Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»

Кафедра «Вычислительные методы и программирование»

Шестакович В. П.

Электронный учебно-методический комплекс

по дисциплине

«ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ»

Для студентов специальностей

36 04 01 «Электронно-оптические системы и технологии»,
39 02 02 «Проектирование и производство радиоэлектронных средств»,
39 02 03 «Медицинская электроника»,
39 02 01 «Моделирование и компьютерное проектирование радиоэлектронных средств»,
38 02 03 «Техническое обеспечение безопасности».

Курс лекций

Минск 2010

СОДЕРЖАНИЕ

ЛЕКЦИЯ 1. КАК УСТРОЕНА ЭВМ И КАК ОНА РАБОТАЕТ 7
1.1. История создания ЭВМ 7
1.2. Структура ПЭВМ 8
1.3. Размещение данных и программ в памяти ПЭВМ..... 9
1.4.Файловая система хранения информации 10
1.5.Операционная система..... 11
ЛЕКЦИЯ 2. КАК СОСТАВЛЯЮТСЯ И ВЫПОЛНЯЮТСЯ ПРОГРАММЫ В СИСТЕМЕ DELPHI 12
2.1. Понятие алгоритма и способы его записи 12
2.2. Общая характеристика языка Pascal..... 14
2.3. Как составляется программа в системе Delphi..... 15
2.4. Наша первая программа реализует линейный алгоритм..... 18
ЛЕКЦИЯ 3. БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА ПАСКАЛЬ..... 22
3.1. Данные и их типы..... 22
3.2. Операции над переменными основных скалярных типов 26
ЛЕКЦИЯ 4. ПРОГРАМИРОВАНИЕ РАЗВЕТВЛЯЮЩИХСЯ АЛГОРИТМОВ..... 31
4.1. Понятие разветвляющегося алгоритма 31
4.2. Оператор условия *if*..... 31
4.3. Оператор выбора *Case* 34
4.4. Некоторые возможности, предоставляемые Delphi для организации разветвлений..... 34
ЛЕКЦИЯ 5. СОСТАВЛЕНИЕ И ПРОГРАМИРОВАНИЕ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ 35
5.1. Понятие цикла..... 35
5.2. Оператор *Repeat...Until* 35
5.3. Оператор *While...do* 36
5.4. Оператор *For...do*..... 38
5.5. Вложенные циклы 39
5.6. Примеры некоторых часто встречающихся циклических алгоритмов 39
ЛЕКЦИЯ 6. ОТЛАДКА ПРОГРАММ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ..... 41
6.1. Ошибки на этапе компиляции..... 41
6.2. Ошибки на этапе выполнения..... 41
6.3. Понятие исключительной ситуации..... 42
6.4. Защищенные блоки 43
6.5. Некоторые стандартные типы исключительных ситуаций..... 45
6.6. Инициирование собственных исключительных ситуаций 45
6.7. Примеры фрагментов программ 45
ЛЕКЦИЯ 7. СОСТАВЛЕНИЕ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ МАССИВОВ 51

7.1. Понятие массива 51
7.2. Некоторые возможности ввода-вывода в Delphi 52
7.3. Примеры часто встречающихся алгоритмов работы с массивами 54
ЛЕКЦИЯ 8. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ 58
8.1. Статическое и динамическое распределение оперативной памяти 58
8.2. Понятие указателя 58
8.3. Наложение переменных 59
8.4. Динамическое распределение памяти 60
8.5. Организация динамических массивов 60
ЛЕКЦИЯ 9. ПОДПРОГРАММЫ И БИБЛИОТЕКИ 63
9.1. Понятие подпрограммы 63
9.2. Описание подпрограмм 63
9.3. Передача данных между подпрограммой и вызывающей ее программой 65
9.4. Оформление подпрограмм в библиотечный модуль 68
9.5. Примеры подпрограмм, оформленных в отдельные библиотечные модули 70
ЛЕКЦИЯ 10. ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ ТИПА МНОЖЕСТВА 73
10.1. Понятие множества 73
10.2. Операции над множествами 73
10.3. Примеры работы с множествами 74
ЛЕКЦИЯ 11. ИСПОЛЬЗОВАНИЕ СТРОКОВЫХ ДАННЫХ 78
11.1. Зачем нужны строки 78
11.2. Описание переменных строкового типа 78
11.3. Основные операции над переменными строкового типа 80
11.4. Некоторые процедуры и функции обработки строк 80
11.5. Примеры алгоритмов обработки строк 81
ЛЕКЦИЯ 12. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ЗАПИСЕЙ 83
12.1. Понятие записи 83
12.2. Операции над записями 84
12.3. Использование записей для работы с комплексными числами 85
ЛЕКЦИЯ 13. ИСПОЛЬЗОВАНИЕ ФАЙЛОВ 87
13.1. Понятие файла 87
13.2. Операции над файлами 88
13.2.1. Типизированные файлы 88
13.2.2. Текстовые файлы 89
13.2.3. Нетипизированные файлы 90
13.3. Подпрограммы работы с файлами 90
13.4. Компоненты *OpenDialog* и *SaveDialog* 91

ЛЕКЦИЯ 14. ПРОГРАММИРОВАНИЕ С ОТОБРАЖЕНИЕМ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ 100
14.1. Как рисуются изображения 100
14.2. Построение графиков с помощью компонента *Chart* 101
ЛЕКЦИЯ 15. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ РЕКУРСИИ 106
15.1. Понятие рекурсии 106
15.2. Примеры рекурсивных вычислений 107
ЛЕКЦИЯ 16. ПОИСК И СОРТИРОВКА МАССИВОВ 109
16.1. Организация работы с базами данных 109
16.2. Поиск в массиве записей 109
16.3. Сортировка массивов 110
16.3.1. Метод пузырька 111
16.3.2. Метод прямого выбора 112
16.3.3. Метод Шелла 113
16.3.4. Метод Хоара (Hoare) 114
ЛЕКЦИЯ 17. РАБОТА СО СПИСКАМИ НА ОСНОВЕ ДИНАМИЧЕСКИХ МАССИВОВ 116
17.1. Работа со списками 116
17.2. Добавление нового элемента в список на заданную позицию 116
17.3. Удаления элемента с заданным номером 117
17.4. Пример программы 117
ЛЕКЦИЯ 18. СВЯЗАННЫЕ СПИСКИ НА ОСНОВЕ РЕКУРСИВНЫХ ДАННЫХ 122
18.1. Что такое стек и очередь 122
18.2. Понятие рекурсивных данных и однонаправленные списки. 122
18.3. Процедуры для работы со стеками 125
18.4. Процедуры для работы с односвязными очередями 128
18.5. Работа с двухсвязными очередями 135
18.6. Процедуры для работы с двусвязными очередями 137
ЛЕКЦИЯ 19. АЛГОРИТМЫ РЕШЕНИЯ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ 146
19.1. Основные понятия и определения 146
19.2. Прямые методы решения СЛАУ 147
19.2.1. Метод Гаусса 147
19.2.2. Метод прогонки 148
19.2.3. Метод квадратного корня 150
19.3. Итерационные методы решения СЛАУ 151
19.3.1. Метод простой итерации 151
19.3.2. Метод Зейделя 151
19.3.3. Понятие релаксации 152
ЛЕКЦИЯ 20. АППРОКСИМАЦИЯ ФУНКЦИЙ 153
20.1. Зачем нужна аппроксимация функций? 153

