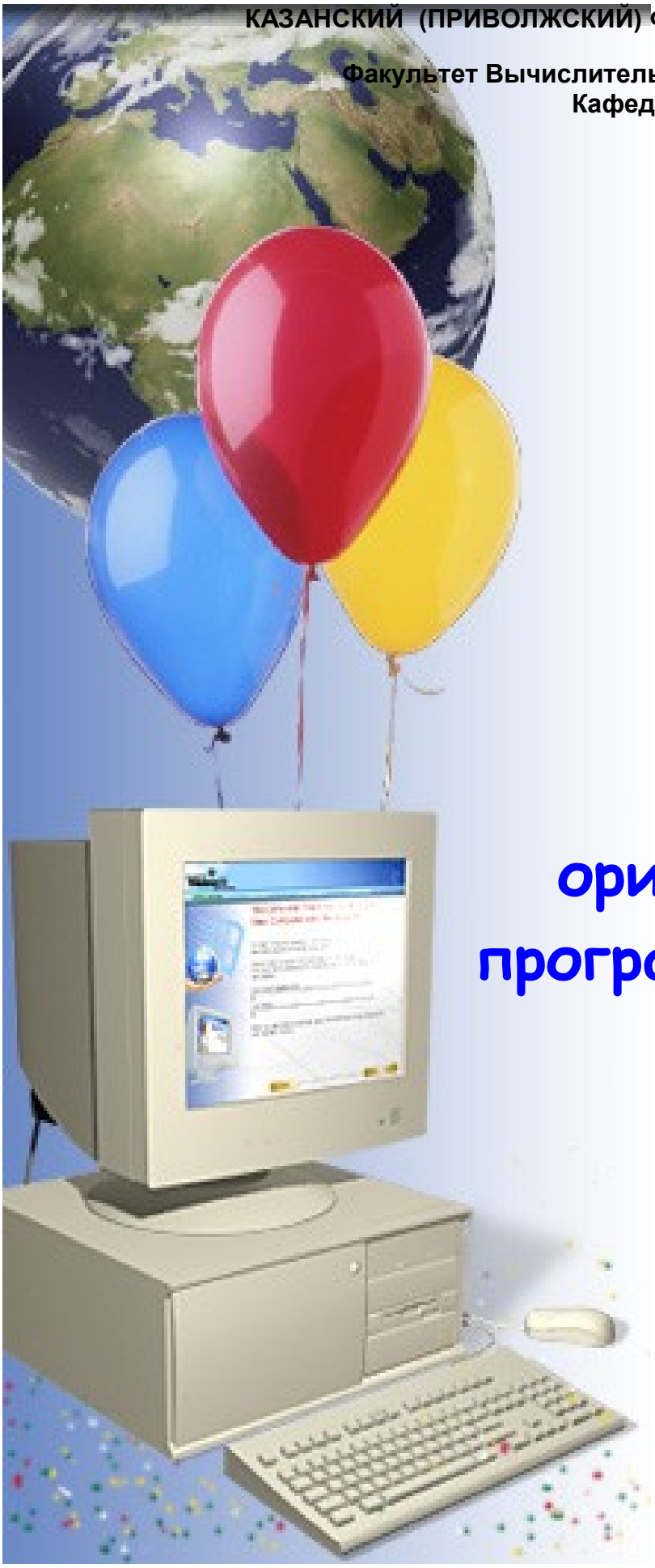
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Факультет Вычислительной Математики и Кибернетики
Кафедра Экономической Кибернетики



АНДРИАНОВА А.А.,
ИСМАГИЛОВ Л.Н.,
МУХТАРОВА Т.М.

Объектно-ориентированное программирование на C++

КАЗАНЬ - 2010

УДК 004.43

ББК 32.973.26 – 018.1

Печатается по постановлению редакционно-издательского совета
факультета вычислительной математики и кибернетики
Казанского (Приволжского) федерального университета

Рецензенты:

кандидат технических наук, доцент кафедры систем информационной безопасности КГТУ им. А.Н. Туполева **Л.А. Александрова**

старший преподаватель кафедры системного анализа и информационных технологий Казанского (Приволжского) федерального университета **Р.Р. Тагиров**

Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М.
Объектно-ориентированное программирование на C++: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский (Приволжского) федерального университет, 2010. – 230 с.

Учебное пособие посвящено принципам разработки программ на языке программирования C++, использующим объектно-ориентированный подход и предназначено для ведения практических занятий на 1, 2 курсах специальностей «Математические методы в экономике» и «Бизнес-информатика», а также может использоваться студентами других специальностей.

Оглавление

Чтобы написать хорошую программу, нужно иметь ум, вкус и терпение. С первого раза у Вас ничего не получится; старайтесь, пробуйте!

Бьярн Страуструп

Введение

Учебное пособие посвящено принципам разработки программ на языке программирования C++, использующим объектно-ориентированный подход. В настоящее время этот подход является основным для различных языков программирования. Объектно-ориентированное программирование отличается от процедурного программирования в первую очередь тем, что при проектировании основной акцент ставится на разработку структур хранения и управления данными, а не на алгоритмы, реализованные в программе. Согласно объектно-ориентированному подходу любая программа представляет собой набор взаимодействующих друг с другом объектов, имеющих состояние и поведение. Разработка программы сводится к определению этого набора объектов.

Язык программирования C++ является одним из наиболее распространенных средств разработки в рамках объектно-ориентированного подхода. Этот язык был создан в начале 1980-х годов Бьярном Страуструпом и быстро стал очень популярным среди программистов. В настоящее время на базе языка C++ возникли новые объектно-ориентированные языки программирования, например, языки Java и C#.

Учебное пособие предназначено для проведения практических занятий по курсу «Объектно-ориентированное программирование» для студентов 1-2 курсов факультета ВМК специальностей «Математические методы в экономике» и «Бизнес-информатика», а также может использоваться студентами других специальностей, изучающих программирование.

Данное учебное пособие состоит из трех глав. Первая глава посвящена возможностям языка C++, которые не поддерживались в языке C: обзор механизмов использования объектно-ориентированной

библиотеки ввода/вывода, новый подход к обработке ошибок – генерация исключительных ситуаций, перегрузка функций, шаблоны функций и указатели на функции.

Во второй главе описаны принципы объектно-ориентированного программирования и их использование при проектировании программ– построение объектной модели приложения, абстрагирование, инкапсуляция, использование классов и объектов в программе, наследование, полиморфизм и параметризация классов.

В третьей главе представлены примеры классов для решения различных задач. Для каждого примера приводится поэтапный процесс разработки необходимых классов, обосновывается целесообразность использования принципов объектно-ориентированного программирования и демонстрируется их применение в программах.

При разборе каждой задачи приводится ее постановка, обсуждение используемого алгоритма решения, а также фрагменты программ с комментариями, написанные на языке программирования C++. Программы отлажены с помощью оболочки проектирования Microsoft Visual Studio 2008.

Глава 1. Особенности языка C++ в структурном программировании

1.1. Потоковый ввод/вывод данных

В языке C++ имеется своя (в отличие от языка C) библиотека ввода/вывода. В ней основным инструментом является поток байтов. При вводе программа читает байты из потока ввода, при выводе записывает байты в поток вывода. Понятие потока позволяет абстрагироваться от того, с каким устройством ввода/вывода работает программа. Например, байты потока ввода могут поступать с клавиатуры, из файла на диске, из другой программы. Аналогично, байты потока вывода могут выводиться на экран, в файл на диске или на вход другой программы. Реализация потоков осуществляется через буфер – специальную область оперативной памяти (подробнее см. в [1]).

Разработчики библиотек ввода/вывода языка C++ использовали широкий набор средств объектно-ориентированного программирования (классы, наследование, виртуальные функции и т.д.). Но, несмотря на сложность реализации, применять потоковые классы ввода/вывода удобно и просто. Опишем далее способы их применения для организации обмена данными с консолью и работы с текстовыми файлами.

Консольный ввод/вывод. Для организации консольного ввода/вывода, т.е. ввода с клавиатуры (стандартный поток ввода) и вывода на экран (стандартный поток вывода), создана библиотека `iostream`. В ней определены классы для поддержки работы с потоками и два основных объекта, которые обеспечивают стандартный ввод/вывод:

- `cin` – стандартный поток ввода (объект класса `istream`);
- `cout` – стандартный поток вывода (объект класса `ostream`);.

Эти объекты становятся доступными в программе при наличии директивы `#include <iostream>`.

Форматированный ввод/вывод реализуется через две операции: операция вывода (вставки, помещения, включения) в поток `"<<"` и операция ввода (извлечения) из потока `">>"`.

Рассмотрим пример использования этих объектов и операций. Пусть требуется ввести целое число, выполнить над ним некоторую операцию и вывести результат.

```
.   .    .
int x = 5;
int y;
std::cin >> y;     // ввод значения в переменную y
x = x + y;
//вывод сообщения вида "x = значение x"
std::cout << "x = " << x << std::endl;
.   .    .
```

Вводить и выводить данные с помощью указанных операций можно по отдельности, разделяя их символом соответствующей операции. В частности, это дает возможность вводить (выводить) в одном операторе данные различных типов. Примером такого вывода является последняя строка фрагмента программы. В ней ключевое слово `endl` означает включение в поток вывода символа конца строки.

`std` – это пространство имен, в котором определены классы, функции и объекты. К объектам `cin` и `cout` следует обращаться с указанием пространства имен: `std::cin` и `std::cout`. Существует возможность обращения к ним без указания префикса `"std::"`. Для этого применяют директиву `using`:

```
using namespace std;
```

Тогда тот же программный код будет выглядеть таким образом:

```
.    .    .
int x = 5;
cin >> y;
x = x + y;
cout << "x = " << x << endl;
.    .    .
```

Помимо объектов `cin` и `cout` и соответствующих операций библиотека `iostream` содержит большой набор средств, реализующих форматированный и неформатированный ввод и вывод. К ним относятся так называемые манипуляторы и методы классов ввода/вывода. К манипуляторам относятся, например, включение в поток символа конца строки `endl`, преобразование числа в шестнадцатеричную систему счисления `hex` и т.д. К методам относятся, к примеру, метод получения кода извлеченного из потока символа `get()` и метод вывода в поток некоторого символа `put(c)` (подробнее см. в [1]).

Файловый ввод/вывод. Для поддержки файлового ввода/вывода на основе потоков используются классы библиотеки `fstream`:

- `ifstream` – класс, с помощью которого осуществляется чтение из файла;
- `ofstream` – класс, с помощью которого осуществляется запись в файл;
- `fstream` – класс, с помощью которого осуществляется чтение и запись в файл.

Эти классы становятся доступными в программе при наличии директивы `#include <fstream>`.

Работа с файлами предполагает следующие операции:

1. создание потокового объекта;
2. открытие потока и связывание его с файлом;
3. осуществление чтения/записи;
4. закрытие файла.

Приведем пример записи данных в текстовый файл, название которого вводится пользователем:

```
.    .    .
char file[256];
cout << "Введите имя файла" << endl;
cin >> file;
// создание потокового объекта, связывание его  с файлом
ofstream os(file);
// запись  в файл
os << "5 + 3 = " << (5 + 3);
// закрытие файла
os.close();
.    .    .
```

Аналогичный программный код можно написать для осуществления чтения данных из текстового файла:

```
.    .    .
char file[256];
cout << "Введите имя файла" << endl;
cin >> file;
char text[256];
// создание потокового объекта, связывание его с файлом
ifstream is(file);
// чтение  из файла и запись считанных данных
// в символьную строку
is >> text;
```

```
// закрытие файла
is.close();
.    .    .
```

При чтении данных из входного файла часто требуется контролировать, достигнут ли конец файла после очередной операции чтения. Это позволяет делать метод `eof()` потокового класса, возвращающий `0`, если конец файла еще не достигнут, и не `0` в противном случае. Условие конца файла возникает только тогда, когда программа пытается считывать данные за последним элементом файла.

Приведем пример печати содержимого текстового файла:

```
.    .    .
char file[256];
cout << "Введите имя файла" << endl;
cin >> file;
char text[256];
// создание потокового объекта, связывание его с файлом
ifstream is(file);
// проверка, открыт ли файл
if (is.is_open())
{
     // считывание строк файла,
     // пока не достигнут его конец
     while (is.eof() == 0)
     {
          // чтение  из файла и запись считанных данных
          // в символьную строку
          is.getline(text,255);
          // вывод считанной строки на экран
          cout << text << endl;
     }
     // закрытие файла
     is.close();
}
else
{
     cout << "Ошибка открытия файла" << endl;
}
.    .    .
```

Домашнее задание

1. Написать программу, которая выполняет операции сложения, умножения и транспонирования прямоугольных матриц. Ввод и

вывод матриц должен осуществляться с помощью объектов `cin` и `cout`.

2. В задании 1 организовать ввод матрицы из текстового файла с помощью объекта класса `ifstream` и вывод матрицы в текстовый файл с помощью объекта класса `ofstream`.
3. Дан текстовый файл, в котором записана последовательность целых чисел. Создать новый текстовый файл, в котором числа идут в обратном порядке.
4. Дан текстовый файл, содержащий информацию о городах мира (название города, название страны, численность населения, площадь). Выполнить следующие задания:
   - Найти самый населенный город в Германии;
   - Найти город, который имеет наибольшую плотность населения;
   - Распечатать названия всех стран, которые имеют города с населением более `1000000` человек
   - Найти количество городов, которые расположены во Франции.
5. Во входном потоке содержится текст на английском языке, заканчивающийся точкой (другие символы "." в тексте отсутствуют). Требуется написать программу, которая будет определять и выводить на экран английскую букву, встречающуюся в этом тексте чаще всего, и количество таких букв. Строчные и прописные буквы при этом считаются неразличимыми. Если искомых букв несколько, то программа должна выводить на экран первую из них по алфавиту.
6. Во входном потоке содержатся фамилии и имена студентов. Известно, что общее количество студентов не превосходит `100`. В первой строке вводится количество зарегистрированных студентов (`N`). Далее следуют `N` строк, имеющих следующий формат: <Фамилия> <Имя>. Требуется написать программу, которая формирует и печатает уникальный логин для каждого студента по следующему правилу: если фамилия встречается первый раз, то логин – это данная фамилия, если фамилия встречается второй раз, то логин – это фамилия, в конец которой приписывается число `2` и т.д.
7. На автозаправочных станциях (АЗС) продается бензин с маркировкой `92`, `95` и `98`. В городе `N` был проведен мониторинг

цены бензина на различных АЗС. Напишите программу, которая будет определять для каждого вида бензина сколько АЗС продают его дешевле всего. Первая строка входного потока содержит количество данных (`N`) о стоимости бензина. В каждой из последующих `N` строк находится информация в следующем формате: <Компания> <Улица> <Марка> <Цена>.

8. В файле содержатся сведения о телефонах всех сотрудников некоторого учреждения. В первой строке сообщается количество сотрудников `N`, каждая из следующих `N` строк имеет следующий формат: <Фамилия> <Инициалы> <Телефон>. Сотрудники одного подразделения имеют один и тот же номер телефона. Номера телефонов в учреждении отличаются только двумя последними цифрами. Требуется написать программу, которая будет выводить на экран информацию, сколько в среднем сотрудников работает в одном подразделении данного учреждения.
9. Входной поток содержит произвольные алфавитно-цифровые символы. Ввод этих символов заканчивается точкой. Требуется написать программу, которая будет печатать последовательность строчных английских букв (`'a' 'b'... 'z'`) из входной последовательности и частоты их повторения. Печать должна происходить в алфавитном порядке.

1.2. Обработка исключений

При выполнении программы могут возникнуть проблемы, препятствующие ее нормальному функционированию. Такими проблемами могут быть открытие несуществующего файла, обращение к несуществующему элементу массива и пр. Для решения этих проблем ранее могло быть предпринято одно из следующих действий:

- прервать выполнение программы;
- вывести значение, означающее ошибку;
- вывести сообщение об ошибке и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжить работу.

В C++ появился еще один механизм уведомления об ошибках и их обработки, который называется ***обработкой исключительных ситуаций***, или исключений. Перечислим его преимущества:

- исключения невозможно проигнорировать, т.е. обязательно любая исключительная ситуация должна быть обработана;
- исключения выделяют обработку ошибок и восстановление после них из основного потока управления, т.е. алгоритм обработки данных описывается отдельно от обработки исключительных ситуаций, которые могут в нем возникнуть;
- при использовании в программах классов и объектов исключения иногда оказываются единственной возможностью обработки ошибок.

Для обработки исключений в язык C++ введены дополнительные операторы: `try`, `catch` и `throw`. Обработка исключений состоит из нескольких этапов:

- выделение контролируемого блока, т.е. блока, в котором может возникнуть исключительная ситуация – блок `try`;
- генерация одного или нескольких исключений с помощью оператора `throw` внутри блока `try` или внутри функций, которые вызываются из этого блока;
- размещение сразу за блоком `try` одного или нескольких обработчиков исключений `catch`.

Рассмотрим программный код для обработки исключительной ситуации деления на `0`. Исключение будет генерироваться до тех пор, пока пользователем не будут введены корректные данные.

```
.    .    .
while(true)
{
     // ввод делимого (x) и делителя (y)
     int x,y;
     cout << "Введите x,y: " << endl;
     cin >> x >> y;

     // контролируемый блок, в котором могут
     // возникнуть исключения
     try
     {
          // если делитель y=0, нужно
          // сгенерировать ислючение
          if(y == 0)
               // генерируется исключение типа int
               throw 0;
          // если частное можно вычислить,
          // вычисляем и выводим результат
```

```
            cout << " x/y = " << (x / y) << endl;
            // выход из цикла, поскольку
            // деление прошло успешно
            break;
      }
      // обработчик исключения типа int
      catch(int e)
      {
            // выводим сообщение об ошибке
            cout << "Деление на 0" << endl;
      }
}
.    .    .
```

Один контролируемый блок `try {. . .}` может быть предназначен для обработки нескольких видов исключительных ситуаций. Ниже приведем пример такой обработки при вычислении обратной матрицы – обратная матрица существует только для квадратных невырожденных матриц. Тогда функция вычисления обратной матрицы должна генерировать исключения в случаях, когда, во-первых, матрица является неквадратной и, во-вторых, когда определитель матрицы равен 0.

```
double**  InverseMatrix(double** a, int m, int n)
{
      // проверяем, является ли матрица квадратной
      if(m != n)
            throw "Матрица должна быть квадратной";
      // проверяем, является ли матрица вырожденной
      // вычисление определителя осуществляет функция
      // Determinant (double** , int)
      double det = Determinant (a, n);
      if(det == 0.0)
            throw 0;
      // далее должен следовать код
      // вычисления обратной матрицы
      .    .    .
}
```

Программа, которая вычисляет обратную матрицу, должна содержать блок `try {. . .}` для выявления указанных выше исключений и соответствующие обработчики `catch`.

```
double** a;    // объявление матрицы
int m, n; // количество строк и столбцов матрицы
.    .    .
```

```
try
{
     double** b = InverseMatrix (a, m, n);
}
catch(char* str)
{
     cout << str << endl;
}
catch(int e)
{
     cout << "Матрица является вырожденной" << endl;
}
.    .    .
```

Отметим, что в приведенном примере генерация исключительных ситуаций происходит в функции вычисления обратной матрицы, а их обработка – в вызывающей ее функции. Таким образом, генерация исключений и их обработка могут быть разделены между различными функциями.

Если в блоке `try` присутствуют вызовы нескольких функций, которые могут генерировать исключения разного вида, следует предусмотреть обработку каждого из них в вызывающей программе. Например, в блоке `try` производится сначала умножение двух матриц, которое может сгенерировать исключение типа `int` со значением `1`, когда размеры перемножаемых матриц не соответствуют, а затем вычисление обратной матрицы:

```
.    .    .
double** a, **b;    // объявление матриц
int m, n, k, l;     // размеры матрицы
.    .    .
try
{
     // умножение матрицы a размерности m x n
     // на матрицу b размерности k x l
     double** c = MultiplyMatrix (a, m, n, b, k, l);
     // получение обратной матрицы для матрицы a
     double** d = InverseMatrix (a, m, n);
}
catch(char* s)
{
     cout << s << endl;
}
catch(int i)
{
```

```
        if (i == 0)
             cout << "Матрица является вырожденной" << endl;
        if(i == 1)
        cout << "Матрицы таких размеров перемножать нельзя"
                                                    << endl;
}
.    .    .
```

В этом случае возникновение ошибки при выполнении операции перемножения матриц приведет к ее обработке и передаче управления операторам, которые следуют за блоком `try {. . .} catch`. Таким образом, вычисление обратной матрицы производиться не будет, даже если это можно сделать без каких-либо ошибок. Такой подход обычно используют, когда операторы блока `try` образуют взаимосвязанную последовательность действий и без корректного выполнения предшествующих действий невозможно правильно выполнить следующие.

Можно действовать по-другому – разбить контролируемый блок на части, каждой из которых будет соответствовать свой набор исключений. Далее каждая часть заключается в свой собственный блок `try`:

```
.    .    .
double** a, **b;    // объявление матриц
int m, n, k, l;     // размеры матрицы
.    .    .
try
{
     // умножение матрицы a размерности m x n
     // на матрицу b размерности k x l
     double** c = MultiplyMatrix (a, m, n, b, k, l);
}
catch(int i)
{
     cout << "Матрицы таких размеров перемножать нельзя"
                                                    << endl;
}
try
{
     // получение обратной матрицы для матрицы a
     double** d = InverseMatrix (a, m, n);
}
catch(char* s)
{
     cout << s << endl;
}
catch(int i)
```

```
{
     cout << "Матрица является вырожденной" << endl;
}
```

В этом случае ошибка, возникшая при перемножении матриц, никак не повлияет на получение обратной матрицы. Такой подход удобнее применять, когда решаемые в блоке `try` задачи не зависят друг от друга. Тогда все задачи, которые могут быть выполнены без ошибок, будут решены.

Домашнее задание

1. Написать программу, которая выполняет операции сложения, умножения и транспонирования прямоугольных матриц. Обработать с помощью генерации исключений ситуацию, когда невозможно выделить память под матрицу (переменные, задающие размеры имеют отрицательные или слишком большие размеры).
2. К заданию 1 добавить обработку исключения, связанного с ситуацией, когда невозможно осуществить сложение и умножение матриц из-за несоответствия их размеров.
3. Написать программу работы с текстовыми файлами. Имя файла должно вводиться пользователем. Обработать исключительную ситуацию отсутствия требуемого файла.
4. Написать функцию поиска местоположения заданного элемента в массиве. Функция должна возвращать номер найденного элемента. Обработать с помощью генерации исключения ситуацию, когда заданный элемент в массиве не найден.
5. Написать функцию ввода в массив результатов забега спортсменов на `1500` метров. Каждый результат записывается в переменной типа структура, содержащей время (минуты, секунды и доли секунды). Обработать с помощью генерации исключения ситуацию ввода недопустимых значений времени (отрицательные значения и количество секунд, большее `60`).
6. Написать функцию ввода в массив информации о студентах учебной группы. Информация об одном студенте хранится в переменной типа структура, содержащей фамилию студента, его имя и дату рождения (день, месяц, год). Обработать с помощью генерации исключения ситуацию ввода недопустимой даты.

7. Написать функцию поиска седловой точки матрицы. Функция должна возвращать объект структуры, содержащий индексы седловой точки. Обработать с помощью генерации исключения ситуацию, когда седловых точек у матрицы не существует.
8. Написать функции, реализующие работу со стеком и очередью (добавление и извлечение элементов). Хранение данных стека и очереди организовать в массиве заданного размера. Обработать с помощью генерации исключения ситуацию, когда делается попытка извлечь элемент из пустого стека или из пустой очереди, и ситуацию, возникающую при добавлении нового элемента, когда стек и очередь уже заполнены.
9. Написать функцию поиска местоположения заданного ключа в дереве сортировки. Функция должна возвращать ссылку на узел, содержащий найденный элемент. Обработать с помощью генерации исключения ситуацию, когда заданного ключа в дереве не существует.

1.3. Перегрузка функций

Часто приходится разрабатывать функции, выполняющие одинаковые действия с различными типами данных, например, для обмена значениями двух переменных целого, вещественного или символьного типов. Удобно было бы называть их одинаково. Поэтому в языке C++ была предусмотрена возможность создавать функции с одинаковыми именами, но различными параметрами (параметры должны различаться количеством и/или типом данных). Такие ***функции*** называются ***перегруженными***. Типы возвращаемых значений у них также могут отличаться, однако использование функций, которые отличаются типом возвращаемого значения, но имеют одинаковые параметры, недопустимо.

Разберем процесс разработки перегруженных функций и их применения на примере функций обмена значениями двух объектов.

```
// функция обмена значениями двух переменных целого типа
void Swap(int& a, int& b)
{
     int t = a; a = b; b = t;
}
```

```
// функция обмена значениями двух переменных
// вещественного типа
void Swap(double& a, double& b)
{
     double t = a; a = b; b = t;
}

// функция обмена значениями двух символьных строк
void Swap(char*& a, char*& b)
{
     char* t = a; a = b; b = t;
}

// функция обмена значениями двух целочисленных массивов
void Swap(int*& a, int& n, int*& b, int& m)
{
     int t = n; n = m; m = t;
     int* ta = a; a = b; b = ta;
}
```

Отметим, что последняя реализация функции `Swap()` отличается не только типами параметров, но и их количеством. Именно по типу фактических параметров и их количеству компилятор определяет, какую функцию требуется вызвать в тот или другой момент. Например,

```
.    .    .
int a = 10, b = 30;
// вызов функции обмена значений переменных целого типа
Swap(a, b);

double x = 100.5, y = 30.2;
// вызов функции обмена значений переменных
// вещественного типа
Swap(x, y);

char * str1="Первая строка", *str2 = "Вторая строка";
// вызов функции обмена двух символьных строк
Swap(str1, str2);

int m = 10, n = 20;
int* array1 = new int[m];
int* array2 = new int[n];
for(int i = 0; i < m; i++)
     cin >> array1[i];
for(int i = 0; i < n; i++)
     cin >> array2[i];
// вызов функции обмена двух массивов целых чисел
```

```
Swap(array1, m, array2, n);
.    .    .
```

При разработке семейства перегруженных функций следует избегать следующих ситуаций, когда нельзя однозначно определить, какую из функций вызвать.

- Функции не должны отличаться только типом возвращаемого значения. Пусть имеются функции:

```
void f(int a) {.        .    .}
int f(int a)  {.        .    .}
```

Тогда при следующем вызове

```
f(10);
```

возникает неоднозначная ситуация для компилятора, которая приводит к ошибке.

- Если две функции имеют параметры одного и того же типа, но в одной из них параметры передаются по значению, а в другой – по ссылке, вызов функции будет содержать неоднозначность, поэтому компилятор рассматривает это как ошибку. Например, имеются функции:

```
void f(int a)  {.   .    .}
void f(int& a) {.   .    .}
```

Тогда при следующем вызове

```
int n = 19;
f(n);
```

возникает неоднозначность, поскольку синтаксис передачи параметров при вызове функции по значению и по ссылке одинаковый.

Ошибки не возникнет в случае передачи в качестве фактического параметра константы, например,

```
f(10);
```

Тогда будет вызвана первая функция.

- Если списки параметров двух функций отличаются только наличием в одной из них дополнительных параметров, которые

имеют значения по умолчанию, вызов функции также может интерпретироваться неоднозначно. Например, имеются функции:

```
void f(int a) {.    .    .}
void f(int a, int b = 10) {.  .    .}
```

При следующем вызове

```
int n = 19;
f(n);
```

возникает неоднозначность, поскольку такой вызов может трактоваться и как вызов первой функции, и как вызов второй функции с параметром по умолчанию. Таким образом, первую функцию вызвать нельзя (возникнет ошибка), а вторую – можно только при указании обоих параметров.

Домашнее задание

1. Напишите перегруженные функции, реализующие операции с векторами:
   - умножения вектора на число;
   - скалярного умножения двух векторов;
   - умножения числа на вектор.
2. Определите структуру «Время», содержащую поля для хранения часов, минут, секунд. Напишите перегруженные функции, реализующие операцию добавления к имеющемуся времени:
   - заданного количества секунд;
   - заданного количества минут и секунд;
   - заданного количества часов, минут и секунд.
3. Определите структуру «Дата», содержащую поля для хранения года, месяца, дня. Напишите перегруженные функции, реализующие операцию добавления к имеющейся дате:
   - заданного количества дней;
   - заданного количества месяцев и дней;
   - заданного количества лет, месяцев и дней.
4. Напишите перегруженные функции, осуществляющие преобразование числа в символьную строку:
   - Для целых чисел;

- Для действительных чисел;
- Для комплексных чисел (комплексное число хранится в структуре, содержащей реальную и мнимую части).

1.4. Шаблоны функций

Шаблоны функций определяют обобщенное описание функций, т.е. описание для обобщенного типа данных. При вызове шаблона функции указывается конкретный тип данных (`int`, `double` или любой пользовательский тип – структура или класс), который подставляется на место обобщенного типа. Таким образом, компилятор генерирует функцию для этого конкретного типа. Подставляя разные типы данных, можно создать множество функций на основе шаблона, реализующих единый алгоритм.

Шаблоны функций выполняют то же назначение, что и перегрузка функций. Однако здесь имеются и существенные отличия. Во-первых, при перегрузке функций их реализация может различаться, шаблон же определяет единый алгоритм для любых типов данных. Во-вторых, шаблоны определяют множество функций с одинаковым количеством параметров, а при перегрузке функций количество параметров может быть разным.

Шаблон функции определяется следующим образом:

```
template <список_обобщенных_типов>
тип_функции имя_функции(список_параметров)
{
     .    .    .
}
```

`список_обобщенных_типов :=`
`class T1 [,class T2, . . ., class TN], Ti` – произвольный идентификатор обобщенного типа;

`тип_функции` – тип возвращаемого значения, в качестве которого может быть указан как конкретный тип данных, так и один из списка обобщенных типов;

`список_параметров` – список формальных параметров шаблона функции, которые могут быть описаны как с указанием конкретного типа данных, так и одного типа из списка обобщенных типов.

При определении шаблона прототип функции предваряется ключевым словом `template` и указанием в угловых скобках списка имен обобщенных типов данных, применяемых в данном шаблоне. Обязательным условием является применение ключевого слова `class` перед каждым именем типа.

Создадим шаблон функции обмена значениями переменных различных типов данных:

```
// шаблон функции обмена значениями
// двух переменных обобщенного типа T
template <class T> void Swap(T& a, T& b)
{  T t = a; a = b; b = t;  }
```

Тип фактических параметров, указываемых при вызове функции, определяет конкретную функцию, сгенерированную для этого типа на основе шаблона. Например,

```
int a = 10, b = 30;
// генерация и вызов функции обмена значений
// переменных целого типа
Swap(a, b);

double x = 100.5, y = 30.2;
// генерация и вызов функции обмена значений
// переменных вещественного типа
Swap(x, y);

char * str1="Первая строка", *str2 = "Вторая строка";
// генерация и вызов функции обмена двух символьных строк
Swap(str1, str2);
```

При вызове функции можно явно указать типы, подставляемые вместо обобщенных. Эти типы указываются в угловых скобках "<>" после имени функции.

```
int a = 10, b = 30;
// генерация и вызов функции обмена значений
// переменных целого типа
Swap<int>(a, b);

double x = 100.5, y = 30.2;
// генерация и вызов функции обмена значений
// переменных вещественного типа
Swap<double>(x, y);
```

Заметим, что если функция `Swap()` будет вызвана с параметрами различных типов данных, компилятор на основе шаблона не сможет сгенерировать функцию, поскольку согласно определению шаблона параметры должны иметь одинаковый тип.

На практике могут возникнуть ситуации, когда один и тот же алгоритм нельзя использовать для всех типов данных. Тогда можно использовать ***перегрузку*** определений ***шаблонов функции***, т.е. создать два и более шаблона, либо создать функции для конкретных типов данных. Например, предыдущий шаблон функции `Swap()` для двух массивов неприменим, поскольку необходимо дополнительно передавать в качестве параметров количество элементов в массивах. Поэтому шаблон можно перегрузить следующим образом:

```
// шаблон функции обмена значениями двух массивов
// с элементами типа T
template <class T> void Swap(T*& a, int& m, T*& b, int& n)
{
     int t = n; n = m; m = t;
     T* ta = a; a = b; b = ta;
}
```

Приведем еще один пример, в котором используется несколько обобщенных типов. Пусть требуется написать шаблон функции умножения вектора с элементами произвольного типа на число (целое или вещественное, дробь, комплексное число). Тип элементов вектора-результата зависит от типа элементов исходного вектора и от типа числа, на которое он умножается. Поэтому внутри шаблона функции тип результата определить сложно. Для этого лучше создать еще один обобщенный тип:

```
// шаблон функции умножения вектора на число
template <class T1, class T2, class T3>
T3* multiply (T1* obj, int n, T2 number)
{
     T3* res;
     res = new T3[n];
     for(int i = 0; i < n; i++)
          res[i] = (T3) obj[i] * number;
     return res;
}
```

В этом случае вызов функции будет выглядеть несколько иначе, так как требуется явно указать все три типа данных, для генерации конкретной функции:

```
int *a = new int[5];
int n = 5;
.    .    .
// вызов функции
double *b = multiply<int, double, double>(a, n, 2.4);
.    .    .
```

Существует большое количество задач, которые решаются одинаково независимо от типов используемых данных. Часто используемые алгоритмы удобно оформить в виде шаблонов. Примерами таких алгоритмов являются поиск, сортировка, слияние структур данных и пр. Рассмотрим шаблон функции сортировки элементов массива произвольного типа с помощью алгоритма Хоара (алгоритм быстрой сортировки).

Идея быстрой сортировки состоит в том, чтобы разделить сортируемый массив на две части таким образом, чтобы можно было каждую из них упорядочить по отдельности, а затем объединить эти части в единый упорядоченный массив. Для такого объединения необходимо, чтобы все элементы первой части массива были меньше любого элемента из второй части. Для разделения массива на две части можно выбрать произвольный элемент (в нашем случае – первый), а затем все элементы просматривать одновременно с начала и с конца. Как только в начале массива найдется некоторый элемент больше выбранного, а в конце – элемент меньше выбранного, нужно поменять их местами и продолжить просмотр. Когда индексы просмотра будут указывать на один и тот же элемент, то выбранный элемент помещается в эту позицию. Далее продолжаем сортировку каждой части по отдельности с помощью рекурсивного вызова.

```
// m – адрес первого элемента сортируемой части массива
// n – количество элементов в сортируемой части массива
template <class T> void QuickSort(T* m, int n)
{
     // если в массиве остался один элемент,
     // массив отсортирован
     if(n <= 1)
          return;
     // i – индекс просмотра массива с начала
```

```
        // j – индекс просмотра массива с конца
        int i = 0, j = n - 1;

        // selected – элемент, относительно которого
        // разделяется массив
        T selected = m[0], temp;

        // цикл разделения массива на две части
        // относительно элемента selected
        while(i != j)
        {
             // поиск с конца элемента меньшего,
             // чем элемент selected
             while(m[j] >= selected && j > i)
                  j--;
             // если элемент найден
             if(j > i)
             {
                  m[i] = m[j];
                  // поиск с начала элемента большего,
                  // чем элемент selected
                  while(m[i] <= selected && i < j)
                       i++;
                  m[j] = m[i];
             }
        }
        // помещаем выбранный элемент в нужную позицию
        m[i] = selected;
        // сортируем две части массива по отдельности
        QuickSort(m, i);
        QuickSort(m + i + 1, n – i - 1);
}
```

Домашнее задание

1. Написать шаблон функции поиска местоположения элемента в массиве. Тип элементов массива может быть произвольным.
2. Написать шаблон функции проверки симметричности массива. Тип элементов массива может быть произвольным.
3. Написать шаблон функции проверки того, что массив образует возрастающую последовательность. Тип элементов массива может быть произвольным.
4. Написать шаблон функции поиска максимального и минимального элементов массива. Тип элементов массива может быть произвольным.

5. Перегрузить шаблон функции поиска максимального и минимального элементов массива для случая, когда элементами массива являются вектора. Вектор задается с помощью массива. Сравнивать вектора следует по их длине.
6. Пусть элементы множества хранятся в массиве. Написать шаблон функции проверки включения одного множества в другое. Тип элементов множества может быть любым.
7. Пусть элементы множества хранятся в массиве. Написать шаблоны функций получения:
   - пересечения двух множеств;
   - объединения двух множеств;
   - разности двух множеств.

   Тип элементов множества может быть любым.
8. Пусть элементы множества хранятся в массиве. Написать шаблон функции проверки равенства двух множеств. Тип элементов множества может быть любым.

1.5. Указатели на функции

При выполнении программы машинный код откомпилированной функции загружается в оперативную память компьютера. Таким образом, к функциям можно обращаться по их адресу. Для этого в языке C++ существует специальный тип данных – указатель на функцию, значением которого является ее адрес.

Указатель на функцию определяется прототипом той функции, на которую он будет ссылаться, т. е. типом возвращаемого значения и списком формальных параметров. Поэтому адреса различных функций, имеющих одинаковый прототип, могут быть присвоены одной переменной-указателю. Имя функции соответствует ее адресу.

Указатель на функцию определяется следующим образом:

```
тип_функции (*имя_указателя) (список_параметров);
```

где `тип_функции` определяет тип возвращаемого значения функции, `список_параметров` – список формальных параметров.

Покажем на примере способы работы с указателями на функции. Пусть имеются следующие математические функции:

```
double sin(double);      // y = sin(x)
double ln(double);       // y = ln(x)
double exp(double);      // y = exp(x)
// и  т.д.
```

Опишем переменную-указатель с именем f, которой можно присвоить адреса каждой из выше приведенных функций:

```
double (*f)(double);
```

Здесь определяется переменная f как указатель на функцию, которая возвращает значение типа double и имеет один аргумент типа double.

Следует обратить внимание на наличие скобок в описании переменной. Отсутствие этих скобок трактует данную запись как прототип функции с именем f и параметром типа double, возвращающую значение указателя на тип double.

Присвоим переменной f адрес функции вычисления синуса:

```
f = sin;  // sin – адрес начала машинного кода функции
          // double sin(double)
```

Таким же образом можно было присвоить переменной f адрес любой из вышеперечисленных функций.

Далее вызывать функцию вычисления синуса можно через переменную-указатель:

```
double y = f(3.1415926); // вызов функции sin(3.1415926)
```

или через операцию разадресации переменной-указателя:

```
double y = (*f)(3.1415926);
```

Указатели на функции часто используются в качестве параметров других функций. Проиллюстрируем это при решении следующей задачи: найти корень уравнения $F(x) = 0$ методом деления отрезка пополам. Предусмотреть возможность использования в качестве $F(x)$ различных функций. Это обеспечивается посредством введения четвертого параметра, который является указателем на функцию вычисления $F(x)$.

```
// функция решения уравнения F(x) = 0 на отрезке [a; b]
// с точностью eps>0 методом деления отрезка пополам
double Root(double a, double b, double eps,
                                  double (*f)(double))
```

```
{
     // корень гарантировано существует,
     // если на концах отрезка
     // функция принимает значения различных знаков.
     if(f(a) * f(b) > 0)
          throw 1;
     // обеспечиваем выполнение условия:
     // a – левый конец отрезка, b – правый
     if(a > b)
     {
          double t = a;
          a = b;
          b = t;
     }
     // цикл метода деления отрезка пополам
     while(b - a > eps)
     {
          double c = (a + b) / 2;
          if(f(c) == 0)
               return c;
          if(f(a)*f(c) < 0)
               b = c;
          else
               a = c;
     }
     return (a + b) / 2;
}
```

Еще одним классическим примером использования указателей на функции является функция сортировки некоторого массива объектов.

Сортировка, как правило, состоит из трех частей: сравнения, определяющего упорядоченность пары объектов; перестановки, меняющей местами пару объектов, и сортирующего алгоритма, который осуществляет сравнения и перестановки до тех пор, пока все объекты не будут упорядочены. Алгоритм сортировки не зависит от операций сравнения, так что, передавая ему в качестве параметров различные функции, его можно настроить на различные критерии сравнения.

