Лекция 20

Множественное наследование

Производный класс может иметь не один, а сразу несколько базовых классов. В этом случае говорят о *множественном наследовании*.

Продемонстрируем применение множественного наследования на примере программы для решения систем линейных алгебраических уравнений.

Сначала создаются два класса: алгебраические вектора и матрицы. Система линейных алгебраических уравнений есть совокупность матрицы системы и вектора правой части, поэтому естественно представить класс систем линейных алгебраических уравнений как производный одновременно от класса матриц и класса векторов.

Программа 20.1. Системы уравнений

Класс алгебраических векторов Vect

Представим алгебраический вектор как динамический массив с элементами типа `double`. В заголовочном файле `Vect.h` разместим объявление класса алгебраических векторов `Vect`.

```
//Файл Vect.h

#ifndef VectH
#define VectH
#include <iostream>
using namespace std;

class Vect {                          // Класс алгебраических векторов
    double* v;                        // Компоненты вектора
    int size;                         // Размер вектора
public:
    Vect(int r=1);                    // Конструктор
    Vect(const Vect&);                // Конструктор копирования
    ~Vect()                           // Деструктор освобождает
    { delete[] v; }                   // память вектора
    double& operator[](int i)         // Доступ к элементу
    { return v[i]; }
    double operator[](int i) const    // Получить значения элемента
    { return v[i]; }
    int GetSize() const               // Получить размер вектора
    { return size; }
```

```
    Vect& operator=(const Vect&);     // Оператор присваивания
    Vect  operator+(const Vect&);     // Сложение векторов
    Vect  operator-(const Vect&);     // Вычитание векторов
    Vect  operator*( double);         // Умножение вектора на число
    friend ostream& operator<<(ostream&, const Vect&);// Вывод вектора
    friend istream& operator>>(istream&, Vect&);      // Ввод вектора
};
#endif
```

Алгебраические вектора реализованы в виде динамического массива. Для доступа к элементам вектора перегружен оператор [], возвращающий *ссылку* на компонент вектора. *Значение* элемента вектора возвращает *константная* функция-оператор []. Константная функция не изменяет значения объекта, для которого вызвана.

Для повышения эффективности аргументы в функции передаются по ссылке. В этом случае не создаются их копии в функции. В случае, если передаваемый по ссылке аргумент не должен изменяться внутри функции, перед ним можно поставить const, тогда компилятор будет выдавать ошибку при попытке изменить такой аргумент. Такой прием уменьшает вероятность ошибок.

В составе класса предусмотрены функции, реализующие обычные операции над векторами.

В файл Vect.cpp поместим реализацию класса Vect.

```
// Файл Vect.cpp

#include <cmath>
#include "Vect.h"

//  Реализация класса алгебраических векторов

Vect::Vect(int n)                         // Конструктор
{
    size = n;
    v = new double[n];                    // При создании вектора
    for(int i = 0; i < n; i++)            // его элементы
        v[i] = 0.0;                       // обнуляются
}

Vect :: Vect(const Vect& b)               // Конструктор копирования
{
    size = b.size;                        // Создается
    v = new double[size];                 // копия
    for(int i = 0; i < size; i++)         // вектора b
        v[i] = b.v[i];
}

Vect& Vect::operator=(const Vect& b)      // Оператор присваивания
{
    if(this != &b){                       // Присваивание не самому себе,
        delete[] v;                       // освобождение старой памяти
```

```
        size = b.size;                  // Новый размер
        v = new double[size];           // Выделение новой памяти
        for(int i = 0; i < b.size; i++) // Копирование
            v[i] = b.v[i];              // компонентов вектора
    }
    return *this;
}

Vect Vect :: operator+(const Vect& b)   // Сложение векторов
{
    Vect sum = *this;                   // Копия первого слагаемого
    for(int i = 0; i < size; i++)       // Добавляем
        sum[i] += b.v[i];               // второе слагаемое
    return sum;
}

Vect Vect :: operator-(const Vect& b)   // Вычитание векторов
{
    Vect dif = *this;                   // Копия уменьшаемого
    for(int i = 0; i < size; i++)       // Вычитание элементов
        dif.v[i] -= b.v[i];             // второго вектора
    return dif;
}

Vect Vect :: operator*(double d)        // Умножение на число
{
    Vect prod = *this;                  // Копия вектора
    for(int i = 0; i < size; i++)       // Умножение элементов
        prod.v[i] *= d;                 // вектора на число
    return prod;
}

ostream& operator<<(ostream& stream, const Vect& b)   // Оператор
{                                                     // вывода вектора
    for(int i = 0; i < b.size; i++)
        stream << b.v[i] << endl;
    return stream;
}

istream& operator>>(istream& stream, Vect& b)     // Оператор
{                                                 // ввода вектора
    for(int i = 0; i < b.size; i++)
        stream >> b.v[i];
    return stream;
}
```

