Лабораторная работа н.2

Представление вектор-столбцов и матриц в виде одномерных массивов и перегрузка оператор-функций

Петров Артем Евгеньевич

Содержание

Цель работы 3

Задание 4

Выполнение лабороторной работы 5

Описание классов vect и matr, содержащих поля int dim, doubleb (doublea), int num, static int count; 5

Конструкторы и деструктор, содержащие вывод сообщений о выполненном действии 9

Набор оператор-функций (компонентных и внешних) для операций векторной алгебры, содержащих вывод сообщений о выполненных действиях: 1(v+v, v-v, -v, vv, kv, v=v), 2(m+m, m-m, -m, mm, km, m=m), 3(m*v). 9

Функция main, содержащая сценарий работы с векторами и матрицами. . 17

Вывод 20

Цель работы

Целью лабороторной работы служит обучение создания матриц и вектор-столбцов в виде одномерных массивов и перегрузка их операторов.

Задание

Написать компьютерную программу, содержащую:

- Описание классов vect и matr, содержащb[поля int dim, doubleb (doublea), int num, static int count;
- Конструкторы и деструктор, содержащие вывод сообщений о выполненном действии;
- Набор оператор-функций (компонентных и внешних) для операций векторной алгебры, содержащих вывод сообщений о выполненных действиях:

  v+v, v-v, -v, vv, kv, v=v,

  m+m, m-m, -m, mm, km, m=m,

  m*v.

- Функцию main, содержащую сценарий работы с векторами и матрицами.

Представить результаты в виде двух файлов:

- Компьютерная программа на C++;
- Отчет о выполнении лабораторной работы с описанием алгоритма и структуры программы

Выполнение лабороторной работы

Описание классов vect и matr, содержащих поля int dim, doubleb (doublea), int num, static int count;

```
class vect
{
    int dim; // razmernost'
    double *V; // massiv

    public:
        int num;
        static int count; //schetchik vektorov

        vect(int d, double *x);
        vect(vect &b); // copy
        vect(){}; //constructor

        ~vect(){}; //destructor

        void set(int d, double *x)
        {
            dim=d;
            this->V=x;
```

```
    }

    void print()
    {
        for (int i = 0; i < dim; i++)
        {
            cout << "(";
            if (i == dim-1) cout<<this->V[i];
            else if(i < dim) cout<<this->V[i] << ",";
            cout << ")\n";
        }
    }

    vect& operator=(const vect &r); //+
    vect& operator+(const vect& r); //+
    friend vect& operator-(vect &l, vect &r); //+
    vect& operator-();
    double operator*(vect &r);
    friend vect& operator*(double k, vect& r);
    double operator[](int i);
    friend class matr;
};

class matr
{
    double *X;
    int dim;
public:
    static int countMatr; int num; double idk; //idk protiv UB, ya de bil
    matr(int d, double *x);
```

```
matr()
{
}
matr(const matr &b)
{
    cout<< "copying";
    X=new double[b.dim*b.dim];
    for (int i = 0; i < b.dim*b.dim; i++)
    {
        X[i]=b.X[i];
    }
    this->dim=b.dim;
}



void print() //hope there is no need to tell whats this
{
    int counter=0;
    cout << "\nPrintin matrix\n";
    if (this->X[dim+1] == idk) //chtobi otlovit' UB pri M*V
    {
        for (int i = 0; i < dim; i++)
        {
            cout << "\n|" << this->X[i] << "|";
        }
        return;
    }
```

```
        for (int i = 0; i < dim; i++)
        {
            cout << "|";
            for (int i=0; i < dim; i++)
            {

                cout << this->X[counter] << " ";
                counter++;
            }
            cout << "|\n";
        }

    }
    //overloading operators for class matrix
    matr& operator=(const matr &r); //+
    matr& operator+(const matr& r); //+
    friend matr& operator-(matr &l,matr &r);
    matr& operator-();
    matr& operator*(vect& r);
    matr& operator*(matr &r);
    friend matr& operator*(double k, matr& r);
    double operator[](int i);
    matr& operator=(vect *r);
};
```