Пусть существует список студентов, у которых известны фамилия, имя, отчество (ФИО), номер группы и средний балл успеваемости. Используя различные критерии, например, упорядочивание списка студентов в лексикографическом порядке по ФИО, по среднему баллу успеваемости, по группам и т. д., необходимо отсортировать этот список студентов по убыванию или по возрастанию.

```
// структура, содержащая информацию о студенте
```

```
struct Student
{
     char fio[100];      // ФИО студента
     char group[10];     // номер группы
     float mark;         // средний балл успеваемости
};

// функция сравнения по фамилии студентов
bool GreaterFio(Student s1, Student s2)
{
     if(strcmp(s1.fio, s2.fio) > 0)
          return true;
     return false;
}

// функция сравнения по среднему баллу студентов
bool GreaterMark(Student s1, Student s2)
{
     if(s1.mark > s2.mark)
          return true;
     return false;
}

// функция сравнения по группе и фамилии студентов
bool GreaterGroupFio(Student s1, Student s2)
{
     if(strcmp(s1.group, s2.group) > 0)
          return true;
     if(strcmp(s1.group, s2.group) == 0)
          return GreaterFio(s1, s2);
     return false;
}

// функция сравнения по номеру группы и баллу студентов
bool GreaterGroupMark(Student s1, Student s2)
{
     if(strcmp(s1.group, s2.group) > 0)
          return true;
     if(strcmp(s1.group, s2.group) == 0)
          return GreaterMark(s1, s2);
     return false;
}

// функция сортировки списка студентов методом пузырька
void Sort(Student* a, int n, bool (*f)(Student, Student))
{
     for(int i = 0; i < n - 1; i++)
     {
          bool flag = false;
```

```
            for(int j = 0; j < n - i - 1; j++)
                if(f(a[j], a[j+1]))
                {
                    Student temp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = temp;
                    flag = true;
                }
            if (!flag)
                break;
        }
}
```

Вызвать функцию сортировки с указанием критерия сравнения можно следующим образом:

```
// сортировка по группе и среднему баллу
Sort(a, n, GreaterGroupMark);

// сортировка по группе и фамилии
Sort(a, n, GreaterGroupFio);

// сортировка по среднему баллу
Sort(a, n, GreaterMark);

// сортировка по фамилии
Sort(a, n, GreaterFio);
```

В результате вызова одной из функций изменяется порядок студентов в исходном массиве в соответствии с выбранным критерием сравнения.

Домашнее задание

1. Написать функцию поиска максимального элемента в массиве из объектов «Студент», обучающихся в конкретной группе. Передать в качестве параметра указатель на функцию, определяющую отношение сравнения двух студентов. Получить информацию о студенте, который имеет максимальный балл успеваемости, о студенте, который быстрее всех пробежал дистанцию 100 м на соревнованиях, о студенте, который старше всех в группе. Если нужно, добавить в структуру «Студент» новые поля с дополнительной информацией о студенте.

2. Найти минимум одномерной унимодальной функции $f(x)$ на отрезке $[a,b]$ с помощью следующих методов:

- метода деления отрезка пополам;
- метода золотого сечения;

Написать программу, которая находит минимум функции на отрезке тем методом, который выбирает пользователь. Метод минимизации задается с помощью параметра – указателя на функцию.

Метод деления отрезка пополам заключается в следующем. Выбирается произвольно число d: $0 < d < b - a$. Находятся две точки, которые расположены симметрично относительно середины отрезка на расстоянии $d/2$: $x_1 = \frac{a + b - d}{2}$, $x_2 = \frac{a + b + d}{2}$. Если $f(x_1) \le f(x_2)$, то полагают $b = x_2$, в противном случае, $a = x_1$. Процесс продолжается до тех пор, пока длина отрезка не станет меньше заданной точности ε ($\varepsilon > d$).

Метод золотого сечения заключается в следующем. Золотым сечением отрезка $[a,b]$ называется деление отрезка на две неравные части так, чтобы отношение длины всего отрезка к длине большей части равнялось отношению длины большей части к длине меньшей части отрезка. Такое деление осуществляют две точки отрезка, расположенные симметрично относительно его середины. Выбираются две точки, образующие золотое сечение отрезка $[a,b]$: $x_1 = a + \frac{3 - \sqrt{5}}{2}(b - a)$, $x_2 = a + \frac{\sqrt{5} - 1}{2}(b - a)$. Если $f(x_1) \le f(x_2)$, то полагают $b = x_2$, в противном случае, $a = x_1$. Процесс продолжается до тех пор, пока длина отрезка не станет меньше заданной точности ε ($\varepsilon > 0$).

Глава 2. Основы объектно-ориентированного программирования

2.1. Объектная модель

2.1.1. Основные элементы объектной модели

На начальном этапе развития компьютерной техники и программирования основными являлись вычислительные задачи. В это время центральным понятием в программировании являлся алгоритм – предписание выполнить точно определенную последовательность операций, которая преобразовывает входные данные в результат. Программа представлялась как средство реализации некоторого алгоритма. Для этого были созданы специальные языки программирования, которые позволили преобразовывать отдельные вычислительные операции в соответствующий программный код.

Со временем вычислительные задачи становились все сложнее, и решающие их программы увеличивались в размерах. Это привело к изменению подходов к их разработке. Программы становились все больше, их приходилось разделять на все более мелкие фрагменты, которые решали конкретные подзадачи. Основой для такого разбиения стала процедурная (функциональная) декомпозиция. Программа, таким образом, превратилась в совокупность процедур, каждая из которых представляла собой законченную последовательность действий, направленных на решение отдельной задачи. Отдельно выделялась главная процедура, определяющая процесс решения задачи путем вызова в определенном порядке отдельных процедур. Такой подход в методологии создания программ назвали ***структурным программированием***. Одна из основных особенностей такой методологии заключалась в том, что появилась возможность создавать библиотеки подпрограмм (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта. Период наибольшей популярности идей структурного программирования пришелся на конец 1970-х – начало 1980-х годов.

В 1980-е годы, когда массовое распространение получили персональные компьютеры, вычислительные и расчетно-алгоритмические задачи стали занимать второстепенное место. Компьютер перестал

восприниматься в качестве простого вычислителя, он превратился в среду решения различных прикладных задач обработки и манипулирования данными. На первый план вышли задачи организации простого и удобного человеко-машинного взаимодействия, разработка программ с удобным графическим интерфейсом, создание автоматизированных систем управления и пр. При решении этих задач принципы структурного программирования стали неэффективны, поскольку в процессе разработки таких приложений часто могли изменяться функциональные требования, что усложняло процесс создания программного обеспечения. Также увеличивались размеры программ, что требовало привлечения немалого числа программистов и дополнительных ресурсов для организации их согласованной работы.

В этот момент ведущим при разработке программ стал объектно-ориентированный подход, который основан на представлении любого объекта в виде набора взаимодействующих друг с другом составных частей, которые также являются объектами.

Сложность окружающего мира и многообразие объектов в нем всегда побуждало человечество в своей деятельности вырабатывать упрощенные схемы, модели для его описания и изучения (макет системы орошения, макет строящегося жилищного комплекса, модель летательного аппарата, эскизы картин и т.д.). Эти модели позволяют представить объекты и взаимосвязи между ними в наглядной форме. Так и в программировании, перед тем как написать программный код необходимо получить некоторое представление о предметной области решаемой задачи в виде основных объектов и отношений между ними. Это представление и получило название ***объектной*** (***объектно-информационной***) ***моделью***.

Основой объектно-информационной модели являются объекты. ***Объект*** – это часть окружающей нас действительности, воспринимаемая человеком как единое целое. Объекты могут быть материальными (предметы и явления) и нематериальными (идеи и образы), например, стол, стул, собака, сердце – это материальные объекты; матрица, теория относительности, законы Ньютона, философское учение Платона – примеры нематериальных объектов. Отдельно можно выделить объекты, которые добавляются в процессе программной реализации и не имеют никакого отношения к окружающей нас реальности – вектор, динамический список, бинарное дерево, хэш-таблица и пр.

Каждый объект характеризуется множеством ***свойств***. Информационная модель объекта выделяет из этого множества только

некоторые свойства, существенные для решения конкретной задачи, позволяющие отделить этот объект от других.

Объекты могут находиться в различных состояниях. ***Состояние*** объекта характеризуется перечнем всех его свойств и их текущими значениями.

В следующей таблице приведены примеры некоторых объектов, их свойств и значений

Имя объекта	**Свойства**	**Значения свойств**
Студент 09-155	Имя Специальность Курс Форма обучения Номер зачетной книжки Номер учебной группы	Петров Андрей Сергеевич Математические методы в экономике 1 курс Контрактная форма 09-155 900Э
Мой жесткий диск	Объем Количество занятой памяти	200 Гб 103 Гб
Матрица А	Количество строк Количество столбцов Элементы матрицы	5 5 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Обычно выделяется свойство (набор свойств), однозначно идентифицирующее объект во множестве объектов того же типа. Это свойство (набор свойств) называют ***идентичностью***. Например, свойством, отвечающим за идентичность объекта «Студент», может быть

номер его зачетной книжки (в таблице номер зачетной книжки студента Петрова А.С. – 09-155).

Как правило, объекты не остаются неизменными. Изменение состояния объекта отражается в его модели изменением значений его свойств. Например, на жестком диске изменяется объем занятой памяти, студенты переводятся на следующие курсы и пр.

В объектно-информационной модели отражаются не только свойства, но также и поведение объекта. ***Поведение объекта*** – действия, которые могут выполняться над объектом или которые может выполнять сам объект. Именно поведение объекта определяет его переход из одного состояния в другое. Опишем поведение объектов из нашего примера.

Имя объекта	**Поведение (действия)**
Студент 09-155	Посещение занятий Решение контрольных работ Ответ на семинарах Сдача зачетов Сдача экзаменов Перевод на следующий курс Отчисление Оплата обучения
Мой жесткий диск	Форматирование Копирование
Матрица А	Транспонирование Определение, является ли матрица квадратной Вычисление обратной матрицы Сложение матриц

Множество объектов с одинаковым набором свойств и поведением называется ***классом***. Все студенты обладают одним и тем же набором свойств (имя, специальность, курс, форма обучения, номер зачетной книжки, номер учебной группы, посещение занятий, сдача зачетов, сдача экзаменов и др.) и поэтому образуют класс объектов «Студент». Каждый конкретный студент – экземпляр этого класса (или объект). Следовательно, «Студент 09-155» – экземпляр класса «Студент».

Аналогично можно ввести класс «Жесткий диск», объединив в нем все жесткие диски. Тогда «Мой жесткий диск» – экземпляр класса «Жесткий диск».

Таким образом, ***экземпляр класса*** – это конкретный предмет или объект, а класс определяет множество объектов с одинаковым набором свойств и поведением. Класс может порождать произвольное число объектов, однако любой объект относится к строго фиксированному классу. Класс объекта – это его неявное свойство.

2.1.2. Отношения между объектами и классами

Проектирование объектной модели сводится не только к определению классов, которые описывают предметную область. Классы не существуют автономно – они взаимодействуют между собой. Поэтому в объектную модель включается также описание связей (отношений) между классами.

Наиболее распространенными при описании предметной области модели являются следующие три типа связей – ассоциация, обобщение и зависимость.

Ассоциацией называется структурное отношение, показывающее, что объекты одного типа связаны с объектами другого типа. Например, высказывание «студент учится в вузе» определяет ассоциацию между объектами классов «Студент» и «Вуз». Эта ассоциация является простой, т.е. ни один из классов, участвующих в ней, не является более важным, чем другой. Отношение ассоциации изображено на рис.1.

Рис. 1. Ассоциация «Студент-Вуз».

Ассоциации обычно описываются именем, отражающим природу отношения между объектами. На рисунке именем ассоциации служит

«учится в». При определении ассоциации указывается, какое количество объектов каждого класса участвует в отношении. Это количество называют ***кратностью ассоциации***. Так, в примере ассоциации «Студент-Вуз» кратность характеризуется высказыванием «В одном вузе учится много студентов, но каждый студент учится только в одном вузе». Заметим, что любой человек может учиться в нескольких вузах одновременно, но в этом случае роль студента он выполняет для каждого вуза в отдельности.

Особым видом ассоциации является ***агрегирование*** – отношение типа «является частью» («is-part-of»), когда объект-целое состоит из нескольких объектов-частей. Например, высказывание «группа состоит из студентов» определяет отношение агрегации между объектами классов «Группа» и «Студент» (рис.2).

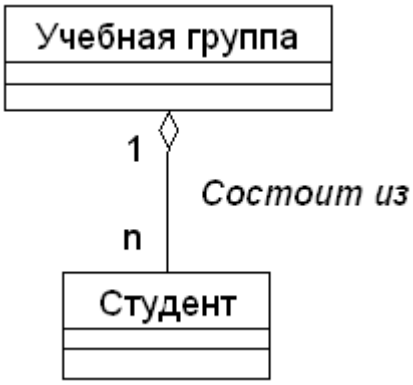


Рис.2. Агрегирование «Учебная группа-Студент».

Частным случаем агрегирования является ***композиция*** – отношение, когда время жизни частей и целого совпадает. Примером такой связи является отношение «Вуз-Факультет» – после ликвидации вуза факультеты как самостоятельные единицы существовать не могут (рис.3).

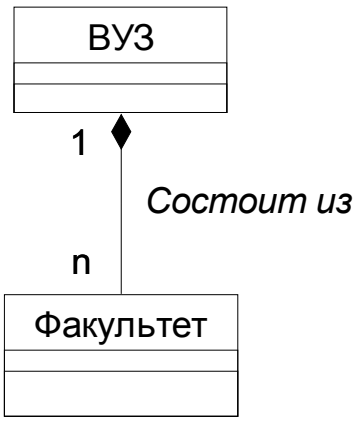


Рис.3. Композиция «Вуз-Факультет».

В отличие от этого агрегация «Группа-Студент» не обладает таким свойством – при распределении студентов по специализациям осуществляется переформирование групп (прежние группы упраздняются, новые группы формируются). При этом объекты-студенты не уничтожаются.

Обобщение – это отношение между общим классом (суперклассом, родителем) и одной или несколькими его вариациями (подклассами, потомками). Обобщение объединяет классы по их общим свойствам и поведению, что обеспечивает структурирование описания объектов.

Обобщение иногда называют отношениями типа ***«является» («is-a»)***, имея в виду, что одна сущность (класс «Студент-контрактник») является частным случаем другой, более общей (класс «Студент»). Обобщение означает, что объекты класса-потомка могут использоваться всюду, где встречаются объекты класса-родителя, но не наоборот. Потомок может быть подставлен вместо родителя. При этом он наследует свойства родителя – его атрибуты и операции. Часто, хотя и не всегда, у потомков есть и свои собственные атрибуты и операции, помимо тех, что существуют у родителя (рис. 4).

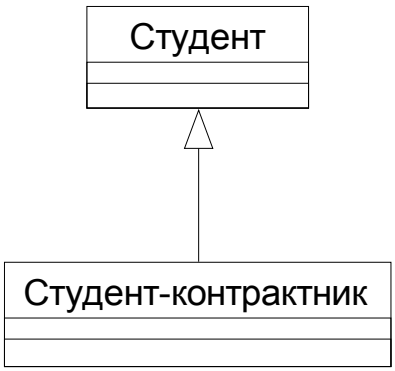


Рис.4. Обобщение между классами «Студент»
и «Студент-контрактник».

В случаях, когда класс-потомок не содержит собственных атрибутов и операций, но реализация некоторых унаследованных им методов отличается от родительских, определяется отношение типа ***«является подобным» («is-like-a»)***. Примером такого отношения является иерархия классов для студентов разных курсов. Все студенты сдают зачеты и экзамены, количество которых может быть различным в зависимости от года обучения. Поэтому методы допуска к экзаменационной сессии,

перевода на следующий курс и т.д. будут иметь одинаковый прототип, но их реализация будет различной.

Отношение ***зависимости*** – это такой тип отношения, при котором изменение в определении одного класса приводит к изменению реализации другого класса. Например, изменение в классе «Студент» (добавление новых методов, изменение прототипов существующих методов и пр.) может привести к изменениям в классе «Учебная группа». Чаще всего такая связь возникает в случаях, когда классы находятся в отношении агрегации или когда объекты одного класса являются параметрами методов другого класса.

2.1.3. Объектная модель вузовской информационной системы

Приведем несколько примеров объектных моделей, связанных с различными приложениями вузовской информационной системы. Как уже было сказано, для определения объектной модели необходимо описать объекты предметной области и отношения между ними.

Пример 1. Объектная модель системы «Абитуриент».

Классами, которые определяют основные объекты системы «Абитуриент», являются:

- «Абитуриент», свойствами которого являются ФИО, место жительства, номер школы, паспортные данные, выбранная специальность, результаты ЕГЭ и т.д;
- «Результат вступительных испытаний», определяющий баллы, которые получил конкретный абитуриент при сдаче экзамена по одному предмету (ЕГЭ или вступительные экзамены);
- «Специальность», которая описывается следующими атрибутами: код, название, план приема, стоимость обучения, проходной балл и т.д.
- «Факультет», у которого задаются: код, название факультета, список специальностей;
- «Приказ о зачислении», характеризующийся номером, датой и списком абитуриентов, зачисленных в студенты по различным специальностям;

- «Студент», основная информация о котором совпадает с атрибутами класса «Абитуриент», а также имеются новые свойства, необходимые для других приложений.

Опишем связи между этими классами, необходимые для данного приложения.

1. Между классами «Факультет» и «Специальность» существует ассоциация, показывающая, что на каждом факультете ведется подготовка специалистов по конкретным специальностям. На одном факультете может быть несколько специальностей, подготовка по каждой специальности может вестись только на одном факультете.
2. Между классами «Абитуриент» и «Специальность» существует ассоциация, показывающая, что абитуриент подает заявление на участие в конкурсе по конкретной специальности. Один абитуриент может подать заявления на несколько специальностей, на каждую специальность может быть подано большое количество заявлений от абитуриентов.
3. Между классами «Абитуриент» и «Результат экзамена» существует ассоциация, демонстрирующая, что любой абитуриент должен предоставить свои результаты ЕГЭ или сдать вступительный экзамен для участия в конкурсе. Один абитуриент предоставляет результаты нескольких экзаменов, каждый результат принадлежит только одному абитуриенту.
4. Между классами «Абитуриент», «Специальность» и «Приказ о зачислении» существует тернарная (между тремя классами) ассоциация, которая описывает формирование приказов о зачислении абитуриентов на специальности. Приказ может содержать списки абитуриентов, зачисленных в студенты по разным специальностям. Каждый абитуриент при этом может быть зачислен только на одну специальность, т.е. его фамилия присутствует в приказе только один раз.
5. Между классами «Приказ» и «Студент» существует ассоциация, которая производит изменение статуса зачисленных абитуриентов в студенты (рис. 5).
6. Как правило, выделяются различные типы абитуриентов – граждане РФ и иностранные граждане. Отличия этих типов состоит в отсутствии результатов ЕГЭ у иностранных граждан. Следовательно, можно создать два новых класса –

«Абитуриент – гражданин РФ» и «Абитуриент – иностранный гражданин». Эти классы наследуют свойства и методы класса «Абитуриент», определяя тем самым отношение обобщения между ними (отношение типа «is-like-a»).

7. Предположим, что только абитуриентам – гражданам РФ могут быть предоставлены льготные условия при поступлении (сиротам, военнослужащим, инвалидам и пр.). Тогда между классами «Абитуриент – гражданин РФ» и новыми классами «Абитуриент с льготами» и «Абитуриент без льгот» можно также выделить отношение обобщения (рис. 6).
8. Другой вариант представления этой связи приводит к появлению отношения зависимости между классом «Абитуриент» и «Типы абитуриентов», который будет в себя включать перечисление типов абитуриентов (имеющие или не имеющие льготы). Зачисление в студенты будет происходить в зависимости от типа абитуриента, т.е. будет по-разному реализовано. Такое представление взаимосвязей между классами считается более правильным (рис. 7).

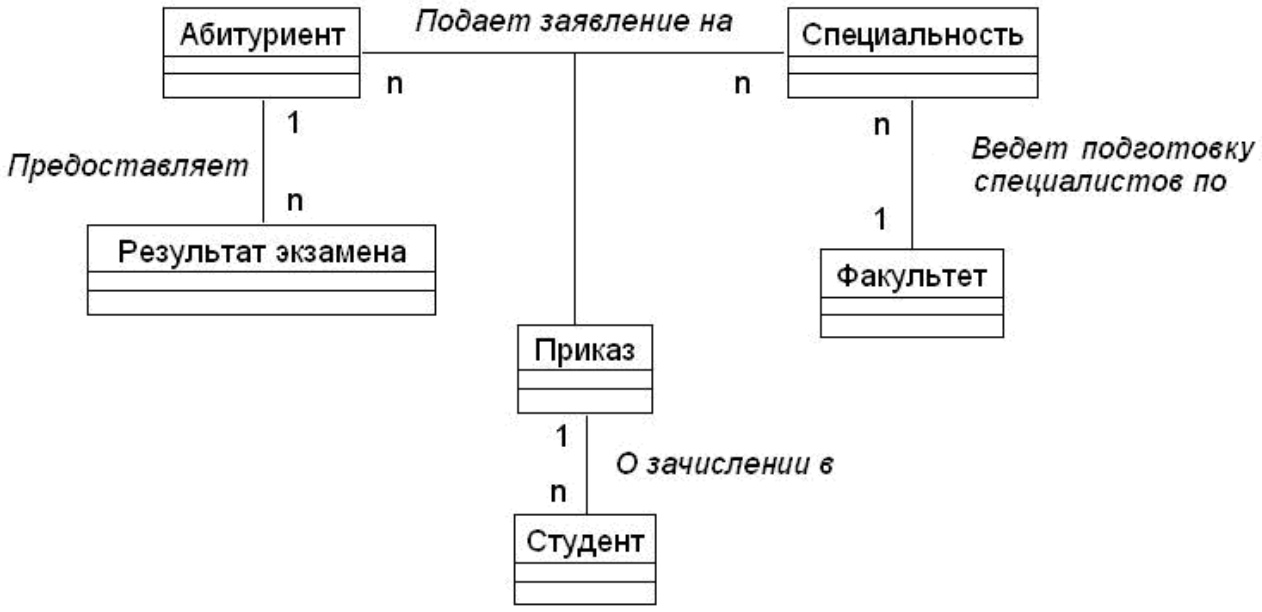


Рис.5. Объектная модель системы «Абитуриент».

Рис. 6. Отношения обобщения в системе «Абитуриент».

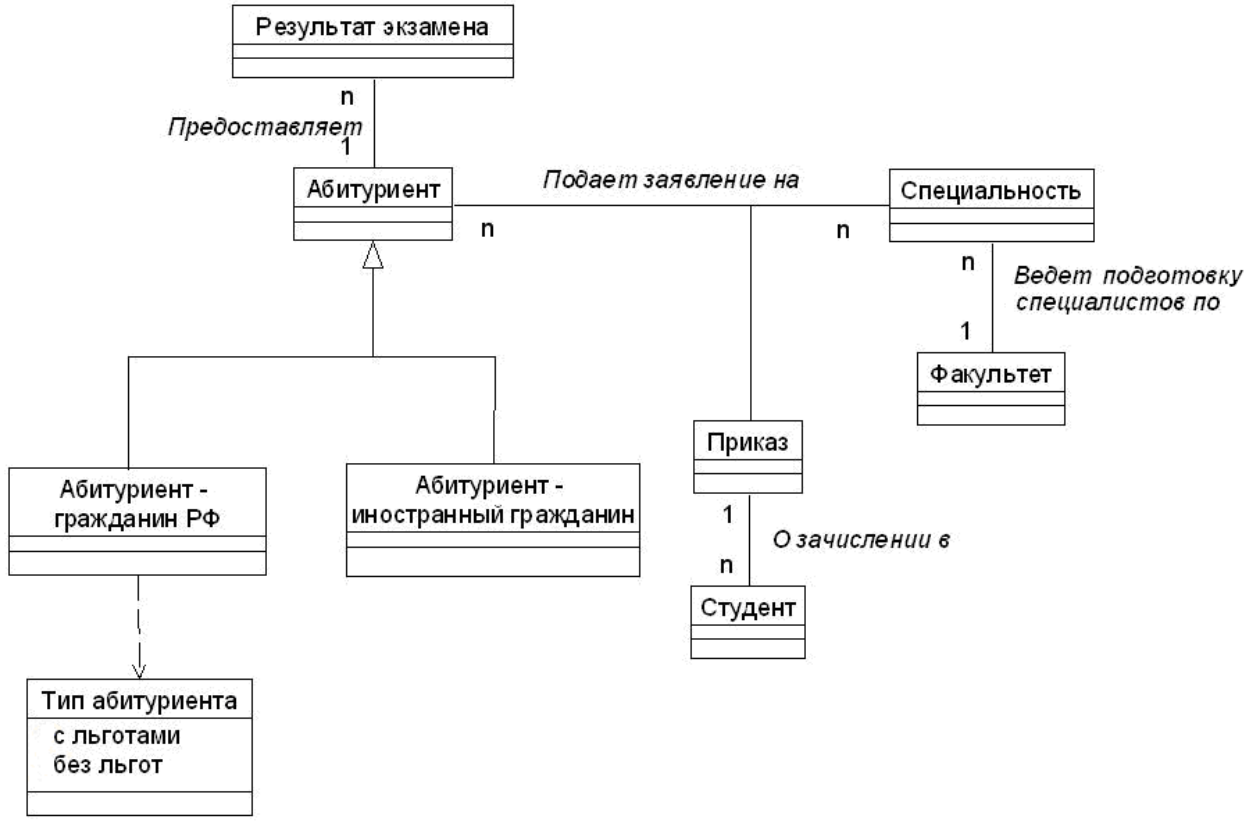


Рис. 7. Отношение зависимости в системе «Абитуриент».

Пример 2. Объектная модель системы «Формирование учебного расписания».

Основные объекты системы «Формирование учебного расписания» определяются следующими классами;

- «Расписание». Его свойствами являются название факультета и список занятий;
- «Учебное занятие». Оно задается названием предмета, типом занятия (лекция, практика, консультация и т.д.), преподавателем, который его проводит, учебной группой, днем недели и временем проведения, аудиторией;
- «Преподаватель», проводящий занятия;
- «Предмет», по которому проводятся занятия;
- «Учебная группа», для которой проводится занятие;
- «Аудитория», в которой проводится занятие.

Расписание содержит в себе информацию обо всех учебных занятиях, т.е. определяет отношение типа «часть/целое» с классом «Учебное занятие». Данное отношение является отношением композиции, поскольку учебные занятия не существуют как самостоятельные сущности без расписания. Между классом «Учебное занятие» и оставшимися классами существуют ассоциации. Одно учебное занятие проводит один преподаватель, при этом один преподавать может проводить несколько занятий в разное время. Одно учебное занятие может проводиться для студентов нескольких учебных групп, при этом каждая группа посещает несколько занятий в разное время. Одно учебное занятие ведется по конкретному предмету, но для каждого предмета может быть несколько занятий в расписании. Каждое учебное занятие проходит только в одной аудитории, но в этой же аудитории в другое время могут проходить другие занятия.

Как и в предыдущем примере, типы занятий можно выделить в отдельный класс, который будет накладывать ограничение на аудитории, в которых может проводиться занятие (в зависимости от вместительности аудитории или наличия определенной техники). Тем самым, образуется отношение зависимости между классами «Тип занятия» и «Аудитория».

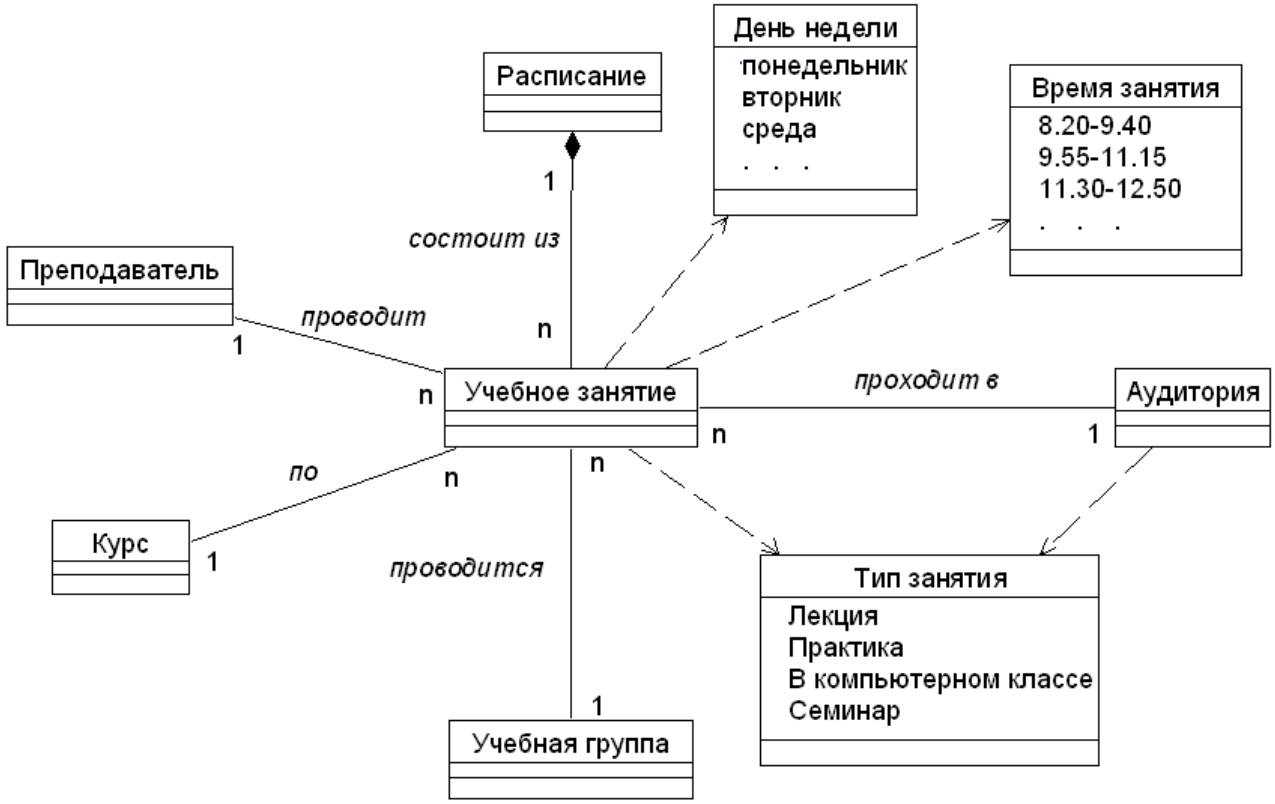


Рис.8. Объектная модель системы «Формирование учебного расписания».

В заключение данного раздела отметим, что процесс разработки программ при использовании объектно-ориентированного подхода претерпел существенные изменения, главными из которых являются:

- процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Действительно, до того как начать программирование классов, их свойств и методов, необходимо определить, чем же являются сами эти классы. Более того, нужно дать ответы на такие вопросы, как: сколько и какие классы нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания классам требуемого поведения, а также установить взаимосвязи между классами;
- объектная декомпозиция программ. Программа является совокупностью взаимодействующих объектов, каждый из которых принадлежит некоторому классу.

Указанные изменения подразумевают выделение обязательного начального этапа разработки программы, связанного с предварительным анализом предметной области задачи. Все эти обстоятельства привели к появлению специальной методологии, получившей название методологии объектно-ориентированного анализа и проектирования (ООАП).

Построение объектной модели на начальном этапе разработки программ дает определенные преимущества:

• объектная модель более естественная для разработки, поскольку ориентирована на человеческое восприятие мира, а не на компьютерную реализацию;

• объектная модель сложной программной системы может быть разделена на объектные модели ее подсистем. Поэтому процесс разработки носит обычно эволюционный характер, что предполагает развитие системы на базе уже разработанных небольших подсистем;

• объектная модель в дальнейшем позволяет использовать выразительные возможности объектных и объектно-ориентированных языков программирования, таких как C++, C#, Java и т.д.

Домашнее задание

1. Создать объектную модель информационной системы «Домашняя библиотека».
2. Создать объектную модель информационной системы «Электронный магазин».
3. Создать объектную модель информационной системы «Адресная книга».
4. Создать объектную модель информационной системы «Салон видеопроката».
5. Создать объектную модель информационной системы «Бронирование номеров в гостинице».

2.2. Основные принципы объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части: 1) ООП использует в качестве базовых элементов объекты, а не алгоритмы; 2) каждый объект является экземпляром какого-либо определенного класса;

3) классы организованы иерархически. Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований.

В объектно-ориентированной технологии используется особый подход к разработке программ, основанный на использовании объектных моделей и нескольких базовых концепциях. К этим концепциям относятся абстрагирование, инкапсуляция, полиморфизм, наследование.

2.2.1. Абстрагирование

Любая объектная модель содержит описание объектов, необходимых для работы приложения, и их взаимосвязей. Любой объект обладает большим количеством различных свойств. Каждый человек воспринимает объект по-своему, исходя из того, какие задачи приходится ему решать, работая с этим объектом. В этом случае для описания объекта выделяется конечное количество его характеристик, существенных для решения задачи. К характеристикам объекта относятся его свойства, как с точки зрения его структуры, так и с точки зрения его поведения. Например, при приобретении велосипеда покупатель обращает внимание на:

- структурные свойства: возрастная группа велосипедиста (детский, подростковый, взрослый), тип велосипеда (спортивный, прогулочный, горный, шоссейный), размер колес, количество передач, материал, из которого сделан велосипед, фирма-производитель, цвет, стоимость и др.;
- и поведение: переключение скорости, движение, торможение и др.

Из всех этих свойств пользователь в данный момент выделяет только существенные. Предположим, покупателю нужен велосипед для обучения ребенка езде. В этом случае для него несущественными могут быть следующие свойства: количество передач, фирма-производитель, цвет, переключение скорости и др.

Так и в программировании, разработчики концентрируют свое внимание на существенных свойствах, необходимых для описания объекта, и на операциях, которые описывают его поведение. В этом и заключается ***абстрагирование***.

При абстрагировании выделяются существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и,

таким образом, четко определяются его концептуальные границы с точки зрения наблюдателя. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Классы объектной модели представляют собой абстракции сущностей предметной области задачи. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Абстракция структурного свойства объекта определяется своим именем и множеством значений, которые это свойство может принимать. Например, свойство велосипеда «цвет» может принимать значения «красный», «зеленый», «серебристый» и др., свойство «количество передач» – 1, 4, 8 и др., «стоимость» - 15 000 рублей, 40 000 рублей и т.д. С точки зрения программирования множество значений характеризуется типом данных переменной, которая будет хранить значение этого свойства.

В процессе использования объекта значения структурных свойств могут изменяться. Например, увеличилась стоимость велосипеда или поменялся его цвет. Эти изменения, как правило, происходят в результате воздействия на объект, которое осуществляется абстракцией его поведенческого свойства. Воздействие на объект порождает некоторую реакцию этого объекта. Действия, которые можно выполнить по отношению к данному объекту, и реакция объекта на них определяют абстракцию поведенческого свойства. С точки зрения программирования она задается специальной функцией (***методом***).

Абстракцию совокупности объектов, которые имеют общий набор свойств и обладают одинаковым поведением, называют ***классом***. Каждый объект в этом случае рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют полностью одинаковых свойств или не обладают одинаковым поведением, по определению, не могут быть отнесены к одному классу.

В языке C++ класс объявляется следующим образом:

```
class имя_класса
{
     // определение структурных свойств
     .    .    .
     // определение поведенческих свойств
     .    .    .
};
```

Определяя новый класс, мы создаем новый пользовательский тип данных с идентификатором, совпадающим с именем класса.

Приведем простой пример класса «Велосипед».

```
// определение новых типов данных
// для задания свойств велосипеда

// возрастная группа
typedef enum{child, teenager, adult} AgeGroup;
// цвет
typedef enum{red, green, blue} Color;
// тип велосипеда
typedef enum{sport, walk, mountain} BicycleType;

class Bicycle
{
     // определение структурных свойств
     AgeGroup ageGroup;
     BicycleType type;
     double wheelDim;
     int countGear;
     char material[50];
     char firm[100];
     Color color;
     double price;
     bool moveState;     // состояние велосипеда –
                         // стоит или двигается
     double valSpeed;    // величина текущей скорости
     int numGear;        // номер текущей скорости

     // определение поведенческих свойств
public:
     void HighSpeed(double);  // метод увеличения скорости
     void LowSpeed(double);   // метод уменьшения скорости
     void HighGear();         // метод повышения передачи
     void LowGear();          // метод понижения передачи
     void Move();             // метод "начать движение"
     void Stop();             // метод "остановиться"
};
```

В класс `Bicycle` были добавлены свойства, которые характеризуют текущее состояние велосипеда в процессе использования – движение (стоит или двигается), текущая скорость и номер передачи. Изменяют эти свойства соответствующие им методы. Например, метод `Stop()` может изменить скорость велосипеда `valSpeed` на значение `0.0` и его текущее состояние `moveState` на значение `false`.

Еще одним примером класса является класс «Матрица действительных чисел»:

```
class Matrix
{
     // определение структурных свойств
     double** a;    // адрес двумерного массива
                    // для хранения матрицы
     int m, n;      //количество строк и столбцов в матрице

     // определение поведенческих свойств
public:
     Matrix Transp();     // метод транспонирования матрицы
     double Determinant();  // метод вычисления
                            // определителя квадратной
                            // матрицы
     int RangMatrix();      // метод вычисления ранга
                            //  матрицы
     Matrix Summ(Matrix);   // метод сложения двух матриц
     Matrix Substr(Matrix); // метод вычитания
                            // для двух матриц
     Matrix Mult(Matrix);  // метод умножения двух матриц
     Matrix Mult(double);  // метод умножения матрицы
                           // на число
     void InputMatrix();   // метод ввода матрицы
     void OutputMatrix();  // метод вывода матрицы
     .    .    .
};
```

Пример класса «Матрица действительных чисел» будет использоваться для демонстрации других принципов ООП.

2.2.2. Инкапсуляция

Следующий принцип ООП – ***инкапсуляция***. Этот термин характеризует сокрытие отдельных деталей внутреннего устройства класса от внешних по отношению к нему объектов или пользователей. Действительно, пользователю нет необходимости знать, из каких частей состоит экземпляр класса и каким образом реализовано его поведение. Реализация присущих классу структурных и поведенческих свойств, является его собственным делом. Более того, отдельные свойства и методы вообще могут быть невидимы за пределами этого класса. В этом случае пользователю предоставляется набор средств для доступа и управления объектом, чаще всего реализованных в виде методов класса.

Покажем на примере класса «Матрица действительных чисел» (`Matrix`) использование принципа инкапсуляции. Этот класс содержит переменные, доступ к которым может быть осуществлен только из методов класса. Такой подход используется для того, чтобы защитить переменные от несанкционированного доступа. Например, если дать возможность программисту, который будет использовать класс `Matrix`, изменять напрямую значения переменных для хранения размера матрицы, то эти значения могут стать некорректными (отрицательными или очень большими). А, выполняя то же самое с помощью методов класса, можно включить в них проверку корректности введенных значений. Для нашего класса таким методом может быть метод ввода матрицы. В примере приведен фрагмент кода, в котором обрабатывается ввод некорректных значений количества строк и столбцов матрицы.

```
class Matrix
{
     .    .    .
public:

     void InputMatrix()
     {
          cout << "Введите количество строк матрицы"
                                                  << endl;
          cin >> m;
          if (m <= 0)
               throw "Количество строк должно быть
                                   положительным" << endl;
          cout << "Введите количество столбцов матрицы"
                                                  << endl;
          cin >> n;
          if (n <= 0)
               throw "Количество столбцов должно быть
                                          положительным ";
          .    .    .
     }
     .    .    .
};
```

Помимо структурных свойств класса сокрытию подлежит и реализация операций класса. Для пользователя нет необходимости знать, как реализован тот или иной метод. Надо знать только то, что именно метод выполняет, как к нему обратиться и как воспользоваться результатом его работы. Например, определитель квадратной матрицы

можно вычислить различными способами: методом исключений Гаусса или по формуле Лапласа – разложением по строке или столбцу. Для пользователя не имеет значения, какой способ вычисления определителя реализован в методе класса.