20.2. Что такое интерполяция 154
20.3. Многочлены и способы интерполяции 156
20.3.1. Интерполяционный многочлен Ньютона 156
20.3.2. Линейная и квадратичная интерполяция 156
20.3.3. Интерполяционный многочлен Лагранжа 157
20.3.4. Интерполяция общего вида, использующая прямое решение системы (20.2) методом Гаусса 157
20.4. Среднеквадратичная аппроксимация 157
20.4.1. Метод наименьших квадратов 158
ЛЕКЦИЯ 21. ВЫЧИСЛЕНИЕ ПРОИЗВОДНЫХ И ИНТЕГРАЛОВ 160
21.1. Формулы численного дифференцирования 160
21.2. Формулы численного интегрирования 161
21.2.1. Формула средних 161
21.2.2. Формула трапеций 162
21.2.3. Формула Симпсона 162
21.2.4. Формулы Гаусса 162
ЛЕКЦИЯ 22. МЕТОДЫ РЕШЕНИЯ НЕЛИНЕЙНЫХ УРАВНЕНИЙ .. 164
22.1. Как решаются нелинейные уравнения 164
22.2. Итерационные методы уточнения корней 165
22.2.1. Метод простой итерации 165
22.2.2. Метод Ньютона 166
22.2.3. Метод секущих 166
22.2.4. Метод Вегстейна 167
22.2.5. Метод парабол 167
22.2.6. Метод деления отрезка пополам 168
ЛЕКЦИЯ 23. МЕТОДЫ ОПТИМИЗАЦИИ 169
23.1. Постановка задач оптимизации, их классификация 169
23.2. Методы нахождения минимума функции одной переменной 170
23.2.1. Метод деления отрезка пополам 171
23.2.2. Метод золотого сечения 171
23.2.3. Метод Фибоначчи 173
23.2.4. Метод последовательного перебора 174
23.2.5. Метод квадратичной параболы 174
23.2.6. Метод кубической параболы 175
ЛЕКЦИЯ 24. РЕШЕНИЕ ЗАДАЧИ КОШИ ДЛЯ ОБЫКНОВЕННЫХ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ 177
24.1. Задачи для обыкновенных дифференциальных уравнений ... 177
24.2. Основные положения метода сеток для решения задачи Коши 178
24.2.1. Явная схема 1-го порядка (метод Эйлера) 179
24.2.2. Неявная схема 1-го порядка 179
24.2.3. Неявная схема 2-го порядка 180
24.2.4. Схема предиктор-корректор (Рунге-Кутта) 2-го порядка 180

24.2.5. Схема Рунге-Кутта 4-го порядка 181
24.3. Многошаговые схемы Адамса 181
24.3.1. Явная экстраполяционная схема Адамса 2-го порядка .. 182
24.3.2. Явная экстраполяционная схема Адамса 3-го порядка .. 182
24.3.3. Неявная схема Адамса 3-го порядка 182
ПРИЛОЖЕНИЕ 1. Процедуры и функции для преобразования строкового представления чисел 184
ПРИЛОЖЕНИЕ 2. Математические формулы 185
ПРИЛОЖЕНИЕ 3. Таблицы символов ASCII 186
ЛИТЕРАТУРА 187

ЛЕКЦИЯ 1. КАК УСТРОЕНА ЭВМ И КАК ОНА РАБОТАЕТ

1.1. История создания ЭВМ

Проблема вычислений сопровождает человечество на всем историческом отрезке его существования. Первый счетный инструмент ***абак*** был известен еще в Y веке до нашей эры в Египте, Финикии, Греции и представлял дощечку, покрытую слоем песка, на которой острой палочкой проводили линии и в получавшихся колонках по позиционному принципу размещали камешки. В древнем Риме абак назывался Calculi. От этого слова произошло в дальнейшем латинское calculatore (вычислять).

С конца XY столетия в Западной Европе получил распространение тип абака, известный как «счет на линиях». На разлинованную таблицу накладывались специальные жетоны, горизонтальные линии таблицы соответствовали единицам, десяткам и т.д., вертикальные линии образовывали столбцы для отдельных слагаемых или множителей.

Первую счетную машину для выполнения сложений и вычитаний изобрел и сконструировал в 1623 году профессор математики и астрономии Тюбингенского университета В.Шинкард. Изготовленная в одном экземпляре машина Шинкарда сгорела во время пожара в 1624 году и не оказала влияния на развитие идей счетной техники.

Биография механических счетных машин ведется от арифметической машины французского математика, физика и философа Б.Паскаля, созданной в 1642 году. Над счетной машиной Б.Паскаль работал 12 лет и сделал около 50 действующих моделей. Первый арифмометр, выполняющий все четыре арифметических действия, был предложен в 1670 году немецким ученым Г.В.Лейбницем. В Беларуси первая суммирующая машина была изобретена и изготовлена в 1770 году Евной Якобсоном, часовым матером и механиком в г. Несвиже.

Идею ***универсальной вычислительной машины с программным управлением*** впервые предложил в своем неосуществленном проекте в 1834 году английский ученый Ч.Бэббедж. Ее структура совпадала по существу со структурой современных ЭВМ. Однако, большинство из современников ученого не поняли его идей и имя истинного «отца компьютеров» было на долгие годы забыто.

Отличительной особенностью ***электронных вычислительных машин*** (ЭВМ) от счетных машин является наличие устройства управления вычислениями и принцип хранения программы. Еще одной особенностью современных ЭВМ является применение двоичной системы счисления.

Двоичную арифметику разработал Г.В.Лейбниц. Он также предложил арифметизацию логики за 200 лет до создания алгебры Дж.Буля (1815). Так же как двоичная арифметика представляет все числа с помощью двух символов (0,1) так и Булева алгебра оперирует с двумя понятиями (истина, ложь) и тремя операциями (и, или, не).

С помощью этих понятий можно смоделировать любые логические цепочки и построить 16 логических функций. На этой основе строятся все современные логические схемы различной сложности, реализуемые в ЭВМ.

Первая ЭВМ была создана в 1945 году (США), представляла огромное сооружение, содержащее 18000 электронных ламп, 1500 реле и выполняла около 3000 умножений в секунду. Она использовалась для баллистических расчетов, предсказаний погоды и некоторых научно-технических расчетов для военных целей. Мировой парк ЭВМ к 1965 году насчитывал порядка 50 тыс. компьютеров, к началу 1975 – более 200 тысяч.

Первые ***персональные ЭВМ*** (ПЭВМ) появились в начале 70 годов. ПЭВМ содержат клавиатуру, близкую к клавиатуре печатной машинки, системный блок, реализованный на одной плате и графический дисплей. Все это при желании можно разместить в «дипломате». Скорость вычислений достигает 10^8 операций в секунду. За счет выпуска больших партий достигнута относительная дешевизна ПЭВМ и уже близко то время, когда электронный помощник будет в каждой квартире.

1.2. Структура ПЭВМ

Схема ПЭВМ представлена на рис. 1.

