на неравенство два битовых поля. Если они различны, то возвращает 1, иначе 0.

*TBitField& operator=(const TBitField &bf)* – принимает ссылку на объект класса *TBitField*, присваивает полученное битовое поле к тому, которым был вызван оператор.

*TBitField operator|(const TBitField &bf)* – принимает ссылку на объект класса *TBitField*, возвращает новый объект класса *TBitField* полученный путем использования операции побитового «ИЛИ» между исходным и пришедшим битовыми полями.

*TBitField operator&(const TBitField &bf) –* принимает ссылку на объект класса *TBitField*, возвращает новый объект класса *TBitField* полученный путем использования операции побитового «И» между исходным и пришедшим битовыми полями.

*TBitField operator~()* – возвращает объект битового поля, полученный путем применения операции побитового отрицания для исходного битового поля.

Дружественные функции:

*friend istream &operator>>(istream &istr, TBitField &bf)* – ввод битового поля с консоли.

*friend ostream &operator<<(ostream &ostr, const TBitField &bf)* – вывод битового поля на консоль.

#### Класс TSet – множество

**Элементы, объявленные со спецификатором private:**

*int maxPower* – максимальная мощность множества.

*TBitField bitField* – битовое поле для хранения характеристического вектора.

**Со спецификатором доступа public:**

*TSet(int mp)* – конструктор-инициализатор. Принимает максимальную мощность множества.

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

*TSet(const TBitField &bf)* – конструктор преобразования типа. Принимает ссылку на объект с типом TBitField.

**Метод преобразования типа битовому полю:**

*operator TBitField()* – возвращает битовое поле характеристического вектора.

**Методы доступа к битам:**

*int GetMaxPower(void) const* – возвращает максимальную мощность множества.

*void InsElem(const int Elem)* – добавляет элемент *Elem* в множество.

*void DelElem(const int Elem)* – удаляет элемент *Elem* из множества.

*int IsMember(const int Elem) const* – проверяет наличие элемента Elem в множестве. Если элемент есть, то возвращает 1, иначе 0.

**Теоретико-множественные операторы:**

*int operator==(const TSet &s) const* – проверяет на совпадение два битовых поля. Если они идентичны, то возвращает 1, иначе 0.

*int operator!=(const TSet &s) const* – проверяет на несовпадение два битовых поля. Если они не совпадают, то возвращает 1, иначе 0.

*TSet& operator=(const TSet &s)* – присваивает полученное множество в то, которое вызвало оператор.

*TSet operator+(const int Elem*) – к исходному множеству добавляет *Elem* (если его там нет, иначе ничего не меняет) и возвращает новый объект класса *TSet*.

*TSet operator-(const int Elem)* – из исходного множества удаляет *Elem* (если он там есть, иначе ничего не меняет) и возвращает новый объект класса *TSet*.

*TSet operator+(const TSet &s)* – объединяет множество, которое вызвало оператор, с пришедшим в качестве аргумента, результат возвращается как новый объект класса *TSet*.

*TSet operator\*(const TSet &s)* – выполняет операцию пересечения двух множеств, результат возвращает как новый объект класса *TSet*.

*TSet operator~(void)* – возвращает дополнение к исходному множеству как новый элемент класса *TSet.*

**Дружественные функции:**

*friend istream &operator>>(istream &istr, TSet &bf)* – ввод множества с консоли.

*friend ostream &operator<<(ostream &ostr, const TSet &bf)* – вывод характеристического вектора множества.

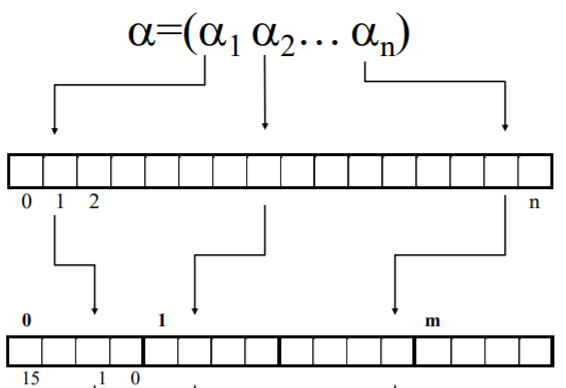
## Описание алгоритмов

Рассмотрим некоторые алгоритмы, работа которых не очевидна на первый взгляд.

**Создание битового поля.**

Необходимо представить в памяти множество натуральных чисел. Для этого будем использовать битовую строку, в которой каждый бит будет принимать либо значение «1», если элемент присутствует во входном множестве, либо «0» в противном случае.