Класс прямоугольных матриц

Для моделирования прямоугольных матриц разработаем класс `Matrix`. Представим матрицу в виде одномерного массива, в котором последовательно расположены ее строки (рис. 20.1). В программе `nrow` обозначает число строк матрицы, `ncol` – число столбцов. Указатель `a` содержит адрес начала массива, в котором расположены элементы

матрицы. Каждая строка матрицы содержит ncol элементов, поэтому выражения a, a + ncol, a + ncol * 2,…, a + ncol * (nrow – 1) есть адреса первых элементов строк 0, 1, 2,…, nrow – 1. Адрес j-го элемента i-й строки вычисляется с помощью выражения: a + ncol * i + j.

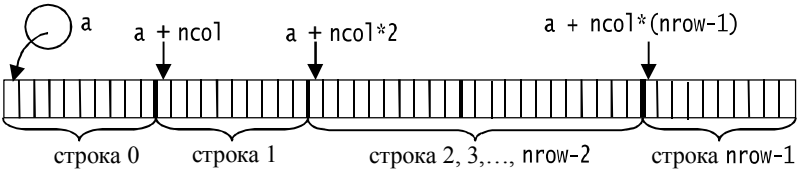

Рис. 20.1. Схема представления матрицы в памяти

Объявление класса Matrix

Объявление класса матриц разместим в заголовочном файле Matrix.h.

```
// Файл Matrix.h

#ifndef MatrixH
#define MatrixH

#include "Vect.h"

class Matrix{                              // Матрица
    int nrow, ncol;                        // Число строк и столбцов матрицы
    double* a;          // Указатель на первый элемент первой строки
        // Строки матрицы расположены последовательно в одномерном массиве
public:
    Matrix(int n = 1, int m = 1);          // Конструктор
    Matrix(const Matrix&);                 // Конструктор копирования
    ~Matrix()                              // Деструктор
    {
        delete[] a;
    }
    Matrix& operator=(const Matrix&);      // Оператор присваивания

    double& operator()(int i, int j)       // Ссылка на элемент матрицы
    {return *(a + i * ncol + j);}

    double operator()(int i, int j) const // Значение элемента матрицы
    {return *(a + i * ncol + j);}
    Matrix operator~();                    // Обращение матрицы
    Matrix operator+(const Matrix&);       // Сложение матриц
    Matrix operator-(const Matrix&);       // Вычитание матриц
    Matrix operator*(const Matrix&);       // Умножение матрицы на матрицу
    Vect operator*(const Vect&);           // Умножение матрицы на вектор
    void SwapLines(int i, int j);          // Поменять местами строки i и j
    void DivLine(int i, double divisor);  // Деление i-й строки на divisor
```

```
    // Вычитание из k-й строки i-й строки, умноженной на factor
    void DiffLines(int k, int i, double factor);
    Vect GetLine(int i);                  // Получить i-ю строку матрицы

    struct MatrixException{               // Класс исключений
        char* Mess;                       // Сообщение о проблеме
        MatrixException(char* problem)    // Конструктор класса исключений
        { Mess = problem; }
    };

    // Возвращает номер максимального по модулю элемента j - го столбца,
    int NumbMainElement(int j, int i);    // начиная с номера i

    friend ostream& operator<<(ostream&, const Matrix&);  // Вывод матрицы
    friend istream& operator>>(istream&, Matrix&);        // Ввод матрицы
};
#endif
```

Для доступа к элементам матрицы перегружен оператор вызова функции (): `operator()(int i, int j)`, что позволяет обращаться к `j`-му элементу `i`-й строки некоторой матрицы `A` в достаточно естественном виде `A(i, j)`. Неконстантный вариант оператора () возвращает *ссылку* на элемент матрицы, что позволяет изменять его, а константный вариант позволяет получать *значение* элемента матрицы.