Конструкторы и деструктор, содержащие вывод сообщений о выполненном действии

- Изначально я прибег к созданию пользовательского конструктора, который в будущем будет банально создавать нужные мне вектора без необходимости явно редактировать экземпляры класса, созданные конструктором компилятора. Оба конструктора непосредственно в теле класса были в виде протипа, поэтому вне классов мне пришлось указывать их поле видимости, что, честно говоря, не совсем и проблема

vect constructor vect::vect(int d, double *X) { cout << "\nVector constructor\n"; this->dim=d; this->V=X; }

matr constructor matr::matr(int d, double *x) { cout << "\nmatr Constructor\n"; dim=d; this->X=x; }

- Оба случая аналогично решаются: делаем потоковый вывод с информацией о функциях и передаем данные из передаваемых в функцию данных в данные экземпляра класса. Где d-заданная пользователем размерность, а указатель типа double-указатель на массив.

Набор оператор-функций (компонентных и внешних) для операций векторной алгебры, содержащих вывод сообщений о выполненных действиях: 1(v+v, v-v, -v, vv, kv, v=v), 2(m+m, m-m, -m, mm, km, m=m), 3(m*v).

1. v+v, v-v, -v, vv, kv, v=v

- V+V. Опять же, в самом классе есть лишь прототип, поэтому указываем поле видимости. Тип функции ссылочный для того, чтобы непосредственно

оперировать значениями в памяти. В функцию передается лишь правый вектор(константный, чтобы случайно не попал под UB). Вывводим информацию о начале выполнения функции(1). Дальше же циклом(2), с количеством итераций равным размерности вектора, присваем экземпляру класса, в котором и запускается функция, новое значение, вида: i-тый эл. массива экземпляра(через указатель this на массив экземпляра(левого вектора)) прибавить i-тый элемент правого вектора(3). Возвращаем же указатель на новоизмененный левый вектор(4). Таким образом, результат суммирования в конце сложения векторов присвоится уже левому вектору безповоротно(честно говоря, я бы мог все обернуть так, чтобы результат присваивался временной переменной экземпляра класса, но нам разрешили делать и так, не судите строго, пожалуйста, все исправлю, если скажете)

Перегрузка оператора сложения vect& vect::operator+(const vect& r) //сложение векторов { cout << "Addition of vectors\n"; //(1) for(int i=0; i<r.dim; i++) //(2) { this->V[i]=this->V[i]+r.V[i]; //(3) } return *this; //(4) }

- V-V. В случае вычитания векторов мы рассматриваем оператор, как бинарный оператор по той причине, что нахождение противоположного по знаку вектора будет бинарной. В остальном же, перегрузка в точности повторяет перегруженное сложение, только передается в этот раз оператор-фунуции сразу два адреса-левого и правого векторов. В этом случае нет необходимости указывать поле видимости по той причине, что мы “подружили” наш прототип в теле класса, что позволяет нам довериться компилятору в непростом деле по поиску прототипа по заданным параметрам.

Перегрузка оператора вычитания vect& operator-(vect& l, vect& r) //вычитание векторов { cout << "substracting vectors\n"; for (int i = 0; i < l.dim; i++) { l.V[i]=l.V[i]-r.V[i]; } return l; }

- -V. В этом случае оператору передается лишь экзепляр класса, который и вызывает его, поэтому мы лишь воспользуемся указателем this->, указав через

иттерации последовательно на все элементы массива экземпляра, умножим их при этом на -1.