Для явного определения способа доступа к данным и методам класса в языке C++ используются модификаторы `public`, `protected` и `private`.

```
class A
{
private:
     // определение скрытых данных и методов, к которым
     // можно обращаться только из самого класса
     .    .    .
protected:
     // определение данных и методов, к которым можно
     // обращаться из самого класса и
     // из производных от него классов
     .    .    .
public:
     // определение общедоступных данных и методов, к
     // которым можно обращаться из любого места программы
     .    .    .
};
```

Принципы абстрагирования и инкапсуляции используются совместно при разработке классов, дополняя друг друга. Уже на этапе выбора структурных и поведенческих свойств класса (при применении принципа абстрагирования) определяется способ доступа к этим свойствам, т.е. применяется принцип инкапсуляции. Тем самым определяется внешний интерфейс класса – набор средств, которым можно пользоваться извне при работе с объектами этого класса. Каждое такое средство определяет некоторое внешнее поведение объекта. Внутренняя же реализация этих средств скрыта от других объектов.

2.2.3. Классы и объекты

Как было указано ранее, ***класс*** представляет собой пользовательский тип данных, который определяет множество объектов с одинаковой структурой и одинаковым поведением. Поэтому объекты класса можно использовать как переменные любого стандартного типа данных (`int`, `double` и др.): создавать объекты класса посредством объявления переменной класса, передавать объекты как аргументы в функцию, определять с их помощью возвращаемое значение функции, использовать для их ввода/вывода потоки `cin` и `cout`, создавать средства автоматического преобразования типов между объектами подобных классов.

В большинстве случаев определение класса состоит из двух частей:

- объявления класса, в котором описаны компоненты данных и общедоступный интерфейс;
- определения методов класса, в которых реализуется конкретное поведение данного класса.

Если метод класса определяется вне объявления класса, то перед именем метода указывается его принадлежность классу посредством добавления имени класса и операции **`"::"`**:

```
тип_функции имя_класса::имя_метода(список_параметров)
{
     .    .    .
}
```

Приведем примеры определений классов. Пусть требуется написать программу, автоматизирующую работу деканата, в которой используется информация о студентах факультета и учебных группах. В этой программе должны быть определены классы `Student` (студент) и `Group` (учебная группа).

Класс `Student` описывает множество объектов-студентов. Используя принципы абстрагирования и инкапсуляции, включим в класс следующие структурные свойства, доступные только внутри него самого:

- фамилия, имя и отчество студента;
- дата рождения студента;
- номер зачетной книжки;
- средний балл успеваемости студента.

Поведение объектов данного класса описывается набором общедоступных методов:

- метод инициализации данных о студенте;
- методы получения значений закрытых свойств класса: фамилии, имени и отчества студента, даты рождения, номера зачетной книжки и среднего балла успеваемости;
- метод изменения значения среднего балла успеваемости (другие свойства изменяются крайне редко);
- метод вывода информации о студенте.

```
// объявление класса «Студент»
class Student
{
private:
     char fio[100]; // фамилия, имя, отчество
     int dayBirth;  // дата рождения
     int monthBirth;
     int yearBirth;
     int numberOfTestBook; // номер зачетной книжки
     double averageMark; // средний балл успеваемости
public:
     // метод заполнения свойств студента
     void Fill(char*, int, int, int, int, double);
     // метод инициализации данных студента
     void Init(char*, int, int, int, int, double = 0.0);
     // метод получения фамилии,
     // имени и отчества студента
     char* GetFio();
     // метод получения даты рождения
     void GetBirthday(int&, int&, int&);
     // метод получения номера зачетной книжки
     int GetNumberOfTestBook();
     // метод получения среднего балла
     // успеваемости студента
     double GetAverageMark();
     // метод изменения среднего балла
     // успеваемости студента
     void SetAverageMark(double);
     // метод распечатки информации о студента
     void Print();
};
```

Класс `Group` описывает учебную группу. Этот класс содержит следующие закрытые свойства:

- номер учебной группы;

- текущее количество студентов в группе (полагаем, что общее количество студентов в группе не должно превышать `20` человек);
- список студентов этой группы. Данное свойство описывает отношение агрегирования между классами `Student` и `Group`.

Для каждой группы могут быть определены следующие общедоступные методы:

- метод инициализации данных о группе;
- методы добавления студента в группу. Добавить студента можно по уже существующим данным или запросить его данные для ввода с клавиатуры;
- метод исключения студента из группы;
- метод поиска студента по номеру зачетной книжки;
- метод вывода информации об учебной группе.

```
// объявление класса «Учебная группа»
class Group
{
private:
     Student s[20];      // массив студентов группы
     int count;          // количество студентов в группе
     char numberGr[10];  // номер группы
public:
     // метод инициализации информации о группе
     void Init(char*, int);
     // метод добавления студента в группу –
     // информация о новом студенте вводится
     // с клавиатуры
     void AddStudent();
     // метод добавления студента в группу
     void AddStudent(Student);
     // метод исключения студента из группы
     void DeleteStudent(Student);
     // метод поиска студента по номеру зачетной книжки
     bool FindStudent(int, Student&, int&);
     // метод распечатки информации об учебной группе
     void Print();
};
```

Далее приведем описание некоторых методов этих классов. Разберем метод `Fill()` заполнения свойств объекта класса `Student`. Он имеет параметры, содержащие значения свойств объекта. Метод должен осуществить проверку корректности даты рождения и среднего балла. В

случае недопустимых значений генерируются исключения. При проверке даты рождения исключения генерируются в самом методе `Fill()`, при проверке значения среднего балла – в методе `SetAverageMark()` – изменения среднего балла успеваемости.

```
// метод заполнения свойств студента
void Student::Fill(char* f, int d, int m, int y, int ntb,
                                          double mark)
{
     strcpy(fio, f);
     numberOfTestBook = ntb;
     // проверка на корректность введенной даты
     // в случае некорректной даты генерируются исключения
     if(y < 1950 || y > 2000)
          throw 1;
     if(m < 1 || m > 12)
          throw 2;
     if ((m == 1 || m == 3 || m == 5 || m == 7 || m == 8
               || m == 10 || m == 12) && (d < 1 || d > 31))
          throw 3;
     if ((m == 4 || m == 6 || m == 9 || m == 11) &&
                                          (d < 1 || d > 30))
          throw 3;
     if (m == 2 && (d < 1 || (((y % 4 == 0 &&
          y % 100 != 0) || y % 400 == 0) && d > 29 )
          || (!((y % 4 == 0 &&  y % 100 != 0)
          || y % 400 == 0) && d > 28)))
          throw 3;
     dayBirth = d;
     monthBirth = m;
     yearBirth = y;
     SetAverageMark(mark);
}

// метод изменения среднего балла успеваемости студента
void Student::SetAverageMark(double mark)
{
     if(mark < 0.0)
          throw 1;
     averageMark = mark;
}
```

Разберем метод `Init()` инициализации объекта класса `Student`. В нем вызвается метод `Fill()` заполнения свойств объекта-студента. При начальной инициализации студентов первого курса текущая успеваемость отсутствует, поэтому значение параметра `mark` по

умолчанию устанавливается равным нулю (это указывается при объявлении метода в классе).

```
// метод инициализации данных студента
void Student::Init(char* f, int d, int m, int y, int ntb,
                                                  double mark)
{
     Fill(f, d, m, y, ntb, mark);
}
```

Остальные методы в комментариях не нуждаются и будут приведены далее.

Разберем методы класса `Group`. Сначала опишем методы добавления студента в группу. Было отмечено, что добавить студента в группу можно по существующим данным о нем или при вводе информации о новом студенте. Поэтому класс `Group` содержит два перегруженных метода с одинаковым именем `AddStudent()` и различными списками формальных параметров (без параметров и с объектом класса `Student`).

В случае ввода информации о новом студенте требуется выполнить проверку уникальности номера зачетной книжки. Для этого можно воспользоваться методом поиска студента по номеру зачетной книжки `FindStudent()`. Если номер зачетной книжки повторяется, требуется ввести другой номер. Ввод некорректной даты рождения отслеживается в методе `Fill()` заполнения свойств объекта класса `Student` посредством генерации исключения. Обработчик данного исключения вновь запрашивает ввод корректной даты рождения.

В случае, когда группа уже заполнена, метод `AddStudent()` генерирует собственное исключение. Ввод информации о новом студенте осуществляется, если группа неукомплектована.

```
// метод добавления студента в группу –  информация о
новом // студенте вводится с клавиатуры
void Group::AddStudent()
{
     if(count == 20)
          throw "Группа заполнена";
     char f[100];
     cout << "Введите фамилию, имя и отчество студента: ";
     cin.getline(f, 100);
     int ntb;
     Student temp;
```

```
        while(true)
        {
            cout << "Введите номер зачетной книжки:";
            cin >> ntb;
            int i;
            if(!FindStudent(ntb, temp, i))
                break;
            else
                cout<<"Такой номер зачетной книжки
                                        существует"<<endl;
        }
        int d, m, y;
        cout << "Введите дату рождения:";
        cin >> d >> m >> y;
        cin.get();
        try
        {
            s[count].Init(f, d, m, y, ntb);
        }
        catch(int e)
        {
            while(true)
            {
                cout << "Введите корректную дату рождения:";
                cin >> d >> m >> y;
                cin.get();
                try
                {
                    s[count].Init(f, d, m, y, ntb);
                }
                catch(int e)
                {}
                break;
            }
        }
        count++;
    }

    // метод добавления студента в группу
    void Group::AddStudent(Student std)
    {
        if(count == 20)
            throw "Группа заполнена";
        int d, m, y;
        std.GetBirthday(d, m, y);
        s[count].Init(std.GetFio(), d, m, y,
            std.GetNumberOfTestBook(), std.GetAverageMark());
        count++;
    }
```

Для инициализации группы в параметрах передается номер группы и количество студентов в ней. Метод также запрашивает ввод информации о студентах с клавиатуры, который осуществляется с помощью функции `AddStudent()`. В случае, если количество студентов, передаваемое в параметре, будет больше максимально возможного (`20`), метод `AddStudent()` сгенерирует исключение, при обработке которого выведется сообщение о заполненности группы и будет прекращен ввод студентов.

```
// метод инициализации информации о группе
void Group::Init(char* ng, int ct)
{
     count = 0;
     strcpy (numberGr, ng);
     for(int i = 0; i < ct; i++)
          try
          {
               AddStudent();
          }
          catch(char* str)
          {
               cout << str;
               return;
          }
}
```

При поиске студента может потребоваться определить, существует ли студент с заданным номером зачетной книжки, а также данные этого студента и его порядковый номер в списке группы. В нашем примере метод возвращает значение `true`, если студент найден, и `false` – в противном случае. Если студент найден, информация о нем и его порядковом номере в группе заполняется в параметры `Student& std`, `int& istud`, передаваемые по ссылке. Исходными данными для поиска этой информации является номер зачетной книжки.

```
// метод поиска студента по номеру зачетной книжки
bool Group::FindStudent(int ntb, Student& std, int& istud)
{
     for(int i =0; i < count; i++)
          if(s[i].GetNumberOfTestBook() == ntb)
          {
               std = s[i];
               istud = i;
               return true;
```

```
            }
        return false;
    }
```

Особенностью реализации метода исключения студента из группы является использование метода `Fill()` класса `Student` не для начальной инициализации объекта, а для присваивания нового значения объекту класса `Student`.

```
// метод исключения студента из группы
void Group::DeleteStudent(Student std)
{
     Student st;
     int istud;
     if(FindStudent(std.GetNumberOfTestBook(), st, istud))
     {
          for(int i = istud; i < count - 1; i++)
          {
               int d, m, y;
               s[i + 1].GetBirthday(d, m, y);
               s[i].Fill(s[i + 1].GetFio(),d, m, y,
                    s[i + 1].GetNumberOfTestBook(),
                    s[i + 1].GetAverageMark());
          }
          count--;
     }
}
```

Рассмотрим функцию `main()`, которая демонстрирует использование объектов классов `Group` и `Student`.

```
void main()
{
     Group gr;
     gr.Init("991Э", 3);
     gr.AddStudent();
     gr.Print();
     Student st;
     int index;
     gr.FindStudent(111, st, index);
     st.Print();
     gr.DeleteStudent(st);
     gr.Print();
}
```

Создание объектов классов `Group` и `Student` в данном случае не отличается от создания переменной любого другого типа данных, и, в общем случае, выглядит так:

```
имя_класса имя_объекта;
```

Обращение к элементам (данным и методам) объекта класса осуществляется через операцию «доступа к элементу», которая имеет синтаксис

```
имя_объекта.имя_элемента
```

В качестве элементов объекта могут выступать атрибуты объекта и его методы. Например,

```
Group gr;
gr.Init("991Э",3);
```

Если доступ к элементам происходит внутри метода класса, то обращение осуществляется к элементам объекта, для которого был вызван этот метод. Поэтому достаточно указать только имя элемента. Обратиться к элементам объекта, для которого вызван метод, можно также явно с помощью указателя `this`, который хранит адрес расположения объекта в памяти:

```
this->имя_элемента
```

или

```
(*this).имя_элемента
```

Приведем полный код данной программы.

Так как определение класса состоит из двух частей – объявления и реализации, его можно поместить в два файла: файл объявления класса (имеет расширение `*.h`) и файл реализации методов класса (имеет расширение `*.cpp`). Заметим, что такое разбиение необязательно и можно в один файл реализации поместить полное определение класса.

Будем определять каждый класс в двух файлах: класс `Student` – в файлах “`Student.h`” и “`Student.cpp`”, класс `Group` – в файлах “`Group.h`” и “`Group.cpp`”.

Файл реализации, как и любой другой файл, использующий некоторый класс, должен содержать директиву подключения файла с объявлением класса. Его следует включать в тот или иной файл только

один раз. Но может случиться так, что один и тот же h-файл может быть подключен несколько раз косвенно через другие заголовочные файлы. В нашем примере файл “Student.h” должен быть подключен в файлах “Student.cpp”, “Group.h” и “main.cpp”, поскольку в них используется класс Student для создания объектов. При этом в файле “main.cpp” происходит подключение файла “Student.h” дважды: явно и через “Group.h”. В этом случае возникает ошибка. Чтобы избежать этого, существует стандартный прием. В его основу положена директива

```
#ifndef _STUDENT_H_
#define _STUDENT_H_
// объявление класса Student
.    .    .
#endif
```

Она означает: выполнить операторы, заключенные между #ifndef и #endif только в случае, если имя _STUDENT_H_ не определено, т.е. ранее не встречалась директива #define _STUDENT_H_. Тогда файл “Student.h” будет подключен к другому файлу только один раз.

```
// файл Student.h

#ifndef _STUDENT_H_
#define _STUDENT_H_

// объявление класса «Студент»
class Student
{
private:
     char fio[100]; // фамилия, имя, отчество
     int dayBirth;  // дата рождения
     int monthBirth;
     int yearBirth;
     int numberOfTestBook; // номер зачетной книжки
     double averageMark; // средний балл успеваемости
public:
     // метод инициализации данных студента
     void Init(char*, int, int, int, int, double = 0.0);
     // метод заполнения свойств студента
     void Fill(char*, int, int, int, int, double);
     // метод получения фамилии,
     // имени и отчества студента
     char* GetFio();
     // метод получения даты рождения
```

```
     void GetBirthday(int&, int&, int&);
     // метод получения номера зачетной книжки
     int GetNumberOfTestBook();
     // метод получения среднего балла
     // успеваемости студента
     double GetAverageMark();
     // метод изменения среднего балла
     // успеваемости студента
     void SetAverageMark(double);
     // метод распечатки информации о студента
     void Print();
};
#endif
```

// файл Student.cpp

```
# include <iostream>
# include <cstring>

// подключения файла объявления класса
# include "Student.h"

using namespace std;

// метод заполнения свойств студента
void Student::Fill(char* f, int d, int m, int y, int ntb,
                                              double mark)
{
     strcpy(fio, f);
     numberOfTestBook = ntb;
     // проверка на корректность введенной даты
     // в случае некорректной даты генерируются исключения
     if(y < 1950 || y > 2000)
          throw 1;
     if(m < 1 || m > 12)
          throw 2;
     if ((m == 1 || m == 3 || m == 5 || m == 7 || m == 8
               || m == 10 || m == 12) && (d < 1 || d > 31))
          throw 3;
     if ((m == 4 || m == 6 || m == 9 || m == 11) &&
                                        (d < 1 || d > 30))
          throw 3;
     if (m == 2 && (d < 1 || (((y % 4 == 0 &&
           y % 100 != 0) || y % 400 == 0) && d > 29 )
           || (!((y % 4 == 0 && y % 100 != 0)
           || y % 400 == 0) && d > 28)))
          throw 3;
     dayBirth = d;
     monthBirth = m;
```

```
     yearBirth = y;
     SetAverageMark(mark);
}

// метод инициализации данных студента
void Student::Init(char* f, int d, int m, int y, int ntb,
                                                  double mark)
{
     Fill(f, d, m, y, ntb, mark);
}


// метод получения фамилии, имени и отчества студента
char* Student::GetFio()
{
     return fio;
}


// метод получения даты рождения
void Student::GetBirthday(int& d, int& m, int& y)
{
     d = dayBirth;
     m = monthBirth;
     y = yearBirth;
}


// метод получения номера зачетной книжки
int Student::GetNumberOfTestBook()
{
     return numberOfTestBook;
}

// метод получения среднего балла успеваемости студента
double Student::GetAverageMark()
{
     return averageMark;
}

// метод изменения среднего балла успеваемости студента
void Student::SetAverageMark(double mark)
{
     if(mark < 0.0)
          throw 1;
     averageMark = mark;
}

// метод распечатки информации о студента
```

```
void Student::Print()
{
     cout << "Фамилия, Имя, Отчество:" << fio;
     cout << " Дата Рождение: " << dayBirth << "." <<
                          monthBirth << "." << yearBirth;
     cout<< "Номер зачетной книжки:" << numberOfTestBook;
     cout<< " Средний балл успеваемости:" <<
                                     averageMark << endl;
}
```

// файл Group.h

```
#include "Student.h"
#ifndef _GROUP_H_
#define _GROUP_H_

#include "Student.h"

// объявление класса "Учебная группа"
class Group
{
private:
     Student students[20];    // массив студентов группы
     int count;          // количество студентов в группе
     char numberGr[15];       // номер группы
public:
     // метод инициализации информации о группе
     void Init(char*, int);
     // метод добавления студента в группу –
     // информация о новом студенте
     // вводится с клавиатуры
     void AddStudent();
     // метод добавления студента в группу
     void AddStudent(Student);
     // метод исключения студента из группы
     void DeleteStudent(Student);
     // метод поиска студента по номеру зачетной книжки
     bool FindStudent(int, Student&, int&);
     // метод распечатки информации об учебной группе
     void Print();
};
#endif
```

// файл Group.cpp

```
# include <iostream>
# include <cstring>
```

```
// подключения файла объявления класса
# include "Group.h"

using namespace std;

// метод инициализации информации о группе
void Group::Init(char* ng, int ct)
{
     count = 0;
     strcpy (numberGr, ng);
     for(int i = 0; i < ct; i++)
     try
     {
          AddStudent();
     }
     catch(char* str)
     {
          cout << str;
          return;
     }
}

// метод добавления студента в группу – информация о новом
// студенте вводится с клавиатуры
void Group::AddStudent()
{
     if(count == 20)
          throw "Группа заполнена";
     char f[100];
     cout << "Введите фамилию, имя и отчество студента: ";
     cin.getline(f, 100);
     int ntb;
     Student temp;
     while(true)
     {
          cout <<"Введите номер зачетной книжки:";
          cin >> ntb;
          int i;
          if(!FindStudent(ntb, temp, i))
               break;
          else
               cout<<"Такой номер зачетной книжки
                                   существует"<<endl;
     }
     int d, m, y;
     cout << "Введите дату рождения:";
     cin >> d >> m >> y;
     cin.get();
     try
```

```
        {
            students[count].Init(f, d, m, y, ntb);
        }
        catch(int e)
        {
            while(true)
            {
                cout << "Введите корректную дату рождения:";
                cin >> d >> m >> y;
                cin.get();
                try
                {
                    students[count].Init(f, d, m, y, ntb);
                }
                catch(int e)
                {
                }
                break;
            }
        }
        count++;
}

// метод добавления студента в группу
void Group::AddStudent(Student std)
{
        if(count == 20)
            throw "Группа заполнена";
        int d, m, y;
        std.GetBirthday(d, m, y);
        students[count].Init(std.GetFio(), d, m, y,
            std.GetNumberOfTestBook(), std.GetAverageMark());
}

// метод исключения студента из группы
void Group::DeleteStudent(Student std)
{
        Student st;
        int istud;
        if(FindStudent(std.GetNumberOfTestBook(), st, istud))
        {
            for(int i = istud; i < count - 1; i++)
            {
                int d, m, y;
                students[i + 1].GetBirthday(d, m, y);
                students[i].Fill(students[i + 1].GetFio(),
                d,m,y,students[i + 1].GetNumberOfTestBook(),
                students[i + 1].GetAverageMark());
```

```
            }
            count--;
        }
}

// метод поиска студента по номеру зачетной книжки
bool Group::FindStudent(int ntb, Student& std, int& istud)
{
        for(int i = 0; i < count; i++)
        if(students[i].GetNumberOfTestBook() == ntb)
        {
            std = students[i];
            istud = i;
            return true;
        }
        return false;
}

// метод распечатки информации об учебной группе
void Group::Print()
{
        cout << "Группа " << numberGr << endl;
        for(int i = 0; i < count; i++)
        {
            cout << (i + 1) << ". ";
            students[i].Print();
        }
}
```

// файл main.cpp

```
# include <iostream>
# include <cstring>
# include "Student.h"
# include "Group.h"

using namespace std;

void main()
{
        Group gr;
        gr.Init("991Э",3);
        gr.AddStudent();
        gr.Print();
        Student st;
        int index;
        gr.FindStudent(111, st, index);
        st.Print();
```

```
        gr.DeleteStudent(st);
        gr.Print();
}
```

2.2.4. Конструкторы и деструктор

Обратимся к примеру о студентах и учебной группе, приведенному в разделе 2.2.3. Как уже было сказано, объекты класса являются обычными переменными. Поэтому использование объектов класса должно быть таким же удобным, как и использование переменных. Например, хотелось бы иметь возможность инициализировать объект класса при его создании так же, как и объект структуры:

```
// ошибка !!!
Student std={"Иванов Иван Иванович", 21, 5,
                                    1989, 123, 0.0};
```

Однако в этом случае возникнет ошибка. Причина ее в том, что закрытые свойства класса недоступны извне. К ним могут обращаться только методы этого класса. Именно поэтому в нашем примере созданы функции инициализации объектов классов `Student` и `Group`. С их помощью инициализация объекта происходит в два этапа: создание переменной и заполнение ее данными:

```
// правильно !
Student std;
std.Init("Иванов Иван Иванович", 21, 5, 1989, 123, 0.0);
```

Но даже при использовании этого приема для создания объекта могут возникнуть ошибки. Допустим, при создании учебной группы программист забыл вызвать метод инициализации. Тогда свойство `count` (количество студентов в группе) будет иметь некорректное значение. В дальнейшем вызов методов класса `Group` может привести к возникновению ошибки, поскольку это свойство используется при обращении к элементам массива объектов-студентов. Этого можно избежать, если предусмотреть обработку этих ошибок во всех методах класса. Но такой подход очень трудоемкий. Гораздо легче избежать ошибок, если будет производиться корректная инициализация данных.

Подобные проблемы решаются с помощью специальных функций, которые вызываются при создании объекта и называются

конструкторами. В задачу конструктора входят построение новых объектов и их инициализация.

Прототип конструктора имеет следующий вид:

```
имя_класса (список_параметров);
```

В отличие от других методов при описании конструктора не указывается тип возвращаемого значения, а его имя совпадает с именем класса. В примере методы инициализации можно заменить конструкторами с такими прототипами:

```
// прототип конструктора класса Student
Student(char*, int, int, int, int, double);

// прототип конструктора класса Group
Group(char*, int);
```

При создании объектов одного класса возникают ситуации, когда их надо инициализировать разными данными. Например, создать учебную группу можно, задав только ее номер, а можно, как в примере, указать еще и количество студентов в ней. В этом случае определяют два перегруженных конструктора:

```
// прототип конструктора класса Group
Group(char*);
// прототип конструктора класса Group
Group(char*, int);
```

Как и перегруженные функции, эти конструкторы отличаются списками формальных параметров.

Наличие конструкторов в классе позволяет создавать и инициализировать объекты класса. Конструктор можно явно вызвать как любую функцию:

```
Student std = Student("Иванов Иван Иванович",
                                    21, 5, 1989, 123, 0.0);
```

Еще один способ предусматривает неявный вызов конструктора:

```
Student std("Иванов Иван Иванович",
                                    21, 5, 1989, 123, 0.0);
```

При создании объекта среди конструкторов класса осуществляется вызов того конструктора, у которого список параметров соответсвует списку аргументов.

Объект класса можно создать с помощью операции `new`. Тогда конструктор будет вызываться явно.

```
Student* pStd = new Student("Иванов Иван Иванович",
                                    21, 5, 1989, 123, 0.0);
```

Конструктор не может быть вызван объектами, поскольку он используется для их создания.

Если в определении класса отсутствуют конструкторы, компилятор предоставляет конструктор, заданный по умолчанию. Для класса `Student` он будет выглядеть следующим образом:

```
Student::Student()
{}
```

Этот конструктор выделяет память для хранения объекта без его инициализации. Он не имеет параметров, поэтому используется, когда явные значения для инициализации объекта отсутствуют. Если существует хотя бы один конструктор в определении класса, то конструктор по умолчанию не генерируется и его можно определить явно. Он может быть пустым (совпадающим с тем, который предоставляется компилятором), а может содержать операторы, выполняющие вспомогательные действия. Например, для класса `Group` конструктор по умолчанию должен обнулять количество студентов в группе:

```
Group :: Group ()
{
     count = 0;
}
```

Обязательно наличие конструктора по умолчанию в классе в случае использования массивов объектов этого класса. В нашем примере в классе `Group` содержится массив объектов-студентов. При создании объекта класса `Group` автоматически вызывается конструктор по умолчанию для каждого элемента массива `students`. Поскольку в классе `Student` метод инициализации был заменен конструктором с

параметрами, конструктор по умолчанию генерироваться не будет, и нам необходимо создать его самим.

Рассмотренный нами пример имеет недостаток, связанный с неэффективным использованием памяти. При создании объекта класса `Group` память под массив студентов выделяется статически для хранения двадцати элементов. Тем самым, с одной стороны, мы ограничиваем количество студентов в группе (оно не должно превышать `20`). С другой стороны, если в группе будет двенадцать студентов, останется неиспользованной память для восьми элементов массива. Этот недостаток может быть устранен посредством динамического выделения памяти под массив студентов. В этом случае определение класса `Group` должно измениться:

- в конструкторе класса `Group` должна выделяться память под массив студентов;
- при добавлении или исключении студента из группы должен меняться размер массива, для чего выделяется новая область памяти достаточного размера, и в новый массив записывается модифицированный набор данных;
- по-разному должны быть реализованы функции добавления нового студента, если нет выделенной памяти для его хранения, и если требуется заполнить уже выделенную память данными о студенте. В класс добавляется еще одна функция `FillStudent()` для заполнения уже существующего объекта, а функция `AddStudent()` будет осуществлять добавление нового студента, изменяя размер массива;
- в функции удаления студента теперь также необходимо изменять размер массива при удалении одного элемента. Это повлечет изменения в функции `DeleteStudent()`.

Приведем новые и модифицированные методы класса `Group`. Конструкторы класса имеют следующий вид:

```
// Конструктор по умолчанию
Group::Group()
{
     count = 0;
     students = NULL;
}

// Конструктор с параметрами
Group::Group(char* ng, int ct)
```

```
{
     count = 0;
     students = new Student[ct];
     strcpy (numberGr, ng);
     for(int i = 0; i < ct; i++)
          FillStudent();
}
```

Метод `FillStudent()` является аналогом метода `AddStudent()` предыдущего варианта программы. В нем также осуществляется ввод информации о студенте и заполнение соответствующего элемента массива `students`.

```
// метод заполнения свойств студента
void Group::FillStudent()
{
     char f[100];
     cout << "Введите фамилию, имя и отчество студента: ";
     cin.getline(f, 100);
     int ntb;
     Student temp;
     while(true)
     {
          cout << "Введите номер зачетной книжки:";
          cin >> ntb;
          int i;
          if(!FindStudent(ntb, temp, i))
               break;
          else
               cout << "Такой номер зачетной книжки
                                        существует" << endl;
     }
     int d, m, y;
     cout << "Введите дату рождения:";
     cin >> d >> m >> y;
     cin.get();
     try
     {
          students[count].Fill(f, d, m, y, ntb, 0.0);
     }
     catch(int e)
     {
          while(true)
          {
               cout << "Введите корректную дату рождения:";
               cin >> d >> m >> y;
               cin.get();
```

```
                try
                {
                     students[count].Fill(f, d, m, y,
                                                ntb, 0.0);
                }
                catch(int e)
                {}
                break;
           }
      }
      count++;
}
```

При добавлении студента в группу создается новый массив размерностью (`count + 1`) и в него копируется прежний список студентов. Информация о новом студенте заносится в последний элемент массива с помощью метода `FillStudent()`. Память, занятая прежним массивом студентов, освобождается, и сформированный массив студентов становится атрибутом объекта класса `Group`.

```
// метод добавления студента в группу – информация о новом
// студенте вводится с клавиатуры
void Group::AddStudent()
{
     Student* tmp = new Student[count + 1];
     for(int i = 0; i < count; i++)
          tmp[i]=students[i];
     delete [] students;
     students = tmp;
     FillStudent();
}
```

Отличие метода `AddStudent(Student std)` заключается в том, что информация о новом студенте, которая передается в качестве параметра, заносится в последний элемент массива.

```
// метод добавления студента в группу
void Group::AddStudent(Student std)
{
     Student* tmp = new Student[count + 1];
     for(int i = 0; i < count; i++)
          tmp[i] = students[i];
     delete [] students;
     students = tmp;
     students[count] = std;
     count++;
}
```

Метод `DeleteStudent(Student std)` отличается от прежнего варианта тем, что осуществляется формирование массива студентов размерности `(count – 1)`, в который не включен удаляемый элемент. Память, занятая прежним массивом студентов, освобождается, и сформированный массив студентов становится атрибутом объекта класса `Group`.

```
// метод исключения студента из группы
void Group::DeleteStudent(Student std)
{
     Student st;
     int istud;


     if(FindStudent(std.GetNumberOfTestBook(), st, istud))
     {
          Student* tmp = new Student[count - 1];
          for(int i = 0; i < istud; i++)
               tmp[i] = students[i];
          for(int i = istud; i < count - 1; i++)
          {
               tmp[i] = students[i + 1];
          }
          delete [] students;
          students = tmp;
          count--;
     }
}
```

При уничтожении объектов в ходе работы программ необходимо освобождать занятые им ресурсы. Поэтому кроме конструкторов, класс содержит еще один специальный метод, который выполняет данную задачу. Этот метод называется ***деструктором***. Деструктор не имеет возвращаемого значения и параметров. Имя деструктора совпадает с именем класса с предшествующим символом “~”. Деструктор в классе может быть только один. Он вызывается автоматически при уничтожении объекта класса. Если в классе не определен деструктор, то он предоставляется компилятором и освобождает память, занимаемую объектом. Деструктор обычно переопределяют в случаях, когда в методах класса происходит выделение памяти с помощью операции `new`.

В нашем примере память, занятую массивом студентов в объекте класса `Group`, надо освобождать при уничтожении объекта. Для этого добавляем в определение класса деструктор:

```
// деструктор
Group::~Group()
{
     delete [] students;
     cout << "Группа удалена" << endl;
}
```

Существует еще один вид конструктора, который называется ***конструктором копирования***. Он вызывается, когда объекты класса являются параметрами или возвращаемыми значениями некоторых функций. Также он используется в случае явного создания копии уже существующего объекта.

Например, необходимо объединить две учебные группы в одну. Для этого создадим метод `IntegrateGroup()` класса `Group`:

```
// Метод объединения двух групп
Group Group::IntegrateGroup(Group gr)
{

     Group newGroup(*this); // или
                            // Group newGroup = (*this);
     for(int i = 0; i < gr.count; i++)
          newGroup.AddStudent(gr.students[i]);
     return newGroup;
}
```

Объединение групп происходит следующим образом: к объекту-группе, для которого вызывается метод `IntegrateGroup()`, добавляются студенты из группы, передаваемой в качестве параметра. В методе создается новый объект-группа, который является копией первой группы, а затем с помощью метода `AddStudent(Student)` добавляются студенты из второй группы. Метод возвращает созданную группу.

Без определенного в классе конструктора копирования работа метода будет некорректной. Конструктор копирования, так же как и конструктор по умолчанию, предоставляется компилятором в случае его отсутствия в определении класса. Он осуществляет поэлементное копирование значений всех атрибутов объекта. При создании копии объекта класса

Group в переменную students нового объекта будет копироваться адрес массива студентов из исходного объекта. Таким образом, получается, что два объекта-группы ссылаются на один и тот же массив. Ошибка возникает при удалении одного из объектов, поскольку деструктор освободит память, занимаемую массивом студентов, и работать с другим объектом станет невозможно.

Поэтому необходимо определить конструктор копирования класса Group, который создаст новый массив студентов с теми же значениями, что и у исходного объекта:

```
// Конструктор копирования
Group::Group(const Group& gr)
{
     count = gr.count;
     strcpy (numberGr, gr.numberGr);
     // Формирование копии массива студентов
     students = new Student[gr.count];
     for(int i = 0; i < count; i++)
          students[i] = gr.students[i];
}
```

Предположим, в программе имеются объекты gr991 и gr992 класса Group. При вызове метода IntegrateGroup() для объединения этих групп конструктор копирования срабатывает в следующих случаях:

- при передаче объекта gr992 класса Group в качестве параметра метода по значению:

```
Group gr990 = gr991.IntegrateGroup(gr992);
```

- при создании копии объекта-группы в методе IntegrateGroup();

```
Group newGroup(*this);
```

- при возврате объекта-группы в качестве результата метода IntegrateGroup():

```
return newGroup;
```

В последнем случае возвращается временный объект, который является копией объекта `newGroup`, сам объект `newGroup` уничтожается, поскольку является локальной переменной функции `IntegrateGroup()`.

2.2.5. Полиморфизм

Слово «полиморфизм» означает «имеющий множество форм». В программировании под полиморфизмом понимают использование одного и того же имени для выполнения различных задач.

Полиморфизм – достаточно широкое понятие, в котором можно выделить следующие формы:

- перегрузка функций;
- использование шаблонов функций;
- перегрузка операций;
- использование методов с одним и тем же именем в различных классах, включая виртуальные функции;
- шаблоны классов.

Мы уже рассматривали перегруженные функции и шаблоны функций, которые являются частным случаем полиморфизма (см. разделы 1.3, 1.4). Шаблоны классов будут рассмотрены позднее.

В этом разделе остановимся на изучении перегрузки операций и использовании методов с одним и тем же именем в различных классах.

Перегрузка операций.

При проектировании классов, характеризующих поведение математических объектов, удобно использовать традиционные математические знаки операций для выполнения соответствующих действий. Например, при сложении двух матриц было бы понятней использовать операцию **"+"**, а не вызывать функцию `Summ()` (тем более, что другой программист может назвать эту функцию иным именем). Для таких ситуаций в C++ появилось очень удобное средство, называемое ***перегрузкой операций***. Фактически многие операции языка C++ перегружены изначально. Например, операция **"*"**, примененная к паре чисел, дает их произведение, а примененная к переменной-указателю, дает то значение, которое хранится по этому адресу (операция разадресации). Другим примером уже перегруженной операции является операция **"/"**. Примененная к целым числам, она является операцией целочисленного деления (25 / 10 = 2), а если хотя бы одно из чисел

является вещественным, результатом деления будет вещественное число (25.0 / 10.0 = 2.5). В языке С++ у программиста появляется возможность задать для собственных типов данных свои методы обработки, закрепленные за обозначением той или иной операции.

Большинству операций языка С++ соответствуют специальные операторные функции, имеющие следующий прототип:

```
тип_возвращаемого_значения operator#
                          (список_формальных_параметров);
```

Здесь "#" – это знак операции С++. Не все операции можно перегружать. К запрещенным для перегрузки операциям относятся "." , "::", "?:" и некоторые другие. Существует несколько ограничений, которые следует учитывать при перегрузке операций. Во-первых, нельзя менять приоритет операций. Во-вторых, нельзя изменять число операндов операции. К примеру, операция "!" имеет только один операнд, поэтому и ее перегруженная реализация должна быть унарной. В остальном, правила перегрузки операций совпадают с правилами перегрузки функций. Перегруженные операции должны отличаться списками параметров. Например, операция "*" для матриц может быть перегружена как функция умножения двух матриц или функция умножения матрицы на число.

Операторные функции могут быть как методами класса, так и внешними по отношению к классу функциями. Разберем принципы создания таких функций для реализации операции умножения в классе Matrix:

```
class Matrix
{
     // определение структурных свойств
     double** a;    // адрес двумерного массива
                    // для хранения матрицы
     int m, n;      //количество строк и столбцов в матрице
     // определение поведенческих свойств
public:
     // конструктор создания матрицы определенного размера
     Matrix(int, int);
     // конструктор копирования
     Matrix(const Matrix&);
     // деструктор
     ~Matrix();
     void InputMatrix();// метод ввода матрицы
```

```
        void OutputMatrix();// метод вывода матрицы
        .    .    .
};
```

Перегрузим в классе `Matrix` операцию умножения двух матриц. Данная операция является бинарной, т.е. имеет два операнда типа `Matrix`. Если опрерация является методом класса `Matrix`, то ее вызов возможен только через объект класса. Этим объектом является первый операнд операции умножения. Он передается в метод неявно и к его свойствам и методам можно обращаться напрямую или с помощью указателя `this`. В качестве параметра в этом случае передается только второй операнд операции.

```
// операция перемножения двух матриц
Matrix Matrix::operator * (Matrix& ob)
{
     if(n == ob.m)
     {
          // создание матрицы-результата
          Matrix temp(m, ob.n);
          // заполнение матрицы-результата
          for(int i = 0; i < m; i++)
               for(int j = 0; j < ob.n; j++)
                    for(int k = 0; k < n; k++)
                         temp.a[i][j] +=
                                   (a[i][k] * ob.a[k][j]);
          return temp;
     }
     else
          // количество столбцов первой матрицы должно
          // быть равно количеству строк второй матрицы
          throw "Матрицы таких размеров
                                   перемножать нельзя";
}
```

Вызов перегруженной операции можно осуществить также как и вызов других методов класса:

```
.    .    .
Matrix a(2,3), b(3,4);
a.InputMatrix();
b.InputMatrix();
Matrix c = a.operator*(b);
.    .    .
```

Однако значительно удобнее использовать более естественное обращение к операции:

```
.    .    .
Matrix a(2,3), b(3,4);
a.InputMatrix();
b.InputMatrix();
Matrix c = a * b;
.    .    .
```

Для умножения матрицы на число должна быть перегружена операция "*", она также будет бинарной, но будет отличаться типом второго операнда:

```
// операция умножения матрицы на число
Matrix Matrix::operator * (double b)
{
     Matrix temp(m, n);
     // заполнение матрицы-результата
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               temp.a[i][j] = a[i][j] * b;
     return temp;
}
```

Если при обращении к операции умножения второй операнд будет являться числовой константой или переменной числового типа, будет вызываться последняя реализация перегруженной операции.

```
.    .    .
Matrix a(2,3);
a.InputMatrix();
Matrix c = a * 5;   // эквивалентное обращение
                    // Matrix c = a.operator*(5);
.    .    .
```

Однако обращение к операции умножения в случае, когда число является первым операндом, пока невозможно. Поэтому имеет смысл перегрузить операцию "*" для случая, когда первый операнд является числом, а второй – матрицей. Но для этого нельзя использовать операторную функцию, которая является методом класса, поскольку для нее первым операндом обязательно должен быть объект класса.