Битовую строку будем хранить в виде массива. Заметим, что в битовой строке индексация бит происходит слева направо, в массиве аналогично, однако биты элемента массива нумеруются справа налево (Рис. 1).

Множество

Битовая строка

Битовое поле

Рис. 1 Представление битового поля.

Для удобства работы с методами получения, очищения, добавления бита и подобным в классе *TBitField* используются вспомогательные методы:

1. *GetMemIndex* – для n-го бита определяется индекс элемента m в массиве pMem, в котором хранится битовое поле. Заметим, что m является целой частью от деления номера бита n на количество битов выделяемых на один элемент массива.

|  |
| --- |
|  |

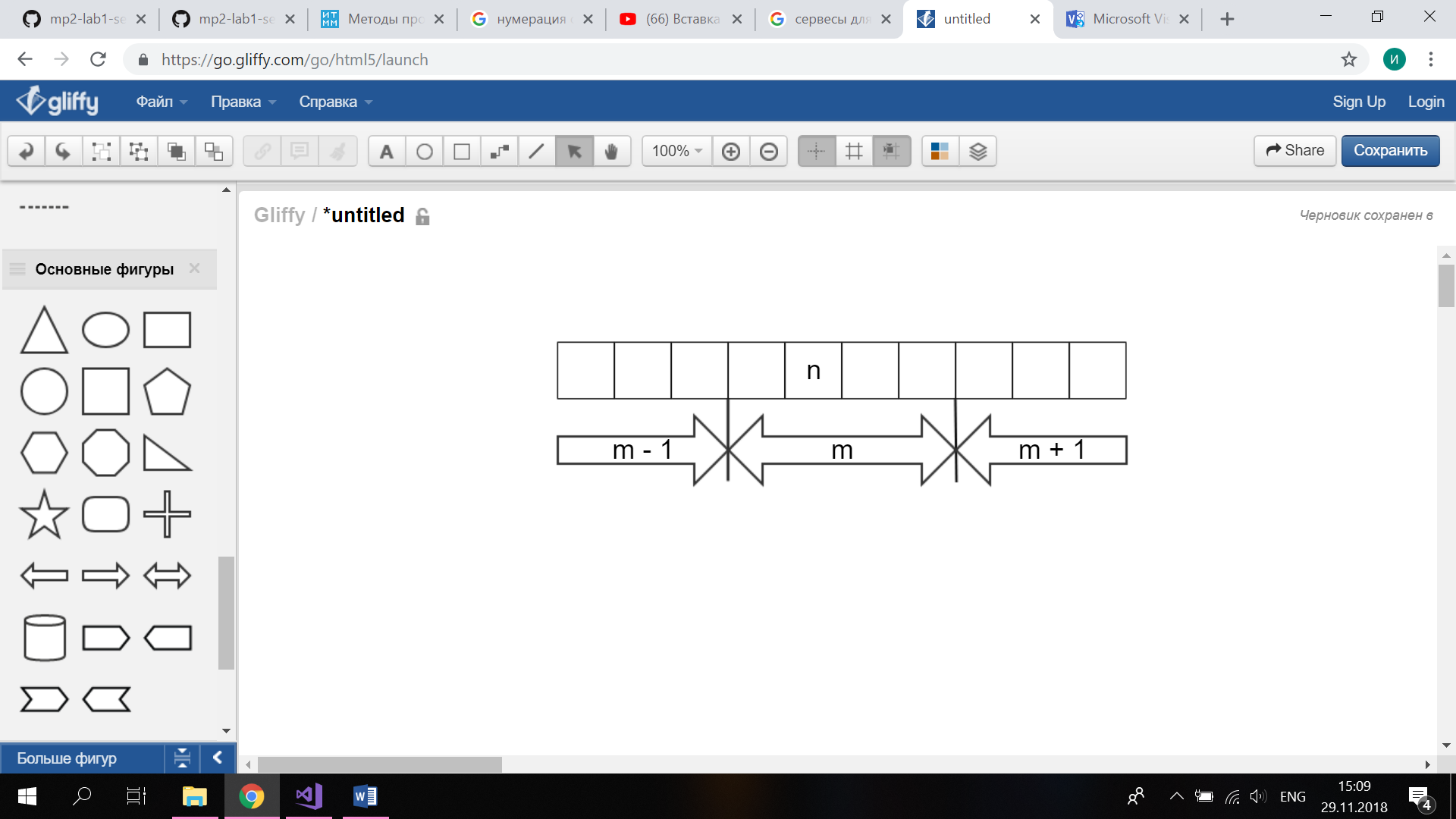


Рис 2. Пример того, где может находится бит n для элемента m.