Перегрузка оператора нахождения противоположного элемента vect& vect::operator-() // получение противоположного по знаку вектора { cout << "turning negative\n"; for (int i = 0; i < this->dim; i++) { this->V[i]*=-1; } return *this; }

- V * V. Скалярное произведение векторов-попарное перемножение и их сумма. Опять передаем только адрес правого вектор, но только в этот раз результатом будет лишь тип double(значение с плавающей запятой, но только с двойной точностью). Инициализируем временную переменную в теле оператор-функции, равную нуль для предотвращения UB, и результат вычисления скалярного произведения будем передавать в нее с постепенным сложением суммы прошлых результатов с текущим(1). В конце же вернем эту переменную. В будущем она будет выводится через команду cout.

```
double vect::operator*(vect &r) //скалярное произведение векторов
{
    cout << "multyplying vect by vect\n";
    double num1=0;
    for (int i = 0; i < this->dim; i++)
    {
        num1+=this->V[i]*r.V[i]; //(1)
    }
    return num1;
}
```

- k * V. В данном случае все действия имеют огромную схожесть с предыдущими, только в этот раз операция умножения будет бинарной. Левый операнд-число, правый-вектор. Просто-напросто перемножаем все элементы с помощью цикла

на данное число. Протип этой функции-оператора мы тоже “подружили”, поэтому поле видимости не указываем.

```
vect& operator*(double k, vect& r) //произведение вектора на число
{
    cout << "multyplying vect by a num\n";
    for (int i = 0; i < r.dim; i++)
    {
        r.V[i]=r.V[i]*k;
    }
    return r;
}
```

- V=V. Операция присваивания тоже повторяет предыдущие действия. Мы переприсваем значениям массива левого операнда значения правого циклом. Возвращаем же указатель но изменный левый вектор.

```
vect& vect::operator=(const vect &r) //присваивание вектору вектора
{
    cout << "assignment\n";
    for(int i=0; i < r.dim; i++)
    {
        this->V[i]=r.V[i];
    }
    return *this;
}
```

2. m+m, m-m, -m, mm, km, m=m.

В нашем представлении матрица имеет так же вид одномерного массива, поэтому все действия, кроме умножения матрицы на матрицу и матрицы на вектор принимают абсолютно схожие очертания, лишь количество иттераций в цикле принимает вид dim*dim. Надеюсь, вы поверить мне наслово, что я понимаю, что пишу uWu

- M+M. Аналогична сложению векторов, лишь итерации в кол-ве dim*dim.

```
matr& matr::operator+(const matr& r)
{
    cout << "\nsumming matrixes\n";
    for (int i = 0; i < r.dim*r.dim; i++)
    {
        this->X[i]+=r.X[i];
    }
    return *this;
}
```

- M-M. Аналогична операции вычитания векторов. Иттераций в цикле-dim*dim.

```
matr& operator-(matr &l, matr &r)
{
    cout << "\nsubstracting matrixes\n";
    for (int i = 0; i < r.dim*r.dim; i++)
    {
        l.X[i]-=r.X[i];
    }
    return l;
}
```

- -M. Все так же. Аналогия та же.

```
matr& matr::operator-()
{
    cout << "\nneg. matrix\n";
    for (int i = 0; i < this->dim*this->dim; i++)
    {
        this->X[i]*=-1;
```

```
    }
    return *this;
}
```

- M=M. Без комментариев.

```
matr& matr::operator=(const matr &r) //++++
{
    cout << "\noperator prisvaivaniya\n";
    for (int i = 0; i < r.dim*r.dim; i++)
    {
        this->X[i]=r.X[i];
    }
    return *this;
}
```

-M * V. Результатом умножения матрицы на вектор будет служить другой вектор. В этот раз создаем временный вектор, который будет сохранять результаты вычислений(1). Размерности его присваиваем значение правого вектора и создаем пустой массив из dim элементов. Теперь самое сложное-процессы вычисления. Нам придется сделать двойной цикл, где иттерации внешнего цикла будут присваивать результат вычислений для нашего вектора размера dim(2). Внутренний цикл уже послужит для последовательного сложения(первая строка матрицы умножить на вектор(для этого при каждом шаге мы будем складывать результат умножения предыдущих эл. плюс результат умножения следующих элементов) и т.д.). Однако, мы столкнемся с тем, что двойной цикл будет присваивать каждому элементу вектора значение умножения первой строки на первый элемент вектора r. Чтобы этого избежать, нам надо будет перешагивать массив матрицы на dim, т.к. это расстояние между первым эл. одной строки и первым элементом следующей строки. Итого, таких перешагиваний циклу придется сделать dim умножить на количество строк. Так как внешний цикл как раз за него и отвечает, мы будем брать первый элемент каждой строки