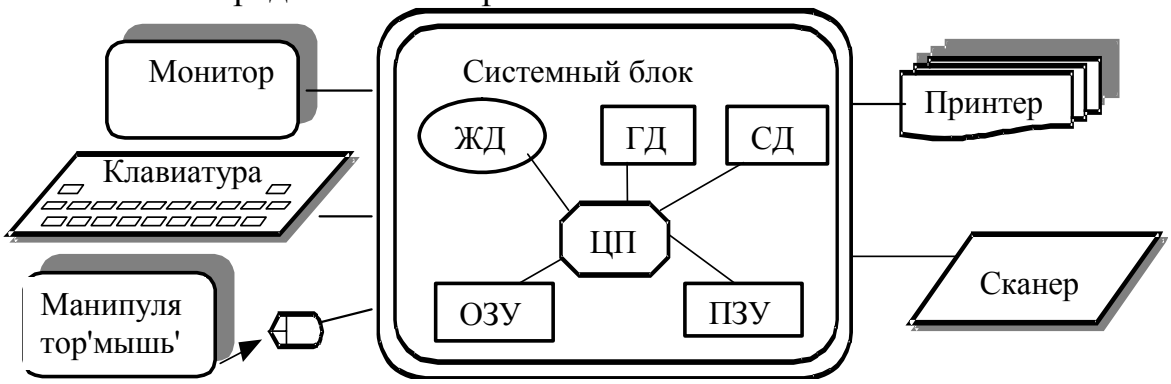


Рис. 1.

Системный блок ПЭВМ содержит следующие блоки:

– ***центральный процессор*** (ЦП), который осуществляет управление работой и выполнение расчетов по программе;

– ***блок «быстрой» оперативной памяти*** (ОЗУ), в которой во время работы компьютера располагаются выполняемые программы (заметим, что при выключении компьютера эта память очищается);

– ***блок постоянной памяти*** (ПЗУ), которая содержит программы, необходимые для запуска компьютера (эти программы не стираются при его выключении);

– ***жесткий магнитный диск*** (ЖД), получивший название винчестер;

– ***дисковод*** (ГД) для сменных, гибких магнитных дисков (дискет).

– ***сидиром*** (СД) – для чтения с компакт-дисков.

В системный блок встроены электронные схемы, управляющие работой различных устройств, входящих в состав компьютера. К системному блоку

подключаются дисплей (монитор) для отображения информации, клавиатура для ввода данных и команд, устройство для визуального управления (мышь), печатающее устройство (принтер), устройство для считывания и ввода графической информации (сканер).

1.3. Размещение данных и программ в памяти ПЭВМ

Данные и программы во время работы ПЭВМ размещаются в оперативной памяти, которая представляет собой последовательность пронумерованных ячеек. По указанному номеру процессор находит нужную ячейку, поэтому номер ячейки называется ее адресом. Минимальная адресованная ячейка (согласно стандарту IBM), с точки зрения программиста, состоит из 8 двоичных позиций, т.е. в каждую позицию могут быть записаны либо 0, либо 1. Объем информации, который помещается в одну двоичную позицию, называется ***бит***. Объем информации равный 8 бит называется ***байтом***.

Таким образом, в одной ячейке из 8 двоичных разрядов помещается объем информации в один байт. Поэтому объем памяти принято оценивать количеством байт (1024 Байт = 1 Килобайт, 1024 Килобайт = 1048576 Байт = 1 Мегабайт, 1024 Мегабайт = 1073741824 Байт = 1 Гигабайт,).

Для помещения данных в такие ячейки производится их запись с помощью нулей и единиц (кодирование). При кодировании каждый символ, допускаемый на клавиатуре, заменяется последовательностью из 8 двоичных разрядов в соответствии со стандартной кодовой таблицей. (т.е. один символ размещается в одном байте). Например, в соответствии с таблицей кодов ASCII D=(01000100); F=(00100110); 4=(00110100).

При кодировании чисел они предварительно преобразуются в двоичное представление. Например

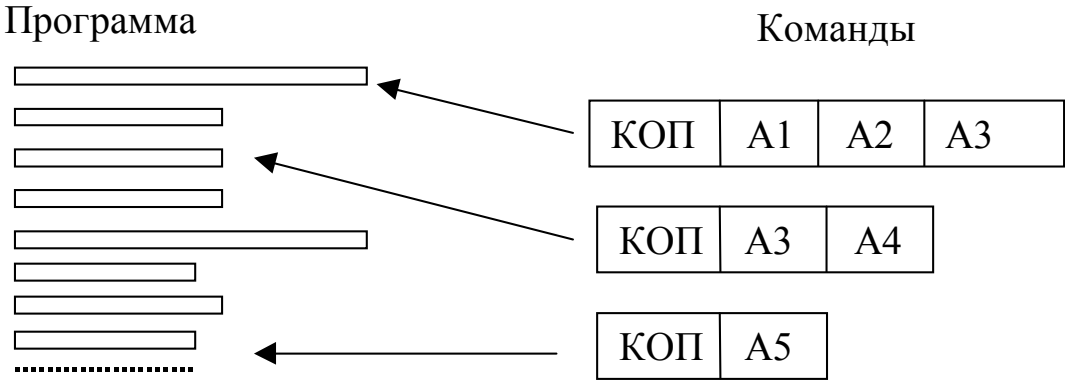
$$2=1\cdot2^1+0\cdot2^0=10_2;\ 5=1\cdot2^2+0\cdot2^1+1\cdot2^0=101_2;\ 256=1\cdot2^8=100000000_2.$$

Заметим, что с увеличением числа количество разрядов для его представления в двоичной системе резко возрастает. Поэтому для размещения большого числа выделяется несколько подряд расположенных однобайтных ячеек. В этом случае адресом такой расширенной ячейки является адрес первого байта. Один бит такой ячейки выделяется под знак числа. Числа, размещенные таким естественным образом, называются ***числами с фиксированной запятой***, или просто ***целыми***.

Для хранения дробных чисел, или слишком больших, их предварительно приводят к нормализованному виду. Например, $-35{,}6 = -0.356\cdot10^{+2}$, где – -356 – мантисса, +2 – порядок. После этого переводят порядок и мантиссу в двоичную систему. Такое число запоминается в комбинированной ячейке, один байт которой содержит порядок, несколько других содержат мантиссу. Числа, размещенные таким образом, называются ***числами с плавающей запятой*** или просто ***действительными***.

Программа – это последовательность ***команд***, которые помещаются в памяти и выполняются процессором в указанном порядке.

Команда размещается в комбинированной ячейке следующим образом. Первый байт содержит код операции (КОП), (например + или – или *), которую необходимо выполнить над содержимым ячеек памяти. В одной, двух или трех ячейках (операндах команды) по 2 или 4 байта содержатся адреса ячеек (А1, А2, А3 над которыми нужно выполнить указанную операцию. Номер первого байта команды называется ее адресом. Последовательность из этих команд называется ***программой в машинных кодах***.