Для обозначения унарной по своей природе операции обращения матрицы выбран унарный оператор ~.

В состав класса `Matrix` включен локальный класс `MatrixException`, предназначенный для передачи информации о возможных проблемах при обращении матриц.

Реализация класса Matrix

```
// Файл Matrix.cpp

#include "Matrix.h"
#include <cmath>
using namespace std;

Matrix :: Matrix(int n, int m)                // Конструктор
{
    nrow = n; ncol = m;
    a = new double[nrow * ncol];
    for(int k = 0; k < nrow * ncol; ++k)      // Зануление
        *(a + k) = 0.0;                       // матрицы
}

Matrix::Matrix(const Matrix& B)               // Конструктор копирования
{                                             // Создает копию матрицы B
    nrow = B.nrow; ncol = B.ncol;             // Копирование размеров
    a = new double[nrow * ncol];
    for(int k = 0; k < nrow * ncol; ++k)      // Копирование
        a[k] = B.a[k];                        // элементов матрицы B
```

```
}

Matrix& Matrix :: operator=(const Matrix& B)  // Присваивание матриц
{
    if(this == &B)                          // Если присваивание самому себе,
        return *this;                       // ничего не делаем
    if(nrow * ncol != B.nrow * B.ncol){   // Размеры памяти не совпадают,
        delete [] a;                        // удаление старой памяти,
        a = new double[B.nrow * B.ncol]; // выделение новой памяти
    }
    nrow = B.nrow; ncol = B.ncol;         // Присваиваним новые размерности
    for(int k = 0; k < nrow * ncol; ++k)  // Копирование
        a[k] = B.a[k];                      // элементов матрицы
    return *this;
}

Matrix Matrix :: operator+(const Matrix& B)   // Сумма матриц
{
    Matrix sum(*this);                        // Копия первого слагаемого
    for(int i = 0; i < nrow; i++)             // Добавление к первому
        for(int j = 0; j < ncol; j++)         // слагаемому второго
            sum(i, j) += B(i, j);             // слагаемого
    return sum;
}

Matrix Matrix :: operator-(const Matrix& B)   // Разность матриц
{
    Matrix div(*this);                        // Копия уменьшаемого
    for(int i = 0; i < nrow; i++)             // Вычитание из уменьшаемого
        for(int j = 0; j < ncol; j++)         // элементов
            div(i, j) -= B(i, j);             // второй матрицы
    return div;
}

Vect Matrix :: operator*(const Vect& b)       //Умножение матрицы на вектор
{
    if(ncol != b.GetSize())
        throw MatrixException("Несоответствие размера умножаемой матрицы"
                              " и вектора");
    Vect prod(nrow);                          // Вектор с нулевыми элементами
    for(int i = 0; i < nrow; i++)         // Перебор строк матрицы
        for(int j = 0; j < ncol; j++)     // j-й элемент i-й строки матрицы
            prod[i] += (*this)(i, j) * b[j]; // умножается на j-й элемент
                                              // вектора
    // Возможные варианты:
    //      prod[i] = operator()(i, j) * b[j];
    //      prod[i] = this->operator()(i, j) * b[j];
    //      prod[i] = (*this).operator()(i, j) * b[j];
    return prod;
}

Vect Matrix :: GetLine(int i)             // Получение i-й строки матрицы
{
    Vect ai(ncol);                        // Вектор для i-й строки
    for(int j =0; j < ncol; j++)          // Заполнение вектора
```

```
        ai[j] = (*this)(i, j);           // элементами i-й строки
    return ai;
}

Matrix Matrix :: operator*(const Matrix& B)       // Умножение матрицы
{                                                 // на матрицу  A*B
    if(ncol != B.nrow)
        throw MatrixException("Размеры перемножаемых матриц"
                              " не соответствуют друг другу");
    Matrix c(nrow, B.ncol);              // Матрица-произведение
    double tmp;
    for(int i = 0; i < nrow; i++)        // Перебор строк 1-й матрицы
        for(int j = 0; j < B.ncol; j++){ // Перебор столбцов 2-й матрицы
            tmp = 0.0;
// Умножение i-й строки 1-й матрицы на j-й столбец 2-й матрицы
            for(int k = 0; k < ncol; k++)
                tmp += (*this)(i, k) * B(k, j);
            c(i, j) = tmp;               // Элемент произведения матриц
        }
        return c;
}

// Перестановка i-й и j-й строк матрицы