по результату умножения i на dim. Однако, этим мы добьемся лишь получения новой строки для следующей итерации. Чтобы пройти эту строку, нам уже поможет внутренний цикл, который уже отвечает за перемножение n-того элемента строки на n-тый элемент строки матрици. Поэтому значения из матричного массива мы будем брать по формуле dim *i+j(3).

```
matr& matr::operator*(vect& r)
{
    cout << "Multyplying left matrix by right vector";
    vect pm; //(1)
    pm.dim=r.dim;
    pm.V = new double[pm.dim]();
    for (int i = 0; i < pm.dim; i++) //(2)
        for (int j = 0; j < pm.dim; j++)
            pm.V[i] = pm.V[i] + this->X[j+pm.dim*i] * r.V[j]; //(3)

    delete this->X;//(5)
    this->X=pm.V;//(5)
    this->X[dim+1]=this->idk; (6)
    return *this;
}
```

Однако, на нашем пути встанет очередная проблема-мы не можем перевести класс матрицы в класс вектор без наследование первым второго. Т.к. мы еще не дошли до этой темы, я постараюсь обойтись без него тоже, интересно же. Теперь мы очищаем память от старого массива матриц и присваиваем экземпляру массив временного вектора(4). В описание класса встроенна переменна idk, которая присваивается следующему элементу после последнего элемента массива, поэтому при печати мы делаем отдельное ветвление на этот случай(6).

Честно говоря, я думал, стоит ли создавать в классе временную переменную класса вектора. Я посчитал, что затраты памяти в обеих случая не критично отличаются

друг от друга. Для всех других экземплярова класса матриц будет существовать бесхозная переменная, а их больше, чем всего одного экземпляра в нашем случае при проверке операции умножения матрицы на вектор. Понять и простить, я сделаль.

- M * M. Штош, в этом случае мы имеем схожую сущность с умножением матрицы на вектор, только в этот раз мы рассмотрим более сложный случай: тройной цикл, так как каждая строка будет преобразовывать уже не в один элемент, а сразу в ее изначальное количество. Теперь мы создадим временный экземпляр класса матриц с схожей размерность исходной. Только теперь мы будем шагать по каждому элементу этой матрици, коих уже не просто dim. Поэтому мы применяем схожую идею с двойным циклом и добавляем третрить во внутрь уже для прошагивания данных при инициализации двух матриц. При шаге первого цилка(внешнего) мы должны переходить на новую строку временного цикла, поэтому берем эл из массива по формуле j+pm.dimi, левая матрица повторяет все действия. Однако, правая матрица уже должна шагать через элементы намного интенсивнее-за шаг на след. элемент столбца и потом перешагивать на след. столбец. В этом нам поможет два вложенных массива. Итого элементы у правой матрицы будем брать по формуле j+h dim

```
matr& matr::operator*(matr &r)
{
    cout << "Multiplying left matrix by matrix\n";
    matr pm;
    pm.dim=r.dim;
    pm.X = new double[pm.dim*pm.dim]();
    for (int i = 0; i<pm.dim; i++)
        for (int j = 0; j<pm.dim; j++)
            for (int h = 0; h<pm.dim; h++)
        pm.X[j+pm.dim*i] = pm.X[j+pm.dim*i] + this->X[h+pm.dim*i] * r.X[h*pm.dim+j];
    *this=pm;
    return *this;
```

```
}
```