Составление программ в современных компьютерах автоматизировано. Программист пишет программу на специальном ***языке высокого уровня***, т.е. наиболее удобном для записи алгоритма решения определенного класса задач. С помощью символов клавиатуры вводит ее текст в память компьютера. Далее происходит перевод алгоритма в команды машины (***трансляция***) подключение необходимых стандартных программ (***компоновка***) и лишь после этого ***выполнение***. Заметим, что полученная программа в кодах машины может быть записана на диск и для ее многоразового выполнения уже не требуется этапов трансляции и компоновки.

1.4.Файловая система хранения информации

Для размещения программ и всевозможных наборов данных на различных устройствах компьютера и последующей работы с этими данными была разработана концепция файлов.

Под ***файлом*** понимается поименованное место на некотором устройстве ПЭВМ (память, диск, принтер, сканер…), отведенное для помещения и (или) чтения некоторой информации. При этом файл может быть пуст, т.е. место отведено, поименовано, но какая-либо другая информация отсутствует. Информация, помещенная в файл, приобретает имя этого файла. Поэтому файлом часто называют эту информацию.

За работу с файлами в компьютере отвечают специальные программы, набор которых называется ***файловой системой***, основные функции которой заключаются в том, чтобы предоставить программисту удобные средства для работы с данными на всевозможных носителях.

Имя, которое присваивается файлу, может иметь тип. Имя и тип разделяются точкой. При отсутствии типа, точка необязательна. Например, имена могут быть такими:

Prog.pas, prog.dat, prog.out, prog, A, statya.doc

В компьютере обычно хранится несколько тысяч имен файлов. Для их более удобного размещения введены каталоги.

Каталог – это группа файлов на одном носителе. Каталог имеет свое имя. Он может быть вложен внутрь другого каталога. В этом случае он является ***подкаталогом.*** Такая вложенность может быть многократной и образуется иерархическая (древовидная) структура хранения данных.

Для удобства хранения внешним носителям присваиваются имена. Для дисков, например, имена обозначаются одной буквой a:, b:, c:,…. При этом на одном винчестере для удобства размещения файлов может быть организовано несколько логических дисков с разными именами.

Маршрут (путь) файла. При сложной структуре хранения файлов может возникать такая ситуация, когда имеются разные файлы с одинаковым именем, но расположенные в разных каталогах или дисках. Для точной идентификации (указания) файла необходимо кроме имени указать на каком диске, и в какой цепочке подкаталогов он находится. Такая цепочка и называется путем к файлу.

Примеры имен с указанием пути:

C:/sin/doc/lec.doc
D:/delphi/prog.pas

Для работы с файлами обычно используют специальные программы, наибольшее распространение получили ***Total Commander***, ***FAR***, ***Проводник***.

1.5.Операционная система

Вся работа компьютера осуществляется под управлением большого набора специальных программ называемых операционной системой (ОС). С точки зрения пользователя ОС представляет широкий набор системных команд, задавая которые, он может потребовать от ЭВМ выполнения многих полезных для него процедур и действий.

ОС поддерживает целый спектр языков программирования. Файловая система является одной из составных частей ОС.

Часть программ ОС предназначена для управления процессом прохождения задач. Имеется группа программ, так называемого администратора системы, позволяющая следить за работой группы пользователей в рамках системы. В последних системах важное место занимает блок программ, обеспечивающих обмен сообщениями, между пользователями сети, в том числе через интернет.

Удобства, предоставляемые пользователю, существенно зависят от качества ОС, которые по мере совершенствования компьютеров постоянно развиваются. В настоящее время наибольшее распространение на ПЭВМ получили ОС WINDOWS 98, WINDOWS 2000, WINDOWS XP, обеспечивающие визуальный доступ (с помощью мыши) пользователя к набору своих программ.

ЛЕКЦИЯ 2. КАК СОСТАВЛЯЮТСЯ И ВЫПОЛНЯЮТСЯ ПРОГРАММЫ В СИСТЕМЕ DELPHI

2.1. Понятие алгоритма и способы его записи

Фундаментальным понятием при программировании является алгоритм. Написанию программы предшествует разработка алгоритма решения задачи.

Алгоритм – это последовательность действий, в результате выполнения которых получается решение поставленной задачи.

Алгоритмы можно классифицировать по виду выполняемых действий. Например, правила ГАИ – это набор алгоритмов описывающих действия водителей и пешеходов, поваренная книга – набор алгоритмов приготовления пищи.

Назовем ***вычислительными*** – алгоритмы, последовательность действий которых «умеет» выполнять компьютер.

Программа представляет запись вычислительного алгоритма на языке, который умеет читать компьютер.

Мы уже знаем, что основными действиями, которые умеет выполнять компьютер, являются операции над содержимым ячеек памяти. Составление программы в командах машины крайне сложно и неудобно. И после того, как был накоплен опыт применения ЭВМ для решения различных классов задач, было осознано, что компьютер – это универсальный инструмент, который может выполнять любую формализованную работу по переработке информации. Именно такой задачей является перевод программы с одного формального языка на другой. Возникли первые ***языки*** программирования, которые сейчас принято называть ***языками высокого*** уровня, удобные для записи тех или иных классов алгоритмов.

В языках программирования вместо номеров ячейкам принято давать имена (идентификаторы), а содержимое ячеек называть ***переменными*** или ***константами***, в зависимости от того изменяется оно или нет в процессе работы.

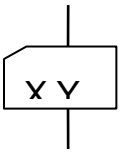
Во всех языках фундаментальным понятием является ***оператор***, который представляет описание определенного набора действий ЭВМ. Программа, написанная на языке программирования, состоит из последовательности операторов.

Одним из распространенных операторов является ***оператор присваивания*** ячейке памяти результата арифметических операций над содержимым других ячеек. Последовательность таких операций записывается, например, следующим образом:

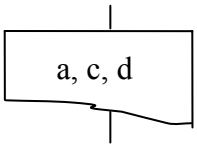
```
b:=0; c:=1; f:=5;
a:=b+c-f;
b:=b+c;
```

Здесь := означает присвоить; a, b, c, f – имена ячеек (переменных); каждый оператор в Паскале заканчивается точкой с запятой.

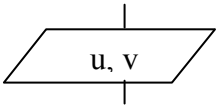
Во всех языках программирования имеется набор операторов, реализующих следующие основные действия:



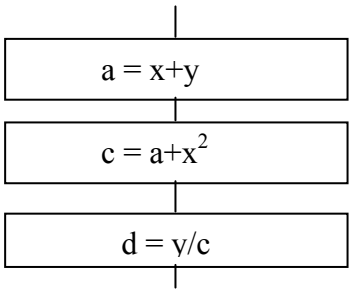
Ввод данных с внешнего носителя



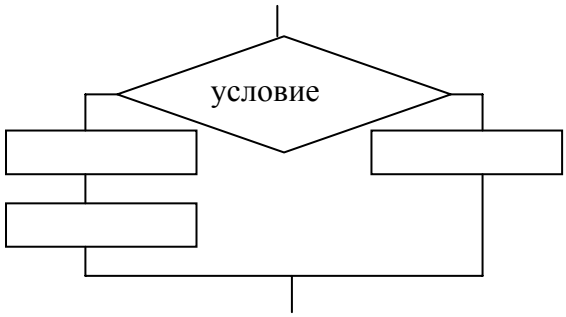
Вывод результатов на внешний носитель



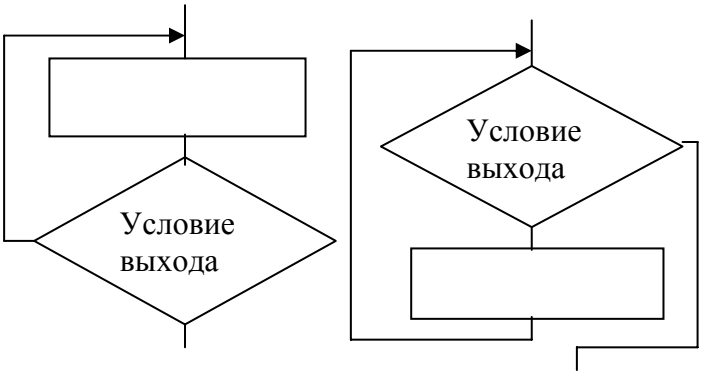
Обмен информацией между программами



Линейный вычислительный процесс



Разветвление алгоритма в зависимости от условия



Циклы, т.е. организация повторений некоторой последовательности операторов

2.2. Общая характеристика языка Pascal

В 1970 году появилось сообщение еще об одном языке программирования, названного *Pascal*, в честь знаменитого французского математика Блеза Паскаля. Автор языка Никлаус Вирт, профессор, директор института информатики Швейцарской высшей политехнической школы. Этот язык создавался для целей начального обучения студентов языкам программирования. К тому времени были в распространении такие известные языки как *Бейсик, Фортран, Алгол, СИ.* Уже проявились как достоинства, так и недостатки этих языков. Поэтому концепция нового языка оказалась настолько удачной, что он быстро завоевал прочные позиции в мире программирования. Как правило, практически во всех учебных заведениях мира изучение языков программирования начинают с Pascal.

Pascal – язык программирования высокого уровня, позволяющий значительно облегчить подготовку программ. Его синтаксис максимально приближен к обычному математическому языку.

В концепции Паскаля заложены следующие основные понятия:

а) ***принцип структурирования***, который позволяет автоматизировать проверку правильности программ. Грубо говоря, это значит, что возможности языка позволяют записать алгоритм любой сложности без использования меток и операторов *GOTO*. Оператор *GOTO* в Pascal оставлен для программистов низкого уровня;

б) ***принцип надежности***. В отличие от всех других языков, в Pascal все переменные, которые используются, должны быть описаны в заголовке программы, т.е. указаны их типы, границы изменения;

в) ***простота реализации транслятора*** позволяет его адаптацию практически на любом типе компьютера и высокую скорость компиляции.

г) ***развитая структура данных*** делает этот язык универсальным языком многоцелевого назначения, позволяющим реализовать как обычные программы, так и современные технологии объектно-ориентированного программирования.

В процессе совершенствования вычислительной техники Pascal прошел ряд этапов в своем развитии. Начиная с простейшего языка обучения, имелось очень большое число версий наиболее известные из которых: *Microsoft Pascal Compiler, Quick Pascal, Pascal-2, Professional Pascal, USCD Pascal,* одна из самых популярных версий - *Pascal-5.5, Pascal-7, Object Pascal.*

В 1996 году появилась современная высокопроизводительная система визуального программирования Delphi на основе языка *Object Pascal*. По своим возможностям она не уступает системе *VisualC++* (которая появилась немного позже Delphi), а по удобству языка и простоте обучения Delphi значительно привлекательнее.

Система Delphi реализует ***технологию объектно-ориентированного визуального программирования*** (**ООП**). В ООП главным элементом является

объект. Объектно-ориентированная программа – это совокупность объектов и способов их взаимодействия. Пользователь программы является главным (управляющим) объектом. Обмен между объектами происходит посредством сообщений. Пользователь посылает свои сообщения посредством нажатия кнопок на панели программы. Когда сообщение поступило некоторому объекту (произошло определенное событие), он начинает выполнять свои определенные действия и сообщает результаты пользователю визуально через окна на панели программы.

Объект представляет собой объединение под одним именем (имя объекта) данных и методов работы с ними. Методы чтения и записи данных называются *свойствами*. Вызов метода производится указанием имени объекта и через точку имени метода, например, *Memo.Clear*.

2.3. Как составляется программа в системе Delphi

Среда разработки программ Delphi визуально реализуется в виде нескольких одновременно раскрытых на экране монитора окон. Количество, расположение, размер и вид окон может меняться программистом в зависимости от его текущих нужд, что значительно повышает производительность работы. При запуске Delphi вы можете увидеть на экране картинку, подобную представленной на рис. 2.1.

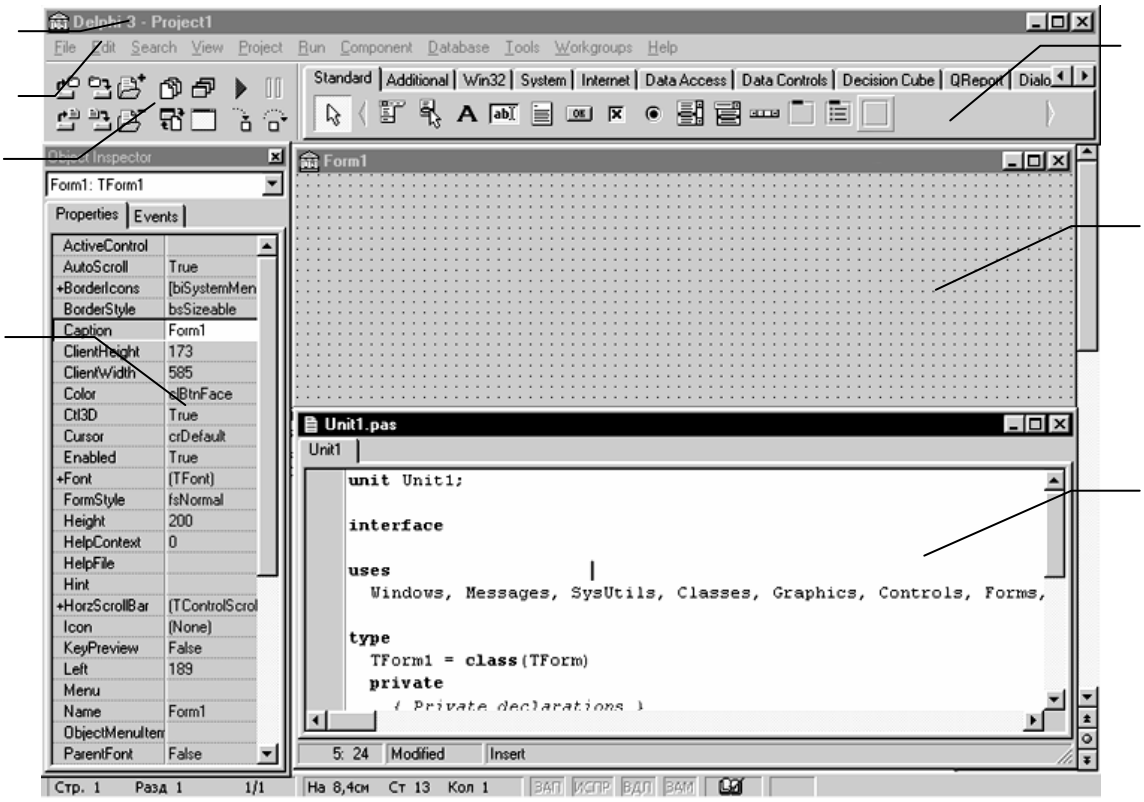


Рис.2.1.