void Matrix :: SwapLines(int i, int j)
{
    double tmp;
    for(int k = 0; k < ncol; k++){
        tmp = (*this)(i, k);
        (*this)(i, k) = (*this)(j, k);
        (*this)(j, k) = tmp;
    }
}


void Matrix :: DivLine(int i, double divisor)     // Деление i-й строки на r
{                                                 // divisor
    for(int j = 0; j < ncol; ++j)
        (*this)(i, j) /= divisor;
}

// Вычитание из k-й строки матрицы i-й строки, умноженной на factor
void Matrix :: DiffLines(int k, int i, double factor)
{
    for(int j = 0; j < ncol; j++)
        (*this)(k, j) -= (*this)(i, j) * factor;
}

// NumbMainElement: Возвращает номер максимального по модулю (главного)
// элемента столбца j, номер которого в столбце >= i

int Matrix :: NumbMainElement(int j, int i)
{
    double MainEl = fabs(a[i * ncol + j]);    // Значение главного элемента
    int NumbMain = i;                         // Номер главного элемента
```

```
// Перебор элементов столбца j, начиная с элемента, номер которого i + 1
    double tmp;
    for(int k = i + 1; k < nrow; k++)
        if(MainEl < (tmp = fabs((*this)(k, j)))){
            MainEl = tmp;
            NumbMain = k;
        }
        return NumbMain;
}

Matrix Matrix::operator~()                  // Обращение матрицы
/*
Исходная матрица путем элементарных преобразований превращается в единичную.
Те же преобразования параллельно выполняются над другой матрицей E,
которая вначале является единичной. В результате матрица E превращается
в матрицу, обратную исходной матрице.
Элементарные преобразования реализуют метод Гаусса
с выбором главного элемента.
При создании матрицы E она обнуляются в конструкторе,
на главную диагональ помещаются единицы.
*/
{
    int i, imain, k;
    if(nrow != ncol) // Если матрица не квадратная, генерируется исключение

        throw MatrixException("Попытка обратить прямоугольную матрицу");

    Matrix Tmp(*this);                      // Запоминаем исходную матрицу
    Matrix E(nrow, nrow);                   // Вспомогательная матрица
    for(i = 0; i < nrow; i++)               // Делаем вспомогательную матрицу
        E(i, i) = 1.0;                      // единичной

    for(i = 0; i < nrow; i++){              // Перебор строк
        imain = NumbMainElement(i, i);      // Номер главн. элем. i-го столбца
        SwapLines(i, imain);                // Перестановка строк, чтобы на
        E.SwapLines(i, imain);              // диагонали стоял главный элемент
        double divisor = (*this)(i, i);     // Главный элемент

        if(0 == divisor)
            throw MatrixException("Главный элемент = 0."
                                  " Матрица вырожденная");

        DivLine(i, divisor);     // Деление строки на главный элемент
        E.DivLine(i, divisor);

        for(k = 0; k < nrow; k++){              // Вычитание i-й строки
            if(k == i) continue;                // из остальных строк для
            double factor = (*this)(k, i);      // обнуления всех элементов
            DiffLines(k, i, factor);            // i -го столбца,
            E.DiffLines(k, i, factor);          // кроме диагонального
        }
    }
    *this = Tmp;                        // Восстановление исходной матрицы
    return E;                           // Возвращение обратной матрицы
}
```

```
ostream& operator<<(ostream& stream, const Matrix& m) // Оператор
{                                                     // вывода матрицы
    for(int i = 0; i < m.nrow; i++){
        for(int j = 0; j < m.ncol; j++)
            stream << " " << m(i, j);
        stream << endl;
    }
    return stream;
}

istream& operator>>(istream& stream, Matrix& m)       // Оператор ввода
{                                                     // для матрицы
    for(int i = 0; i < m.nrow; i++)
        for(int j = 0; j < m.ncol; j++)
            stream >> m(i, j);
    return stream;
}
```

Функция `NumbMainElement()` используется при обращении матриц методом Гаусса. *Главным* называется максимальный по модулю элемент столбца матрицы. Для исключения неизвестных по методу Гаусса следует использовать строку, содержащую главный элемент, что гарантирует от случайного деления на нуль. Кроме того, при использовании главного элемента уменьшаются ошибки округления.