В таких случаях перегрузка операций осуществляется с помощью внешних по отношению к классу функций. Такая функция не является

методом какого-либо класса и поэтому все операнды операции указываются в этой функции в качестве параметров. При этом возникает проблема доступа к данным объекта класса. Напомним, что структурные свойства класса обычно являются закрытыми членами класса и недоступны для других функций. Существует механизм, который позволяет дать возможность доступа к закрытым членам класса из некоторой внешней функции или методов другого класса. Данный механизм называют ***дружественностью***. Если функция является дружественной классу, в объявление класса включается ее прототип, предваренный ключевым словом `friend`:

```
class Matrix
{
     .    .    .
     friend Matrix operator*(double, Matrix);
     .    .    .
};
```

Реализация дружественной функции будет иметь следующий вид:

```
// операция умножения числа на матрицу
Matrix operator * (double b, Matrix ob)
{
     Matrix temp(ob.m, ob.n);
     // заполнение матрицы-результата
     for(int i = 0; i < ob.m; i++)
          for(int j = 0; j < ob.n; j++)
               temp.a[i][j] = ob.a[i][j] * b;
     return temp;
}
```

Вызов этой операции **"*"** будет таким:

```
.    .    .
Matrix a(2,3);
a.InputMatrix();
Matrix c = 5 * a;
.    .    .
```

Отметим, что в качестве дружетвенных функций могут выступать не только операции, но и обычные функции, которым необходим доступ к закрытым элементам класса.

Когда операндами являются объекты разных классов (например, перемножение вектора и матрицы), то можно реализовать перегрузку одним из двух способов:

- создать внешнюю функцию и сделать ее дружественной для обоих классов (функция будет иметь два параметра);
- перегрузить функцию в классе первого операнда, а во втором классе объявить ее дружественной. В этом случае функция будет иметь один параметр – второй операнд операции.

Перегрузка унарных операций аналогична перегрузке бинарных операций за исключением того, что она происходит не с двумя, а с одним операндом. В связи с этим при перегрузке унарной операции в классе этот метод не имеет параметров. Объект, вызывающий операцию, будет являться ее единственным операндом.

В качестве примера перегрузим операции "!" и "~" для класса `Matrix`. Операция "!" будет выполнять транспонирование матрицы, а операция "~" – получение обратной матрицы.

```
// перегрузка операции "!" - транспонирование матрицы
Matrix Matrix::operator !()
{
     Matrix temp(n, m);
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               temp.a[j][i] = a[i][j];
     return temp;
}
```

Для вычисления обратной матрицы предполагаем, что в классе `Matrix` разработаны методы вычисления определителя квадратной матрицы `Determinant()` и получения ее подматрицы, путем удаления заданных строки и столбца `SubMatrix()`.

```
// перегрузка операции "~" - получение обратной матрицы
Matrix Matrix::operator ~()
{
     // если матрица неквадратная, то обратной матрицы
     // не существует
     if(n != m)
          throw 1;
     Matrix res(n, n);
     // вычисление определителя матрицы
     double det = Determinant();
     // если определитель матрицы равен 0,
```

```
        // обратной матрицы не существует
        if(det == 0)
             throw 2;
        // обратная матрица – это транспонированная матрица
        // алгебраических дополнений, поделенных на
        // значение определителя исходной матрицы
        int z;
        for(int i = 0; i < n; i++)
        {
             z = i%2==0 ? 1 : -1;
             for(int j = 0; j < n; j++)
             {
                  Matrix temp = SubMatrix(i, j);
                  res.a[j][i] = z * temp.Determinant() / det;
                  z = -z;
             }
        }
        return res;
   }
```

Операторная функция не должна содержать дополнительных параметров, кроме тех, которые определяют ее операнды. Поэтому, в случае возникновения ошибки при выполнении операции, сообщение о ней не может быть передано через параметры или возвращаемое значение. Для обработки таких ситуаций удобно использовать генерацию исключений, что было продемонстрировано в примерах: в операции умножения матриц при некорректных размерах и в операции вычисления обратной матрицы в случаях, когда исходная матрица не является квадратной или когда определитель матрицы равен нулю.

Если разрабатывается класс, в котором происходит выделение и освобождение памяти, то в дополнение к конструкторам и деструктору класса требуется перегрузить операцию присваивания (операция "="). Если она не перегружена, то при ее выполнении происходит поэлементное копирование значений из объекта, стоящего справа от символа "=", в объект, стоящий слева. Таким образом, два или более объектов будут ссылаться на одну и ту же область памяти. Поэтому при уничтожении одного из этих объектов другими пользоваться будет нельзя. В перегруженной версии необходимо освобождать память в объекте – левом операнде, в нем же выделять память, необходимую для хранения данных правого операнда, и произвести копирование данных правого операнда в левый. Приведем реализацию перегруженного оператора присваивания для класса `Matrix`:

```
// операция присваивания матриц
Matrix& Matrix::operator = (Matrix& ob)
{
     if(n != ob.n || m != ob.m)
     {
          // освобождение памяти в левом операнде
          for(int i = 0; i < m; i++)
               delete [] a[i];
          delete [] a;
          // выделение памяти в левом операнде
          n = ob.n;
          m = ob.m;
          a = new double* [m];
          for(int i = 0; i < m; i++)
               a[i] = new double [n];
     }
     // копирование данных правого операнда в левый
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = ob.a[i][j];
     return *this;
}
```

Результатом оператора присваивания является левый операнд. При такой реализации возможно следующее его использование:

```
.    .    .
Matrix a(2,2), b(2,2), c(2,2);
a.InputMatrix();
c = b = a;
c.OutputMatrix();
.    .    .
```

В данном случае матрицы `b` и `c` будут содержать те же данные, что и матрица `a`.

Отметим, что оператор присваивания по своей реализации аналогичен конструктору копирования. Отличие состоит в том, что конструктор копирования выполняется в момент создания объекта-копии, а оператор присваивания используется для существующих объектов.

Перегрузка операций ввода/вывода.

В C++ ввод и вывод выполняется с помощью специальных операций потокового ввода/вывода, использование которых мы рассматривали в разделе 1.1. Эти же операции можно использовать и для организации ввода/вывода объектов пользовательских типов данных. Это осуществляется с помощью их перегрузки.

Операции ввода/вывода нельзя определить как методы класса из-за того, что первым операндом этой операции должен быть объект входного или выходного потока. Таким образом, перегрузка этих операций осуществляется с помощью дружественных функций.

Определим для класса `Matrix` ввод/вывод:

```
class Matrix
{
     .    .    .
     friend istream& operator >> (istream&, Matrix&);
     friend ostream& operator << (ostream&, Matrix);
     .    .    .
};

// перегруженная операция ввода матрицы
istream& operator >> (istream& in, Matrix& ob)
{
     if(ob.a != NULL)
     {
          for(int i = 0; i < ob.m; i++)
               for(int j = 0;j < ob.n; j++)
                    in >> ob.a[i][j];
     }
     return in;
}

// перегруженная операция вывода матрицы
ostream& operator << (ostream& out, Matrix ob)
{
     if(ob.a != NULL)
     {
          out << "Матрица:" << endl;
     for(int i = 0; i < ob.m; i++)
     {
          for(int j = 0; j < ob.n; j++)
               out << ob.a[i][j] << "\t";
          out << endl;
     }
     }
     else
          out << "Матрица пуста" << endl;
     return out;
}
```

Первым параметром операций ввода/вывода является ссылка на поток ввода или вывода. В функции ввода второй параметр должен передаваться по ссылке, поскольку в нем сохраняются вводимые данные.

Возвращаемым значением этих функций также должна быть ссылка на соответствующий поток. Это необходимо в случае, если ввод/вывод осуществляется в один поток частями. Например,

```
.    .    .
Matrix a(2,2), b(2,2), c(2,2);
cin >> a >> b >> c;
.    .    .
cout << a << b << c;
.    .    .
```

Перегрузка операций преобразования типов.

Класс может содержать методы для осуществления операции ***преобразования*** в другие типы данных или из них. Они реализуются:

- с помощью конструкторов, когда преобразование происходит к типу класса;
- с помощью специальных операторов, когда объект класса преобразуется к другим типам данных.

Оператор преобразования к типу данных имеет вид:

```
operator имя_типа(); // имя_типа – имя оператора
```

Этот оператор не имеет возвращаемого типа, поскольку он совпадает с именем оператора. Также он не имеет аргументов, поскольку является унарным.

Приведем пример перегрузки операции преобразования типа для класса `Group` из приложения, автоматизирующего работу деканата, рассмотренного ранее. Добавим метод получения средней оценки студентов в группе, оформив его в виде оператора преобразования к типу `double`.

```
// операция получения среднего балла успеваемости
//студентов группы – операция преобразования типа
Group::operator double()
{
     double averageMark = 0.0;
     for (int i = 0; i < count; i++)
          averageMark += students[i].GetAverageMark();
     averageMark /= count;
     return averageMark;
}
```

Вызвать данный оператор явно с указанием типа данных, к которому осуществляется преобразование, можно следующим образом:

```
.    .    .
Group gr991("991Э", 12);
cout << "Средний балл учебной группы " << (double) gr991;
.    .    .
```

Также возможен неявный вызов данного оператора:

```
double aM =  gr990;
```

Когда преобразование осуществляется к типу класса, должен создаваться объект этого класса. Такое преобразование делают конструкторы с одним параметром. Например, пусть требуется преобразовать символьную строку, содержащую номер группы, в объект класса `Group`. К этому конструктору можно обратиться как к оператору преобразования типа, указав имя класса явно, или, не указывая его (неявное преобразование):

```
.    .    .
char* numGr = "991Б";
Group gr991B = (Group)numGr;// явное преобразование
Group gr991B = numGr;    // неявное преобразование
.    .    .
```

Использование методов с одинаковым именем в разных классах.

Несколько классов могут иметь методы с одинаковыми именами. Это означает, что действия, выполняемые одноименными методами, могут различаться в зависимости от того, к какому из классов относится тот или иной метод. Например, в классе `Matrix` определим оператор сложения двух матриц:

```
// операция сложения двух матриц
Matrix Matrix::operator + (Matrix& ob)
{
     if(ob.n == n && ob.m == m)
     {
          Matrix temp(m, n);
          for(int i = 0; i < m; i++)
               for(int j = 0; j < n; j++)
                    temp.a[i][j] = a[i][j] + ob.a[i][j];
          return temp;
```

```
        }
        else
            throw 1;
}
```

Также определим оператор сложения для класса `Group`, который будет объединять две учебные группы в одну:

```
// операция сложения – объединение двух групп
Group Group::operator + (Group& gr)
{
    Group newGroup(*this);
    strcpy(newGroup.numberGr, numberGr);
    strcat(newGroup.numberGr, "-");
    strcat(newGroup.numberGr, gr.numberGr);
    for(int i = 0; i < gr.count; i++)
        newGroup.AddStudent(gr.students[i]);
    return newGroup;
}
```

В обоих классах перегружается один и тот же оператор, который для объектов каждого класса выполняет разные действия.

```
.    .    .
Matrix a(2,2), b(2,2);
cin >> a >> b;
Matrix c;
c = a + b;
Group gr991("991Э", 12), gr992("992Э", 15);
Group newGr;
newGr = gr991 + gr992;
.    .    .
```

Выбор метода для выполнения определяется типом объекта, для которого он вызывается. В нашем случае оператор, который необходимо выполнить, определяется типом первого операнда.

2.2.6. Наследование

Для любого объектно-ориентированного приложения необходимо определить объекты, которые оно будет содержать. В некоторых случаях эти объекты могут быть схожими по своей структуре и поведению, но при этом иметь существенные различия. Также бывают ситуации, когда

некоторые объекты являются частями других объектов. Подобные взаимосвязи образуют иерархию объектов – их упорядочивание. Основными видами иерархических структур применительно к объектно-ориентированным приложениям являются структура классов (иерархия «is-a» – «является», «is-like-a» – «является подобным») и структура объектов (иерархия «is-part-of» – «является частью»), которые определяются отношениями агрегации и обобщения.

Иерархия «is-part-of» была представлена ранее в приложении, автоматизирующем работу деканата. Ее определяет взаимосвязь классов «Учебная группа» (`Group`) и «Студент» (`Student`) – «студент является частью учебной группы».

Продемонстрируем использование другого вида иерархии. В том же приложении можно определить два класса студентов: бюджетной (`Student`) и контрактной (`ContractStudent`) формы обучения. Большая часть их структурных и поведенческих свойств совпадает. Объекты этих классов описываются фамилией, именем и отчеством студента, его датой рождения, номером зачетной книжки, средним баллом успеваемости и т.д. И те и другие сдают зачеты и экзамены, получают баллы, переводятся на следующий курс. При этом у студентов-контрактников имеются дополнительные характеристики: объекты этого класса могут содержать информацию об общей стоимости обучения за год, внесенной сумме, оставшейся задолженности, методы доступа и изменения этих свойств, методы проверки задолженности по оплате обучения и т.д.

Для того чтобы не определять общие свойства и методы для этих классов дважды используются принцип ООП, называемый наследованием. ***Наследование*** – это механизм, который позволяет создавать новые классы на основе существующих, используя их структурные и поведенческие свойства. Новые классы называют дочерними (производными или подклассами), а классы, на основе которых происходит наследование, – родительскими (базовыми или суперклассами). Кроме наследуемых свойств дочерние классы обладают дополнительными свойствами, которые и отличают их от родительских.

В нашем примере в качестве родительского класса выступает класс `Student`, который наследуется дочерним классом `ContractStudent`. Данное отношение классов образует иерархию вида «is-a» – «студент-контрактник является студентом».

Использование принципа наследования позволяет расширить базовый класс посредством создания производного класса, содержащего дополнительно:

- новые данные, которые будут определять дочерний класс. Например, расширяя структурные свойства класса `Student`, добавим переменные для хранения значений общей стоимости обучения за год, внесенной суммы, оставшейся задолженности и, тем самым, образуем структурные свойства нового класса `ContractStudent`;
- новые поведенческие свойства дочернего класса. Например, новым поведением объектов класса `ContractStudent` может служить внесение оплаты за обучение;
- переопределенные поведенческие свойства базового класса. Например, если для студентов бюджетной формы обучения для определения допуска к сессии необходимо наличие определенного количества набранных баллов, то для студентов-контрактников, кроме этого, необходимо отсутствие задолженности по оплате. Это приведет к переопределению в дочернем классе метода базового класса, осуществляющего проверку допуска студента к сессии. Переопределенный метод имеет тот же прототип и скрывает в производном классе метод базового класса.

Теперь подробно рассмотрим применение принципа наследования на примере приложения, автоматизирующего работу деканата. Как было указано выше, родительским классом является класс `Student`:

```
// объявление класса "Студент"
class Student
{
protected:
     char fio[100]; // фамилия, имя, отчество
     int dayBirth;  // дата рождения
     int monthBirth;
     int yearBirth;
     int numberOfTestBook; // номер зачетной книжки
     double averageMark;    // средний балл успеваемости
     static int countTests; // количество зачетов
                            // в текущую сессию
     static int countExams; // количество экзаменов
                            // в текущую сессию
     double tests[6]; // баллы за текущую работу (зачеты)
```

```
        double exams[4]; // баллы за экзамены
    public:
        // конструкторы
        Student(char*, int, int, int, int, double = 0.0);
        Student();
        // метод заполнения свойств студента
        void Fill(char*, int, int, int, int, double = 0.0);
        // метод получения фамилии, имени и отчества студента
        char* GetFio();
        // метод получения даты рождения
        void GetBirthday(int&, int&, int&);
        // метод получения номера зачетной книжки
        int GetNumberOfTestBook();
        // метод получения среднего балла
        // успеваемости студента
        double GetAverageMark();
        // метод изменения среднего балла
        // успеваемости студента
        void SetAverageMark();
        // метод распечатки информации о студента
        void Print();
        // метод сдать зачет
        void SetTest(int, double);
        // метод сдать экзамен
        void SetExam(int, double);
        // метод проверки допуска к экзаменационной сессии
        bool PassedTests();
        // метод проверки сдана сессии
        bool PassedExams();
        // метод допуска к обучению в новой сессии
        bool NextSession();
    };
```

По сравнению с предыдущей версией, в класс были добавлены новые структурные и поведенческие свойства: количество зачетов и экзаменов, которые необходимо сдать в текущем семестре, массивы полученных баллов за зачеты и экзамены, методы регистрации баллов, полученных за зачет и экзамен, проверки допуска к экзаменационной сессии, проверки сдачи экзаменационной сессии и метод допуска студента к обучению в следующем семестре.

Предположим, что каждый студент во время зачетной и экзаменационной сессий должен сдать шесть зачетов и четыре экзамена. Соответствующие свойства класса (количество зачетов и количество экзаменов) будут иметь одинаковые значения для всех объектов этого класса. Поэтому удобно хранить такие значения в одном экземпляре. Для этого в языке C++ существуют ***статические переменные класса***. Такие

переменные создаются один раз и доступны для любого объекта этого класса. Значение статической переменной класса должно быть проинициализировано глобально:

```
int Student::countTests = 6;
int Student::countExams = 4;
```

Кроме новых свойств и методов в новой версии класса `Student` имеется еще одно отличие: все структурные свойства объявлены со спецификатором `protected`. К `protected`-элементам можно обращаться из самого класса и из производных от него классов. В рассматриваемом примере это позволяет получать доступ к элементам класса `Student` из его наследника `ContractStudent`.

Теперь определим производный класс `ContractStudent`:

```
// объявление класса "Студент-контрактник"
class ContractStudent: public Student
{
private:
     double payment; // оплата обучения за семестр
     double credit; // долг по оплате
public:
     // конструкторы класса
     ContractStudent(char*, int, int, int, int,
                                    double, double = 0.0);
     ContractStudent();
     // метод получения суммы оплаты обучения
     double GetPayment();
     // метод изменения суммы оплаты обучения
     void SetPayment(double);
     // метод получения суммы долга
     double GetCredit();
     // метод уменьшения суммы долга
     void DescCredit(double);
      // метод сдать зачет
     void SetTest(int, double);
     // метод проверки допуска к экзаменационной сессии
     bool PassedTests();
     // метод проверки, сдана ли сессия
     bool PassedExams();
     // метод сдать экзамен
     void SetExam(int, double);
     // метод допуска к обучению в новой сессии
     bool NextSession();
     // метод распечатки информации о
```

```
        // студенте-контрактнике
        void Print();
};
```

В объявлении производного класса после его имени через двоеточие необходимо указать имя базового класса:

```
class ContractStudent: public Student
{
      .    .    .
};
```

Вместе с именем базового класса указывается способ наследования: `public`, `protected` или `private`.

Public-наследование (общедоступное наследование) не изменяет режима доступа к элементам базового класса из производного. При таком наследовании общедоступные (`public`) элементы базового класса останутся общедоступными элементами в производном классе. Закрытые (`private`) элементы станут частью производного класса, но к ним можно будет обращаться только посредством общедоступных и защищенных методов базового класса. Защищенные (`protected`) элементы базового класса будут также защищенными и в производном классе.

Private-наследование (скрытое наследование) осуществляет изменение режима доступа к элементам базового класса: общедоступные и защищенные элементы базового класса становятся закрытыми элементами в производном классе. Это означает, что методы базового класса уже не являются частью общедоступного интерфейса производного класса и могут использоваться только внутри функций производного класса. Часто такое наследование используют, чтобы ограничить использование методов базового класса в производном.

Protected-наследование (защищенное наследование) является разновидностью скрытого наследования. В этом случае общедоступные элементы базового класса становятся защищенными. Основное отличие `private` и `protected`-наследования проявляется при создании нового класса из производного. При `private`-наследовании класс третьего поколения не будет иметь доступа к методам класса первого поколения, т.к. они становятся `private`-методами в классе второго поколения.

Схема изменения режима доступа к элементам базового класса

при `private` -наследования

Класс первого поколения	Класс второго поколения	Класс третьего поколения

`class A`
`{`
`public:`
`//открытые элементы`
`// класса первого`
`// поколения`
`void f();`
`. . .`
`};`
	`class B : private A`
	`{`
	`. . .`
	`};`
	
	`// метод f() класса`
	`//A стал private-`
	`//методом класса B`
		`class C : public B`
		`{`
		`. . .`
		`};`
		
		`// метод f() класса`
		`A // можно вызывать`
		`// только с помощью`
		`// методов класса B`

Схема изменения режима доступа к элементам базового класса при `protected`-наследования

Класс первого поколения	Класс второго поколения	Класс третьего поколения

`class A`
`{`
`public:`
`//открытые элементы`
`// класса первого`
`// поколения`
`void f();`
`. . .`
`};`
	`class B : protected A`
	`{`
	`. . .`
	`};`
	
	`// метод f() класса`
	`// A стал`
	`// protected-`
	`// методом класса B`
		`class C : public B`
		`{`
		`. . .`
		`};`
		
		`// метод f() класса A`
		`// можно вызвать в`
		`// классе C`

Вернемся к определению классов `Student` и `ContractStudent`. Как уже было сказано, производный класс содержит все данные и методы базового класса и собственные элементы. Так, в класс `ContractStudent` добавлены следующие новые элементы:

- структурные свойства: данные о плате за обучение (`payment`) и о задолженности студента (`credit`);
- конструктор производного класса:

```
// прототип конструктора
```

```
ContractStudent(char*, int, int, int, int, double,
                                          double = 0.0);

// определение конструктора
ContractStudent::ContractStudent(char* fio, int d,
     int m, int y, int ntb, double pay, double mark):
                    Student(fio, d, m, y, ntb, mark)
{
     if (pay < 0)
          throw 1;
     payment = pay;
     credit = payment;
}
```

- новые методы:

```
// метод получения суммы оплаты обучения
double GetPayment();
// метод изменения суммы оплаты обучения
void SetPayment(double);
// метод получения суммы долга
double GetCredit();
// метод уменьшения суммы долга
void DescCredit(double);
```

- переопределенные методы базового класса:

```
// метод сдать зачет
void SetTest(int, double);
// метод проверки допуска к экзаменационной сессии
bool PassedTests();
// метод проверки, сдана ли сессия
bool PassedExams();
// метод сдать экзамен
void SetExam(int, double);
// метод допуска к обучению в новой сессии
bool NextSession();
// метод распечатки информации
// о студенте-контрактнике
void Print();
```

При создании объекта производного класса сначала происходит создание его базовой части посредством вызова конструктора базового класса. Для такого вызова используется специальный синтаксис, называемый инициализацией в заголовке:

```
ContractStudent::ContractStudent(char* fio, int d, int m,
               int y, int ntb, double pay, double mark):
```

```
                                    Student(fio, d, m, y, ntb, mark)
{
      .     .     .
}
```

Если вызов конструктора базового класса не указывается явно, автоматически осуществляется вызов конструктора базового класса, не имеющего параметров.

Заметим, что конструктор производного класса должен инициализировать как базовую компоненту, так и собственную. Поэтому параметры конструктора производного класса содержат данные для инициализации обеих компонент. Далее параметры, инициализирующие структурные свойства базового класса передаются его конструктору, а конструктор производного класса инициализирует только собственную часть (для нашего примера свойства стоимости обучения и задолженности по оплате).

Аналогично, при разрушении объекта должны вызываться оба деструктора – базового и производного классов. Порядок вызова деструкторов обратный – сначала разрушается собственная компонента производного класса, потом автоматически вызывается деструктор базового класса для разрушения базовой компоненты.

Переопределение методов базового класса в производном вызвано необходимостью добавления новой функциональности к поведению, определенному базовым классом, или с определением принципиально другого поведения в производном классе.

При переопределении в производный класс добавляется метод с тем же именем и прототипом, что и метод базового класса. Таким образом, производный класс будет содержать два метода с одним и тем же прототипом: унаследованный от базового класса и собственный метод, который скрывает базовый метод. Объект производного класса будет вызывать переопределенный метод. Тем не менее, остается возможность вызова метода базового класса. Это бывает необходимо, когда объект производного класса должен выполнить те же действия, что и объект базового класса, и в дополнение к ним другие действия.

Например, метод класса `Student`, проверяющий, допущен ли студент к сессии, должен проверить только, сданы ли студентом все зачеты:

```
// метод проверки допуска к экзаменационной сессии
// для класса Student
```

```
bool Student::PassedTests()
{
     for(int i = 0; i < countTests; i++)
          if (tests[i] < 51)
               return false;
     return true;
}
```

В производном классе ContractStudent нужна еще проверка отсутствия задолженности по оплате. Поэтому переопределим метод PassedTest() в классе ContractStrudent. В нем после определения отсутствия задолженности по оплате вызываем одноименный метод базового класса, чтобы выяснить, имеются ли у студента академические задолженности. Это осуществляется с помощью указания имени базового класса и "::" перед именем метода при вызове:

```
// метод проверки допуска к экзаменационной сессии
// для класса ContractStudent
bool ContractStudent::PassedTests()
{
     // проверка наличия задолженности по оплате
     if (credit > 0)
          return false;
     // вызов метода базового класса для проверки,
     // сданы ли зачеты
     return Student::PassedTests();
}
```

Приведем полный программный код приложения, использующего классы Student и ContractStudent:

```
// файл "Student.h"

#ifndef _STUDENT_H_
#define _STUDENT_H_

// объявление класса "Студент"
class Student
{
protected:
     char fio[100]; // фамилия, имя, отчество
     int dayBirth;  // дата рождения
     int monthBirth;
     int yearBirth;
     int numberOfTestBook; // номер зачетной книжки
     double averageMark; // средний балл успеваемости
```

```
    static int countTests; // количество зачетов
                           //в текущую сессию
    static int countExams; // количество экзаменов
                             // в текущую сессию
    double tests[6]; // баллы за текущую работу (зачеты)
    double exams[4]; // баллы за экзамены
public:
    // конструкторы класса Student
    Student(char*, int, int, int, int, double = 0.0);
    Student();
    // метод заполнения свойств студента
    void Fill(char*, int, int, int, int, double = 0.0);
    // метод получения фамилии, имени и отчества студента
    char* GetFio();
    // метод получения даты рождения
    void GetBirthday(int&, int&, int&);
    // метод получения номера зачетной книжки
    int GetNumberOfTestBook();
    // метод получения среднего балла
    // успеваемости студента
    double GetAverageMark();
    // метод изменения среднего балла
    // успеваемости студента
    void SetAverageMark();
    // метод распечатки информации о студента
    void Print();
    // метод сдать зачет
    void SetTest(int, double);
    // метод сдать экзамен
    void SetExam(int, double);
    // метод проверки допуска к экзаменационной сессии
    bool PassedTests();
    // метод проверки сдана сессии
    bool PassedExams();
    // метод допуска к обучению в новой сессии
    bool NextSession();
};
#endif
```

// файл "Student.cpp"

```
# include <iostream>
# include <cstring>

// подключения файла объявления класса
# include "Student.h"

using namespace std;
```

```
// инициализация статических переменных класса Student
int Student::countTests = 6;
int Student::countExams = 4;

// метод заполнения информации о студенте
void Student::Fill(char* f, int d, int m, int y,
                                    int ntb, double mark)
{
     strcpy(fio, f);
     numberOfTestBook = ntb;
     // проверка на корректность введенной даты
     // в случае некорректной даты генерируются исключения
     if(y < 1950 || y > 2000)
          throw 1;
     if(m < 1 || m > 12)
          throw 2;
     if ((m == 1 || m == 3 || m == 5 || m == 7 || m == 8
           || m == 10 || m == 12) && (d < 1 || d > 31))
               throw 3;
     if ((m == 4 || m == 6 || m == 9 || m == 11) &&
          (d < 1 || d > 30))
          throw 3;
     if (m == 2 && (d < 1 || (((y % 4 == 0 &&
          y % 100 != 0) || y % 400 == 0) && d > 29)
          || (!((y % 4 == 0 && y % 100 != 0)
          || y % 400 == 0) && d > 28)))
          throw 3;
     dayBirth = d;
     monthBirth = m;
     yearBirth = y;
     averageMark = mark;
     for(int i = 0; i < countTests; i++)
          tests[i] = 0.0;
     for(int i = 0; i < countExams; i++)
          exams[i] = 0.0;
}

// конструктор класса Student
Student::Student(char* f, int d, int m, int y,
                                int ntb, double mark)
{
     Fill(f, d, m, y, ntb, mark);
}

// конструктор по умолчанию класса Student
Student::Student()
{
}
```

```
// метод получения фамилии студента
char* Student::GetFio()
{
     return fio;
}

// метод получения даты рождения студента
void Student::GetBirthday(int& d, int& m, int& y)
{
     d = dayBirth;
     m = monthBirth;
     y = yearBirth;
}

// метод получения номера зачетной книжки
int Student::GetNumberOfTestBook()
{
     return numberOfTestBook;
}

// метод получения среднего балла успечаемости студента
double Student::GetAverageMark()
{
     return averageMark;
}

// метод вычисления среднего балла успеваемости
void Student::SetAverageMark()
{
     // подсчет среднего балла за текущую сессию
     double mark = 0.0;
     for(int i = 0; i < countExams; i++)
          mark += exams[i];
     // корректировка среднего балла
     // с учетом текущей сессии
     averageMark = averageMark > 0 ?
                    (averageMark + mark/countExams)/2 :
                     mark / countExams;
}

// метод распечатки информации о студенте
void Student::Print()
{
     cout << "Фамилия, Имя, Отчество: " << fio << endl;
     cout << "Дата Рождение: " << dayBirth << "."
          << monthBirth << "."<<yearBirth << endl;
     cout << "Номер зачетной книжки: "
```

```
          << numberOfTestBook << endl;
     cout << "Средний балл успеваемости: "
               << averageMark << endl;
     cout<< "Баллы за зачетную сессию:" << endl;
     for(int i = 0; i < countTests; i++)
          cout << i + 1 << ". " << tests[i] << endl;
     cout<< "Баллы за экзаменационную сессию:" << endl;
     for(int i = 0; i < countExams; i++)
          cout << i + 1 << ". " << exams[i] << endl;
}

// метод сдачи зачета
void Student::SetTest(int numTest, double ball)
{
     if (ball < 0 || ball > 100)
          throw 1;
     if(numTest < 0 || numTest > countTests)
          throw 3;
     tests[numTest] = ball;
}

// метод сдачи экзамена
void Student::SetExam(int numExam, double ball)
{
     if(!PassedTests())
          throw 1;
     if (ball < 0 || ball > 100)
          throw 2;
     if(numExam < 0 || numExam > countExams)
          throw 3;
     exams[numExam] = ball;
}

// метод проверки допуска студента
// к экзаменационной сессии
bool Student::PassedTests()
{
     for(int i = 0; i < countTests; i++)
          if (tests[i] < 51)
               return false;
     return true;
}

// метод проверки, сдал ли студент экзаменационную сессию
bool Student::PassedExams()
{
     for(int i = 0; i < countExams; i++)
          if (exams[i] < 51)
               return false;
```

```
     return true;
}

// метод допуска студента к занятиям в следующем семестре
bool Student::NextSession()
{
     // если студент не сдал зачеты
     // и экзамены, допуска нет
     if (!(PassedTests() && PassedExams()))
          return false;
     // корректировка среднего балла
     // с учетом прошедшей сессии
     SetAverageMark();
     // подготовка массивов набранных баллов
     // к следующему семестру
     for(int i = 0; i < countTests; i++)
          tests[i] = 0.0;
     for(int i = 0; i < countExams; i++)
          exams[i] = 0.0;
     return true;
}
```

// файл "ContractStudent.h"

```
# include "Student.h"

#ifndef _CONTRACTSTUDENT_H_
#define _CONTRACTSTUDENT_H_

// объявление класса "Студент-контрактник"
class ContractStudent: public Student
{
private:
     double payment; // оплата обучения за семестр
     double credit; // долг по оплате
public:
     // конструкторы класса ContractStudent
     ContractStudent(char*, int, int, int, int,
                                    double, double = 0.0);
     ContractStudent();
     // метод получения суммы оплаты обучения
     double GetPayment();
     // метод изменения суммы оплаты обучения
     void SetPayment(double);
     // метод получения суммы долга
     double GetCredit();
     // метод уменьшения суммы долга
     void DescCredit(double);
```

```
        // метод сдать зачет
        void SetTest(int, double);
        // метод проверки допуска к экзаменационной сессии
        bool PassedTests();
        // метод проверки, сдана ли сессия
        bool PassedExams();
        // метод сдать экзамен
        void SetExam(int, double);
        // метод допуска к обучению в новой сессии
        bool NextSession();
        // метод распечатки информации
        // о студенте-контрактнике
        void Print();
};
#endif
```

// файл "ContractStudent.cpp"

```
# include <iostream>
# include <cstring>

// подключения файла объявления класса
# include "ContractStudent.h"

using namespace std;

// конструктор класса ContractStudent
ContractStudent::ContractStudent(char* fio, int d, int m,
                int y, int ntb, double pay, double mark):
                     Student(fio, d, m, y, ntb, mark)
{
     if (pay < 0)
          throw 1;
     payment = pay;
     credit = payment;
}

// конструктор по умолчанию
ContractStudent::ContractStudent()
{}

// переопределенные методы базового класса
// метод распечатки информации о студенте-контрактнике
void ContractStudent::Print()
{
     Student::Print();
     cout << "Долг по оплате: " << credit << endl;
}
```

```
// метод сдать зачет
void ContractStudent::SetTest(int numTest, double ball)
{
     if (credit > 0)
          throw 1;
     Student::SetTest(numTest, ball);
}

// метод сдать экзамен
void ContractStudent::SetExam(int numExam, double ball)
{
     if(!PassedTests())
          throw 1;
     Student::SetExam(numExam, ball);
}

// метод проверки допуска к экзаменационной сессии
bool ContractStudent::PassedTests()
{
     if (credit > 0)
          return false;
     return Student::PassedTests();
}

// метод проверки, сдана ли сессия
bool ContractStudent::PassedExams()
{
     if (credit > 0)
          return false;
     return Student::PassedExams();
}

// метод допуска к обучению в новой сессии
bool ContractStudent::NextSession()
{
     if (!(PassedTests() && PassedExams()))
          return false;
     SetAverageMark();
     for(int i = 0; i < countTests; i++)
          tests[i] = 0.0;
     for(int i = 0; i < countExams; i++)
          exams[i] = 0.0;
     // определение задолженности на следующий семестр
     credit = payment;
     return true;
}

// метод получения суммы оплаты обучения
```

```
double ContractStudent::GetPayment()
{
     return payment;
}

// метод изменения суммы оплаты обучения
void ContractStudent::SetPayment(double newPay)
{
     if (newPay < 0)
          throw 1;
     payment = newPay;
}

// метод получения суммы долга
double ContractStudent::GetCredit()
{
     return credit;
}

// метод уменьшения суммы долга
void ContractStudent::DescCredit(double pay)
{
     if (pay < 0)
          throw 1;
     credit -= pay;
     if (credit < 0)
          credit = 0.0;
}
```

// файл "main.cpp"

```
# include <iostream>
# include <cstring>
# include "Student.h"
# include "ContractStudent.h"

using namespace std;

void main()
{
     // создание объекта класса ContractStudent
     ContractStudent petr("Петров Петр Петрович", 2, 2,
                          1990, 222, 50000.0);
     petr.Print();
     // погашение задолженности студента
     petr.DescCredit(50000.0);
     try
     {
          // фиксация сдачи зачетов и экзаменов
```

```
            petr.SetTest(0, 51.5);
            .    .    .
            petr.SetExam(0, 60.0);
            .    .    .
            petr.Print();
            // перевод к следующему семестру
            petr.NextSession();
      }
      catch (int e)
      {
            // обработка исключений,
            // связанных с невозможностью фиксации
            // сдачи зачетов и экзаменов
            if (e == 1)
                 cout << "Студент на допущен к сессии"
                                                    << endl;
            if (e == 2)
                 cout << "Некорректные баллы" << endl;
            if (e == 3)
                 cout << "Некорректный номер зачета
                           или экзамена" << endl;
      }
      petr.Print();
}
```

2.2.7. Виртуальные функции и абстрактные классы

Внесем изменения в класс «Учебная группа». Предыдущая версия класса `Group` содержала массив объектов класса `Student`. Допустим, что в группе могут учиться также и контрактные студенты. Для их хранения в класс `Group` можно добавить еще один массив объектов класса `ContractStudent`. В этом случае придется в методах класса `Group` обрабатывать оба массива по отдельности, хотя, возможно, одинаковым образом. Принцип наследования, который был применен при разработке двух классов-студентов, позволяет использовать более эффективные средства ООП для единообразной обработки студентов обоих типов. Этим средством является использование виртуальных функций, которые определяются в базовом классе и переопределяются в производном.

Механизм использования ***виртуальных функций*** основывается на возможности хранения в переменной, являющейся указателем или ссылкой на базовый класс, адреса объекта производного класса. По

умолчанию выбор вызываемой функции осуществляется в соответствии с типом указателя или ссылки. В случае вызова виртуальной функции через указатель или ссылку на базовый класс программа будет использовать метод, определенный для типа объекта, а не метод, определенный для типа указателя или ссылки.

Допустим, метод `Print()` класса `Student` является виртуальным. Он выводит информацию об одном студенте. Это осуществляется добавлением в начало объявления метода в базовом классе ключевого слова `virtual`:

```
virtual void Print();
```

В классе `ContractStudent` этот метод переопределяется, распечатывая также информацию о задолженностях студента по оплате обучения.

Продемонстрируем работу виртуального метода на примере:

```
Student ivan("Иванов Иван Иванович", 1, 1, 1990, 111);
ContractStudent petr("Петров петр Петрович",
                              2, 2, 1990, 222, 50000.0);
Student* pStud;

pStud = &ivan; // указателю на базовый класс присваивается
               // адрес объекта класса Student
pStud -> Print(); // вызов метода Print() класса Student
pStud = &petr;  // указателю на базовый класс
                // присваивается адрес объекта
                //класса ContractStudent
pStud -> Print(); // вызов метода Print()
                  // класса ContractStudent
```

В данном случае две одинаковые строки кода будут приводить к вызову двух разных методов базового и производного классов соответственно.

Конструкторы классов не могут являться виртуальными функциями, поскольку производным классом не наследуется конструктор базового класса.

Деструктор класса должен быть виртуальным, если при уничтожении объекта производного класса происходит освобождение ресурсов, которые использовались в нем. Виртуальность деструктора проявляется в случае создания объекта с помощью операции выделения

памяти new и присваивании адреса этого объекта указателю на базовый класс:

```
Student* pStud = new
              ContractStudent("Петров Петр Петрович",
                              2, 2, 1990, 222, 50000.0);
.    .    .
delete pStud;
```

При вызове операции delete для уничтожения объекта с помощью указателя на базовый класс должен вызваться деструктор производного класса.