Функция main, содержащая сценарий работы с векторами и матрицами.

```
int main()
{
    //class vector
    double f[3]={1, 4, 3};
    vect first;
    first.set(3, f);

    vect second;
    double s[3]={1, 2, 3};
    second.set(3, s);

    first+second;
    first.print();
    second.print();

    first=first-second;
    first.print();

    cout << first*second << "\n";
    first.print();

    3*first;
    first.print();
```

```
cout << first[0] << "\n";

-first;
first.print();

//class matrix
double third[4]={1, 2, 3, 4};
double fourth[4]={5, 6, 7, 8};

matr third1(2, third);
matr fourth1(2, fourth);

third1.print();
third1=fourth1;

third1.print();
third1+fourth1;

third1.print();
third1-fourth1;

third1.print();

third1*fourth1;
third1.print();

double r[2]={1, 2};
vect fifth;
fifth.set(2, r);
```

```
    third1*fifth;
    third1.print();

    fourth1=fifth;
    return 0;
}
```

- Резульат работы(рис. [-@fig:001]):

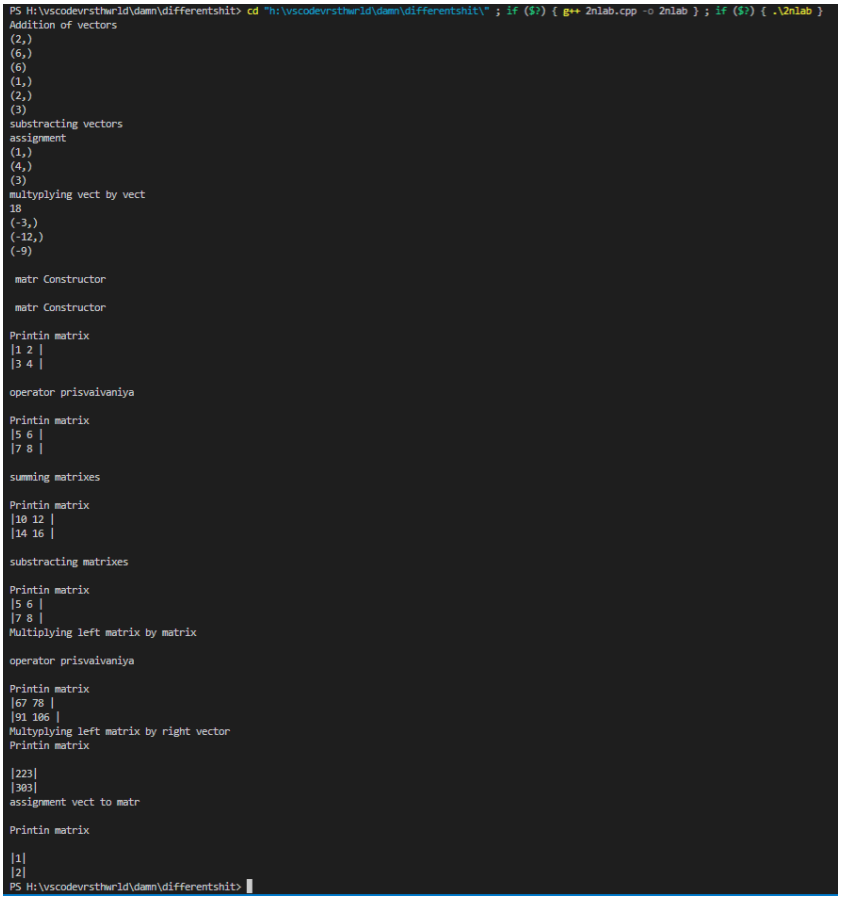

Рис. 0.1: Выполнение программы

Вывод

Благодаря данной лабораторной работе я познакомился с работой с классами и перегрузкой операторов в C++. Несмотря на то, что мой код очень похож на спагетти-код, мне довелось достаточно хорошо ознакомиться с этими разделами программирования на этом языке. В моих силах довести программу до правильной структуры, чем я и постараюсь заняться летом. Честно, последние две с половиной недели я был очень занят переездом из общежития в незнакомую мне обстановку, поэтому достаточно много дел были запущены, что не позволило мне довести программу до абсолютной выверенности в каждой строчке. Обещаю, что обязательно летом она будет переписана и привидена в идеальный вид, извините. В остальном же, программа выполняет необходимые действия и, кроме того, мне очень понравилось работать над этой лабораторной работой. Спасибо!