1. *GetMemMask* – определяет битовую маску для n-го бита. Ставим значение 1 в позицию k, определяемую по формуле 2. Битовую маску рассматриваем как двоичную запись числа.

|  |
| --- |
|  |

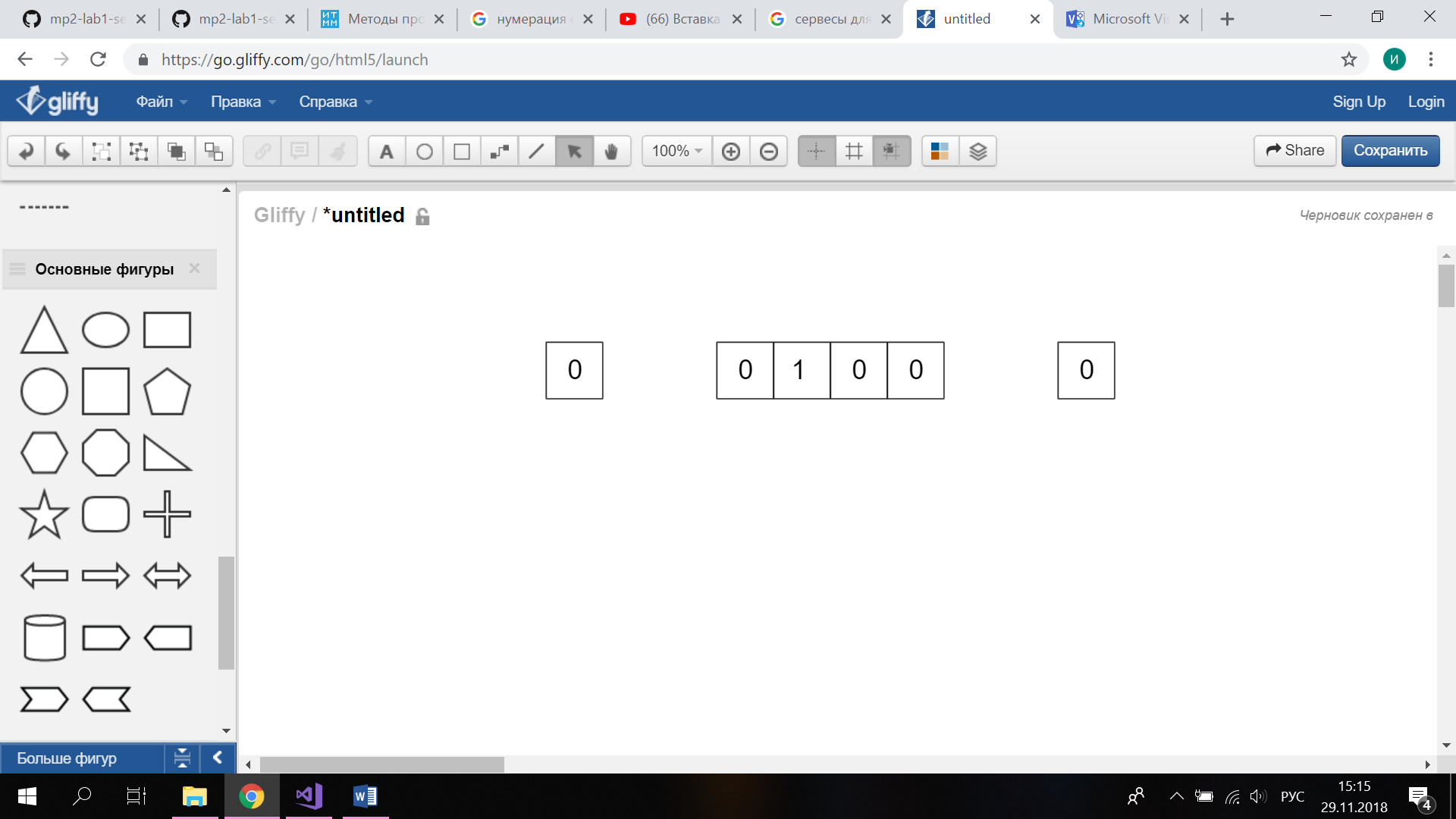


Рис. 3. Битовая маска бита n для Рис.2.

Рассмотрим непосредственно методы работы с битами.