Продемонстрируем использование механизма виртуальных функций на примере класса Group. Теперь все методы класса Student, переопределенные в классе ContractStudent, будут являться виртуальными. В классе Group будем хранить массив указателей на объекты класса Student, которые могут содержать адреса студентов как бюджетной формы обучения, так и контрактной. В этом случае при определении списка студентов в группе необходимо указывать, к какому классу нужно обращаться при создании объекта-студента. Для этого в методе заполнения FillStudent() запрашивается тип студента (переменная type) и далее создание объекта происходит в зависимости от значения этой переменной:

```
.    .    .
int type = 0;
while(true)
{
     cout << "Введите 1, если студент-бюджетник,
                         2 - если студент-контрактник";
     cin >> type;
     if(type == 1 || type == 2)
           break;
}
double pay = 0;
if(type == 2)
{
     while(true)
     {
          cout << "Введите размер платы
                              за обучение в семестре:";
          cin >> pay;
          if(pay > 0)
               break;
```

```
        }
    }
    cin.get();
    try
    {
        if(type == 1)
            students[count]= new Student(f, d, m, y, ntb);
            else
                students[count]=new
                    ContractStudent(f, d, m, y, ntb, pay);
    }
    catch(int e)
    {
        .    .    .
    }
```

Теперь обработка всех студентов группы может происходить одинаково. Например, можно добавить в класс `Group` следующие методы

```
// внесение оплаты студентом (по номеру зачетной книжки)
void DescPayment(int, double);
// сдача зачета одним студентом
// (по номеру зачетной книжки)
void SetTest(int, int, double);
// сдача экзамена одним студентом
// (по номеру зачетной книжки)
void SetExam(int, int, double);
// сдача зачета всеми студентами группы
void SetTest(int, double*);
// сдача экзамена всеми студентами группы
void SetExam(int, double*);
// распечатка информации о сдаче зачета студентами группы
void PrintTest(int);
// распечатка информации о сдаче экзамена
// студентами группы
void PrintExam(int);
// метод распечатки информации о должниках
void PrintCredits();
// метод перевода студентов на следующий год
// (с исключением несдавших)
void NextSession();
```

Все они обрабатывают список студентов, выполняя соответствующие методы в зависимости от того, к какому классу принадлежит студент, используя механизм виртуальных функций. Например, метод распечатки информации о студентах, не сдавших

сессию, и студентов-контрактников, не допущенных к сессии из-за задолженности по оплате, использует виртуальный метод `PassedExams()`:

```
// метод распечатки информации о должниках
void Group::PrintCredits()
{
     cout << "Задолжники " << endl;
     for(int i = 0; i < count; i++)
          if(!students[i] -> PassedExams())
               cout << students[i] -> GetFio() << endl;
}
```

Аналогичный прием используется и в других методах (выделенные строки представляют собой использование виртуальных методов).

```
// сдача зачета одним студентом
// (по номеру зачетной книжки)
void Group::SetTest(int ntb, int numTest, double ball)
{
     int i;
     Student* std;
     if(FindStudent(ntb, std, i))
          std->SetTest(numTest, ball);
}

// сдача экзамена одним студентом
// (по номеру зачетной книжки)
void Group::SetExam(int ntb, int numExam, double ball)
{
     int i;
     Student* std;
     if(FindStudent(ntb, std, i))
          std->SetExam(numExam, ball);
}

// сдача зачета всеми студентами группы
void Group::SetTest(int numTest, double* balls)
{
     for(int i = 0; i < count; i++)
          try
          {
               students[i]->SetTest(numTest, balls[i]);
          }
          catch(int e)
          {
               if (e == 1)
```

```
                    cout << "Студент не допущен" << endl;
               if (e == 2)
                    cout << "Некорректные баллы" << endl;
               if (e == 3)
                    cout << "Некорретный индекс студента"
                                                  << endl;
          }
}

// сдача экзамена всеми студентами группы
void Group::SetExam(int numExam, double* balls)
{
     for(int i = 0; i < count; i++)
          try
          {
               students[i]->SetExam(numExam, balls[i]);
          }
          catch(int e)
          {
               if (e == 1)
                    cout << "Студент не допущен" << endl;
               if (e == 2)
                    cout << "Некорректные баллы" << endl;
               if (e == 3)
                    cout << "Некорретный индекс студента"
                                                  << endl;
          }
}

// метод перевода студентов на следующий год
// (с исключением несдавших)
void Group::NextSession()
{
     for(int i = count - 1; i >= 0; i--)
          if(!students[i]->NextSession())
               DeleteStudent(students[i]);
}


// распечатка информации о сдаче зачета студентами группы
void Group::PrintTest(int numTest)
{
     cout << "Зачет №" << (numTest + 1) << endl;
     for(int i = 0; i < count; i++)
          cout << (i + 1) << "."
               << students[i]->GetFio() << "\t"
               << students[i]->GetTest(numTest) << endl;
}
```

```
// распечатка информации о сдаче экзамена
// студентами группы
void Group::PrintExam(int numExam)
{
     cout << "Экзамен №" << (numExam + 1) << endl;
     for(int i = 0; i < count; i++)
          cout << (i + 1) << "."
               << students[i]->GetFio() << "\t"
               << students[i] -> GetExam(numExam) << endl;
}
```

Отдельно рассмотрим метод внесение оплаты студентом. Поскольку эта операция относится только к студентам-контрактникам, при просмотре списка студентов нужно определять тип объекта. Это осуществляется с помощью оператора `typeid`, который возвращает величину, идентифицирующую точный тип объекта. Этот оператор определен в библиотеке ***RTTI*** (runtime type information – тип информации времени выполнения), которая содержит стандартные возможности программы по определению типа объекта во время ее выполнения.

```
// внесение оплаты студентом (по номеру зачетной книжки)
void Group::DescPayment(int ntb, double pay)
{
     int i;
     Student* std;
     if(FindStudent(ntb, std, i))
          // если студент-контрактник – вносится оплата
          if(typeid(ContractStudent) == typeid(*std))
               ((ContractStudent*)std) -> DescCredit(pay);
}
```

Если тип студента является `ContractStudent`, то осуществим явное преобразование указателя `std` к `ContractStudent*` для вызова метода `DescCredit()`, который есть только в классе `ContractStudent`.

Иногда в классе нельзя задать определение виртуальных функций, поскольку поведение объектов производных классов очень сильно отличается. Например, пусть создается приложение, которое формирует рисунок, состоящий из графических примитивов (прямоугольник, эллипс, надпись и т.д.). Все они обладают общими методами – их требуется рисовать, перемещать, поворачивать, изменять размеры и т.д. Но реализация этих методов для различных видов графических примитивов будет сильно отличаться. Для унифицированной обработки всего набора

графических объектов их классы наследуются от общего родителя, который содержит объявления всех виртуальных методов, определяющих операции с фигурами. Таким образом, создается класс «фигура вообще», который определяет некоторое абстрактное понятие с абстрактным поведением. Такие классы называют абстрактными. Поскольку поведение этого класса определить невозможно, виртуальные функции, определяющие операции с фигурами, только объявляются в классе, но не определяются. Такие методы называются ***чисто виртуальными методами***. При объявлении чисто виртуального метода следует сразу после его прототипа указать "=0":

```
virtual void f() = 0;
```

Таким образом, класс является ***абстрактным***, если в нем объявлен хотя бы один чисто виртуальный метод. Абстрактный класс не может использоваться для создания объектов, поскольку не полностью определена его реализация. Его можно применять только в качестве базового класса, определив в его потомках чисто виртуальные функции. Если в производном классе не будет определена хотя бы одна чисто виртуальная функция базового класса, то он также будет абстрактным. В программах можно создавать указатели и ссылки на абстрактный класс для хранения адресов объектов классов-потомков. Применительно к нашему примеру, рисунок можно хранить как список указателей на базовый класс «фигура вообще», которые могут хранить адреса объектов различных типов (прямоугольник, эллипс, надпись и т.д.).

2.2.8. Шаблоны классов (параметризация классов)

Напомним, что существует возможность определять описание функций для обобщенного типа данных. Это описание называется шаблоном функции. Аналогично можно описывать классы, использующие обобщенные типы данных. Соответственно, в этом случае мы говорим о ***шаблонах классов***.

Шаблон класса определяется следующим образом:

```
template <список_обобщенных_типов>
class имя_шаблона_класса
{
     .    .    .
};
```

список_обобщенных_типов := class T1 [,class T2, . . ., class TN], где Ti – произвольный идентификатор обобщенного типа.

Все методы шаблона класса являются шаблонами функций. Поэтому при определении каждого метода следует указать то, что он является шаблоном:

```
template <список_обобщенных_типов>
тип_возвращаемого_значения
имя_шаблона_класса<список_идентификаторов_обобщенных_типов>::
            имя_метода(список_формальных_параметров)
            {
            .    .    .
            }
```

Список формальных параметров метода может содержать параметры как конкретных типов данных (int, double, Student и т.д.), так и обобщенных. Это же относится и к типу возвращаемого значения метода.

Например, создадим шаблон класса «Квадратная матрица», в котором в качестве обобщенного типа данных выступает тип элемента матрицы.

```
template <class T> class QMatrix
{
private:
     T** a;    // массив элементов матрицы
     int n;    // размер матрицы
public:
     // конструкторы и деструктор
     QMatrix(int);
     QMatrix()
     QMatrix(const QMatrix<T>&);
     ~QMatrix();
     // метод получения размера матрицы
     int GetSize();
     // перегруженные операторы
     QMatrix<T> operator+(QMatrix<T>&);
     QMatrix<T> operator*(QMatrix<T>&);
     QMatrix<T>& operator=(QMatrix<T>&);
     T* operator[](int);
     QMatrix<T> operator!();
     // метод вычисления определителя матрицы
     T Determinant();
     // метод получения минора матрицы
```

```
        QMatrix<T> SubMatrix(int, int);
        // метод получения обратной матрицы
        QMatrix<T> InverseMatrix();
        // дружественные функции ввода и вывода матрицы
        template <class T> friend ostream& operator <<
                              (ostream&, QMatrix<T>&);
        template <class T> friend istream& operator >>
                              (istream&, QMatrix<T>&);
};
```

Приведем определение некоторых методов данного шаблона класса. Напомним, что при определении метода перед его именем указывается имя класса, которому он принадлежит. Также следует поступать и при определении метода шаблона класса, только в этом случае указывается параметризированное имя шаблона:

```
// конструктор из шаблона класса квадратной матрицы
template <class T> QMatrix<T>::QMatrix(int n1)
{
     n = n1;
     a = new T* [n];
     for(int i = 0; i < n; i++)
          a[i] = new T [n];
     for(int i = 0; i < n; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = 0;
}

// операция сложения из шаблона класса квадратной матрицы
template <class T>
QMatrix<T> QMatrix<T>::operator+(QMatrix<T>& ob)
{
     if(ob.n == n)
     {
          QMatrix<T> temp(n);
          for(int i = 0; i < n; i++)
               for(int j = 0; j < n; j++)
                    temp.a[i][j] = a[i][j] + ob.a[i][j];
          return temp;
     }
     else
          throw 1;
}


// операция умножения из шаблона класса квадратной матрицы
template <class T>
QMatrix<T> QMatrix<T>::operator*(QMatrix<T>& ob)
```

```
{
     if(n == ob.n)
     {
          QMatrix temp(ob.n);
          for(int i = 0; i < n; i++)
               for(int j = 0; j < ob.n; j++)
                    for(int k = 0; k < n; k++)
                         temp.a[i][j] = temp.a[i][j] +
                                   (a[i][k] * ob.a[k][j]);
          return temp;
     }
     else
        throw 1;
}
```

На основе этого шаблона можно создавать объекты сгенерированных классов `QMatrix` с указанием конкретных типов данных для элементов матрицы (`int`, `double`, `Complex` – комплексное число, `Fraction` – дробь и т.д.). При генерации класса с указанием пользовательских типов данных нужно убедиться, что данный пользовательский тип содержит переопределение основных функций и операций, которые используются в методах шаблона. Например, пусть разработан класс «Рациональная дробь» `Fraction` и создается матрица, элементами которой являются рациональные дроби:

```
QMatrix<Fraction> ob(4);
```

При таком создании объекта происходит генерация класса `QMatrix` с элементами типа `Fraction` (также будут сгенерированы все методы этого класса). Затем будет вызван сгенерированный конструктор с одним параметром. При выполнении данного конструктора (его определение было указано выше) происходит инициализация элементов матрицы нулями. Для инициализации дроби необходимо, чтобы в классе `Fraction` был определен конструктор с одним параметром целого типа, который преобразует целое число (в данном случае 0) в дробь.

Для корректной работы метода сложения двух матриц придется определить в классе `Fraction` операторы присваивания и сложения двух дробей.

Особое внимание следует обратить на использование дружественных функций в шаблонах классов. Часто с помощью дружественных функций переопределяются операции ввода/вывода для конкретного класса. При

определении таких функций для шаблона класса неясен тип объекта, который следует ввести или вывести. Этот тип зависит от типа данных, для которого формируется объект на основе шаблона класса. Поэтому дружественные функции ввода/вывода также должны быть шаблонными и это должно быть указано при их объявлении в шаблоне класса:

```
class QMatrix
{
.    .    .
     template <class T>
     friend istream& operator >>
                         (istream& is, QMatrix<T>& ob);
     template <class T>
     ostream& operator << (ostream& os, QMatrix<T>& ob);
     .    .    .
};

// шаблон операции ввода квадратной матрицы
template <class T>
istream& operator >> (istream& in, QMatrix<T>& ob)
{
     if(ob.n != 0)
     {
          for(int i = 0; i < ob.n; i++)
               for(int j = 0;j < ob.n; j++)
                    in >> ob[i][j];
     }
     return in;
}

// шаблон операции вывода квадратной матрицы
template <class T>
ostream& operator << (ostream& out, QMatrix<T>& ob)
{
     if(ob.n != 0)
     {
          out << "Матрица:" << endl;
          for(int i = 0; i < ob.n; i++)
          {
               for(int j = 0; j < ob.n; j++)
                    out << ob[i][j] << "\t";
               out << endl;
          }
     }
     else
          out << "Матрица пуста" << endl;
     return out;
}
```

Шаблоны функций и классов широко используются при разработке программ. В языке C++ существует библиотека ***STL*** (Standard Template Library), в которой реализованы мощные и гибкие средства на основе шаблонов функций и классов [см. 1, 10].

Глава 3. Примеры реализаций классов

В данной главе демонстрируется применение основных принципов объектно-ориентированного программирования на ряде примеров.

3.1. Реализация класса «Рациональное число»

Рациональное число (лат. *ratio* – отношение, деление, дробь) – число, представляемое обыкновенной дробью $\frac{m}{n}$, где *m, n* – целые числа.

Правильной называется дробь, у которой модуль числителя меньше модуля знаменателя. Правильные дроби представляют рациональные числа, принадлежащие интервалу (-1, 1). Дробь, не являющаяся правильной, называется неправильной. У такой дроби модуль числителя больше или равен модулю знаменателя.

Неправильную дробь можно представить в виде суммы целого числа и правильной дроби. Такая запись числа называется смешанной дробью.

В качестве примера разберем создание класса «Рациональное число», который должен реализовывать стандартные операции над числами: сложение, вычитание, умножение, деление и операции сравнения. В классе также необходимо предусмотреть средства приведения дроби к смешанному виду.

3.1.1. Переменные и методы класса «Рациональное число»

Из определения следует, что любое рациональное число в смешанном виде определяется четырьмя составляющими:

- знаком числа (число положительное или отрицательное);
- целой частью;
- числителем;
- знаменателем.

Все составляющие дроби являются целыми числами. Знак дроби тоже будем представлять как целое число (1 – положительная дробь, -1 – отрицательная дробь), поскольку это удобно при реализации арифметических операций:

```
class Fraction
{
private:
     int sign;           // знак дроби (+ или -)
     int wholeNumber;    // целая часть дроби
     int numerator;      // числитель дроби
     int denominator;    // знаменатель дроби
     .    .    .
};
```

При определении операций с дробями предполагаем, что объекты класса `Fraction` находятся в смешанном виде. Результатом операции над дробями может быть неправильная дробь, которую, согласно предположению, необходимо перевести в смешанный вид. Для этого необходимы методы «преобразования в смешанный вид», «сокращения дроби» и «выделения целой части». Данные методы будут применяться при выполнении арифметических операций над дробями или при создании дроби, гарантируя, что дробь после завершения операции будет находиться в смешанном виде. Таким образом, пользователю класса нет необходимости выполнять операции приведения дроби к смешанному виду, поскольку эта операция выполняется автоматически. Поэтому методы преобразования в смешанный вид, сокращения дроби и выделения целой части можно описать как закрытые элементы класса.

```
class Fraction
{
private:
     int sign;           // знак дроби (+ или -)
     int intPart;        // целая часть дроби
     int numerator;      // числитель дроби
     int denominator;    // знаменатель дроби
     //преобразование в смешанный вид
     void GetMixedView();
     void Cancellation();     //сокращение дроби
     void GetIntPart();       //выделение целой части дроби
     .    .    .
};
```

К доступным элементам класса `Fraction` относятся конструкторы, деструктор, методы, реализующие арифметические операции, методы

сравнения, метод преобразования в вещественное число. Для ввода/вывода дроби и для некоторых операций также необходимо разработать дружественные функции.

Таким образом, полное объявление класса может выглядеть так:

```
class Fraction
{
private:
     int sign;           // знак дроби (+ или -)
     int intPart;        // целая часть дроби
     int numerator;      // числитель дроби
     int denominator;    // знаменатель дроби
     //преобразование в смешанный вид
     void GetMixedView();
     void Cancellation();     //сокращение дроби
     void GetIntPart();       //выделение целой части дроби
public:
     Fraction(); // конструктор без параметров
     //конструктор с параметрами
     Fraction(int, int, int = 0, int = 1);
     //деструктор
     ~Fraction();
     // метод сложения двух дробей
     Fraction operator + (Fraction);
     // метод сложения дроби с целым числом
     Fraction operator + (int);
     // метод вычитания двух дробей
     Fraction operator - (Fraction);
     // метод вычитания из дроби целого числа
     Fraction operator - (int);
     // метод умножения двух дробей
     Fraction operator * (Fraction);
     // метод умножения дроби на целыое число
     Fraction operator * (int);
     // метод деления двух дробей
     Fraction operator / (Fraction);
     // метод деления дроби на целое число
     Fraction operator / (int);
     // метод умножения на (-1)
     Fraction operator - ();
     // дружественные функции
     // функция сложения целого числа и дроби
     friend Fraction operator + (int, Fraction);
     // функция вычитания дроби из целого числа
     friend Fraction operator - (int, Fraction);
     // функция умножения целого числа и дроби
     friend Fraction operator * (int, Fraction);
     // функция деления целого числа на дробь
```

```
        friend Fraction operator / (int, Fraction);
        // метод преобразования в тип double
        operator double();
        // методы сравнения двух дробей
        bool operator > (Fraction);
        bool operator < (Fraction);
        bool operator >= (Fraction);
        bool operator <= (Fraction);
        bool operator != (Fraction);
        bool operator == (Fraction);
        //функция ввода дроби
        friend istream& operator >> (istream&, Fraction&);
        //функция вывода дроби
        friend ostream& operator << (ostream&, Fraction&);
};
```

3.1.2. Конструкторы и деструктор класса «Рациональное число»

Для создания объекта определим конструктор с четырьмя параметрами, соответствующими четырем структурным элементам класса:

- значение числителя;
- значение знаменателя;
- значение целой части;
- знак числа.

Прототип конструктора имеет следующий вид:

```
//конструктор с параметрами
Fraction(int, int, int = 0, int = 1);
```

Если при создании объекта не указываются значения целой части и знака, то по умолчанию целая часть числа равна нулю и число является положительным. Это определяется заданием значений по умолчанию соответствующим параметрам конструктора в его объявлении (прототипе). Параметры, для которых указываются значения по умолчанию, должны располагаться в конце списка формальных параметров.

```
// конструктор класса "Рациональная дробь"
Fraction::Fraction(int n, int d, int i, int s)
{
```

```
        intPart = i;
        numerator = n;
        denominator = d;
        sign = s;
        GetMixedView();
}
```

При создании объекта конструктору могут быть переданы значения числителя и знаменателя, образующие неправильную или сократимую дробь. В этом случае в теле конструктора после инициализации свойств нужно преобразовать дробь в смешанный вид. Это можно сделать путем вызова метода преобразования `GetMixedView()`.

Также определим в классе конструктор без параметров, который может использоваться при создании дроби, равной нулю, а также в специальных случаях, например, при создании массива или матрицы дробей. В конструкторе без параметров структурным свойствам присваиваются конкретные значения:

```
// конструктор по умолчанию класса "Рациональная дробь"
Fraction::Fraction()
{
        intPart = 0;
        numerator = 0;
        denominator = 1;
        sign = 1;
}
```

Отдельно рассмотрим метод преобразования дроби в смешанную и несократимую. В случаях, если значения числителя и знаменателя задают неправильную или сократимую дробь, в методе происходит выделение целой части, а затем осуществляется ее сокращение.

```
// метод преобразования дроби в смешанный вид
void Fraction::GetMixedView()
{
        GetIntPart();  //выделение целой части числа
        Cancellation();     //сокращение дроби
}
```

Если числитель дроби больше знаменателя, то выделяется целая часть:

```
// метод выделения целой части рационального числа
void Fraction::GetIntPart()
{
     if(numerator >= denominator)
     {
          intPart += (numerator / denominator);
          numerator %= denominator;
     }
}
```

Сокращение дроби осуществляется путем деления числителя и знаменателя дроби на их наибольший общий делитель, который вычисляется с помощью алгоритма Евклида.

```
// метод сокращения рациональной дроби
void Fraction::Cancellation()
{
     if(numerator != 0)
     {
          int m = denominator,
          n = numerator,
          ost = m%n;
          // вычисление НОД(числитель, знаменатель)
          // алгоритмом Евклида
          while(ost != 0)
          {
               m = n; n = ost;
               ost = m % n;
          }
          int nod = n;
          if(nod != 1)
          {
               numerator /= nod; denominator /= nod;
          }
     }
}
```

Деструктор класса выводит сообщение о том, что уничтожен объект класса `Fraction`.

```
// деструктор дроби
Fraction::~Fraction()
{
     cout << "Дробь " << (*this)
                      << " уничтожена." << endl;
}
```

В функции `main()` приведены различные способы создания объектов класса `Fraction` с помощью конструкторов.

```
void main(void)
{
     Fraction d1(2, 3, 0, 1);   // создание дроби 2/3
     Fraction d2(4, 5, 2, -1); // создание дроби -2 4/5
     Fraction d3(4, 3, 1, 1);  // создание дроби 2 1/3
     Fraction d4(10, 6);       // создание дроби 1 2/3
     Fraction d5(3, 7);        // создание дроби 3/7
     Fraction d6(3, 8, 2);     // создание дроби 2 3/8
     Fraction d7;         // создание рационального числа 0
     .    .    .
}
```

3.1.3. Перегрузка операций для класса «Рациональное число»

Для стандартного использования арифметических операций и операций сравнения перегрузим соответствующие операторы.

Поскольку любая дробь является вещественным числом, переопределим оператор явного преобразования объекта класса `Fraction` к вещественному типу данных `double`:

```
// операция преобразования дроби в тип double
Fraction::operator double()
{
     double res = (double)sign *
          (intPart * denominator + numerator) /
                                        denominator;
     return res;
}
```

Данное преобразование удобно будет использовать при сравнении дробей.

Перегрузку операций сравнения двух дробей (больше, больше или равно, меньше, меньше или равно, равно, не равно) удобно осуществлять с помощью методов класса `Fraction`. Эти операторы должны возвращать значение типа `bool`, показывающее, истинно ли указанное сравнение. Операторы `"=="` и `"!="` осуществляют поэлементное сравнение двух дробей, которые представлены в смешанной форме. Остальные операторы сравнения используют преобразование дробей к

вещественному числу и сравнивают полученные значения. Объект, вызвавший метод, сравнивается с объектом, переданным в качестве параметра, при этом обращение к нему осуществляется через указатель `this`.

```
// операции сравнения двух дробей
bool Fraction::operator ==(Fraction a)
{
     if (sign != a.sign || intPart != a.intPart ||
          numerator * a.denominator !=
                         denominator * a.numerator)
               return false;
     return true;
}

bool Fraction::operator !=(Fraction a)
{
     if (sign == a.sign && intPart == a.intPart &&
          numerator * a.denominator ==
                         denominator * a.numerator)
               return false;
     return true;
}

bool Fraction::operator > (Fraction a)
{
     if (double(*this) <= double(a))
          return false;
     return true;
}

bool Fraction::operator < (Fraction a)
{
     if (double(*this) >= double(a))
          return false;
     return true;
}

bool Fraction::operator >= (Fraction a)
{
     if (double(*this) < double(a))
          return false;
     return true;
}

bool Fraction::operator <= (Fraction a)
{
```

```
    if (double(*this) > double(a))
        return false;
    return true;
}
```

Каждая арифметическая операция (+, -, *, /) перегружена тремя функциями-операторами для случаев, когда:

- операндами операции являются объекты класса `Fraction`;
- первый операнд – дробь, второй – целое число;
- первый операнд – целое число, второй – объект-дробь.

Для первых двух случаев создаются методы класса, для третьего случая – дружественная функция с двумя параметрами. Результатом выполнения этих операторов является новая дробь. Оператор сложения двух дробей после формирования результата осуществляет преобразование к смешанному виду.

```
// операция сложения двух дробей
Fraction Fraction::operator + (Fraction a)
{
    Fraction res;
    res.numerator = sign * (intPart * denominator +
          numerator) * a.denominator +
          a.sign *(a.intPart *  a.denominator +
          a.numerator) * denominator;
    res.denominator = denominator * a.denominator;
    if (res.numerator < 0)
    {
        res.numerator *= -1; res.sign = -1;
    }
    res.GetMixedView();
    return res;
}
```

В определении функции сложения дроби с целым числом осуществляется преобразование этого целого числа в дробь (создается новый объект класса `Fraction`, значение которого равно целому числу) и вызов оператора сложения двух дробей.

```
// операция сложения дроби и целого числа
Fraction Fraction::operator + (int a)
{
    Fraction res;
    Fraction b(0, 1, abs(a), a/abs(a));     // b = a
    res = (*this) + b;  //сложение двух дробей
    return res;
```

```
}

// определение дружественной функции сложения
// целого числа и дроби
Fraction operator + (int a, Fraction c)
{
     Fraction res;
     Fraction b(0, 1, abs(a), a/abs(a));     // b = a
     res = b + c;        //сложение двух дробей
     return res;
}
```

Аналогичным образом определяют и другие арифметические операции.

Оператор "<<" осуществляет вывод дроби в привычном математическом виде с учетом существования целой или дробной части. Этот же оператор может использоваться и для записи дроби в файл. Для последующего корректного прочтения данных из файла требуется разделять два числа символом, отличным от пробела (поскольку пробел является разделителем между целой и дробной частями числа). Разделителем в данном случае является символ табуляции (`'\t'`).

```
// операция печати дроби
ostream& operator << (ostream& out, Fraction& a)
{
     // знак числа печатается только
     // если число отрицательно
     if (a.sign < 0)
          out << "-";
     // если целая часть не равна 0, выводим ее
     if (a.intPart != 0)
          out << a.intPart << " ";
     // дробная часть печатается, если числитель не равен
     if (a.numerator != 0)
          out << a.numerator << "/" << a.denominator;
     // если и целая часть и дробная часть равны 0,
     // то число равно 0
     if (a.intPart == 0 && a.numerator == 0)
          out << "0";
     // если вывод осуществляется в файл,
     // используется символ '\t'
     if(typeid(ofstream) == typeid(out))
          out << "\t";
     else out << " ";
     return out;
}
```

В том же формате будет осуществляться и ввод данных из потока. Разделителем между двумя дробными числами, записанными в файле, будет являться символ табуляции (`'\t'`) или символ конца строки (`'\n'`). Данные из входного потока считываются в строку, а затем производится разделение на составные части числа – знак, целую часть, числитель и знаменатель.

```
// операция ввода рациональной дроби
istream& operator >>(istream& fin, Fraction& a)
{
     char buf[30];
     // считывается число в строку
     // если считывание происходит из файла,
     // строка вводится до разделителя
     if(typeid(ifstream) == typeid(fin))
          fin.getline(buf, 29, '\t');
     else
          fin.getline(buf, 29);
     // находим первое вхождение символа '/' в строку
     char* ps = strchr(buf, '/');
     // если символ не найден,
     // т.е. число - без дробной части
     if(ps == NULL)
     {
          // из строки выделяем целую часть
          sscanf(buf, "%d", &a.intPart);
          a.numerator = 0;
          a.denominator = 1;
          //знак числа определяется по знаку целой части
          if(a.intPart >= 0)
               a.sign = 1;
          else
          {
               a.sign = -1;
               a.intPart = -a.intPart;
          }
          return fin;
     }
     // если число без целой части
     if(strchr(buf, ' ')==NULL)
     {
          a.intPart = 0;
          // считываем из строки числитель и знаменатель
          sscanf(buf, "%d/%d",
                          &a.numerator, &a.denominator);
          //знак числа определяется по знаку числителя
```

```
            if(a.numerator > 0)
                a.sign = 1;
            else
            {
                a.sign = -1;
                a.numerator = -a.numerator;
            }
            a.GetMixedView();
            return fin;
        }
        // считывание всех составляющих дроби
        // и определение знака
        sscanf(buf, "%d %d/%d", &a.intPart,
                              &a.numerator, &a.denominator);
        if(a.intPart > 0)
            a.sign = 1;
        else
        {
            a.sign = -1;
            a.intPart = -a.intPart;
        }
        a.GetMixedView();
        return fin;
    }
```

Приведем пример применения объектов класса `Fraction` и операций работы с ними.

```
    void main(void)
    {
        Fraction r1(2, 3, 0, 1);
        cout << "r1 = " << r1;
        Fraction r2(5, 7, 0, 1);
        cout << "r2 = " << r2;
        cout << "-r2= " << (-r2);
        cout<< "r2 = "<<(double)r2;
        cout << endl;
        Fraction d;
        // вызов оператора "==" для двух дробей
        if (r1 == r2)
            cout << "r1 == r2" << endl;
        else
            cout << "r1 != r2" << endl;
        // вызов оператора ">" для двух дробей
        if (r1 > r2)
            cout << "r1 > r2" << endl;
```

```
        else
             cout << "r1 <= r2" << endl;
        // вызов оператора "+" для двух дробей
        d = r1 + r2;
        cout << "r1+r2=" << d;
        // вызов оператора "+" для дроби и числа
        d = r1 + (-11);
        cout << "r1+(-11)=" << d;
        // вызов оператора "*" для числа и дроби
        d = 5 + r1;
        cout << "5 + r1 = " << d;
        // вызов оператора преобразования дроби к типу double
        Fraction q(1,3);
        double f = q;
        .    .    .
}
```

3.1.4. Пример использования массива дробей (быстрая сортировка)

Тип `Fraction` может использоваться в качестве подставляемого вместо обобщенного типа данных при генерации функций и классов на основе шаблонов. Приведем пример использования шаблона функции быстрой сортировки для массива дробей. В сгенерированном методе с типом `Fraction` будут вызываться конструктор копирования и конструктор по умолчанию, операторы ">=", "<=" и "=". Эти функции должны быть определены в классе или предоставлены компилятором. В нашем случае будут корректно использоваться конструктор копирования и оператор "=", предоставленные компилятором.

```
template <class T> void QuickSort (T* m, int n)
{
     if(n <= 1)
          return;
     int i = 0, j = n - 1;
     // вызывается конструктор копирования
     // для создания объекта selected и конструктор
     // по умолчанию для создания объекта temp
     T selected = m[0], temp;
     while(i != j)
     {
          // вызывается оператор ">="
```

```
            while(m[j] >= selected && j > i)
                  j--;
            if(j > i)
            {
                  // вызывается оператор "="
                  m[i] = m[j];
                  // вызывается оператор "<="
                  while(m[i] <= selected && i < j)
                        i++;
                  // вызывается оператор "="
                  m[j] = m[i];
            }
      }
      // вызывается оператор "="
      m[i] = selected;
      QuickSort(m, i);
      QuickSort(m + i + 1, n - i - 1);
}

void main()
{
      // создание массива дробей
      Fraction a[10] = { Fraction(4,2,3,1),
                         Fraction(1,6,5,-1),
                         Fraction(0,2,7,1),
                         Fraction(10,2,3,-1),
                         Fraction(2,13,20,1),
                         Fraction(0,1,0,1),
                         Fraction(1,1,3,1),
                         Fraction(2,5,7,-1),
                         Fraction(4,8,2,-1),
                         Fraction(4,1,3,-1)};
      for(int i = 0; i < 10; i++)
            cout << a[i] << ' ';
      cout << endl;
      // генерация и вызов функции быстрой сортировки
      // для массива дробей
      QuickSort(a, 10);
      for(int i = 0; i < 10; i++)
            cout << a[i] << ' ';
}
```

Домашнее задание

1. Дополнить класс `Fraction` перегруженными арифметическими операциями, в которых одним из операндов является вещественное число.
2. Дополнить класс `Fraction` перегруженными операциями сравнения для дробей и вещественных чисел.
3. Дополнить класс `Fraction` перегруженным конструктором, осуществляющим преобразование вещественного числа к типу `Fraction`. Предполагается, что дробная часть вещественного числа содержит до `10` знаков после запятой.
4. Разработать класс «Комплексное число». Определить в нем конструкторы и деструктор, перегрузить арифметические операции, операции ввода-вывода и сравнения.
5. Разработать класс «Комплексное число в тригонометрической форме». Определить в нем конструкторы и деструктор, перегрузить арифметические операции, операции ввода-вывода и сравнения.
6. Разработать класс «Комплексное число», в котором данные хранятся в двух видах: алгебраической и тригонометрической формах. Определить в нем конструкторы и деструктор, перегрузить арифметические операции, операции ввода-вывода и сравнения, написать функции преобразования числа из одной формы в другую. Протестировать все возможности класса.
7. Разработать класс «Дата». Определить в нем конструкторы и деструктор, перегрузить операцию добавления к дате заданного количества дней, операцию вычитания двух дат, операции ввода-вывода и сравнения.
8. Разработать класс «Время». Определить в нем конструкторы и деструктор, перегрузить операцию добавления к времени заданного количества минут, операцию вычитания двух моментов времени, операции ввода-вывода и сравнения.
9. Разработать класс «Прямоугольник». Определить в нем конструкторы и деструктор, перегрузить операцию пересечения прямоугольников (операция `"*"`), операцию вычисления площади прямоугольника операции ввода-вывода и сравнения (по площади).
10. Разработать класс «Студент» со структурными свойствами: фамилия, имя, отчество, номер группы, оценки по трем предметам текущей сессии. Перегрузить для него операции ввода-вывода и сравнения (по среднему баллу). Применить данный класс для создания массива объектов класса «Студент», ввести данные в

массив из файла, содержащего информацию о студентах, отсортировать этот массив по убыванию среднего балла, результат сортировки записать в другой файл.

11. Разработать класс «Игрушка» со структурными свойствами: название игрушки, ее стоимость, возрастные границы детей, для которых предназначена игрушка. Перегрузить для него операции ввода-вывода и сравнения (по стоимости). Применить данный класс для создания массива объектов класса «Игрушка», ввести данные в массив из файла, содержащего информацию об игрушках, в новый файл вывести информацию о тех игрушках, которые предназначены для детей от `N` до `M` лет, отсортировав их по стоимости.

3.2. Реализация классов прямоугольной и квадратной матриц

Разработать классы «Прямоугольная матрица» и «Квадратная матрица», которые должны осуществлять стандартные операции матричного исчисления: сложение, вычитание, умножение, умножение на число, транспонирование. Класс «Квадратная матрица» должен также содержать методы вычисления определителя и получения обратной матрицы.

3.2.1. Определение класса «Прямоугольная матрица»

Приведем объявление класса «Прямоугольная матрица»:

```
class Matrix
{
protected:
     int m, n;      // количество строк и столбцов матрицы
     double** a;    // массив элементов матрицы
public:
     // конструкторы
     Matrix(int, int);
     Matrix();
     Matrix(const Matrix& ob);
     // деструктор
     ~Matrix();
     // операция сложения двух матриц
     Matrix operator+(Matrix&);
```

```
        // операция вычитания матриц
        Matrix operator-(Matrix&);
         // операция умножения двух матриц
        Matrix operator*(Matrix&);
        // операция умножения матрицы на число
        Matrix operator*(double);
        // операция присваивания
        Matrix& operator=(Matrix&);
        // операция получения строки матрицы с заданным номером
        double* operator[](int);
        // операция транспонирования матрицы
        Matrix operator!();
        // дружественная функция операции умножения
        // числа на матрицу
        friend Matrix operator*(double, Matrix&);
        // дружественная функция операции вывода матрицы
        friend ostream& operator<<(ostream&, const Matrix&);
        // дружественная функция операции ввода матрицы
        friend istream& operator>>(istream&, Matrix&);
};
```

Объект класса `Matrix` определяется размерами матрицы и двумерным массивом ее элементов. Поведенческие свойства класса определяются операциями матричного исчисления.

Так как квадратная матрица есть частный случай прямоугольной, структурные и поведенческие свойства класса «Квадратная матрица» (`QMatrix`) будут идентичны свойствам класса `Matrix`. Поэтому реализуем класс «Квадратная матрица» как наследник класса `Matrix`, добавив в него методы, специфичные для квадратной матрицы (вычисление определителя и получение обратной матрицы). Для обеспечения доступа к переменным базового класса из производного их объявление помещено в секцию `protected` класса `Matrix`.

Создание объекта класса `Matrix` требует выделения памяти для хранения ее элементов. Поэтому класс `Matrix` должен обязательно содержать конструктор копирования, оператор присваивания и деструктор. Кроме того, можно дополнительно определить конструктор по умолчанию и конструктор с параметрами, определяющими размеры матрицы.

```
// конструктор по умолчанию
Matrix::Matrix()
{
     n = 0;
```

```
     m = 0;
     a = NULL;
}


// конструктор с параметрами – выделяет память заданного
// размера и инициализирует элементы матрицы нулями
Matrix:: Matrix(int m1, int n1)
{
     n = n1;
     m = m1;
     a = new double* [m];
     for(int i = 0; i < m; i++)
          a[i] = new double [n];
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = 0;
}

// конструктор копирования
Matrix:: Matrix(const Matrix& ob)
{
     n = ob.n;
          m = ob.m;
     if(ob.a == NULL)
     {
          a = NULL;
          return;
     }
     a = new double*[m];
     for(int i = 0; i < m; i++)
          a[i] = new double [n];
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = ob.a[i][j];
}

// деструктор выполняет освобождение памяти,
// занимаемой двумерным массивом
Matrix::~ Matrix()
{
     if(a != NULL)
     {
          for(int i = 0; i < m; i++)
               delete [] a[i];
          delete [] a;
     }
}
```

```
// оператор присваивания
Matrix& Matrix::operator = (Matrix& ob)
{
     if(n != ob.n || m != ob.m)
     {
          // освобождение памяти в левом операнде
          for(int i = 0; i < m; i++)
               delete [] a[i];
          delete [] a;
          // выделение памяти в левом операнде
          n = ob.n;
          m = ob.m;
          a = new double* [m];
          for(int i=0; i<m; i++)
               a[i] = new double [n];
     }
     // копирование данных правого операнда в левый
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = ob.a[i][j];
     return *this;
}
```

Реализация остальных методов класса `Matrix` и дружественных ему функций не представляет труда. Часть методов была рассмотрена в разделе 2.2.5 при описании правил перегрузки операций, и здесь не приводится.

3.2.2. Обработка исключений в классах Matrix и QMatrix

При работе с матрицами исключительные ситуации могут возникнуть в случаях, когда:

- конструктор получает неположительные значения количества строк или столбцов;
- происходит обращение к элементу матрицы по некорректным индексам;
- при выполнении операции сложения размеры двух матриц не совпадают;
- при выполнении операции умножения количество столбцов первой матрицы не совпадает с количеством строк второй матрицы;

- при вычислении обратной матрицы исходная матрица оказывается вырожденной.

При генерации исключений стандартных типов (целое число или символьная строка) программисту приходится отслеживать типы исключений и создавать для каждого из них свой обработчик (см. раздел 1.2). Например, пусть исключения генерируются в классе `Matrix` при перемножении или сложении двух матриц в случае несоответствия размеров.

```
Matrix Matrix::operator * (Matrix& ob)
{
     if(n == ob.m)
     {
          // перемножение матриц
          .    .    .
     }
     else
          // количество столбцов первой матрицы должно
          // быть равно количеству строк второй матрицы
          throw "Матрицы таких размеров
                                   перемножать нельзя";
}

Matrix Matrix::operator + (Matrix& ob)
{
     if(n == ob.n && m == ob.m)
     {
          // сложение матриц
          .    .    .
     }
     else
          // размеры матриц должны совпадать
          throw 1;
}
```

Тогда часть кода, проверяюшая наличие этих ошибок, будет иметь вид:

```
.    .    .
try
{
     Matrix a(m,n), b(n,k);
     cin >> a >> b;
     Matrix c = a * b;
     cout << "a*b=" << c << endl;
     Matrix d = a + b;
```

```
        cout << "a+b=" << d << endl;
    }
    catch(char* s)
    {
        cout << s << endl;
    }
    catch(int i)
    {
        if(i == 1)
            cout << "Размеры матриц должны совпадать"
                                                    << endl;
    }
    .    .    .
```

Теперь продемонстрируем другой подход, основанный на иерархии классов исключений, который избавляет нас от необходимости обрабатывать каждый тип исключения в отдельности.