Класс систем линейных уравнений

Класс для моделирования систем линейных уравнений назовем `SystemLinearEquations`. Его объявление разместим его в файле `UnSystemEq`.

```
// Файл SystLinEq.h

#ifndef SystLinEqH
#define SystLinEqH
#include "Matrix.h"

// Класс систем уравнений получается множественным наследованием
// классов матриц и векторов

class SystemLinearEquations: public Matrix, public Vect
{
public:
    SystemLinearEquations(int n = 1): Matrix(n, n), Vect(n)
    { }     // Конструктор создает квадратную матрицу и вектор правой части

    Vect SolveSystem(void);      // Функция для решения системы уравнений
};
#endif
```

В файле `SystLinEq.cpp` находится реализация класса `SystemLinearEquations`.

```
// Файл SystLinEq.cpp
```

```
#include "SystLinEq.h"

Vect SystemLinearEquations :: SolveSystem(void)   // Решение системы
{                                                 // линейных алгебраических уравнений
  return ~(*this) * (Vect&)(*this);
}
```

Использованная в данной функции инструкция есть краткая запись следующей:

```
  return operator~().operator*((Vector&)(*this));
```

Здесь видно, что сначала вызывается функция получения обратной матрицы `operator~()`, затем для полученной обратной матрицы вызывается функция `operator*()`, которая умножает обратную матрицу на вектор правой части, в результате чего получается вектор решения.

Приведение типа `(Vect&)(*this)` преобразует матрицу `*this` в вектор. Это необходимо, чтобы разрешить неоднозначность между выбором одной из двух функций:
`Matrix :: operator*(const Matrix&)` (умножение матрицы на матрицу) и `Matrix :: operator*(const Vect&)` (умножение матрицы на вектор).

Пример использования классов

Разместим исходные данные в текстовом файле, например, `InputData.txt`. Первым элементом пусть будет порядок системы, далее пусть построчно располагаются элементы матрицы и, наконец, правая часть. Далее приведен вариант исходных данных:

```
2
1 2
3 4
1
2
```

Исходные данные следует прочитать из файла, решить систему уравнений и записать решение в файл. Имя исходного файла и файла решения будем задавать в командной строке программы.

Аргументы командной строки можно указать непосредственно в среде Visual Studio, выполнив команду меню **Проект**, **Свойства**, открыть ветку **Свойства конфигурации**, **Отладка** и в строке **Командные аргументы** ввести названия входного и выходного файлов (рис.20.2).