Чтобы установить бит на позицию можем использовать побитовое «ИЛИ» между битами элемента массива *pMem*, в котором находится бит n, и между его битовой маской:

|  |
| --- |
|  |

Чтобы очистить бит n используем побитовое «И» между битами элемента массива *pMem*, в котором находится бит n, и между дополнением его битовой маски:

|  |
| --- |
|  |

Чтобы получить значение бита n используем побитовое «И» между битами элемента массива *pMem*, в котором находится бит n, и между его битовой маской:

|  |
| --- |
|  |

# Заключение

В ходе лабораторной работы был получен класс на основе которого была создана библиотека, позволяющая хранить натуральные данные. Затраты памяти на n элементов от 0 до n-1 равны (n/8 + 2) байт. Такой способ хранения выгоден, если необходимо хранить все элементы от 0 до 2n, где n может быть равен 15 и больше, например, если надо будет хранить большие по мощности числа, превосходящие 215, то чтобы сохранить одно число 219 на любой системе (тип unsigned char стандарт С++) понадобится 216 +1 байт. Но для структуры данных, в которой реализовано поэлементное хранение данных пришлось бы потратить 220\*8 байт.

|  |  |  |  |
| --- | --- | --- | --- |
| **Характеристика** | **Tset** | **Vector** | **Set** |
| Обьем памяти на элемент | O(n\*8) | (sizeof(TELEM)) | (sizeof(TELEM)) |
| Время поиска элемента | O(1) | O(log n) | O(log n) |
| Время добавление элемента | O(1) | O(n) | O(log n) |
| Время на добавление элемента без повторения | O(1) | O(nlogn) | O(log n) |

# Приложение: Фрагменты исходного кода

## TbitField.cpp

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// tbitfield.cpp - Copyright (c) Гергель В.П. 07.05.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Битовое поле

#include <vector>

#include "tbitfield.h"

TBitField::TBitField(int len) :BitLen(len), MemLen((len +7) >>3)

{

if (len < 0)

throw("Len is incorrect");

else

pMem = new TELEM[MemLen]{ 0 };

}

TBitField::TBitField(const TBitField& bf) :BitLen(bf.BitLen), MemLen(bf.MemLen) // конструктор копирования

{

pMem = new TELEM[MemLen];

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

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

}

TBitField::~TBitField()

{

delete[] pMem;

pMem = nullptr;

}

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

{

if (n < 0 || n > BitLen)

throw("index is incorrect");

else

return n >> 3;

}

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

{

if (n < 0 || n > BitLen)

throw(" index is incorrect");

else

return 1 << (n&7);

}

// 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

{

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

}

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

{

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

}

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

{

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

}

// битовые операции

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

{

if (this != &bf)

{

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 // сравнение

{

if (BitLen != bf.BitLen) return 0;

else

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

if (pMem[i] != bf.pMem[i])

return 0;

return 1;

}

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

{

if (BitLen != bf.BitLen) return 1;

else

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

if (pMem[i] == bf.pMem[i])

return 0;

return 1;

}

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

{

int len;

(bf.BitLen <= BitLen) ? (len = BitLen) : (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;

(bf.BitLen <= BitLen) ? (len = BitLen) : (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

{

TBitField tmp(BitLen);

for (int i = 0; i < tmp.MemLen - 1; i++)

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

for (int i = (tmp.MemLen - 1) \* sizeof(TELEM) \* 8; i < tmp.BitLen; i++)

{

if (GetBit(i))

tmp.ClrBit(i);

else

tmp.SetBit(i);

}

return tmp;

}

// ввод/вывод

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

{

string tmp;

istr >> tmp;

if (tmp.*size*() > bf.GetLength())

throw "Incorrect length";

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

{

if (tmp[i] == '0')

bf.ClrBit(i);

else if (tmp[i] == '1')

bf.SetBit(i);

else

throw "Incorrect input";

}

return istr;

}

*ostream*& operator<<(*ostream*& ostr, const TBitField& bf) // вывод

{

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

{

if (bf.GetBit(i) )

ostr << i << " ";

}

return ostr;

}

## TSet.cpp

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// tset.cpp - Copyright (c) Гергель В.П. 04.10.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Множество - реализация через битовые поля

#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 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

{

if (Elem <= MaxPower)

BitField.SetBit(Elem);

}

void TSet::DelElem(const int Elem) // исключение элемента множества

{

if (Elem <= MaxPower)

BitField.ClrBit(Elem);

}

// теоретико-множественные операции

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

{

BitField = s.BitField;

MaxPower = s.GetMaxPower();

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(BitField);

tmp.BitField.SetBit(Elem);

return tmp;

}

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

{

TSet tmp(BitField);

tmp.BitField.ClrBit(Elem);

return tmp;

}

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

{

TSet tmp(BitField);

tmp.BitField = tmp.BitField & s.BitField;

return tmp;

}

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

{

TSet tmp(BitField);

tmp.BitField = ~BitField;

return tmp;

}

// перегрузка ввода/вывода

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

{

istr>>s.BitField;

return istr;

}

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

{

ostr << s.BitField << ' ';

return ostr;

}