Иерархия классов исключений может быть представлена в следующем виде:

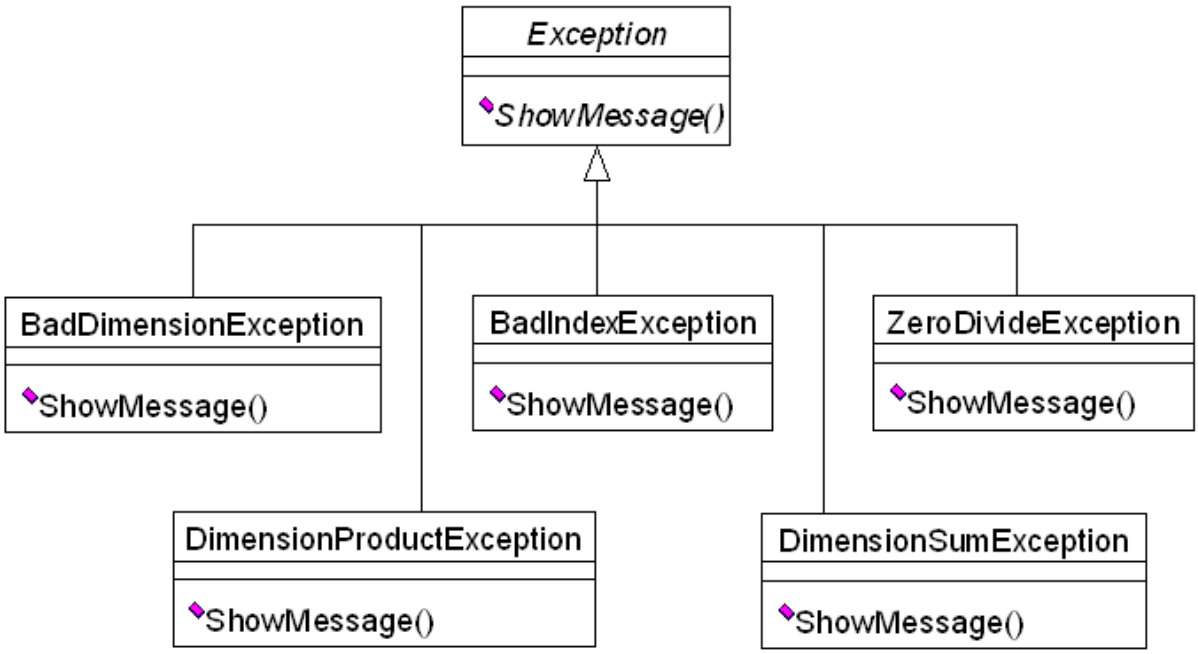


Рис.9. Иерархия исключений, возникающих при работе с матрицами.

При таком представлении классов-исключений достаточно создать обработчик только для исключения базового типа. Универсальная обработка исключений-наследников производится с помощью применения принципа полиморфизма и переопределения виртуальных функций.

Базовый класс иерархии – класс `Exception` – является абстрактным. В нем объявляется виртуальная функция, которая

предназначена для вывода сообщения об ошибке. В данной функции текст сообщения определен быть не может, поскольку заранее неизвестно, какая ошибка возникнет. Следовательно, она должна быть чисто виртуальной и переопределяться в классах-наследниках.

```
// базовый класс иерархии исключений
class Exception
{
     public:
     // метод вывода сообщения об ошибке
     virtual void ShowMessage() = 0;
};
```

Для каждого вида исключительных ситуаций иерархия содержит свой класс, производный от класса `Exception`, в котором переопределен виртуальный метод вывода сообщения об ошибке `ShowMessage()`.

```
// класс исключения, связанного с неправильным
// заданием размеров матрицы
class BadDimensionException : public Exception
{
public:
     void ShowMessage()
     {
          cout << "Размерности матрицы некорректны"
                                                  << endl;
     }
};

// класс исключения, связанного с обращением
// к несуществующему элементу матрицы
class BadIndexException : public Exception
{
public:
     void ShowMessage()
     {
          cout << "Элемента с такими индексами в матрице
                                    не существует" << endl ;
     }
};

// класс исключения несоответствия размеров,
// возникающего при суммировании матриц
class DimensionSumException : public Exception
{
public:
     void ShowMessage()
     {
```

```
            cout << "Размерности суммируемых матриц должны
                                              совпадать" << endl;
      }
};

// класс исключения несоответствия размеров,
// возникающего при умножении матриц
class DimensionProductException : public Exception
{
public:
      void ShowMessage()
      {
            cout << "Количество столбцов первой матрицы
                  должно совпадать с количеством строк
                  второй матрицы" << endl;
      }
};

// класс исключения, возникающего при вычислении обратной
// матрицы - деление на определитель, равный 0.
class ZeroDivideException : public Exception
{
public:
      void ShowMessage()
      {
            cout << "Деление на 0" << endl;
      }
};
```

Каждый класс исключений может содержать дополнительную информацию о возникшей ошибке. Например, в классе `DimensionSumException` могут храниться размеры суммируемых матриц. Далее их можно использовать для вывода сообщения, содержащего информацию об ошибке, возникающей при сложении двух матриц. В этом случае класс-исключение должен содержать конструктор с параметрами, инициализирующий эти элементы.

Классы-исключения могут содержать и другие методы, которые позволяют, если не устранить ошибку, то хотя бы предотвратить ее влияние на последующий ход выполнения программы (корректное освобождение ресурсов, занимаемых объектом, присвоение переменным объекта корректных значений и пр.).

Теперь, имея иерархию исключений, наследуемых от базового класса `Exception`, можно изменить тип генерируемых исключений:

```
// генерация исключения в операции умножения матриц
```

```
Matrix Matrix::operator * (Matrix& ob)
{
     if(n == ob.m)
     {
          // перемножение матриц
          .    .    .
     }
     else
          // количество столбцов первой матрицы должно
          // быть равно количеству строк второй матрицы
          throw DimensionProductException();
}


// генерация исключения в операции сложения матриц
Matrix Matrix::operator + (Matrix& ob)
{
     if(n == ob.n && m == ob.m)
     {
          // сложение матриц
          .    .    .
     }
else
     // размеры матриц должны совпадать
     throw DimensionSumException();
}
```

Как и ранее, можно обрабатывать каждый вид исключения своим обработчиком `catch`:

```
.    .    .
try
{
     Matrix a(m,n), b(n,k);
     cin >> a >> b;
     Matrix c = a * b;
     cout << "a*b=" << c << endl;
     Matrix d = a + b;
     cout << "a+b=" << d << endl;
}
catch(DimensionProductException e)
{
     e.ShowMessage();
}
catch(DimensionSumException e)
{
```

```
        e.ShowMessage();
    }
    .    .    .
```

Используя иерархию классов-исключений, приведенный выше фрагмент кода можно переписать так, что после контролируемого блока `try` следует только один обработчик, имеющий параметр – ссылку на объект базового класса:

```
    .    .    .
    try
    {
        Matrix a(m,n), b(n,k);
        cin >> a >> b;
        Matrix c = a * b;
        cout << "a*b=" << c << endl;
        Matrix d = a + b;
        cout << "a+b=" << d << endl;
    }
    catch(Exception& e)
    {
        e.ShowMessage();
    }
    .    .    .
```

В зависимости от возникающей ошибки согласно принципу полиморфизма ссылка на базовый класс будет указывать на объект производного класса, соответствующего типу ошибки. Тогда в обработчике `catch` будет вызываться метод `ShowMessage()` именно этого класса.

3.2.3. Использование принципа наследования на примере класса «Квадратная матрица»

Так как квадратная матрица есть частный случай прямоугольной, ее структурные и поведенческие свойства могут наследоваться от класса `Matrix`. Кроме этого, в класс `QMatrix` могут быть добавлены новые методы, специфичные для таких матриц (вычисление определителя, получение обратной матрицы).

Применение методов базового класса для выполнения арифметических операций с квадратными матрицами связано с преобразованиями типов. Например, базовый класс `Matrix` содержит

метод, перегружающий операцию "+" для сложения матриц. Этот метод получает в качестве параметра ссылку на объект класса `Matrix` и возвращает объект этого же типа:

```
Matrix Matrix::operator +(Matrix&);
```

Применение этого метода для сложения двух квадратных матриц связано с двумя преобразованиями. Во-первых, параметр типа `QMatrix` должен передаваться через ссылку на базовый класс. Такая передача параметров возможна, поскольку ссылка на базовый класс может указывать на объект производного класса. Проблема возникнет при возвращении результата. Результатом сложения квадратных матриц является квадратная матрица. Поэтому естественно желание присвоить результат работы метода объекту класса `QMatrix`, т.е. делается попытка осуществить преобразование из типа базового класса к производному типу. Для этих целей класс `QMatrix` должен содержать конструктор с параметром типа `Matrix` и оператор присваивания с тем же параметром.

Таким образом, объявление класса «Квадратная матрица» будет следующим:

```
class QMatrix : public Matrix
{
public:
     // конструктор создания квадратный матрицы
     // по ее размеру
     QMatrix(int);
     // конструктор по умолчанию
     QMatrix();
     // конструктор для преобразования
     // к квадратной матрице
     QMatrix(Matrix&);
     // оператор присваивания
     QMatrix& operator=(Matrix&);
     // метод вычисления определителя матрицы
     double Determinant();
     // метод получения подматрицы для нахождения минора -
     // требуется для вычисления определителя
     // и обратной матрицы
     QMatrix SubMatrix(int, int);
     // оператор получения обратной матрицы
     QMatrix operator~();
};
```

Заметим, что операции ввода/вывода для производного класса переопределять не требуется. Дружественные функции, которые перегружают эти операции в базовом классе `Matrix`, получают объект для ввода/вывода по ссылке, которая может указывать и на объект производного класса. Поэтому эти функции будут работать и при вводе/выводе объектов класса `QMatrix`.

Для создания квадратной матрицы заданного размера достаточно вызвать конструктор базового класса с параметрами:

```
QMatrix(int n): Matrix(n,n)
{}
```

Конструктор базового класса всегда вызывается при создании объекта производного класса. Если вызов конструктора класса `Matrix` явно не указан, произойдет вызов конструктора без параметров, который определяет пустой объект (размеры равны `0`, указатель на матрицу равен `NULL`). В этом случае выделение памяти для хранения квадратной матрицы и инициализацию ее размеров необходимо произвести в конструкторе производного класса:

```
QMatrix:: QMatrix(int n1)
{
     n = n1;
     m = n1;
     a = new double* [m];
     for(int i = 0; i < m; i++)
          a[i] = new double [n];
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = 0.0;
}
```

Для преобразования объекта базового класса к производному используются следующие конструктор и оператор присваивания:

```
QMatrix(Matrix& ob) : Matrix(ob)
{ }

QMatrix& QMatrix::operator = (Matrix& ob)
{
     Matrix::operator = (ob);
     return *this;
}
```

Конструктор использует вызов конструктора копирования класса `Matrix` для создания базовой компоненты производного класса. Аналогично, при выполнении присваивания вызывается оператор равно базового класса.

Разберем последовательность вызовов конструкторов и операторов присваивания на примере:

```
.    .    .
QMatrix a(3), b(3);     /*  для  создания  каждого  объекта
                        конструктор  производного  класса
                        вызывает конструктор базового класса. */

cin >> a >> b;          // Ввод матриц.

QMatrix c = a + b;      /* В функции сложения результат операции
                        помещается  в  локальную  матрицу,  для
                        создания которой вызывается конструктор
                        с параметрами класса Matrix. Этот объект
                        является возвращаемым значением функции.
                        При  осуществлении  возврата  вызывается
                        конструктор копирования класса Matrix.
                        Далее локальный объект уничтожается с
                        помощью  деструктора.  Для  присвоения
                        результата новому объекту класса QMatrix
                        его необходимо преобразовать в этот тип
                        путем  использования  конструктора
                        преобразования класса QMatrix. */

cout << c;              // Вывод результата сложения.

c = a * b;              /*  Отличие  от  рассмотренной  операции
                        сложения  состоит  в  том,  что  результат
                        операции  умножения  присваивается  уже
                        существующему  объекту  класса  QMatrix.
                        Поэтому  при  сохранении  результата
                        вызывается  оператор  присваивания  класса
                        QMatrix,  который,  в  свою  очередь,
                        вызывает  оператор  присваивания  базового
                        класса. */

cout << c;              // Вывод результата умножения.
.    .    .
```

Далее определим методы класса `QMatrix`. Для вычисления определителя матрицы и обратной матрицы требуется вычислить ее

миноры. Поэтому в класс `QMatrix` добавим метод формирования подматрицы, полученной из исходной вычеркиванием заданной строки и заданного столбца. Как известно, определитель такой подматрицы является одним из миноров исходной матрицы.

```
// метод получения подматрицы
QMatrix QMatrix::SubMatrix(int i1, int j1)
{
     // матрица-результат имеет
     // порядок на 1 меньше исходной
     QMatrix temp(n - 1);
     // формируем новую матрицу, игнорируя
     // строку с номером i1 и столбец с номером j1
     for(int i = 0; i < i1; i++)
     {
          for(int j = 0;j < j1; j++)
               temp.a[i][j] = a[i][j];
          for(int j = j1 + 1; j < n; j++)
               temp.a[i][j - 1] = a[i][j];
     }
     for(int i = i1 + 1; i < n; i++)
     {
          for(int j = 0; j < j1; j++)
               temp.a[i - 1][j] = a[i][j];
          for(int j = j1 + 1; j < n; j++)
               temp.a[i - 1][j - 1] = a[i][j];
     }
     return temp;
}
```

Согласно теореме Лапласа определитель матрицы *Q* (*det(Q)*) равен сумме произведений элементов строки (столбца) на их алгебраические дополнения:

$$\det(Q) = q_{i1}A_{i1} + q_{i2}A_{i2} + \ldots + q_{in}A_{in},$$

где $A_{ij} = (-1)^{i+j}M_{ij}$, M_{ij} – дополнительный минор элемента q_{ij} (определитель матрицы, полученной вычеркиванием из исходной строки с номером `i` и столбца с номером `j`).

Согласно этой формуле для вычисления определителя `n`-ого порядка требуется вычислить `n` определителей `(n-1)`-ого порядка. Поэтому метод вычисления определителя матрицы будет рекурсивным.

```
// метод вычисления определителя
double QMatrix::Determinant()
{
     double det = 0;
     // определитель 1-ого порядка совпадает
     // с единственным элементом матрицы
     if(n == 1)
          return a[0][0];
     QMatrix temp(n - 1);
     // раскладываем определитель по 0-ой строке
     for(int j = 0; j < n; j++)
     {
          // получаем матрицу для вычисления
          // минора элемента a0j
          temp = SubMatrix(0, j);
          // добавляем очередное произведение элемента
          // на его алгебраическое дополнение
          if(j % 2 == 0)
               det += temp.Determinant() * a[0][j];
          else
               det -= temp.Determinant() * a[0][j];
     }
     return det;
}
```

Для вычисления обратной матрицы для матрицы `Q` требуется найти матрицу алгебраических дополнений (A_{ij}) элементов, транспонировать ее и разделить на значение *det(Q)*. Если матрица является вырожденной (ее определитель равен нулю), обратной матрицы не существует. В этом случае генерируется исключение `ZeroDevideException`.

```
// оператор получения обратной матрицы
QMatrix QMatrix::operator~()
{
     QMatrix res(n);
     // вычисление определителя матрицы
     double det = Determinant();
     // если матрица вырожденная,
     // обратной матрицы не существует
     if(det == 0)
          throw ZeroDevideException();
     // вычисление транспонированной матрицы
     // алгебраических дополнений
     QMatrix temp(n - 1);
     int z;
     for(int i = 0; i < n; i++)
     {
```

```
            z = i%2==0 ? 1 : -1;
            for(int j = 0; j < n; j++)
            {
                temp = SubMatrix(i, j);
                res[j][i] = z * temp.Determinant() / det;
                z = -z;
            }
        }
        return res;
    }
```

Домашнее задание

1. Разработать класс «Множество». Определить конструкторы и деструктор. Переопределить операции объединения, пересечения и разности двух множеств, операции ввода-вывода. Написать методы проверки включения одного множества в другое, проверки равенства двух множеств, проверки пустоты множества.
2. Разработать класс «Многочлен». Определить конструкторы и деструктор. Переопределить операции сложения, вычитания, умножения многочленов и операции ввода-вывода. Написать методы вычисления значения многочлена в точке, получения производной и первообразной многочлена, вычисления определенного интеграла многочлена.
3. Разработать класс «Многоугольник», который хранится в виде массива его вершин. Определить конструкторы и деструктор. Переопределить операции ввода-вывода и операции сравнения по площади. Написать методы вычисления площади многоугольника, определения, принадлежит ли точка многоугольнику, и является ли многоугольник выпуклым.
4. Разработать класс «Целое число в заданной системе счисления». Число должно храниться в виде массива целых чисел (разрядов числа). Определить конструкторы и деструктор. Переопределить операции ввода-вывода, операции сложения, вычитания, умножения, деления и взятия остатка от деления двух чисел и операции сравнения. Написать методы перевода числа из одной системы счисления в другую.
5. Разработать класс «Линейная функция в `n`-мерном пространстве» ($f(x) = \langle b, x \rangle + c$). Определить конструкторы и деструктор. Переопределить операции сложения и вычитания функций,

умножения функции на число и операции ввода-вывода. Написать методы вычисления значения функции в точке, получения градиента функции. Наследовать от этого класса класс «Квадратичная функция в n-мерном пространстве» ($f(x) = \langle Ax, x \rangle + \langle b, x \rangle + c$). Переопределить все указанные операции и методы для класса-наследника.

6. Разработать класс «Граф» в виде матрицы смежности. Определить конструкторы и деструктор. Переопределить операции ввода-вывода. Написать методы проверки связности графа, проверки полноты графа, проверки двудольности графа, получения дополнения графа, нахождения источника графа, нахождения стока графа. Наследовать от этого класса класс «Взвешенный граф». Написать методы получения кратчайшего пути между двумя вершинами алгоритмом Дейкстры, получения каркаса минимального веса алгоритмом Прима и Краскала.

3.3. Реализация класса динамического списка

Для разработки программ большое значение имеют структуры данных, используемые в приложениях. Выбор способа хранения данных имеет важное значение при решении задач, в которых размер данных постоянно меняется. В этом случае говорят об использовании динамических структур данных. Самой простой динамической структурой данных является односвязный (динамический) список.

Разберем пример создания класса «Динамический список» с возможностями добавления в него элементов в заданную позицию, удаления элементов из списка по заданной позиции и по значению. Использовать этот класс для построения классов стека и очереди, хранения разреженных матрицы и создания словарей в виде поисковых хэш-таблиц.

3.3.1. Класс «Динамический список»

Реализуем возможности односвязного списка, содержащего целые числа. Каждый его элемент содержит адрес, по которому расположен

следующий элемент списка. Для работы с таким списком определим два класса: класс отдельного элемента и класс списка в целом.

Класс элемента списка (`Element`) должен предоставлять возможность хранения информационного поля элемента, указателя на следующий элемент списка, а также иметь методы для получения и задания новых значений структурным свойствам класса. Таким образом, класс `Element` имеет следующее объявление:

```
class Element
{
     // информационное поле списка
     int info;
     // указатель на следующий элемент списка
     Element * next;
public:
     // конструктор элемента списка
     Element(int, Element* = NULL);
     // метод получения информационного поля
     // элемента списка
     int GetValue();
     // метод установки нового значения
     // информационного поля
     void SetValue(int);
     // метод получения указателя
     // на следующий элемент списка
     Element* GetPointer();
     // метод установки нового значения
     // адреса следующего элемента
     void SetPointer(Element*);
};
```

Этот класс выполняет вспомогательную роль при определении класса всего списка. Поэтому отдельно от класса-списка он используется крайне редко. В таких случаях существует возможность связать два класса в единое целое, поместив объявление класса `Element` внутрь класса-списка (`List`). Таким образом, будет использоваться вложенный класс. Объявление класса `List` приведем далее:

```
class List
{
     // вложенный класс элемента списка
     class Element
     {
          // информационное поле списка
          int info;
          // указатель на следующий элемент списка
```

```
            Element * next;
        public:
            // конструктор элемента списка
            Element(int, Element* =NULL);
            // метод получения информационного
            // поля элемента списка
            int GetValue();
            // метод установки нового значения
            // информационного поля
            void SetValue(int);
            // метод получения указателя
            // на следующий элемент списка
            Element* GetPointer();
            // метод установки нового значения
            // адреса следующего элемента
            void SetPointer(Element*);
        };

    protected:
        // заголовок списка
        Element* head;
    public:
        // конструктор создания пустого списка
        List();
        // конструктор копирования
        List(const List&);
        // деструктор
        ~List();
        // метод проверки пустоты списка
        bool IsEmpty();
        // метод добавления элемента в заданную позицию
        void AddList(int, int);
        // метод удаления элемента из списка по ключу
        void DeleteByKey(int);
        // метод удаления элемента,
        // находящегося в заданной позиции
        int DeleteByPosition(int);
        // дружественная функция печати списка
        friend ostream& operator<<(ostream&, List&);
    };
```

При таком объявлении классов обращение извне к вложенному классу осуществляется с использованием операции области видимости «имя_внешнего_класса::имя_внутреннего_класса». В нашем случае – List::Element. Это происходит и при определении методов внутреннего класса за его пределами:

```
// конструктор элемента списка
```

```
List::Element::Element(int x, Element* p)
{
     info = x;
     next = p;
}

// метод получения информационного поля элемента списка
int List::Element::GetValue()
{
     return info;
}

// метод установки нового значения информационного поля
void List::Element::SetValue(int x)
{
     info = x;
}

// метод получения указателя на следующий элемент списка
Element* List::Element::GetPointer()
{
     return next;
}

// метод установки нового значения
// адреса следующего элемента
void List::Element::SetPointer(Element* p)
{
     next = p;
}
```

Далее приведем определение всех методов класса `List` и дружественной функции печати списка:

```
// конструктор по умолчанию для создания пустого списка
List::List()
{
     head = NULL;
}

// конструктор копирования
List::List(const List& list1)
{
     head = NULL;
     Element* cur = list1.head;
     int n = 1;
     // просмотр всех элементов списка list1 и добавление
     // в новый список элементов с теми же
     // значениями информационного поля
```

```
        while(cur != NULL)
        {
            AddList(cur -> GetValue(), n++);
            cur = cur -> GetPointer();
        }
}

// деструктор – удаление всех элементов списка
List::~List()
{
        Element * cur, *help;
        cur = head;
        while(cur != NULL)
        {
            help = cur -> GetPointer();
            delete cur;
            cur = help;
        }
}

// метод проверки пустоты списка
bool List::IsEmpty()
{
        if(head == NULL)
            return true;
        return false;
}

// метод добавления элемента в заданную позицию
void List::AddList(int key, int pos)
{
        Element* current,*help;
        // добавление в начало списка (приводит к изменению
        // адреса первого элемента списка)
        if(pos == 1)
        {
            help = new Element(key, head);
            head = help;
            return;
        }
        // поиск элемента, после которого
        // требуется вставить новый
        current = head;
        int i = 1;
        while(i != pos - 1 && current != NULL)
        {
            i++;
            current = current -> GetPointer();
```

```
        }
        // генерация исключения для случая
        // некорректной позиции вставки
        if(current == NULL)
             throw IncorrectPositionException();
        // создание нового элемента и включение его в список
        help = new Element(key, current -> GetPointer());
        current -> SetPointer(help);
}

// метод удаления элементов списка по ключу
void List::DeleteByKey(int key)
{
        Element* current, *help;
        // удаление элементов с заданным ключом
        // из начала списка
        while(head != NULL && head -> GetValue() == key)
        {
             help = head;
             head = head -> GetPointer();
             delete help;
        }
        if(head != NULL)
        {
             // удаление элементов с заданным ключом из
             // середины и/или с конца списка
             current = head;
             while(current->GetPointer() != NULL)
             {
                  help = current->GetPointer();
                  if(help->GetValue() == key)
                  {
                       current -> SetPointer
                                      (help->GetPointer());
                       delete help;
                  }
                  else
                       current = current->GetPointer();
             }
        }
}

// метод удаления элемента списка с заданной позиции
int List::DeleteByPosition(int pos)
{
        Element* current, *help;
        // генерация исключения в случае попытки
        // удаления из пустого списка
        if(IsEmpty())
```

```
            throw ListIsEmptyException ();
      int key;
      // удаление из начала списка (приводит к изменению
      // адреса первого элемента списка)
      if(pos == 1)
      {
            help = head;
            head = head -> GetPointer();
            key = help -> GetValue();
            delete help;
            return key;
      }
      // поиск элемента, после которого
      // требуется удалить элемент
      current = head;
      int i = 1;
      while(i != pos - 1 && current->GetPointer() != NULL)
      {
            i++;
            current = current -> GetPointer();
      }
      // генерация исключения для случая
      // некорректной позиции удаления
      if(current->GetPointer() == NULL)
            throw IncorrectPositionException();
      // удаление элемента из списка
      help = current->GetPointer();
      current -> SetPointer(help->GetPointer());
      key = help->GetValue();
      delete help;
      return key;
}

// дружественная функция печати списка
ostream& operator << (ostream& out, List& list)
{
      if(list.IsEmpty())
            out << "Список пустой" << endl;
      else
      {
            List::Element* current = list.head;
            while(current != NULL)
            {
                  out << current -> GetValue() << " ";
                  current = current -> GetPointer();
            }
      }
      out << endl;
```

```
        return out;
}
```

Для отслеживания исключительных ситуаций при работе со списками была создана иерархия классов-исключений с базовым классом `ListException` и классами-наследниками:

- `IncorrectPositionException` (при попытке обращения к элементу по некорректному номеру позиции);
- `ListIsEmptyException` (при попытке извлечения элемента из пустого списка).

```
// базовый класс иерархии исключений при работе со списком
class ListException
{
public:
     virtual void ShowMessage() = 0;
};


// класс-исключение при попытке обращения
// в некорректную позицию
class IncorrectPositionException : public ListException
{
public:
     void ShowMessage()
     {
          cout << "Некорректная позиция элемента в
                                        списке" << endl;
     }
};



// класс-исключение при попытке извлечения
// элемента из пустого списка
class ListIsEmptyException : public ListException
{
public:
     void ShowMessage()
     {
          cout << "Список пуст" << endl;
     }
};
```

3.3.2. Наследование на основе списка классов стека и очереди

Стек и очередь – это такие динамические структуры данных, в которых добавление и извлечения элементов происходит по четко определенным правилам. Стек работает по правилу LIFO (Last In First Out), т.е. «последним вошел, первым вышел». Очередь работает по правилу FIFO (First In First Out), т.е. «первым вошел, первым вышел». Для реализации стеков и очередей можно использовать динамические списки.

Поскольку структура данных, методы добавления и извлечения элементов определены уже в классе `List`, классы стека (`Stack`) и очереди (`Queue`) удобно наследовать от класса `List`. Класс `List` обладает большими функциональными возможностями, чем это нужно для реализации стека и очереди. Например, из списка можно удалять элементы по ключу, добавлять и удалять элементы из любой позиции. Поэтому использование методов класса `List` в дочерних классах должно быть ограничено. Это можно осуществить, используя private-наследование. В этом случае все методы базового класса становятся `private`-методами для производных классов. Они могут быть вызваны только из методов классов-наследников. Для контролируемого вызова и корректного использования методов базового класса в классы `Queue` и `Stack` добавляем методы помещения и извлечения элемента. Таким образом, объявления производных классов могут быть следующими:

```
class Stack : private List
{
public:
     // метод добавления элемента в стек
     void PushElement(int);
     // метод извлечения элемента из стека
     int PopElement();
     // дружественная функция печати стека
     friend ostream& operator<<(ostream&, Stack&);
};
```

```
class Queue : private List
{
     // количество элементов в очереди
     int count;
public:
     // конструктор пустой очереди
     Queue();
     // метод добавления элемента в очередь
     void PushElement(int);
     // метод извлечения элемента из очереди
     int PopElement();
     // дружественная функция печати очереди
     friend ostream& operator<<(ostream&, Queue&);
};
```

Для добавления элемента в конец очереди необходимо знать количество элементов в ней. Поэтому в класс `Queue` добавляется новое структурное свойство `count`, которое следует проинициализировать в конструкторе класса `Queue`:

```
// конструктор пустой очереди
Queue:: Queue()
{
     count = 0;
}
```

Для добавления элемента в стек и очередь вызывается метод базового класса `AddList`. В стеке добавление производится в первую позицию, в очереди – в конец списка.

```
// метод добавления элемента в стек
void Stack::PushElement(int key)
{
     // вставка элемента в начало списка
     AddList(key, 1);
}

// метод добавления элемента в очередь
void Queue::PushElement(int key)
{
     // добавление элемента в конец списка – номер позиции
     // добавления определяется количеством
     // элементов в очереди
     AddList(key, ++count);
}
```

Для извлечения элемента из очереди и стека вызывается метод базового класса DeleteByPosition. Извлечение происходит из начала списка.

```
// метод извлечения элемента из стека
int Stack::PopElement()
{
     // удаление элемента, стоящего в начале списке
     return DeleteByPosition(1);
}

   // метод извлечения элемента из очереди
int Queue::PopElement()
{
     // уменьшаем количество элементов в очереди
     if(count != 0)
          count--;
     // извлекаем из списка первый элемент
     return DeleteByPosition(1);
}
```

При public-наследовании функцию печати стека и очереди переопределять не требуется, поскольку вступает в силу принцип подстановки – ссылка на базовый класс может указывать на объект производного класса (будет вызвана дружественная функция базового класса). Наследование private-способом данный принцип игнорирует – преобразование из производного класса в базовый возможно только в методах производного класса или в дружественных ему функциях. Поэтому требуется написать свои функции для печати стека и очереди, которые будут осуществлять преобразование объекта к типу базового класса и, таким образом, вызывать функцию печати списка.

```
// дружественная функция печати стека
ostream& operator << (ostream& out, Stack& ob)
{
     // преобразование к базовому классу и вызов
     // операции вывода для базового класса
     out << (List)ob;
     return out;
}

// дружественная функция печати очереди
ostream& operator << (ostream& out, Queue& ob)
{
```

```
        // преобразование к базовому классу и вызов
        // операции вывода для базового класса
        out << (List)ob;
        return out;
}
```

Стеки и очереди очень часто используются для решения различных задач в программировании. Приведем примеры решения нескольких задач, в которых будем использовать разработанные классы `Stack` и `Queue`.

Задача 1. Пусть дано целое положительное число. Требуется перевести его в заданную систему счисления.

Напомним, что при переводе числа в другую систему счисления производится последовательное получение остатков от деления числа на основание системы счисления, которые при записи в обратном порядке образуют требуемое представление числа. Поэтому вычисленные остатки от деления заносятся в стек, а затем извлекаются из него для формирования строкового представления записи числа. Если основание системы счисления больше 10 (в этом случае для записи числа используются буквенные обозначения), в строку заносится соответствующий символ (10 – ‘A’, 11 – ‘B’ и т.д.).

Решение данной задачи оформлено в виде функции, в которой используется объект класса `Stack` и его методы.

```
// определение функции перевода положительного целого
// числа из десятичной системы счисления в любую заданную
// с основанием от 2 до 16.
// number - положительное целое число
// в десятичной системе счисления
// base – основание системы счисления
// функция возвращает строковое представление записи числа
char* PreobrChislo(int number, int base)
{
     Stack s;
     int n = 0;
     // вычисленные остатки от деления числа
     // на основание системы счисления помещаются в стек
     while(number != 0)
     {
          s.PushElement(number % base);
          number = number / base;
          n++;
```

```
        }
        // выделение памяти для символьной строки
        // с представлением числа
        char* strnumber = new char [n + 1];
        int i = 0;
        // формирование строкового представления записи числа
        // путем извлечения значений из стека
        try
        {
             while(true)
             {
                  n = s.PopElement();
                  if(n < 10)
                       strnumber[i] = '0' + n;
                  else
                       strnumber[i] = 'A' + n - 10;
                  i++;
             }
        }
        catch(ListException& )
        {
             // когда стек становится пустым,
             // представление числа сформировано
             strnumber[i] = '\0';
        }
        return strnumber;
}
```

Задача 2. Дана символьная строка, содержащая правильно записанное арифметическое выражение. Написать функцию перевода выражения в постфиксную форму. Постфиксной формой выражения называется такая запись, в которой знак операции следует за операндами. Эта запись не содержит скобок. Например, выражение a * b в постфиксной форме имеет вид ab *, a * b+c преобразуется в ab * c +, a * (b + c) – в abc + *.

Для решения задачи можно использовать следующий алгоритм. Рассматриваются поочередно все символы строки. В стек записывается открывающая скобка, и выражение далее анализируется посимвольно слева направо.

1. Если встречается операнд (число или переменная), то он помещается в очередь.
2. Если встречается открывающая скобка, то она заносится в стек.
3. Если встречается закрывающая скобка, то из стека извлекаются находящиеся там знаки операций до ближайшей открывающей

скобки, которая также удаляется из стека. Все эти знаки в порядке их извлечения помещаются в очередь.

4. Если встречается знак операции, то из стека извлекаются знаки операций, приоритет которых больше или равен приоритету данной операции, и они помещаются в очередь, после чего рассматриваемый знак переносится в стек.
5. Когда выражение заканчивается, выполняются такие же действия, что и при встрече закрывающей скобки.

Решение данной задачи оформлено в виде функции, в которой используются объекты классов `Stack` и `Queue` и их методы.

```
// определение функции перевода арифметического выражения
// в постфиксную форму
// str – строка, содержащая исходное
// арифметическое выражение
// функция возвращает строку
// с постфиксной формой этого выражения
char* Postfix(char* str)
{
     Stack s;
     Queue q;
     int i = 0;
     char c;
     // строка анализируется посимвольно
     while(str[i] != '\0')
     {
       switch(str[i])
       {
          // если текущий символ – знак операции
          case '+': case '-': case '*': case '/':
            if(s.IsEmpty())
              // если стек пустой, помещаем
               // символ операции в стек
               s.PushElement((int)str[i]);
            else
            {
               bool f = true;
               c = (char)s.PopElement();
               if(str[i] == '+' || str[i] == '-')
               {
                 // если текущая операция '+'
                  // или '-', из стека в очередь
                  // перемещаются все знаки
                 // операций либо до достижения
                 // пустоты стека, либо до
                 // достижения символа '('
```

```
                while(c != '(')
                     {
                        q.PushElement((int)c);
                        if(!s.IsEmpty())
                         c = (char)s.PopElement();
                        else
                    {
                      f = false;
                      break;
                    }
                      }
              }
           else
           {
              // если текущая операция '*' или
              // '/',из стека в очередь
              // перемещаются все знаки операций
              // либо до достижения пустоты стека,
              // либо до достижения
              // символов '(', '+'  или '-'
              while(c == '*' || c == '/')
              {
                 q.PushElement((int)c);
                 if(!s.IsEmpty())
                   c = (char)s.PopElement();
                 else
                 {
                    f = false;
                    break;
                 }
              }
            }
            // последний извлеченный символ
            // помещается в стек, если выход из
           // предыдущих циклов осуществился не по
           // достижению конца стека
           if(f)
              s.PushElement((int)c);
           // в стек помещается текущая операция
           s.PushElement((int)str[i]);
        }
        break;
     // если текущий символ '(', он записывается в стек
    case '(':
         s.PushElement((int)str[i]);
         break;
     // если текущий символ ')', из стека извлекаются
     // все знаки операций до ближайшей '(',
```

```
        // которая также извлекается из стека
        case ')':
             c = (char)s.PopElement();
             while(c != '(')
             {
                  q.PushElement((int)c);
                  c = (char)s.PopElement();
             }
             break;
        default:
             // текущий символ – операнд.
             // Он помещается в очередь
             q.PushElement((int)str[i]);
        }
        i++;
    }
    // формирования строки-результата, извлекая сначала
    // все из очереди, а потом из стека
    char* res = new char [i + 1];
    i = 0;
    try
    {
        while(true)
        {
          res[i] = (char)q.PopElement();
          i++;
        }
    }
    catch(ListException& )
    {
        try
        {
           while(true)
          {
             res[i] = (char)s.PopElement();
             i++;
          }
        }
        catch(ListException& )
        {
          res[i] = '\0';
        }
    }
    return res;
}
```

3.3.3.Использование класса «Динамический список» для хранения разреженных матриц

Разреженной называется матрица, которая содержит большое количество нулевых элементов. Такие матрицы нередко возникают в задачах линейной алгебры, математической физики и оптимизации. Хранение этих матриц традиционным способом требует существенных затрат памяти. Особенно это неэффективно при больших размерах матрицы и большом количестве нулевых элементов. В этом случае матрицу можно хранить в виде списка, содержащего только ненулевые элементы.

Для представления разреженной матрицы можно использовать следующую систему классов:

- класс для хранения одного элемента матрицы, который содержит индексы этого элемента и его значение (`MatrixElement`);
- класс для хранения элемента матрицы в списке, содержащий объект класса `MatrixElement` и адрес следующего элемента списка (`ListElement`);
- класс представления всей матрицы в виде списка, содержащий размеры матрицы, количество ненулевых элементов и адрес первого элемента списка (`MatrixList`);
- классы исключений `BadIndexException`, `BadDimension-Exception`, `NonSquareMatrixException`.