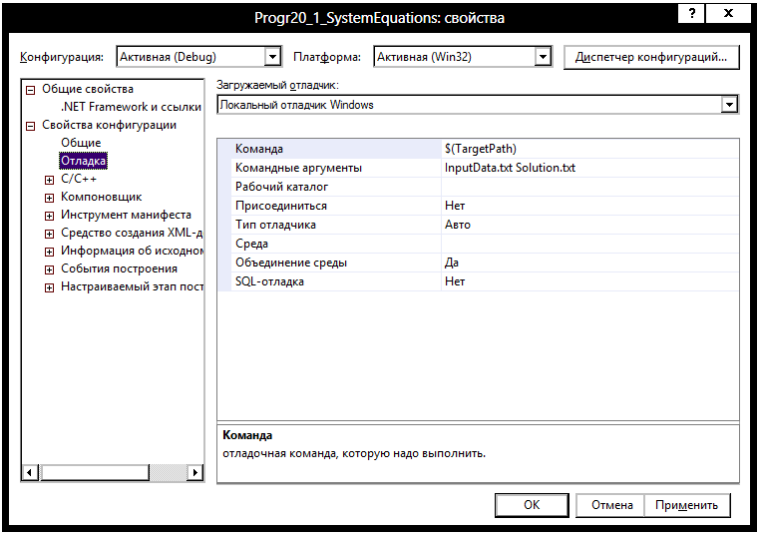

Рис. . 20.2. Ввод параметров командной строки

Далее приводится текст программы, в которой используются разработанные классы.

```
// Файл SolveSystEq.cpp

#include <fstream>
#include "Matrix.h"
#include "SystLinEq.h"
using namespace std;
#include <windows.h>

int main(int argc, char* argv[])
{
    SetConsoleOutputCP(1251);
    if(argc < 3){        // Если в командной строке не заданы аргументы
        cout << "Используйте командную строку\n"
                "Имя_exe-файлпа ФайлИсходнДанных ФайлРешения";
        cin.get();
        exit(1);
    }

    ifstream istrm(argv[1]);          // Поток для исходных данных
    if(!istrm){                       // Если ошибка при открытии файла
        cout << "Не удалось открыть входной файл " << argv[1];
        cin.get();
        exit(1);
    }
    int n;                            // Порядок системы
    istrm >> n;
```

```
    SystemLinearEquations S(n);       // Создание системы уравнений
    istrm >> (Matrix&) S;             // Чтение матрицы системы
    cout << "Матрица системы \n";     // Вывод матрицы
    cout << (Matrix&) S;              // на экран
    istrm >> (Vect&) S;               // Чтение вектора правой части
    cout << "Правая часть системы \n";// Вывод правой части
    cout << (Vect&) S;                // на экран
    try{                              // Попытка решить систему
        Matrix A_1 = ~S;              // Обратная матица системы
        cout << "Обратная матрица \n" << A_1;     // Вывод обратной
        cin.get();                                // матрицы на экран
        cout << "Произведение матрицы на обратную \n"     // Вычисление
        << (Matrix&)((Matrix &) S * A_1); // и вывод произведения
        cin.get();                        // исходной матрицы на обратную
        Vect solution = S.SolveSystem();              // Решение системы
        cout << "Решение системы \n" << solution;     // Вывод решения
        cin.get();
        ofstream ostrm(argv[2]);          // Открытие выходного файла
        if(!ostrm){                       // Проверка открытия файла
            cout << "Не удалось открыть выходной  файл " << argv[2];
            cin.get();
            exit(1);
        }
        ostrm << solution;                    // Запись решения в файл
    }                                         // Конец блока try
    catch(Matrix::MatrixException Exc){       // Обработчик исключений
        cout << Exc.Mess;                     // Вывод сообщения об ошибке
        cin.get();
    }
    return 0;
}
```

Если в нескольких базовых классах, которым наследует производный класс, есть функции с одинаковыми именами, все они становятся членами производного класса, и компилятору надо давать дополнительные указания, какую именно функцию из нескольких одинаковых следует вызывать. Так, для умножения исходной матрицы на обратную будет неверно записать `S * A_1`, так как компилятор не сможет сделать выбор между двумя функциями `Matrix::operator*()` и `Vector::operator*()`, которые обе доступны для системы `S`, так как, с одной стороны система `S` – это матрица, а с другой стороны, – это вектор. Поэтому необходимо преобразование типа для выделения конкретной части производного класса: `Matrix(S)` или `Vector(S)`, в зависимости от того, какую часть системы надо выделить – матрицу или вектор. При таком преобразовании вызываются конструкторы копирования соответствующих классов, которые создают временные объекты, используемые в вычислениях, что связано с определенными накладными расходами.

Если аргументы в функции передаются по ссылке, можно *приводить* производный объект к типу ссылки в виде (Matrix&)S или (Vector&)S, при этом конструкторы не вызываются и работа программы ускоряется.

Для открытия файловых потоков использованы конструкторы потоковых классов, аргументами которых являются имена файлов.

Программа выдает на экран:

```
Матрица системы
 1 2
 3 4
Правая часть системы
1
2
Обратная матрица
 -2 1
 1.5 -0.5
Произведение матрицы на обратную
 1 0
 8.88178e-16 1
Решение системы
0
0.5
```

Полученное решение будет также записано в файл, в данном примере – это Solution.txt.

Если задать вырожденную исходную матрицу, например,

```
1 2
3 6
```

программа выведет:

```
Матрица системы
 1 2
 3 6
Правая часть системы
1
2
Главный элемент = 0. Матрица вырожденная
```

Это результат работы нашего обработчика исключений.