1- главное окно; 2 – основное меню, 3 – пиктограммы основного меню, 4 - окно инспектора объектов; 5 – окно текста программы, 6- окно пустой формы; 7 – меню компонентов.

Главное окно всегда присутствует на экране и предназначено для управления процессом создания программы. Основное меню содержит все необходимые средства для управления проектом. Пиктограммы облегчают доступ к наиболее часто применяемым командам основного меню. Через меню компонентов осуществляется доступ к набору стандартных сервисных программ среды Delphi, которые описывают некоторый визуальный элемент (компонент), помещаемый программистом в окно формы. Каждый компонент имеет определенный набор свойств (параметров), которые программист может задавать. Например, цвет, заголовок окна, надпись на кнопке, размер и тип шрифта и др.

Окно инспектора объектов (вызывается с помощью клавиши *F11*) предназначено для изменения свойств выбранных компонентов и состоит из двух страниц. Страница ***Properties*** (Свойства) предназначена для изменения необходимых свойств компонента. Страница ***Events*** (События) – для определения реакции компонента на то или иное событие (например, нажатие определенной клавиши или щелчок «мышью» по кнопке).

Окно формы представляет собой проект рабочей панели *Windows*-окна программы. В это окно в процессе написания программы помещаются необ-

ходимые компоненты. Причем при выполнении программы помещенные компоненты будут иметь тот же вид, что и на этапе проектирования.

Окно текста программы предназначено для просмотра, написания и редактирования текста программы. В системе Delphi используется язык программирования *Object Pascal.* При первоначальной загрузке в окне текста программы находится исходный текст программного модуля (*Unit*), содержащий минимальный набор операторов для нормального функционирования пустой формы в качестве *Windows*-окна. При помещении некоторого компонента в окно формы, текст программы автоматически дополняется описанием необходимых для его работы библиотек стандартных программ (раздел uses) и типов переменных (раздел *type*).

Переключение между окном формы и окном текста программы осуществляется с помощью клавиши *F12.*

Программа в Delphi состоит из файла проекта (файл с расширением *.dpr*), одного или нескольких файлов исходного текста (с расширением *.pas*), файлов с описанием окон формы (с расширением *.dfm*).

В ***файле проекта*** находится информация о модулях, составляющих данный проект. ***Файл проекта автоматически создается и редактируется средой Delphi и не предназначен для редактирования.***

Файл исходного текста – программный модуль *(Unit)* предназначен для размещения текстов программ. В этом файле программист размещает текст программы, написанный на языке Pascal. В разделе объявлений описываются типы, переменные, заголовки процедур и функций, которые могут быть использованы другими модулями, через операторы подключения библиотек *(Uses)*. В разделе реализации располагаются тела процедур и функций, описанных в разделе объявлений, а также типы переменных, процедуры и функции, которые будут функционировать только в пределах данного модуля. Раздел инициализации используется редко и его можно пропустить.

Модуль имеет следующую структуру:

```
unit Unit1;
                    interface
                // Раздел объявлений
        procedure…

                 implementation
        procedure…
        begin
        . . .   //Раздел реализации
        end;
        . . .
begin
...     //Раздел инициализации
end.
```

Программа в среде Delphi составляется как описание алгоритмов, которые необходимо выполнить, если возникает определённое событие, связанное с формой (например щелчок «мышки»—событие, называемое *OnClick,* создание объекта—*OnCreate*). Для каждого обрабатываемого в форме события с помощью страницы *Events* инспектора объектов (или двойного щелчка мыши), в разделе реализации организуется процедура (*Procedure*), между ключевыми словами *begin* и *end* которой программист записывает на языке *Object Pascal* требуемый алгоритм.

Детальное описание действий при создании *Windows*-окна программы, написании и отладке текстов процедур, а также её выполнении содержится в лабораторной работе №1.

При компиляции программы Delphi создает файл с расширением *.dcu*, содержащий в себе результат перевода в машинные коды содержимого файлов с расширением *.pas* и *.dfm*. Компоновщик преобразует файлы с расширением *.dcu* в единый загружаемый файл с расширением *.exe*. В файлах, имеющих расширение *.~df, .~dp, .~pa* хранятся резервные копии файлов с образом формы, проекта, и исходного текста соответственно.

2.4. Наша первая программа реализует линейный алгоритм

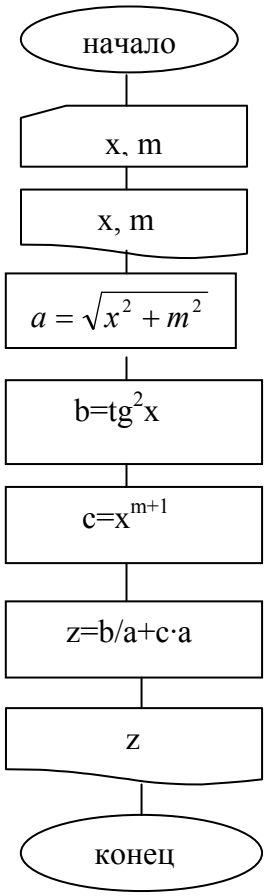
Наиболее часто в практике программирования требуется организовать расчет некоторого арифметического выражения при различных исходных данных. Например, такого

$z = \frac{tg^2 x}{\sqrt{x^2 + m^2}} + x^{(m+1)}\sqrt{x^2 + m^2}$, при x>0 - действительное, m - целое.

Разработка алгоритма обычно начинается с написания схемы. Продумывается оптимальная стратегия вычислений, при которой, например, отсутствуют повторения. При написании алгоритма рекомендуется также переменным присваивать по возможности те же имена, которые фигурируют в заданном арифметическом выражении. Для того, чтобы не было «длинных» операторов исходное выражение полезно разбить на ряд более простых. В нашем случае предлагается схема вычислений, представленная на рис. 2.2. Она содержит ввод и вывод исходных данных, линейный вычислительный процесс, вывод полученного результата. Заметим, что выражение $\sqrt{x^2 + m^2}$ вычисляется только один раз. Введя переменные *a,b,c* мы разбили сложное выражение на ряд простых.