Приведем объявления этих классов:

```
// предварительное объявление класса список
class MatrixList;

// класс одного элемента матрицы
class MatrixElement
{
private:
     int i, j;      // индексы элемента
     double value;  // значение элемента
     friend class MatrixList;
public:
     // конструктор элемента матрицы
     MatrixElement(int, int, double);
     // дружественные функции ввода/вывода
     // элемента матрицы
```

```
        friend ostream& operator <<
                                (ostream&, MatrixElement&);
        friend ostream& operator << (ostream&, MatrixList&);
};

// класс одного элемента списка
class ListElement
{
private:
        // элемент матрицы
        MatrixElement a;
        // следующий элемент списка
        ListElement* next;
        friend class MatrixList;
public:
        // конструктор одного элемента списка
        ListElement(int, int, double);
        // дружественные функции ввода/вывода элемента списка
        friend ostream& operator << (ostream&, MatrixList&);
        friend ostream& operator << (ostream&, ListElement&);
};

// класс разреженной матрицы
class MatrixList
{
private:
        // заголовок списка
        ListElement* head;
        // размеры матрицы
        int m,n;
        // количество ненулевых элементов матрицы
        int count;
public:
        // конструктор разреженной матрицы
        MatrixList(int, int);
        // конструктор копирования
        MatrixList(const MatrixList&);
        // деструктор матрицы
        ~MatrixList();
        // метод уничтожения списка элементов матрицы
        void DestroyList();
        // метод добавления нового элемента в список
        void AddElement(int, int, double);
        // метод удаления элемента из списка
        void DeleteElement(int, int);
        // оператор сложения двух матриц
        MatrixList operator + (MatrixList&);
        // оператор присваивания матрицы
        MatrixList operator = (MatrixList&);
```

```
     // дружественная функция вывода матрица
     friend ostream& operator << (ostream&, MatrixList&);
     // метод проверки, является ли
     // матрицы трехдиагональной
     bool IsTripleDiagonal();
     // метод получения элемента матрицы
     // с заданными индексами
     ListElement* ExistsElement(int, int);
};

// класс исключения обращения
// к несуществующему элементу матрицы
class BadIndexException
{
     int m1, n1;         // некорректные индексы
public:
     BadIndexException(int, int);
     friend ostream& operator <<
                    (ostream&, BadIndexException&);
};

// класс исключения некорректных размеров матриц при сложении
class BadDimensionException
{
     // несовпадающие размеры двух матриц
     int m1, n1, m2, n2;
public:
     BadDimensionException(int, int, int, int);
     friend ostream& operator <<
                    (ostream&, BadDimensionException&);
};


// класс исключения некорректных размеров матрицы
// для операций с квадратными матрицами
class NonSquareMatrixException
{
public:
     NonSquareMatrixException();
     friend ostream& operator <<
                    (ostream&, NonSquareMatrixException&);
};
```

Методы класса `MatrixList` часто обращаются к данным классов элемента списка и элемента матрицы. Поскольку эти данные описаны в `private`-части классов `MatrixElement` и `ListElement`, прямое

обращение к ним из других классов невозможно. Для решения этой проблемы имеется два подхода: включение специальных методов доступа к данным в классы `MatrixElement` и `ListElement` или объявление класса `MatrixList` дружественным этим классам. Будем использовать в нашей программе дружественные классы. Для этого в классы `MatrixElement` и `ListElement` добавляется строка:

```
friend class MatrixList;
```

Поскольку имя класса `MatrixList` мы используем до его объявления, требуется заранее указать о существовании такого класса в программе. Для этого делается предварительное объявление класса `MatrixList.`

Для класса элемента матрицы (`MatrixElement`) требуется определить только конструктор, инициализирующий индексы элемента матрицы и его значение:

```
MatrixElement::MatrixElement(int i1, int j1, double val)
{
     i = i1;
     j = j1;
     value = val;
}
```

Класс элемента списка (`ListElement`) тоже должен содержать конструктор, который вызывает конструктор внедренного объекта класса `MatrixElement:`

```
ListElement::ListElement(int i, int j, double v)
                                                  : a(i,j,v)
{
     next = NULL;
}
```

Далее приводятся определения конструкторов, деструктора, оператора присваивания и функции уничтожения разреженной матрицы. В этих методах матрица рассматривается как список элементов, поэтому их реализация аналогична реализации этих методов для работы с обычным списком.

```
// конструктор матрицы из нулевых элементов
// заданных размеров
```

```
MatrixList::MatrixList(int m1, int n1)
{
     m = m1;
     n = n1;
     head = NULL;
     count = 0;
}

// конструктор копирования матриц
MatrixList::MatrixList(const MatrixList& ob)
{
     // создание списка элементов матрицы путем
     // поэлементного копирования из списка
     // элементов заданной матрицы ob
     m = ob.m;
     n = ob.n;
     if (ob.head == NULL)
     {
          head  = NULL;
          return;
     }
     ListElement* current = ob.head->next;
     ListElement* current1;
     head = new ListElement(ob.head->a.i, ob.head->a.j,
                                          ob.head->a.value);
     head->next = NULL;
     count++;
     current1 = head;
     while(current != NULL)
     {
          current1->next = new ListElement(current->a.i,
                         current->a.j, current->a.value);
          current1 = current1->next;
          current = current->next;
          count++;
     }
}

// операция присваивания матриц
MatrixList MatrixList::operator = (MatrixList& ob)
{
     // уничтожение списка, которому будет
     // присвоено новое значение
     DestroyList();
     // создание списка элементов матрицы путем
     // поэлементного копирования из списка
     // элементов заданной матрицы ob
     m = ob.m;
```

```
        n = ob.n;
        count = ob.count;
        if (ob.head == NULL)
        {
             head =  NULL;
             return *this;
        }
        ListElement* current = ob.head->next;
        ListElement* current1;
        head = new ListElement(ob.head->a.i, ob.head->a.j,
                                        ob.head->a.value);
        head -> next = NULL;
        current1 = head;
        while(current != NULL)
        {
             current1 -> next = new ListElement(current->a.i,
                            current->a.j, current->a.value);
             current1 = current1->next;
             current = current->next;
        }
        return *this;
}

// метод уничтожения списка
void MatrixList::DestroyList()
{
        ListElement* current = head, * help;
        while(current != NULL)
        {
             help = current->next;
             delete current;
             current = help;
        }
}

// деструктор матрицы
MatrixList::~MatrixList()
{
        DestroyList();
}
```

Для эффективного выполнения ряда операций над разреженными матрицами удобно хранить ее элементы, упорядоченными лексикографическим образом по номерам строк и столбцов. Такое представление однозначно определяет место каждого элемента в списке, что позволяет упростить процедуру поиска элемента, находящегося в заданной позиции матрицы. Поиск элемента матрицы, расположенного в

i-ой строке и j-ом столбце заключается в том, что при просмотре списка пропускаются все элементы, расположенные в строках с меньшим номером, чем i, а затем – в заданной строке, но в столбцах с меньшим номером, чем j. Такая процедура поиска используется в методе добавления элемента в матрицу AddElement(int i, int j, double v) для определения места вставки элемента в список, в методе удаления элемента из списка DeleteElement(int i, int j) и в методе проверки существования элемента с заданными индексами ExistsElement(int i, int j).

```
// метод добавления нового элемента в список –
// используется как метод присвоения
// элементу матрицы конкретного значения
void MatrixList::AddElement(int i, int j, double v)
{
     ListElement *current, *help, *prev;
     bool insert = false;
     // проверка корректности позиции
     // вставляемого элемента
     if (i < m && j < n)
     {
          // если новое значение элемента нулевое,
          // удаляем элемент из списка
          if(v == 0)
          {
               DeleteElement(i, j);
               return;
          }
          // проверка, вставляется ли элемент
          // на первое место
          if(head == NULL || i < head->a.i ||
                    (i == head->a.i && j < head->a.j))
          {
               // вставка элемента в начало списка
               help = new ListElement(i, j, v);
               help->next = head;
               head = help;
               count++;
               return;
          }
          // поиск позиции вставляемого элемента
          prev = head;
          current = head -> next;
          // пропускаем элементы, находящиеся
          // в строках с меньшим номером
          while(current != NULL && i > current->a.i)
```

```
            {
                current = current->next;
                prev = prev->next;
            }
            if(current != NULL && i == current->a.i)
            {
                // пропускаем элементы, находящиеся в той же
                // строке,но в столбцах с меньшим номером
                while(current != NULL && j > current->a.j)
                {
                    current = current->next;
                    prev = prev->next;
                }
                // если элемент в заданной позиции уже
                // имеется,изменяем его значение
                if(current != NULL && j == current->a.j)
                {
                    current->a.value = v;
                    insert = true;
                }
            }
            // вставляем новый элемент в список
            if(!insert)
            {
                help = new ListElement(i, j, v);
                prev->next = help;
                help->next = current;
                count++;
            }
        }
        else
            // генерация исключения в случае
            // некорректной позиции вставляемого элемента
            throw BadIndexException(m, n);
}

// метод удаления элемента по индексам,
// т.е. присвоение элементу нулевого значения
void MatrixList::DeleteElement(int i, int j)
{
    ListElement* help, *prev, *current;
    // список пуст, следовательно, матрица нулевая
    if(head == NULL)
        return;
    // удаление первого элемента списка
    if(head->a.i == i && head->a.j == j)
    {
        help = head;
        head = head->next;
```

```
            delete help;
            return;
        }
        // поиск позиции в списке удаляемого элемента
        // и предшествующего ему
        prev = head;
        current = head->next;
        // пропускаем элементы, находящиеся
        // в строках с меньшим номером
        while(current != NULL && i > current->a.i)
        {
            current = current->next;
            prev = prev->next;
        }
        if(current != NULL && i == current->a.i)
        {
            // пропускаем элементы, находящиеся в той же
            // строке, но в столбцах с меньшим номером
            while(current != NULL && j > current->a.j)
            {
                current = current->next;
                prev = prev->next;
            }
            // если элемент с заданной позицией найден,
            // удаляем его
            if(current != NULL && j == current->a.j)
            {
                prev->next = current->next;
                delete current;
            }
        }
}

// метод получения элемента матрицы с заданными индексами
ListElement* MatrixList::ExistsElement(int i, int j)
{
        ListElement* exists = NULL;
        ListElement* current = head;
        // пропускаем элементы, находящиеся
        // до требуемого элемента
        while (current != NULL)
        {
            if (!(current->a.i < i ||
                (current->a.i == i && current->a.j < j)))
                break;
            current = current->next;
        }
        // если элемент найден, запоминаем в exists его
```

```
        // адрес, в противном случае exists
        // остается равным NULL
        if (current != NULL && current->a.i == i &&
                                    current->a.j == j)
            exists = current;
        return exists;
}
```

Сложение двух матриц заключается в создании нового списка на основании двух существующих. Если оба исходных списка содержат элементы с одинаковыми индексами, сумма их значений образует соответствующий элемент нового списка. Остальные элементы обоих списков просто дублируются в новом.

```
// операция сложения двух матриц
MatrixList MatrixList::operator+(MatrixList& ob)
{
        // матрицы должны иметь одинаковые размеры
        if(m != ob.m || n != ob.n)
            throw BadDimensionException(m, n,ob.m, ob.n);
        // создается матрица-результат
        // как копия первого слагаемого
        MatrixList temp(*this);
        ListElement* current = ob.head;
        ListElement* exists;
        // просмотр элементов второй матрицы
        while (current != NULL)
        {
            exists = temp.ExistsElement(current->a.i,
                                        current->a.j);
            if (exists != NULL)
                // если в матрице-результате элемент
                // с такими индексами уже имеется,
                // суммируем элементы
                exists->a.value += current->a.value;
            else
                // если в матрице-результате элемент с
                // такими индексами не существует,
                // добавляем новый элемент
                // в матрицу-результат
                temp.AddElement(current-> a.i, current->a.j,
                    current->a.value);
            current = current->next;
        }
        return temp;
}
```

Матрица является трехдиагональной, если ее ненулевые элементы расположены только на главной диагонали и на двух соседних, параллельных ей.

```
// метод проверки, является ли матрицы трехдиагональной
bool MatrixList::IsTripleDiagonal()
{
     // если матрица неквадратная, генерируется исключение
     if(m != n)
          throw NonSquareMatrixException();
     ListElement* exists1;
     // в каждой строке проверяется наличие ненулевых
     // элементов, расположенных до трех центральных
     // диагоналей и после них. Если ненулевой элемент
     // будет найден, матрица не является
     // трехдиагональной.
     for(int i = 0; i < m; i++)
     {
          for(int j = 0; j < i - 1; j++)
          {
               exists1 = ExistsElement(i, j);
               if(exists1 != NULL)
                    return false;
          }
          for(int j = i + 2; j < n; j++)
          {
               exists1 = ExistsElement(i, j);
               if(exists1 != NULL)
                    return false;
          }
     }
     return true;
}
```

Для вывода разреженной матрицы используются дружественные функции, перегружающие операции вывода для классов `MatrixElement, ListElement, MatrixList.`

```
// операции вывода объектов классов
ostream& operator << (ostream& out, MatrixElement& ob)
{
      out << ob.value;
      return out;
}

ostream& operator << (ostream& out, ListElement& ob)
```

```
{
      out << ob.a;
      return out;
}
```

Вывод элементов списка осуществляется в виде матрицы.

```
ostream& operator << (ostream& out, MatrixList& ob)
{
      int i = 0, j = 0;
      // цикл просмотра элементов списка
      ListElement* current = ob.head;
      while(current != NULL)
      {
            // вывод нулей в качестве элементов
            // предшествующих строк
            for( ; i < current->a.i; i++)
            {
                  for( ; j < ob.n; j++)
                        out << "0\t";
                  out << endl;
                  j = 0;
            }
      // вывод нулей в качестве элементов в той же строке,
      // но в прешествующих столбцах
      for( ;j < current->a.j; j++)
            out << "0\t";
            // вывод текущего элемента
            out << *current << "\t";
            // корректировка индексов для просмотра
            // следующих элементов
            j++;
            if(j == ob.n)
            {
                  i++;
                  j = 0;
                  out << endl;
            }
            current = current->next;
      }
      // вывод нулей в качестве последующих элементов
      // строки, в которой расположен
      // последний элемент списка
      if(j != 0)
      {
            for( ; j < ob.n; j++)
                  out << "0\t";
            out << endl;
            i++;
```

```
        }
        // вывод нулей в качестве элементов строк,
        // расположенных после той,
        // в которой находится последний элемент списка
        for( ;i < ob.m; i++)
        {
             for(j = 0; j < ob.n; j++)
                  out << "0\t";
             out << endl;
        }
        return out;
}
```

3.3.4. Использование класса «Динамический список» для хранения телефонной книги

Нередко возникают приложения, работающие с большим количеством структурированной информации, объем которой постоянно меняется. Для хранения такой информации обычно используют разные виды динамических структур данных. Также в таких приложениях основной функцией является поиск информации по заданным критериям. Поэтому выбор используемой динамической структуры данных должен быть обусловлен эффективностью выполнения поиска информации.

Простым примером подобных приложений является «Телефонная книга». В этом приложении должно храниться произвольное количество записей о контактах (имя абонента и его телефон). Основной функцией приложения является поиск телефона нужного абонента.

Для хранения информации о контактах телефонной книги будем использовать структуру данных, называемую хэш-таблицей.

В самом простом случае все данные, которые хранятся в хэш-таблице, разбиваются на определенное количество групп, для хранения каждой из которых используется отдельная область памяти. Определение номера группы, в которую должны быть помещены данные, происходит по значению некоторого ключевого поля с помощью специальной хэш-функции. Поскольку хэш-функция однозначно определяет номер группы, в которую попадает конкретная запись, при ее последующем поиске значительно сокращается количество просматриваемых данных (достаточно просмотреть только одну группу хэш-таблицы). Для эффективного поиска желательно подбирать хэш-функцию таким образом, чтобы обеспечить равномерное заполнение всех групп.

Для хранения телефонной книги будем использовать хэш-таблицу из 26 групп (по количеству букв латинского алфавита). Ключевым значением поиска будет являться имя абонента. Хэш-функция будет определять номер группы по первой букве этого имени.

Данная хэш-функция является простой, однако она не обеспечивает равномерного распределения информации по группам: некоторые буквы гораздо чаще встречаются в именах, чем другие.

При реализации хэш-таблицы хранение каждой группы будем осуществлять в виде односвязного списка, а сама хэш-таблица будет представлять собой массив этих списков. Таким образом, приложение «Телефонная книга» содержит четыре класса:

- класс информации об абоненте (`Info`);
- класс одного элемента списка (`Element`);
- класс списка для хранения группы хэш-таблицы (`List`);
- класс для хранения хэш-таблицы (`HashTable`).

Приведем объявления этих классов:

```
// предварительное объявление классов
class List;
class HashTable;

// класс информации об абоненте
class Info
{
 private:
      char fio[50];       // имя абонента
      char phone[20];   // номер телефона
 public:
      // конструктор с инициализацией данных об абоненте
      Info(char*, char*);
      // конструктор по умолчанию
      Info();
      // конструктор копирования
      Info(const Info&);
      // методы получения и установки имени абонента
      char* GetFIO();
      void SetFIO(char*);
      // методы получения и установки номера телефона
      char* GetPhone();
      void SetPhone(char*);
      // дружественные классы и функции
      friend class List;
      friend ostream& operator << (ostream&, Info&);
      friend ostream& operator << (ostream&, List&);
```

```
};

// класс элемента списка
class Element
{
private:
     // запись телефонной книги
     Info abonent;
     // указатель на следующую запись в списке
     Element* next;
public:
     // конструктор элемента списка
     Element(char*, char*);
     // дружественные классы и функции
     friend class List;
     friend ostream& operator << (ostream&, List&);
};

// класс группы хэш-таблицы в виде списка
class List
{
private:
     // указатели на первый и последний элементы списка
     Element* head, *tail;
public:
     // конструктор создания пустого списка
     List();
     // деструктор списка
     ~List();
     // метод добавления в конец новой записи об абоненте
     void PushAbonent(char*, char*);
     // метод удаления из списка по имени абонента
     void DeleteAbonent(char*);
     // метод поиска элемента списка по имени абонента
     bool FindAbonent(char*, Info&);
     // дружественные функции вывода
     friend ostream& operator << (ostream&, List&);
     friend ostream& operator << (ostream&, HashTable&);
};

// класс хэш-таблицы в виде массива списков
class HashTable
{
private:
     int n;         // количество групп в хэш-таблице
     List* segments;     // указатель на массив групп
public:
     // конструктор хэш-таблицы по заданному числу групп
```

```
        HashTable(int);
        // деструктор хэш-таблицы
        ~HashTable();
        // метод добавления новой записи об абоненте
        void PushAbonent(char*, char*);
        // метод удаления записи по имени абонента
        void DeleteAbonent(char*);
        // метод поиска записи по имени абонента
        bool FindAbonent(char*, Info&);
        // метод проверки наличия в хэш-таблице
        // заданного имени абонента
        bool HasAbonent (char*);
        // дружественная функция вывода
        friend ostream& operator<<(ostream&, HashTable&);
};
```

Определение методов классов `Info` и `Element` не представляет особой сложности. Класс `List` был подробно разобран ранее. Поэтому остановимся на рассмотрении методов класса `HashTable`.

```
// метод добавления новой записи об абоненте
void HashTable::PushAbonent(char* abonent, char* phone)
{
     // вычисление номера группы по имени абонента
     int num = abonent[0] - 'A';
     // добавление записи в группу с номером num
     segments[num].PushAbonent(abonent, phone);
}


// метод удаления из списка записи по имени абонента
void HashTable::DeleteAbonent(char* abonent)
{
     // вычисление номера группы по имени абонента
     int num = abonent[0] - 'A';
     // удаление записи из группы с номером num
     segments[num].DeleteAbonent(abonent);
}

// метод поиска записи по имени абонента
bool HashTable::FindAbonent(char* abonent, Info& i)
{
     // вычисление номера группы по имени абонента
     int num =abonent[0] - 'A';
     // возвращаем true, если абонент найден в группе num,
     // false – в противном случае
     return segments[num].FindAbonent(abonent, i);
}
```

```
// метод проверки наличия в хэш-таблице
// заданного имени абонента
bool HashTable::HasAbonent(char* abonent)
{
     // вычисление номера группы по имени абонента
     int num =abonent[0] - 'A';
     Info i;
     // возвращаем true, если абонент найден в группе num,
     // false – в противном случае
     return segments[num].FindAbonent(abonent, i);
}
```

Все методы, реализующие операции с записью хэш-таблицы, начинаются с вычисления хэш-функции, т.е. номера группы, который соответствует этой записи. Далее в этой группе производится требуемая операция.

Приведем функцию `main()` приложения, которое формирует и использует телефонную книгу. Приложение выводит меню с операциями, которые можно выполнять с телефонной книгой, и осуществляет выбранную операцию. Для окончания работы приложения в меню содержится команда `"Exit"`.

Будем полагать, что все имена абонентов начинаются с заглавных букв. В случае ввода имени абонента, начинающегося со строчной буквы, в программе осуществляется приведение первой буквы имени к верхнему регистру.

```
int Menu()
{
     int k = 0;
     while(k <= 0 || k > 6)
     {
          cout << "----------- Меню ------------" << endl;
          cout << "Добавить абонента - 1,
                    Удалить абонента - 2,
                    Найти абонента - 3,
                    Проверка существования абонента - 4,
                    Печать телефонной книжки - 5,
                    Выход - 6" << endl;
          cout << "Введите команду:";
          cin >> k;
     }
     return k;
}
```

```
void main(void)
{
     HashTable phoneBook(26);
     char phone[20];
     char str[50];
     Info i;
     while(true)
     {
          switch(Menu())
          {
          case 1:   // вставка новой записи об абоненте
               cout << "--- Добавление абонента ---"
                                                  << endl;
               while(true)
               {
                    cout << " Введите имя:";
                    cin >> str;
                    if(str[0] >= 'a' && str[0] <= 'z')
                         str[0] = 'A' + str[0] - 'a';
                    if(phoneBook.HasKey(str) == 0)
                         break;
               }
               cout << "Введите номер телефона:";
               cin >> phone;
               phoneBook.PushAbonent(str, phone);
               break;
          case 2:   // удаление записи по имени абонента
               cout << "-- Удаление абонента --" << endl;
               cout << "Введите имя:";
               cin >> str;
               if(str[0] >= 'a' && str[0] <= 'z')
                    str[0] = 'A' + str[0] - 'a';
               phoneBook.DeleteAbonent(str);
               break;
          case 3:   // поиск телефона заданного абонента
               cout << "-- поиск абонента ----" << endl;
               cout << "Введите имя:";
               cin >> str;
               if(str[0] >= 'a' && str[0] <= 'z')
                    str[0] = 'A' + str[0] - 'a';
               if(phoneBook.FindAbonent(str, i) == 0)
                    cout << "Абонента не существует"
                                                  << endl;
               else
                    cout << "Телефон - " <<
                                   i.GetPhone() << endl;
               break;
          case 4: // проверка существования абонента
                  // с заданным именем
```

```
                cout << "- Существование абонента -"
                                                    << endl;
                cout << "Введите имя:";
                cin >> str;
                if(str[0] >= 'a' && str[0] <= 'z')
                     str[0] = 'A' + str[0] - 'a';
                if(phoneBook.HasKey(str) == 0)
                     cout << "Абонента не существует" << endl;
                else
                     cout << "Абонент существует" << endl;
                break;
           case 5: // печать телефонной книги
                cout << "- Распечатка телефонной книги -"
                                                    << endl;
                cout << phoneBook;
                break;
           case 6: // выход из приложения
                return;
         }
    }
}
```

Домашнее задание

1. Разработать класс «Граф» в виде списка смежности. Определить конструкторы и деструктор. Переопределить операции ввода-вывода. Написать методы проверки связности графа, проверки полноты графа, проверки двудольности графа, получения дополнения графа, нахождения источника графа, нахождения стока графа. Наследовать от этого класса класс «Взвешенный граф». Написать методы получения кратчайшего пути между двумя вершинами с помощью алгоритма Дейкстры, получения каркаса минимального веса с помощью алгоритма Прима и Краскала.
2. Разработать класс «Товар» со структурными свойствами: название, фирма-производитель, цена, срок годности. Перегрузить для него операции ввода-вывода и сравнения (по названию). Применить данный класс для создания списка объектов класса «Товар», ввести данные в список из файла, предоставить возможность добавления нового товара, поиска товара по названию, по фирме-производителю, удаления товаров с истекшим сроком годности, записать измененный список в другой файл.

3. Разработать класс «Склад», в котором хранятся список товаров, имеющихся в наличии (объекты класса «Товар») с указанием их количества, и список заказов от клиентов на товары. Написать методы поступления товаров на склад, поступления нового заказа, выполнения заказов, удаления товаров с истекшим сроком годности. При поступлении товара на склад меняется количество товара в уже существующем элементе списка, либо создается новый элемент, если такого товара на складе не было. При выполнении заказа требуемое количество товара удаляется со склада (меняется количество товара, либо товар удаляется из списка) и заказ удаляется из списка заказов.
4. Изменить систему классов задания 3. Для каждого товара формировать список заказов на него. Организовать хранения этого списка в виде очереди.
5. Разработать класс «Бинарное дерево сортировки». Написать конструкторы и деструктор, методы добавления нового узла, удаления узла по ключевому значению, вычисления глубины дерева, объединения двух деревьев, вычисления количества узлов на заданном уровне, определения подобия двух деревьев.

3.4. Множество точек на плоскости

В задачах поиска экстремальных точек функции на некотором множестве множество зачастую задается в виде системы ограничений. Каждое ограничение представляет собой уравнение или неравенство, в котором левая часть записывается в виде некоторой функции, определенной в пространстве R_n ($n = 1,2,...$), а правая часть является числом.

Например, в пространстве R_2 множество точек, заданное системой ограничений:

$$\begin{cases} x^2 + y^2 \le 4; \\ y \ge x; \\ -2x \le y; \\ y \ge 0 \end{cases}$$

выглядит так:

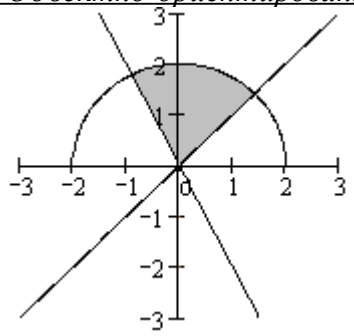


Рис. 10. Множество допустимых точек задачи.

Создадим систему классов для описания какого-либо множества в пространстве R_2.

3.4.1. Структура хранения системы ограничений

Согласно определению, множество допустимых точек задается системой ограничений. Поэтому для задания множества необходимо определить количество ограничений в системе и их набор.

Ограничение может быть представлено следующим образом:

функция в левой части	тип ограничения	правая часть
`f(x,y)`	(=, <, >, ≥, ≤, ≠)	`C (const)`

Для задания ограничения требуется знать функцию его левой части, константу, стоящую в правой части и тип неравенства/равенства.

Определим функции 1-ого и 2-ого порядка, которые будут использоваться в ограничениях:

- линейная – $f(x,y) = ax + by$;
- эллиптическая – $f(x,y) = \frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2}$;
- гиперболическая – $f(x,y) = \frac{(x - x_0)^2}{a^2} - \frac{(y - y_0)^2}{b^2}$;
- параболическая – $f(x,y) = (y - y_0)^2 - 2px$.

Линейная функция задается с помощью коэффициентов `a` и `b`. Для определения эллиптической и гиперболической функций требуется задать коэффициенты `a` и `b`, а также координаты точки (x_0, y_0), задающей смещение графика функции относительно начала координат.

Параболическая функция задается параметром p и смещением графика y_0 по оси OY.

Для определения функции в левой части ограничения можно объявить следующий класс Function:

```
// класс, задающий функцию в левой части ограничения
class Function
{
     int typeFunction; // тип кривой: 1 – линейная,
                       //              2 – эллиптическая,
                       //              3 – гиперболическая,
                       //              4 – параболическая
     // параметры задающие функции разных типов
     double a, b, p, x0, y0;
     .    .    .
};


// перечисление для определения типа ограничения
// le – <=, ge – >=, e – =, l – <, g – >, n – <>
typedef enum{le, ge, e, l, g, n} type_inequation;


// класс, определяющий ограничение
class Constraint
{
    Function  function;    // объект, описывающий функцию
                           // в левой части ограничения
    double b;              // правая часть
    type_inequation type;  // тип ограничения
     .    .    .
};


// класс, определяющий множество
class Set
{
     Constraint* constraints; // массив ограничений
     int n;         // количество ограничений в системе
     .    .    .
};
```

Хранить координаты точки в пространстве R_2 будем с помощью структуры Point:

```
struct Point
{
     double x;
```

```
     double y;
};
```

При решении задач необходимо вычислять значения функций, заданных с помощью объектов класса `Function`, в различных точках пространства. Для этого в классе должен быть определен соответствующий метод:

```
// функция в левой части ограничения
class Function
{
     int typeFunction; // тип кривой: 1 – линейная,
                       //              2 – эллиптическая,
                       //              3 – гиперболическая,
                       //              4 – параболическая
     // параметры задающие функции разных типов
     double a, b, p, x0, y0;
public:
     .    .    .
     double Calculate(Point);
     .    .    .
};


// метод вычисления функции
double Function::Calculate(Point pt)
{
     double value = 0.0;
     switch (typeFunction)
     {
          case 1:
               value = a * pt.x + b * pt.y;
          case 2:
               value = (pt.x - x0) * (pt.x - x0) / (a * a)
                    +(pt.y - y0) * (pt.y - y0) / (b * b);
          case 3:
               value = (pt.x - x0) * (pt.x - x0) / (a * a)
                     -(pt.y - y0) * (pt.y - y0) / (b * b);
          case 4:
               value = (pt.y - y0) * (pt.y - y0)
                        - 2 * p * pt.x;
     default:
          // неизвестен тип функции,
          // поэтому генерируется исключение
          throw FunctionException();
     }
     return value;
```

```
}
```

При такой реализации аналогичная структура программного кода (использование оператора `switch`) будет использоваться во всех методах класса `Function`. Недостатком этой структуры является необходимость внесения изменений во все методы класса, если добавляется новый тип функции или удаляется имеющийся. Если типов функций будет много, то методы становятся объемными и трудно читаемыми.

Еще одним недостатком является наличие неиспользуемых переменных класса `Function`. Например, для задания линейной функции достаточно использовать переменные `typeFunction`, `a` и `b`, а переменные `p`, `x0` и `y0` будут определены, но их значения игнорируются. Это говорит о неэффективном использовании памяти.

3.4.2. Иерархия классов кривых 1-ого и 2-ого порядка

Определить функции можно, используя другую структуру классов. которая не приводит к изменению уже написанного кода при добавлении нового типа функции и хранит для каждого типа функции столько параметров, сколько необходимо для ее задания. В этом случае используются принципы наследования и полиморфизма.

Для каждого типа функции задается собственный класс, например, `Line`, `Ellipse`, `Hyperbola`, `Parabola`. Все эти классы обладают одинаковым поведением – должно вычисляться значение функции, нужно вводить параметры функции и выводить представление функции на экран. Поэтому можно эти методы определить в отдельном классе `Function`, родительском для классов различных типов функций. Поскольку родительский класс «не знает», какая функция вычисляется, как ее распечатать, какие параметры ее определяют, все эти методы должны быть чисто виртуальными, а класс `Function` должен быть абстрактным.

В этом случае классы для реализации функций будут иметь следующие объявления:

```
 // класс, задающий функцию в левой части ограничения
class Function
{
public:
```

```
     // чисто виртуальный метод вычисления функции
     virtual double Calculate(Point) = 0;
     // чисто виртуальный метод ввода параметров функции
     virtual void Input() = 0;
     // чисто виртуальный метод вывода функции на печать
     virtual void Output() = 0;
};


// класс, определяющий линейную функцию
class Line : public Function
{
     double a, b;        // коэффициенты линейной функции
public:
     // конструктор линейной функции
     Line (Point = {0, 0});
     // переопределение виртуальной функции
     // вычисления функции
     double Calculate(Point);
     // переопределение виртуальной функции ввода
     // параметров функции
     void Input();
     // переопределение виртуальной функции
     // вывода на печать
     void Output();
};

// класс, задающий эллиптическую функцию
class Ellipse : public Function
{
     // параметры, задающие эллиптическую функцию
     double a, b;
     double x0, y0;
public:
     // конструктор эллиптической функции
     Ellipse(double = 1, double = 1,
                              double = 1, double = 1);
     // переопределение виртуальной функции
     // вычисления функции
     double Calculate(Point);
     // переопределение виртуальной функции ввода
     // параметров функции
     void Input();
     // переопределение виртуальной функции
     // вывода на печать
     void Output();
};
```

```
// класс, задающий гиперболическую функцию
class Hyperbola : public Function
{
     // параметры, задающие гиперболическую функцию
     double a, b;
     double x0, y0;
public:
     // конструктор гиперболической функции
     Hyperbola(double = 1, double = 1,
                              double = 1, double = 1);
     // переопределение виртуальной функции
     // вычисления функции
     double Calculate(Point);
     // переопределение виртуальной функции ввода
     // параметров функции
     void Input();
     // переопределение виртуальной функции
     // вывода на печать
     void Output();
};

// класс, задающий параболическую функцию
class Parabola: public Function
{
     // параметры, задающие параболическую функцию
     double p;
     double y0;
public:
     // конструктор параболической функции
     Parabola(double = 1, double = 1);
     // переопределение виртуальной функции
     // вычисления функции
     double Calculate(Point);
     // переопределение виртуальной функции ввода
     // параметров функции
     void Input();
     // переопределение виртуальной функции
     // вывода на печать
     void Output();
};
```

При такой структуре хранения функции требуется внести изменения в класс `Constraint`. Внедрение объекта абстрактного типа, определяющего функцию левой части, осуществить нельзя. Поэтому в класс ограничения включается указатель на базовый класс, который может хранить адрес объекта любого дочернего класса. С помощью такого указателя будут вызваны виртуальные методы вычисления, ввода и распечатки функций дочерних классов.

```
// класс, задающий ограничение
class Constraint
{
     Function * function;     // указатель на
                              // объект функции
                              // в левой части ограничения
     double b;                // правая часть
     type_inequation type;    // тип ограничения
     .    .    .
};
```

Теперь приведем объявление классов `Constraint` и `Set` с внесенными изменениями:

```
// класс, определяющий ограничение
class Constraint
{
     Function* function;      // указатель на
                              // объект функции
                              // в левой части ограничения
     double b;                // правая часть
     type_inequation type;    // тип ограничения
public:
     Constraint();            // конструктор класса
     bool IsExecute(Point);   // метод проверки
                              // выполнения ограничения
     bool IsOnBound(Point);   // метод проверки
                              // выполнения равенства
                              // f(x,y) = b для ограничений
                              // типа "<=", ">=", "="
     // дружественные функции ввода, вывода
     friend ostream& operator << (ostream&, Constraint&);
     friend istream& operator >> (istream&, Constraint&);
};
```

```
// класс, определяющий множество
class Set
{
     Constraint* constraints;
     int n;
public:
     //конструктор, задающий количество ограничений
     Set(int);
     ~Set();// деструктор
```

```
     // метод проверки принадлежит ли точка множеству
     bool Belongs(Point);
// метод проверки лежит ли точка на границе множества
     bool IsOnBound(Point);
     // дружественные функции ввода, вывода
     friend ostream& operator << (ostream&, Set&);
     friend istream& operator >> (istream&, Set&);
};
```

Классы, задающие кривые различных типов, имеют одинаковую структуру. Для примера приведем определение методов класса `Ellipse`:

```
// конструктор класса эллиптической функции
Ellipse::Ellipse(double a1, double b1,
                                    double x1, double y1)
{
     // если параметр a или b равен нулю, эллиптическую
     // функцию определить невозможно.
     // Поэтому генерируется исключение
     if(a1 == 0 || b1 == 0)
          throw FunctionException();
     a = a1;
     b = b1;
     x0 = x1;
     y0 = y1;
}


// функция вывода функции на печать
void Ellipse::Output()
{
     cout << "(x-" << x0 << ")^2/" << a * a << "+ (y - "
                              << y0 << ")^2/" << b * b;
}

// функция ввода параметров функции на печать
void Ellipse::Input()
{
     cin >> a >> b >> x0 >> y0;
}

// функция вычисления значения эллиптической функции
double Ellipse::Calculate(Point pt)
{
     return (pt.x - x0) * (pt.x - x0)/(a * a)+
                    (pt.y - y0)*(pt.y - y0) / (b * b);
}
```

В классе `Constraint` используется принцип полиморфизма при работе с объектами функций левой части ограничения. Класс содержит указатель на абстрактный класс `Function`, который может хранить адрес объекта класса `Line`, `Ellipse`, `Hyperbola` или `Parabola`, задающего конкретную функцию. При вводе ограничения у пользователя запрашивается вид нужной функции и создается объект соответствующего класса, адрес которого сохраняется в переменной-указателе `function`. При вызове методов `Input()`, `Output()` и `Calculate(Point)` через указатель `function` будут вызываться виртуальные функции того класса, адрес которого хранится в `function`. Такие вызовы выполняются как в функциях ввода/вывода ограничения, так и в методах проверки выполнения некоторых условий для точки. Таким образом, анализировать в этих методах тип функции левой части ограничения уже не потребуется.

Приведем далее определение методов класса `Constraint` и дружественных ему функций.

```
// функция, определяющая оператор ввода ограничения
istream& operator >> (istream& in, Constraint& ob)
{
     int choice;
     // ввод типа ограничения
     while(true)
     {
          cout << "Линейная - 1,
                    Эллиптическая - 2,
                   Гиперболическая - 3,
                   Параболическая - 4";
          in >> choice;
          if(choice >= 1 && choice <= 4)
               break;
     }
     // создание объекта функции левой части ограничения
     // в зависимости от введенного типа
     switch(choice)
     {
     case 1:
          ob.function = new Line(); break;
     case 2:
          ob.function = new Ellipse();  break;
     case 3:
          ob.function = new Hyperbola(); break;
```

```
        case 4:
             ob.function = new Parabola();  break;
        }
        // ввод параметров создаваемой функции
        ob.function->Input();
        // ввод вида ограничения
        while(true)
        {
             cout << "<= - 0, >= - 1,  = - 2, < - 3,
                                      > - 4, <> - 5" << endl;
             in >> choice;
             if(choice >= 0 && choice <= 5)
                  break;
        }
        ob.type = (type_inequation) choice;
        // ввод правой части
        in >> ob.b;
        return in;
}

// конструктор ограничения – запрашивает ввод
// ограничения с клавиатуры
Constraint::Constraint()
{
        cin >> *this;
}

// функция, переопределяющая оператор вывода ограничения
ostream& operator << (ostream& out, Constraint& ob)
{
        // вывод функции левой части ограничения
        ob.function->Output();
        // вывод знака вида ограничения
        switch(ob.type)
        {
        case le:
             out << "<=";
             break;
        case ge:
             out << ">=";
             break;
        case e:
             out << "=";
             break;
        case l:
             out << "<";
             break;
        case g:
             out << ">";
```

```
            break;
        case n:
            out << "<>";
            break;
        default:
            throw ConstraintException();
        }
        // вывод правой части ограничения
        out << ob.b << endl;
        return out;
}

// метод проверки выполнения ограничения для точки
bool Constraint::IsExecute(Point pt)
{
        // вычисление функции левой части ограничения
        double val = function->Calculate(pt);
        // сравнение с правой частью
        // согласно виду ограничения
        switch(type)
        {
        case le:
            if (val <= b)
                return true;
            break;
        case ge:
            if (val >= b)
                return true;
            break;
        case e:
            if (val == b)
                return true;
            break;
        case l:
            if (val < b)
                return true;
            break;
        case g:
            if (val > b)
                 return true;
            break;
        case n:
            if (val != b)
                return true;
            break;
        }
        return false;
}
```

```
// метод проверки выполнения равенства f(x,y) = b для
// ограничений видов "<=",">=","="
bool Constraint::IsOnBound(Point pt)
{
     // для ограничений видов "<",">", "<>"
     // равенство не должно выполняться
     if(type == l || type == g || type == n)
          return false;
     // вычисление значения функции в точке
     double val = function->Calculate(pt);
     // сравнение с правой частью на выполнение равенства
     if (val == b)
          return true;
     return false;
}
```

Далее определим методы класса `Set`. Отметим, что в этом классе отсутствует метод ввода информации о системе ограничений. Ввод выполняется в конструкторе класса `Set` при создании массива ограничений (в конструкторе каждого ограничения).

```
// конструктор класса «Множество точек на плоскости»
// с заданным количеством ограничений
Set::Set(int n1)
{
     n = n1;
     constraints = new Constraint [n];
}


// деструктор класса «Множество точек на плоскости»
Set::~Set()
{
     delete []  constraints;
}

// метод проверки, принадлежит ли точка множеству
bool Set::Belongs(Point pt)
{
     // точка не принадлежит множеству, если не
     // выполняется хотя бы одно из ограничений,
     // определяющих множество
     for(int i = 0; i < n; i++)
          if (!constraints[i].IsExecute(pt))
               return false;
     return true;
```

```
}

// метод проверки, лежит ли точка на границе множества
bool Set::IsOnBound(Point pt)
{
     // точка лежит на границе, если выполняются все
     // ограничения и хотя бы одно из них – как равенство
     if (Belongs(pt))
          for(int i = 0; i < n; i++)
               if (constraints[i].IsOnBound(pt))
                    return true;
     return false;
}

// функция вывода системы ограничений,
// определяющих множество
ostream& operator << (ostream& out, Set& set)
{
     for(int i = 0; i < set.n; i++)
          out << set.constraints[i];
     return out;
}

void main(void)
{
     Set set(3);
     cout << set;
     Point p1={0,1}, p2={1,1};
     if(set.Belongs(p1))
          cout << "Точка (0,1) принадлежит множеству"
                                                  << endl;
     else
          cout<< "Точка (0,1) не принадлежит множеству"
                                                  << endl;
     if(set.IsOnBound(p1))
          cout << "Точка (0,1) лежит на границе множества"
                                                  << endl;
     else
     cout << "Точка (0,1) не лежит на границе множества"
                                                  << endl;
     if(set.Belongs(p2))
          cout << "Точка (1,1) принадлежит множеству"
                                                  << endl;
     else
          cout<< "Точка (1,1) не принадлежит множеству"
                                                   << endl;
     if(set.IsOnBound(p2))
          cout << "Точка (1,1) лежит на границе множества"
```

```
                                                    << endl;
        else
             cout << "Точка (1,1) не лежит на
                       границе множества" << endl;
    }
```

Для множества, которое приведено на рис. 10 , данная функция `main()` выведет следующие сообщения:

```
"Точка (0,1) принадлежит множеству"
"Точка (0,1) не лежит на границе множества"
"Точка (1,1) принадлежит множеству"
"Точка (1,1) лежит на границе множества"
```

Домашнее задание

1. На основе класса «Товар» из домашнего задания 2 к разделу 3.3 разработать иерархию производных классов (продовольственные и непродовольственные товары, продовольственные товары делятся на молочные, хлебобулочные, мясные и пр., непродовольственные товары делятся на одежду, обувь и пр.). Иерархию можно продолжить. Реализовать виртуальные функции вывода подробной информации о товаре. Создать список товаров в магазине и написать функцию вывода каталога всех товаров магазина.
2. Создать иерархию классов «Вагоны пассажирского поезда» с разделением на купейные, плацкартные, СВ. Каждый класс вагона должен содержать информацию о количестве мест разных типов (нижнее, верхнее, нижнее боковое, верхнее боковое), о наличии дополнительных услуг. С помощью виртуальных функций получить полный доход от эксплуатации вагона. Создать класс «Пассажирский поезд», который хранит список вагонов. Подсчитать доход от одного рейса поезда.
3. Создать абстрактный класс «Функция в n-мерном пространстве». Наследовать от него класс «Линейная функция $f(x) = \langle b,x \rangle + c$» и класс «Квадратичная функция $f(x) = \langle Ax,x \rangle + \langle b,x \rangle + c$», где A – неотрицательно определенная матрица. Реализовать виртуальные методы вычисления значения функции и ее градиента в точке. Определить класс «Множество точек в n-мерном пространстве»,

которое определяется как список неравенств вида $f(x) \le b$, где $f(x)$ – линейные или квадратичные функции. Написать методы, определяющие, принадлежит ли точка множеству и лежит ли точка на границе множества.