При разработке программы, реализующей этот алгоритм, кроме уже известного нам оператора присваивания необходимо знать, как записать операторы ввода и вывода данных. В языке Pascal эти операторы выглядят таким образом:

```
Read(Lr,x,m);
Writeln(Lw,'x=',x:6:2,'   m=',m:2);
Writeln(Lw,'z=',z:8:3);
```



Здесь *Lr* - логическое имя файла, на котором помещены исходные значения *x,y, Lw* - логическое имя файла, на который помещается результат. Набор символов, заключенных в апострофы (например, 'x='), называется *строковой константой.* При выводе строковая константа записывается в файл без изменений. Запись *z:8:3* означает, что, например, действительное число - $0.254 \cdot 10^2$, содержащееся в ячейке *z*, будет преобразовано и 8 позиций будет отведено под всё число, а 3 позиции под его дробную часть. Таким образом, результат выполнения последнего оператора будет *z= -25.400.* Заметим, что при такой организации ввода, исходные данные обычно предварительно помещают в некоторый *текстовый файл*, представляющий последовательность из строк символов. Результат также выводят в текстовый файл. Такие файлы можно готовить, обрабатывать и печатать с помощью различных текстовых редакторов, в частности в окне текста Delphi, используя меню *File.*

Рис.2.2.

В Delphi чаще всего ввод (вывод) небольшого количества данных производят из так называемых окон однострочных редакторов (компонент *Edit*) или табличных строчных редакторов (компонент *StringGrid*).

Вывод большого числа данных обычно осуществляется в окно многострочного редактора (компонент *Memo*). Окна этих редакторов и пояснения к ним размещаются на *Window*-окно программы по усмотрению программиста. В окна строчных редакторов во время активности программы может помещаться любой набор символов, в частности запись чисел, которые необходимо ввести. Каждому окну редакторов *Edit* и *StringGrid*, помещенному на форму в программе в разделе *Type* вводится ячейка памяти типа *String*, в которой содержится строка символов, отражаемая в окне.

Для того, чтобы, например, в окне редактора *Edit1* было помещено число *28.64* в программе достаточно написать *Edit1.Text:='28,64'*. И обратно, если во время активности программы в окно *TEdit1* с помощью клавиатуры, будет помещено число *3,14*, то оно окажется в ячейке *Edit1.Text* (вместо предыдущего *28.64*).

Для реализации нашей программы на форму поместим два окна *Edit1* (под *x*), *Edit2* (под *m*). Ввиду того, что ввод данных происходит в виде строки, а переменные *x, m* представляют числа необходимо воспользоваться функ-

циями перевода строковой записи числа в действительное или целое его представление:

x:=StrToFloat(Edit1.Text);

m:=StrToInt(Edit2.Text);

Для вывода результатов поместим на форму окно многострочного редактора *Memo1*. Окно многострочного редактора (компонент *Memo*) в данном случае предназначено для вывода и представляет собой отображение в окне последовательности строк. Для того, чтобы отобразить новую строку, ее необходимо добавить в окно с помощью оператора *Memo1.Lines.Add('строка')*. Если нужно вывести содержимое действительных или целых переменных их необходимо предварительно преобразовать в строковое представление. Для этого используются функции

FloatToStrF(x,fffixed,6,2); // 6 позиций под все число, 2 десятичных знака

IntToStr(m); // все цифры числа m

Теперь мы готовы написать нашу первую программу. Для этого поместим на форму кнопку, описываемую компонентом *Button1* и сделаем на ней двойной щелчок «мышью». После этого в открывшемся окне текстов наберем следующую программу:

```
Procedure TForm1.Button1Click(Sender:Tobject);
// Этот заголовок среда Delphi вставляет автоматически после
//двойного щелчка «мышью» по кнопке Button1
    Var   x,z,a,b,c:extended; // описание типов
          m:integer;          //  всех используемых переменных
    Begin
    x:=StrToFloat(Edit1.Text);
    m:=StrToInt(Edit2.Text);
    Memo1.Lines.Add('x='+FloatTostrF(x,fffixed,6,2) +'  m='+IntTostr(m));
    a:=sqrt(sqr(x)+m*m);
    b:=sqr(sin(x)/cos(x));
    c:=exp((m+1)*ln(x));
    z:=b/a+c*a;
    Memo1.Lines.Add('полученный результат');
    Memo1.Lines.Add('z='+FloatTostrF(z,fffixed,8,3));
    End.
```

Заметим, что в строке после сдвоенного // пишется произвольный комментарий, который служит для облегчения понимания программы.

После набора программы ее необходимо записать в свою папку. Для создания новой папки можно воспользоваться одной из программ для работы

с файлами. Например, войдем в программу ***Total Commander***, выберем рабочий диск, нажмем клавишу ***F7*** и в появившемся окне наберем имя папки *«Lab1_5»* (Лаб.№1, вариант 5). В меню *«File»* Delphi выберем *«Save Project As»*. После открытия своей папки нажмем *«Save Project As»* и сохраним с указанием имени предлагаемые файлы *Project1* и *Unit1*. После сохранения сделаем запуск программы, для чего в меню *«Run»* выберем команду *«Run»*. Если все сделано верно, после компиляции на экране высветится *Window* - окно вашей первой программы, готовой к выполнению расчетов. В окнах *Edit1, Edit2* наберите исходные данные (*x*, *m)*, после чего щелкните мышью на кнопке *Button1* . В окне *Memo1* появятся результаты расчета вашей программы.

При обнаружении ошибок, в процессе выполнения, переведите программу из ***режима выполнения Running*** в ***режим редактирования***, для чего выберите *«Run Reset»* в меню *«Run»*. Исправьте ошибки, запишите исправленный вариант, нажав *«Save»* в меню *«File»* и запустите программу снова.

Для получения распечатки программы в меню *«File»* выберите *«Print»*, установите запрашиваемые параметры и нажмите *«OK»*.

ЛЕКЦИЯ 3. БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА ПАСКАЛЬ

3.1. Данные и их типы

Любой язык начинается с алфавита. ***Алфавит*** *Object Pascal* оперирует следующим набором символов:

1) прописные и строчные буквы латинского алфавита

A, B, C, D,...,U, V, W, X, Y, Z;

a, b, c, d,...,u, v, w, x, y, z;

2) десятичные арабские цифры от 0 до 9;

3) _ - символ «подчеркивание»,

4) специальные символы

+	плюс	{}	фиг.скобки	:	двоеточие
-	минус	[ ]	кв. скобки	;	точка с запятой
*	звездочка	( )	круг.скобки	'	апостроф
/	дробная черта	#	номер	@	коммерческое а
=	равно		пробел	$	знак ден.единицы
>	больше	.	точка	^	каре
<	меньше	,	запятая		

Комбинации специальных символов могут образовывать составные символы:

:=	присваивание	<=	меньше или равно
..	диапазон значений	>=	больше или равно
		< >	не равно

(. .) альтернатива квадратных скобок

(* *) альтернатива фигурных скобок

5) ключевые (зарезервированные) слова. Например: *Begin, End;*

6) стандартные идентификаторы. Например: *Sin, Cos;*

7) идентификаторы пользователя.

Величины, с которыми должна работать программа принято называть ***данными***. Все данные при работе программы размещаются в различные ячейки памяти, им присваиваются оригинальные имена (идентификаторы) и указываются их типы.

Идентификаторы образуются из латинских букв, символа _ и арабских цифр и могут иметь произвольную длину, но значимыми будут только первые 63 символа. Первым символом должна быть буква или символ _. Пробелы, точки и другие специальные символы не могут входить в имя.

Данные могут быть ***константами*** и ***переменными.***

Константы - это те данные, значения которых известны заранее и в процессе выполнения программы не изменяются.