3.5. Система линейных уравнений

Одна из базовых задач линейной алгебры заключается в решении систем линейных алгебраических уравнений (СЛАУ). В общем случае система m линейных уравнений с n неизвестными имеет следующий вид:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots a_{2n}x_n &= b_2, \\ \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot & \\ a_{m1}x_1 + a_{m2}x_2 + \ldots a_{mn}x_n &= b_m. \end{aligned}$$

где a_{ij} – коэффициенты системы, b_i – свободные члены, x_j – неизвестные, $i = 1..m, j = 1..n$. Решением системы называется такая совокупность n чисел, которая при подстановке в систему на место неизвестных обращает все уравнения системы в тождества.

Систему линейных уравнений удобно представлять в матричном виде:

$$AX = B,$$

где $A = \{a_{ij}\}$ – матрица коэффициентов системы размерности m x n, $X = \{x_j\}$ – вектор-столбец неизвестных размерности n, $B = \{b_i\}$ – вектор-столбец свободных членов размерности m.

Система уравнений называется совместной, если она имеет хотя бы одно решение, и несовместной, если не имеет решений. Совместная система называется определенной, если она имеет единственное решение, и неопределенной, если существует несколько различных решений.

Если матрица A является квадратной, то количество решений системы уравнений определяется по значению определителя матрицы A (*det(A)*). Если определитель *det(A)* не равен 0, то система имеет

единственное решение и получить его можно, к примеру, методом Крамера или по формуле $X = A^{-1}B$, где A^{-1} – обратная матрица к `A`. Если определитель `det(A)` равен `0`, то система может не иметь решения или иметь бесконечное число решений. Решить систему уравнений в этом случае можно, используя метод исключений Жордана-Гаусса. Когда система имеет бесконечное число решений, формируется общее решение СЛАУ, в котором значения одних переменных выражаются через значения других переменных.

Для решения системы уравнений с прямоугольной матрицей A можно применять метод исключений Жордана-Гаусса поиска общего решения системы.

Определим класс `Slau`, описывающий систему линейных уравнений и основные методы ее решения.

Структурными свойствами класса системы уравнений `Slau` являются:

- количество уравнений,
- количество неизвестных,
- матрица коэффициентов,
- вектор свободных членов,
- признак совместности системы,
- ранг матрицы коэффициентов,
- вектор решений.

Значения последних трех свойств определяются в процессе решения СЛАУ.

Поскольку вектор является частным случаем матрицы (вектор – матрица, состоящая из одного столбца), будем использовать для хранения как самой матрицы, так и вектора класс `Matrix`, описанный в разделе 3.2.

```
class Slau
{
private:
     int m;          // количество уравнений
     int n;          // количество переменных
     Matrix a;       // матрица коэф.
     Matrix b;       // вектор правой части
     Matrix x;       // вектор решений
     bool isSolved; // признак совместности
     int* reoder;    // перестановка переменных,
     // полученная в методе Жордана-Гаусса
     int rang;       // ранг матрицы коэффициентов
```

```
    .    .    .
};
```

Помимо конструктора класса и дружественных функций ввода/вывода, класс `Slau` будет содержать методы решения системы линейных уравнений различными способами:

- метод Крамера,
- метод решения с помощью обратной матрицы,
- метод исключения Жордана-Гаусса.

```
// класс, определяющий систему линейных уравнений
class Slau
{
     int m;          // количество уравнений
     int n;          // количество переменных
     Matrix a;       // матрица коэф.
     Matrix b;       // вектор правой части
     Matrix x;       // вектор решений
     bool isSolved; // признак совместности
     int* reoder;    // перестановка переменных,
                     // полученная в методе Жордана-Гаусса
     int rang;       // ранг матрицы коэффициентов
public:
     Slau(int, int);      // конструктор
     void Solve();        // метод решения
     void Kramer();       // метод Крамера
     void InverseMatrix();    // метод X=A^-1B
     void JordanGauss ();     // метод Жордана-Гаусса
     // метод вывода результата
     void PrintSolution(ostream&);
     // дружественные функции ввода/вывода
     friend ostream& operator << (ostream&, Slau&);
     friend istream& operator >> (istream&, Slau&);
};
```

Сначала опишем функции решения системы линейных уравнений различными методами.

Метод Крамера используется для решения системы линейных уравнений с квадратной матрицей коэффициентов. Если матрица коэффициентов СЛАУ является неквадратной, будет генерироваться исключение `NonSquareMatrixException`. Для квадратной матрицы будет вычисляться определитель. Если определитель не равен нулю, единственное решение системы уравнений определяется по формулам Крамера:

$$x_j = \frac{\Delta_j}{\Delta},\ j = 1..n,$$

где Δ – определитель матрицы A, а Δ_j - определитель матрицы, в которой `j`-й столбец исходной матрицы заменен на вектор свободных членов. Если определитель матрицы равен нулю, то будет сгенерировано еще одно исключение `ZeroDevideException`.

```
// Функция решения СЛАУ методом Крамера
void Slau::Kramer()
{
     // проверка, не является ли матрица прямоугольной
     if(m != n)
          throw NonSquareMatrixException();
     // вычисление определителя
     double det = a.Determinant();
     // матрицы коэффициентов
     // проверка определенности системы
     if(det == 0)
          throw ZeroDevideException();
     rang = m;
     // вычисление корней по формулам Крамера
     Matrix temp = a;
     for(int j = 0; j < n; j++)
     {
          for(int i = 0; i < n; i++)
               temp[i][j] = b[0][i];
          x[0][j] = temp.Determinant() / det;
          for(int i = 0; i < n; i++)
               temp[i][j] = a[i][j];
     }
     isSolved = true;
}
```

В случае, если СЛАУ с квадратной матрицей A имеет единственное решение, его можно получить по формуле:

$$X = A^{-1}B, \text{ где } A^{-1} - \text{обратная матрица к } A.$$

Данная формула применима только в случаях, когда обратную матрицу можно вычислить (матрица A – квадратная, ее определитель не равен `0`). Нарушение этих условий приводит к генерации исключений

(генерация исключения равенства нулю определителя предусмотрена в методе вычисления обратной матрицы класса `Matrix`) .

```
// Функция решения СЛАУ с помощью обратной матрицы
void Slau::InverseMatrix()
{
     // проверка, является ли матрица прямоугольной
     if(m != n)
          throw NonSquareMatrixException();
     // вычисление обратной матрицы
     Matrix obr = ~ a;
     // поскольку для эффективного использования памяти
     // вектор хранится как строка, требуется получить
     // соответствующий вектор-столбец посредством
     // транспонирования
     Matrix b = !(this->b);
     // получение решения СЛАУ
     x = obr * b;
     x = ! x;
     rang = m;
     isSolved = true;
}
```

Метод исключения Жордана-Гаусса может быть применен, как в ситуации, когда СЛАУ имеет единственное решение, так и когда этих решений бесконечно много. Проведение исключений всех строк по `i`-ой строке осуществляется по формулам Жордана-Гаусса:

$$a_{ij} = \frac{a_{ij}}{a_{ii}}, \quad j = 1..n$$

$$a_{kj} = a_{kj} - a_{ki}\frac{a_{ij}}{a_{ii}}, \quad j = 1..n, \quad k = 1..m, \quad k \neq i$$

$$b_i = \frac{b_i}{a_{ii}}$$

$$b_k = b_k - a_{ki}\frac{b_i}{a_{ii}}, \quad k = 1..m, \quad k \neq i$$

Эти формулы определяют эквивалентное преобразование СЛАУ, которое не меняет ее решение и позволяет определить ранг матрицы коэффициентов. Напомним, что ранг – это максимальный порядок минора матрицы, отличный от нуля. Посредством исключения нужно

добиться, чтобы один из миноров, соответствующих рангу матрицы, занял положение в ее верхнем левом углу.

Преобразование осуществляется перестановкой строк и столбцов (переменных) матрицы. Согласно формулам Жордана-Гаусса исключение производится с помощью ненулевых элементов строк, расположенных в столбце с тем же номером. Если в строке с номером `i` такой элемент равен нулю, осуществляется поиск ненулевых элементов среди тех, которые расположены под ним в том же столбце. Если будет найден ненулевой элемент в `k`-ой строке, две строки с номерами `i` и `k` меняются местами, отражая изменение порядка следования двух уравнений системы. Решение системы при этом не изменится и можно будет провести исключение по `i`-ой строке. Очевидно, что если ненулевых элементов не будет найдено, `i`-ый столбец является нулевым. Меняем столбцы местами таким образом, чтобы все нулевые столбцы оказались последними (для отслеживания количества нулевых столбцов используется специальная переменная). Поскольку столбцам матрицы соответствуют переменные СЛАУ, изменение порядка столбцов означает изменение порядка переменных. Для фиксации используемого порядка переменных в класс `Slau` следует ввести массив `reoder`, который будет хранить перестановку переменных в результате проведения преобразования системы. Стандартный порядок следования переменных должен быть инициализирован в конструкторе класса `Slau`.

В результате проведения исключений по всем строкам матрицы коэффициентов в ее левом верхнем углу будет расположена единичная матрица. Порядок единичной матрицы равен рангу матрицы коэффициентов. Строки, расположенные ниже единичной матрицы, являются нулевыми.

После определения ранга матрицы могут возникнуть следующие ситуации:

- если какой-либо нулевой строке полученной матрицы будет соответствовать ненулевой элемент в столбце свободных членов, СЛАУ не имеет решений, т.е. является несовместной;
- $m > n$ и ранг матрицы равен количеству переменных. Тогда получаем n уравнений с `n` неизвестными, т.е. система будет иметь единственное решение, которое находится в первых `n` компонентах столбца свободных членов;
- ранг матрицы меньше количества переменных. Тогда СЛАУ имеет множество решений, которые записываются в виде общего

решения системы. Общее решение системы отражает линейную зависимость базисных переменных (их количество равно рангу матрицы), от оставшихся свободных переменных, которые могут принимать любые значения. При подстановке конкретных значений свободных переменных в полученные зависимости значения базисных переменных определяются однозначно.

Для определения общего решения рассмотрим СЛАУ после выполненных преобразований:

$$x_i + a'_{ir+1}x_{r+1} + \ldots + a'_{in}x_n = b'_i, \quad i = 1..r$$

где `r` – ранг матрицы коэффициентов, a'_{ij} – новые коэффициенты при переменных, b'_i – новые значения свободных членов. В таком виде уравнения определяют зависимость базисных переменных от значений свободных переменных, т. е. определяют общее решение системы. В программе его можно представить в виде матрицы:

$$\begin{pmatrix} b'_1 & -a'_{1r+1} & -a'_{1r+2} & \ldots & -a'_{1n} \\ b'_2 & -a'_{2r+1} & -a'_{2r+2} & \ldots & -a'_{2n} \\ b'_3 & -a'_{3r+1} & -a'_{3r+2} & \ldots & -a'_{3n} \\ \ldots & & & & \\ b'_r & -a'_{rr+1} & -a'_{rr+2} & \ldots & -a'_{rn} \end{pmatrix}$$

Порядок следования базисных и свободных переменных будет сохранен в специальном массиве `reoder`.

```
// функция решения СЛАУ с помощью
// метода исключений Жордана-Гаусса
void Slau::JordanGauss()
{
     // создание копий матрицы коэффициентов и свободных
     // членов для последующих преобразований
     Matrix A = a;
     Matrix B = b;
     int count_null_cols = 0;
     // проведение исключений по формулам Жордана-Гаусса
     for(int i = 0; i < m; i++)
```

```
{
        // исключение по i-ой строке
        // проверка возможности исключения
        // по значению ведущего элемента
        if(A[i][i] != 0)
        {
            // исключение во всех строках, кроме ведущей
            for(int k = 0; k < m; k++)
            {
                if(k == i)
                    continue;
                double d = A[k][i] / A[i][i];
                for(int j = i; j < n; j++)
                    A[k][j] = A[k][j] - d * A[i][j];
                B[0][k] = B[0][k] - d * B[0][i];
            }
            // преобразование ведущей строки
            for(int j = i + 1; j < n; j++)
                A[i][j] /= A[i][i];
            // преобразование i-ого свободного члена
            B[0][i] /= A[i][i];
            A[i][i] = 1;
        }
        else
        {
            // элемент главной диагонали
            // в i-ой строке равен нулю
            int k;
            // поиск ненулевого элемента ниже
            // в i-ом столбце
            for(k = i + 1; k < m; k++)
                if(A[k][i] != 0)
                    break;
            if(k == m)
            {
                // все элементы столбца нулевые
                if(i == n - 1 - count_null_cols)
                {
                    // элементов, которые могут быть
                    // ведущими, больше нет
                    count_null_cols++;
                    break;
                }
                // меняем местами столбцы - текущий и
            // последний из непросмотренных
                for(int j = 0; j < m; j++)
                {
                    double t = A[j][i];
                    A[j][i] =
```

```
                              A[j][n-count_null_cols-1];
                         A[j][n-count_null_cols-1] = t;
                    }
                    // отражаем смену столбцов
                    // в перестановке
                    int t = reoder[i];
                    reoder[i] =
                              reoder[n-count_null_cols-1];
                    reoder[n-count_null_cols-1] = t;
                    count_null_cols++;
                    // далее пытаемся провести исключения
                    // с той же строкой
                    i--;
               }
               else
               {
                    // нашли в столбце элемент, который
                    // может быть ведущим –
                    // меняем местами строки
                    double* t = A[i];
                    A[i] = A[k];
                    A[k] = t;
                    double p = B[0][i];
                    B[0][i] = B[0][k];
                    B[0][k] = p;
                    // далее пытаемся провести исключения
                    // с той же строкой
                    i--;
               }
          }
     }

     // вычисление ранга матрицы после
     // проведения исключения
     rang = m < n-count_null_cols ? m : n-count_null_cols;
     // подсчет количества нулевых строк матрицы
     int null_rows = m - rang;
     // проверка на несовместность системы –
     // если в нулевой строке
     // свободный член не равен нулю
     for(int i = rang; i < m; i++)
          if(B[0][i] != 0)
          {
               isSolved = false;
               return;
          }
     // формирование общего решения для совместной СЛАУ
     // путем переноса свободных переменных в правую часть
```

```
        Matrix res(rang, 1 + n - rang);
        for(int i = 0; i < rang; i++)
        {
             res[i][0] = B[0][i];
             for(int j = rang; j < n; j++)
                  res[i][j - rang + 1] = -A[i][j];
        }
        x = res;
        isSolved = true;
   }
```

Выбор метода для решения СЛАУ осуществляется в функции `Solve()`. Если матрица коэффициентов системы является квадратной, то пользователю предлагается задать метод решения системы: метод Крамера или с помощью обратной матрицы. Если определитель квадратной матрицы окажется нулевым, СЛАУ будет решаться методом исключений Жордана-Гаусса. Для решения СЛАУ с прямоугольной матрицей коэффициентов применим только метод Жордана-Гаусса.

```
// функция выбора метода решения
// системы линейных уравнений
void Slau::Solve()
{
     if(m == n)
          try
          {
               // матрица коэффициентов квадратная –
               // предоставлется выбор метода решения
               // пользователю
               cout << "Метод Крамера - 1,
                         С помощью обратной матрицы - 2"
                    << endl;
               int i;
               cin >> i;
               if(i == 1)
                    Kramer();
               else
                    InverseMatrix();
          }
          catch(ZeroDevideException e)
          {
               // генерация исключения происходит при
               // равенстве нулю определителя матрицы
               // коэффициентов осуществляется получение
               // общего решения системы
               JordanGauss();
          }
```

```
      else
            // СЛАУ с прямоугольной матрицей коэффициентов
            // решается методом Жордана-Гаусса для получения
            // общего решения
            JordanGauss();
}
```

В конструкторе класса `Slau` необходимо выделить память для хранения матрицы коэффициентов, вектора свободных членов и вектора решения. Это осуществляется путем вызова конструкторов внедренных объектов класса `Matrix`.

```
// конструктор класса Slau
Slau::Slau(int m1, int n1): a(m1, n1),b(1, m1),x(1, n1)
{
     m = m1;
     n = n1;
     // выделение памяти и заполнение массива
     // для хранения перестановки переменных
     reoder = new int [n];
     for(int i = 0; i < n; i++)
          reoder[i] = i;
}

// дружественная функция ввода СЛАУ
istream& operator >> (istream& in, Slau& ob)
{
     cout << "Матрица коэффициентов: ";
     in >> ob.a;
     cout << "Вектор свободных членов: ";
     in >> ob.b;
     return in;
}

// дружественная функция вывода СЛАУ
ostream& operator << (ostream& out, Slau& ob)
{
     for(int i = 0; i < ob.m; i++)
     {
          for(int j = 0; j < ob.n; j++)
               out << ob.a[i][j] << "\t";
          out << "\t" << ob.b[0][i] << endl;
     }
     try
     {
          out << "Решение СЛАУ: "<<endl;
```

```
            ob.PrintSolution(out);
        }
        catch(Exception& e)
        {
            e.ShowMessage();
        }
        return out;
}

// метод вывода решения СЛАУ
void Slau::PrintSolution(ostream& out)
{
        if(!isSolved)
        {
            out << "Система несовместна" << endl;
            return;
        }
        if(rang < n)
        {
            // получено общее решение системы
            for(int i = 0; i < rang; i++)
            {
                out << "x" << (reoder[i] + 1) <<
                                        " = " << x[i][0];
                for(int j = 1; j <= n - rang; j++)
                {
                    if(x[i][j] == 0)
                        continue;
                    if(x[i][j] > 0)
                        out << "+" << x[i][j] << "*x"
                           << (reoder[rang + j - 1] + 1);
                    else
                        out << x[i][j] << "*x" <<
                             (reoder[rang + j - 1] + 1);
                }
                out << endl;
            }
        }
        else
        {
            // получен единственный вектор решений
            out << "(";
            for(int i = 0; i < n - 1; i++)
                out << x[0][i] << ", ";
            out << x[0][n - 1] << ")" << endl;
        }
}
```

Использование класса `Slau` может быть таким:

```
void main(void)
{
     try
     {
          int m, n;
          cout << "Введите количество уравнений системы:";
          cin >> m;
          cout << "Введите количество переменных
                                              системы:";
          cin >> n;
          // создание объекта системы линейных уравнений
          Slau s(m, n);
          cin >> s;
          // решение системы линейных уравнений
          s.Solve();
          cout << s;
     }
     catch(Exception& e)
     {
          e.ShowMessage();
     }
}
```

Домашнее задание

1. Написать класс для решения задачи поиска точки, минимизирующей полином на отрезке. Реализовать функции решения этой задачи различными методами – методом деления отрезка пополам, методом золотого сечения, методом касательных. Выбор метода решения осуществить в зависимости от унимодальности полинома (полином является унимодальным, если все ненулевые слагаемые имеют степень 0, 1, 2, 4, 6, 8, 10, … 2k, где k – натуральное число).
2. Написать абстрактный класс «Задача», в котором определены три чисто виртуальные функции – ввода задачи, решения задачи и вывода результата. Наследовать от него класс `Slau`, реализованный в данном разделе, и класс поиска точки минимума функции на отрезке из задания 1. Написать консольное приложение, которое по выбору пользователя позволяет решать задачи обоих типов.

3.6. Решение системы линейных уравнений с коэффициентами – рациональными дробями

Несложно изменить классы решения систем линейных уравнений для случая, когда коэффициентами системы будут, к примеру, комплексными числами или рациональными дробями. В этом случае будет удобно реализовать данные классы на основе шаблонов. Поскольку коэффициенты в классе `Slau` хранились в объектах типа `Matrix`, потребуется сделать два шаблона – шаблоны классов `Matrix` и `Slau`. Таким образом, объекты класса, сгенерированного на основе шаблона класса `Slau` для типа `double`, будут содержать внутри себя объекты класса, сгенерированного на основе шаблона класса `Matrix`, для того же типа `double`.

Создание шаблона классов `Matrix` и `Slau` на базе уже разработанных в предыдущем разделе классов не требует особых комментариев. Приведем только программный код разработанных шаблонов, выделив изменения, связанные с использованием обобщенного типа данных.

Объявление шаблона класса матрицы:

```
template <class T> class Matrix
{
protected:
     T** a;     // коэффициенты матрицы имеют
                // обобщенный тип T
     int m, n; // размеры матрицы
public:
     // конструктор матрицы заданных размеров
     Matrix(int ,int);
     // конструктор «пустой» матрицы
     Matrix();
     // конструктор копирования матрицы
     Matrix(const Matrix&);
     // деструктор матрицы
     ~Matrix();
     // оператор суммирования двух матриц
     Matrix operator + (Matrix&);
     // оператор умножения двух матриц
```

```
      Matrix operator * (Matrix&);
      // оператор умножения матрицы на значение
      // типа T (типа данных элемента матрицы)
      Matrix operator * (T);
      // оператор присваивания матрицы
      Matrix& operator = (Matrix&);
      // оператор получения строки матрицы
      T* operator [] (int);
      // оператор получения транспонированной матрицы
      Matrix operator !();
      // метод вычисления определителя матрицы –
      // определитель будет иметь тип данных элемента
      // матрицы, например, определитель матрицы
      // комплексных чисел – комплексное число
      T Determinant();
      // метод получение подматрицы,
      // полученной вычеркиванием
      // одной строки и одного столбца
      Matrix SubMatrix(int, int);
      // оператор получения обратной матрицы
      Matrix operator ~ ();
      // дружественные шаблоны функций для умножения числа
      // на матрицу и реализации операций ввода/вывода
      template <class T>
           friend Matrix<T> operator * (T, Matrix<T>&);
      template <class T>
           friend  ostream& operator <<
                                    (ostream&, Matrix<T>&);
      template <class T>
           friend  istream& operator >>
                                    (istream&, Matrix<T>&);
};
```

Объявление шаблона класса СЛАУ:

```
template <class T> class Slau
{
     int m;          // количество уравнений
     int n;          // количество переменных
     Matrix<T> a;  // матрица коэффициентов
                     // обобщенного типа
     Matrix<T> b;  // вектор элементов правой части
                     // обобщенного типа
     Matrix<T> x;  // решение системы линейных уравнений
     bool isSolved;// определяет, является ли
                     // система совместной
     int* reoder;   // массив перестановки переменных
     int rang;    // ранг матрицы
```

```
public:
     // конструктор СЛАУ с параметрами –
     // количество уравнений и переменных
     Slau(int, int);
     // функция выбора метода решения СЛАУ
     void Solve();
     // функция, реализующая метод Крамера
     void Kramer();
     // функция, реализующая решение СЛАУ
     // с помощью обратной матрицы
     void InverseMatrix();
     // функция получения общего решения СЛАУ
     // методом Жордана-Гаусса
     void JordanGauss();
     // дружественные шаблоны функций,
     // переопределяющие операции ввода/вывода СЛАУ
     template <class T>
          friend ostream& operator <<(ostream&, Slau<T>&);
     template <class T>
          friend istream& operator >>(istream&, Slau<T>&);
     // метод вывода информации про решение СЛАУ
     void PrintSolution(ostream&);
};
```

Определение методов шаблона класса `Matrix` и дружественных функций:

```
// конструктор матрицы заданных размеров
template <class T> Matrix<T>::Matrix(int m1, int n1)
{
     if(n1 <= 0 || m1 <= 0)
          throw BadDimensionException();
     n = n1;
          m = m1;
     a = new T* [m];
     for(int i = 0; i < m; i++)
          a[i] = new T [n];
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = 0;
}

// конструктор «пустой» матрицы
template <class T> Matrix<T>::Matrix()
{
     n = m = 0;
     a = NULL;
}
```

```
// конструктор копирования матрицы
template <class T> Matrix<T>::Matrix(const Matrix<T>& ob)
{
     n = ob.n;
          m = ob.m;
     if(ob.a == NULL)
     {
          a = NULL;
          return;
     }
     a = new T* [m];
     for(int i = 0; i < m; i++)
          a[i] = new T [n];
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = ob.a[i][j];
}

// деструктор матрицы
template <class T> Matrix<T>::~Matrix()
{
     if(a != NULL)
     {
          for(int i = 0; i < m; i++)
               delete [] a[i];
          delete [] a;
     }
}

// шаблон дружественного оператора ввода матрицы
template <class T>
istream& operator >>(istream& in, Matrix<T>& ob)
{
     if(ob.a != NULL)
     {
          for(int i = 0; i < ob.m; i++)
               for(int j = 0; j < ob.n; j++)
                    in >> ob[i][j];
     }
     return in;
}

// шаблон дружественного оператора вывода матрицы
template <class T>
ostream& operator <<(ostream& out, Matrix<T>& ob)
{
     if(ob.a != NULL)
```

```
        {
            out << "Матрица" << endl;
            for(int i = 0; i < ob.m; i++)
            {
                for(int j = 0; j < ob.n; j++)
                    out << ob[i][j] << "\t";
                out << endl;
            }
        }
        else
            out << "Матрица пустая" << endl;
        return out;
}

// оператор получения строки матрицы
template <class T>  T* Matrix<T>::operator [](int i)
{
        return a[i];
}

// оператор сложения двух матриц
template <class T>
Matrix<T> Matrix<T>::operator +(Matrix<T>& ob)
{
        if(ob.n == n && ob.m == m)
        {
            Matrix<T> temp(m, n);
            for(int i = 0; i < m; i++)
                for(int j = 0; j < n; j++)
                    temp.a[i][j] = a[i][j] + ob.a[i][j];
            return temp;
        }
        else
            throw DimensionSumException();
}

// оператор умножения двух матриц
template <class T>
Matrix<T> Matrix<T>::operator *(Matrix<T>& ob)
{
        if(n == ob.m)
        {
            Matrix<T> temp(m, ob.n);
            for(int i = 0; i < m; i++)
                for(int j = 0; j < ob.n; j++)
                    for(int k = 0; k < n; k++)
                        temp.a[i][j] = temp.a[i][j] +
                                       a[i][k] * ob.a[k][j];
            return temp;
```

```
      }
      else
         throw DimensionProductException();
}

// оператор умножения матриц на число типа T справа
template <class T> Matrix<T> Matrix<T>::operator *(T b)
{
     Matrix<T> temp(m, n);
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               temp[i][j] = a[i][j] * b;
     return temp;
}

// оператор умножения матриц на число типа T слева
template <class T>
Matrix<T> operator * (T b, Matrix<T>& ob)
{
     Matrix<T> temp(ob.m, ob.n);
     for(int i = 0; i < ob.m; i++)
          for(int j = 0; j < ob.n; j++)
               temp[i][j] = b * ob[i][j];
     return temp;
}

// оператор присваивания матрицы
template <class T>
Matrix<T>& Matrix<T>::operator =(Matrix<T>& ob)
{
     cout << "Matrix = " << endl;
     if(n != ob.n || m != ob.m)
     {
          for(int i = 0; i < m; i++)
               delete [] a[i];
          delete [] a;
          n = ob.n;
          m = ob.m;
          a = new T* [m];
          for(int i = 0; i < m; i++)
               a[i] = new T [n];
     }
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               a[i][j] = ob.a[i][j];
     return *this;
}
```

```
// оператор траспонирования матрицы
template <class T> Matrix<T> Matrix<T>::operator !()
{
     Matrix<T> temp(n, m);
     for(int i = 0; i < m; i++)
          for(int j = 0; j < n; j++)
               temp.a[j][i] = a[i][j];
     return temp;
}

// функция получения подматрицы путем вычеркиванмя строки
// с номером i1 и столбца с номером j1
template <class T>
Matrix<T> Matrix<T>::SubMatrix(int i1,int j1)
{
     Matrix<T> temp(m - 1, n - 1);
     for(int i = 0; i < i1; i++)
     {
          for(int j = 0; j < j1; j++)
               temp.a[i][j] = a[i][j];
          for(int j = j1 + 1; j < n; j++)
               temp.a[i][j - 1] = a[i][j];
     }
     for(int i = i1 + 1; i < m; i++)
     {
          for(int j = 0; j < j1; j++)
               temp.a[i - 1][j] = a[i][j];
          for(int j = j1 + 1; j < n; j++)
               temp.a[i - 1][j - 1] = a[i][j];
     }
     return temp;
}

// функция вычисления определителя матрицы
template <class T> T Matrix<T>::Determinant()
{
     T det = 0;
     if(m != n)
          throw NonSquareMatrixException();
     if(n == 1)
               return a[0][0];
     Matrix<T> temp(m - 1, n - 1);
     for(int j = 0; j < n; j++)
     {
          temp = SubMatrix(0, j);
          if(j % 2 == 0)
               det = det + temp.Determinant() * a[0][j];
          else
               det = det - temp.Determinant() * a[0][j];
```

```
     }
     return det;
}

// метод вычисления обратной матрицы
template <class T> Matrix<T> Matrix<T>::operator ~()
{
     if(m != n)
          throw NonSquareMatrixException();
     Matrix<T> res(n, n);
     T det = Determinant();
     if(det == 0.0)
          throw ZeroDevideException();
     Matrix<T> temp(n - 1, n - 1);
     int z;
     for(int i = 0; i < n; i++)
     {
          z = i%2==0 ? 1 : -1;
          for(int j = 0; j < n; j++)
          {
               temp = SubMatrix(i, j);
               res[j][i] = z * temp.Determinant() / det;
               z = -z;
          }
     }
     return res;
}

// конструктор класса Slau с заданным количеством
// уравнений m1 и неизвестных n1
template <class T>
Slau<T>::Slau(int m1, int n1): a(m1, n1),b(1, m1),x(1, n1)
{
     m = m1;
     n = n1;
     reoder = new int [n];
     for(int i = 0; i < n; i++)
          reoder[i] = i;
}

// шаблон дружественного оператора ввода СЛАУ
template <class T>
istream& operator >>(istream& in, Slau<T>& ob)
{
     cout << "Матрица коэффициентов: ";
     in >> ob.a;
     cout << "Вектор свободных членов: ";
     in >> ob.b;
```

```
     return in;
}

// функция вывода решения СЛАУ
template <class T>
void Slau<T>::PrintSolution(ostream& out)
{
     if(!isSolved)
     {
          out << "Система несовместна" << endl;
          return;
     }
     if(rang < n)
     {
          for(int i = 0; i < rang; i++)
          {
               out << "x" << (reoder[i] + 1) <<
                                        " = " << x[i][0];
               for(int j = 1; j <= n - rang; j++)
               {
                    if(x[i][j] == 0.0)
                              continue;
                    if(x[i][j] > 0.0)
                         out << "+" << x[i][j] << "*x" <<
                              (reoder[rang + j - 1] + 1);
                    else
                         out << x[i][j] << "*x"
                         << (reoder[rang + j - 1] + 1);
               }
               out << endl;
          }
     }
     else
     {
          out << "(";
          for(int i = 0; i < n - 1; i++)
               out << x[0][i] << ", ";
          out << x[0][n - 1] << ")" << endl;
     }
}

// шаблон дружественного оператора вывода СЛАУ
template <class T>
ostream& operator <<(ostream& out, Slau<T>& ob)
{
     for(int i = 0; i < ob.m; i++)
     {
          for(int j = 0; j < ob.n; j++)
               out << ob.a[i][j] << "\t";
```

```
            out << "\t" << ob.b[0][i];
            out << endl;
        }
        try
        {
            out << "Решение: " << endl;
            ob.PrintSolution(out);
        }
        catch(Exception& e)
        {
            e.ShowMessage();
        }
        return out;
}

// функция решения СЛАУ методом Крамера
template <class T> void Slau<T>::Kramer()
{
        if(m != n)
            throw NonSquareMatrixException();
        T det = a.Determinant();
            if(det == 0.0)
                throw ZeroDevideException();
        rang = m;
        Matrix<T> temp = a;
        for(int j = 0; j < n; j++)
        {
            for(int i = 0; i < n; i++)
                temp[i][j] = b[0][i];
            x[0][j] = temp.Determinant() / det;
            for(int i = 0; i < n; i++)
                temp[i][j] = a[i][j];
        }
        isSolved = true;
}

// функция решения СЛАУ с помощью обратной матрицы
template <class T> void Slau<T>::InverseMatrix()
{
        if(m != n)
            throw NonSquareMatrixException();
        Matrix<T> obr = ~ a;
        Matrix<T> b = !(this -> b);
        x = obr * b;
        x = !x;
        rang = m;
        isSolved = true;
}
```

```
// функция решения СЛАУ методом Жордана-Гаусса
template <class T> void Slau<T>::JordanGauss()
{
     Matrix<T> A = a;
     Matrix<T> B = b;
     bool code = true;
     int count_null_cols = 0;
     for(int i = 0; i < m; i++)
     {
          if(A[i][i] != 0.0)
          {
               for(int k = 0; k < m; k++)
               {
                    if(k == i)
                         continue;
                    T d = A[k][i] / A[i][i];
                    for(int j = i; j < n; j++)
                         A[k][j] = A[k][j] - d * A[i][j];
                    B[0][k] = B[0][k] - d * B[0][i];
               }
               for(int j = i + 1; j < n; j++)
                    A[i][j] = A[i][j] / A[i][i];
               B[0][i] = B[0][i] / A[i][i];
               A[i][i] = 1;
          }
          else
          {
               int k;
               for(k = i + 1; k < m; k++)
                    if(A[k][i] != 0.0)
                              break;
               if(k == m)
               {
                    if(i == n - 1 - count_null_cols)
                    {
                         count_null_cols++;
                         code = false;
                         break;
                    }
                    for(int j = 0; j < m; j++)
                    {
                         T t = A[j][i];
                         A[j][i] =
                              A[j][n-count_null_cols-1];
                         A[j][n-count_null_cols-1] = t;
                    }
                    int t = reoder[i];
                    reoder[i] =
```

```
                                   reoder[n-count_null_cols-1];
                        reoder[n-count_null_cols-1] = t;
                        count_null_cols++;
                        i--;
                   }
                   else
                   {
                        T* t = A[i];
                        A[i] = A[k];
                        A[k] = t;
                        T p = B[0][i];
                        B[0][i] = B[0][k];
                        B[0][k] = p;
                        i--;
                   }
              }
         }
         rang = m < n-count_null_cols ?
                                        m: n - count_null_cols;
         int null_rows = m - rang;
         for(int i = rang; i < m; i++)
              if(B[0][i] != 0.0)
              {
                   isSolved = false;
                   return;
              }
         Matrix<T> res(rang,1 + n - rang);
         for(int i = 0; i < rang; i++)
         {
              res[i][0] = B[0][i];
              for(int j = rang; j < n;j++)
                   res[i][j - rang + 1] =- A[i][j];
         }
         x = res;
         isSolved = true;
}

// функция решения СЛАУ
template <class T> void Slau<T>::Solve()
{
         if(m == n)
              try
              {
                   cout << "Метод Крамера - 1,
                             С помощью обратной матрицы - 2"
                             << endl;
                   int i;
                   cin >> i;
```

```
                if(i == 1)
                    Kramer();
                else
                    InverseMatrix();
            }
            catch(ZeroDevideException e)
            {
                JordanGauss();
            }
        else
            JordanGauss();
    }
```

Отметим, что если в качестве обобщенного типа выступает пользовательский тип данных, то в нем должны быть определены методы, которые используются в шаблоне. Например,

- в шаблоне класса `Matrix` создается массив элементов обобщенного типа `T`:

```
a[i] = new T [n];
```

Для этого вызывается конструктор класса `T` без параметров;

- в шаблоне класса `Matrix` могут создаваться объекты с начальной инициализацией:

```
T det = 0.0;
```

В этом случае конкретные пользовательские классы, используемые при подстановке, должны содержать конструкторы с параметром типа присваиваемого значения. Для приведенного примера в подставляемых классах должен быть определен конструктор с параметром типа `double`;

- в шаблоне класса `Matrix` осуществляется присваивание элементам матрицы новых значений. Поэтому в пользовательском классе, который будет подставлен вместо обобщенного типа, необходимо определить оператор присваивания (кроме случаев, когда предоставляемый компилятором оператор работает корректно);
- если в шаблоне класса используются арифметические операции, операции сравнения с объектами различных типов, операции ввода/вывода и др., то в пользовательском классе, который будет подставлен вместо обобщенного типа, необходимо определить эти операции.

В функции `main()` приведен пример генерации класса `Slau` с указанием конкретного пользовательского типа данных `Fraction` (дробь) для элементов матрицы коэффициентов, правой части, решения системы уравнений. Все методы класса `Slau` также будут сгенерированы для конкретного типа `Fraction`.

```
void main(void)
{
     try
     {
          int m, n;
          cout << "Введите количество уравнений и
                         количество неизвестных:";
          cin >> m >> n;
          cin.get();
          Slau<Fraction> s(m, n);
          cin >> s;
          s.Solve();
          cout << s;
     }
     catch(Exception& e)
     {
          e.ShowMessage();
     }
}
```

Домашнее задание

1. Создать шаблон класса «Односвязный список» и применить его для хранения списка товаров магазина и списка заявок от клиентов на покупку этих товаров (раздел 3.4 задание 1).
2. Создать шаблон класса «Очередь» и применить его для хранения списка заявок на товары от клиентов (для каждого товара своя очередь) при решении предыдущей задачи для.
3. Создать шаблон класса «Ассоциативный массив», индексом которого может быть элемент любого типа данных, включая объект класса, значение также может быть любого типа данных. Применить этот шаблон для хранения телефонной книжки (индекс – имя абонента, значение – массив телефонов). Применить этот шаблон для хранения каталога товаров (индекс – объект класса «Товар», значение– цена товара).

Предметный указатель

Литература

1. Прата, С. Язык программирования C++. Лекции и упражнения [Текст]: пер. с англ. / Стивен Прата. – СПб: ООО «ДиаСофтЮП», 2005. - 1104 с.

2. Страуструп, Б. Язык программирования C++ [Текст]: пер. с англ. / Бьярн Страуструп – СПб: Бином, 1999. – 991 с.

3. Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка [Текст]: пер. с англ. / Джеймс Рамбо, Мартин Блаха. – СПб: Питер, 2007. – 544 с.

4. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ [Текст]: пер. с англ. / Гради Буч. – М: Бином, СПб: Невский диалект, 1999. – 560 с.

5. Пышкин, Е.В. Основные концепции и механизмы объектно-ориентированного программирования [Текст]/ Е.В.Пышкин. – СПб: БХВ-Петербург, 2005. – 640 с.

6. Шилдт, Г.. Самоучитель C++ [Текст]: пер. с англ. / Герберт Шилдт - СПб: БХВ-Петербург, 2005. - 688 с.

7. Дейтел, Х. Как программировать на C++ [Текст]: пер. с англ./ Харви Дейтел, Пол Дейтел. – М: Издательство БИНОМ, 2001. – 1152 с.

8. Павловская, Т.А. C/C++, Программирование на языке высокого уровня [Текст] / Т.А. Павловская. – СПб.: Питер, 2001. – 460 с.

9. Павловская, Т.А. C++. Объектно-ориентированное программирование. Практикум. [Текст]/ Т.А. Павловская, Ю.А. Щупак. – СПб: Питер, 2005. – 265 с.

10. Кубенский, А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ [Текст] /

А.А. Кубенский. – СПб: БХВ-Петербург, 2004. – 464 с.

11. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.

12. Саттер, Г. Стандарты программирования на C++ [Текст]: пер. с англ. / Герб Саттер, Андрей Александреску. – М: Издательский дом «Вильямс», 2005. – 224 с.

13. Шилдт, Г. Справочник программиста по C/C++ [Текст]: пер. с англ. / Герберт Шилдт. – Москва: Издательский дом “Вильямс”, 2001. – 448 с.

14. Крячков, А.В. Программирования на C и C++. Практикум. [Текст]/ А.В. Крячков, И.В. Сухинина, В.К. Томшин – М: Горячая линия-Телеком, 2000. – 344 с.

15. Андрианова, А.А. Практикум по курсу «Алгоритмизация и программирование». Часть 1. [Текст]/ А.А. Андрианова, Т.М. Мухтарова. – Казань: Казанский государственный университет, 2008.- 96 с.

16. Андрианова, А.А. Практикум по курсу «Алгоритмизация и программирование». Часть 2. [Текст]/ А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский государственный университет, 2009.- 132 с.