В качестве констант в *Object Pascal* могут использоваться целые, вещественные и шестнадцатеричные числа, логические константы, символы, строки символов, конструкторы множеств и признак неопределенного указателя nil.

Целые константы - это целые числа со знаком или без в диапазоне от -2^{63} до $2^{63}-1$.

Пример:

286; -17; +1995

Можно использовать целую константу в *шестнадцатеричном* виде. Шестнадцатеричное число состоит из шестнадцатеричных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), которым предшествует знак доллара $ (код символа 36)

Пример:

$3A4F; $100A

Вещественные константы могут быть представлены в двух видах: с фиксированной и плавающей точкой.

Константа с фиксированной точкой - это число, содержащее точку, разделяющую целую и дробную часть (наличие целой и дробной части обязательно).

Пример:

-39.013; 0.256

Константа с плавающей точкой - это число, представленное с десятичным порядком: *mEp* (без пробелов).

Здесь *m* - мантисса (как целые, так и вещественные числа с фиксированной точкой), *E* - признак записи числа с десятичным порядком, *p* - порядок числа (только целые числа).

Пример:

-7.78E-3; -0.785E02; 4.9E5

Логические константы могут принимать только одно из двух значений: *True* (истина) и *False* (ложь).

Символьная константа - это любой символ, заключенный в апострофы. Допускается использование записи символа путем указания его внутреннего кода, которому предшествует символ #.

Пример:

‘w’; ‘!’; #53

Строковые константы - это последовательность символов, заключенная в апострофы.

Пример:

'БГУИР'; #75#58#93

Конструктор множества – это список элементов множества, обрамленный квадратными скобками.

Пример:

[43, 76, 12, 90]; [red, blue, green]

Константы могут иметь свои собственные имена. Этот вариант предпочтительней при многократном использовании константы.

Форма описания констант:

Const <идентификатор> = <значение константы>;

Пример:

```
Const   Min = 1;          // Описание констант
        Max = 150;
        A='Таблица';
```

Переменная - это именованный объект, который в процессе выполнения программы может принимать различные значения. Форма описания переменных:

Var <идентификатор 1, ... > : тип;

Пример:

```
Var
   A, B: Integer:
   Sum, Min : Real;
   C: Char;
```

Тип переменных определяет, какие значения они могут иметь, какая структура ячеек для их размещения используется, и какие операции над ними можно выполнять. В *Object Pascal* имеется развитая система типов данных. Их можно разделить на две группы: скалярные (простые) и структурированные (составные).

К ***скалярному типу*** относятся данные, представляемые одним значением (числом, символом) и размещаемые в одной ячейке из нескольких байтов.

Структурированные типы определяются пользователем через скалярные и описанные ранее структурированные с помощью оператора *Type* следующим образом

```
Type <Tun1> = <определение типа 1>;
     <Tun2> = <определение типа 2>;
```

В языке *Object Pascal* имеется 7 скалярных типов (целые, вещественные, логические, символьные, перечисляемые, интервальные, дата-время) и 10 основных структурированных типов (строки, массивы, множества, записи, файлы, указатели, процедуры, объекты, варианты, классы). В Pascal предусмотрено создания программистом новых типов данных, поэтому общее количество типов, использующихся в программе, может быть сколь угодно большим. По мере изучения языка мы будем детально знакомиться с различными типами.

Характеристики двух основных скалярных типов представлены в следующих таблицах.

Целые типы. Табл.3.1.

Unsigned (Беззнаковые)			Signed (Знаковые)		
Тип	Объем	Диапазон	Тип	Объем	Диапазон
Byte	1	0 .. 255	ShortInt	1	-128 .. 127
Word	2	0 .. 65 535	SmallInt	2	-32 768 .. 32 767
Long, Cardinal	4	0 .. 4 294 967 295	Integer, LongInt	4	-2 147 483 648 … 2 147 483 647
			Int64	8	-2^{63} .. $2^{63}-1$

Вещественные типы Табл.3.2.

Тип	Размер	Значащих цифр	Диапазон
Single	4	7..8	$1,5*10^{-45}$ .. $3,4*10^{38}$
Real, Double	8	15 .. 16	$5*10^{-324}$ .. $1,7*10^{308}$
Extended	10	19 .. 20	$3,4*10^{-4951}$ .. $1,1*10^{4932}$
Comp	8	19 .. 20	-2^{63} .. $2^{63}-1$
Currency	8	19 .. 20	±922 337 203 685 477, 5807

Для экономии памяти следует пользоваться более «короткими» типами, а для повышения точности более «длинными». Следует помнить, что арифметический сопроцессор всегда обрабатывает числа в формате *Extended*, а три других вещественных типа (*Double, Real, Single*) получаются простым усечением результата до нужных размеров и применяются, как правило, для экономии памяти. Следует так же учесть что тип *Real* оптимизирован для работы без сопроцессоров, однако его использование на ПК с сопроцессором приводит к дополнительным затратам (в 2-5 раз) времени на преобразование формата. Если ваш компьютер оснащен сопроцессором, то не рекомендуется использовать тип *Real*.

Из логических типов наиболее часто используется тип *Boolean* (1 байт), другие логические типы *ByteBool* (1 байт), *Bool* (2 байта), *WordBool* (2 байта) и *LongBool* (4 байта) введены для совместимости с *Windows*.

Пользовательские типы переменных

К ним относятся переменные перечисляемого и интервального типов. Переменная типа *перечисление* задается перечислением значений, которые она может принимать.

Форма описания этих переменных:

Type <имя типа> = (список значений);
Var <идентификатор 1,...> : <имя типа>;

или

Var <идентификатор> : (список значений);

Пример:

Type Sezon = (Zima, Vesna, Leto, Osen);

Var S1, S2 : Sezon;

или

Var S1, S2 : (Zima, Vesna, Leto, Osen);

Здесь *S1,S2* - переменные типа перечисление, которые могут принимать любое из заданных значений.

Следует отметить, что описание типа перечисляемой переменной одновременно вводит упорядочение ее значений. Так, для данного примера

Zima < Vesna < Leto < Osen (в операциях сравнения).

Для переменных *интервального* типа указывается некоторое подмножество значений, которые они могут принимать.

Форма описания этих переменных:

Type <имя типа> = <константа 1>..<константа 2>;
Var <идентификатор 1, ... > : <имя типа>;

или

Var <идентификатор 1, ... > : <константа 1> .. <константа 2>;

Здесь *<константа 1>, <константа 2>* - соответственно константы, определяющие левую и правую границы значений, которые может принимать интервальная переменная. Значение первой константы должно быть обязательно меньше значения второй. Эти константы могут быть целого, символьного или перечисляемого типов.

Пример:

```
Type Dni = 1..31;
Var D1, D2 : Dni;
```

В этом примере переменные *D1* и *D2* имеют тип *Dni* и могут принимать любые значения из диапазона *1..31*. Выход из диапазона вызывает программное прерывание.

Можно определять интервальный тип и более универсальным способом, задав границы диапазона не значениями констант, а их именами.

Пример:

```
Const  Min=1; Max=31;
Type  Dni = Min..Max;
Var  D1, D2 : Dni;
```

3.2. Операции над переменными основных скалярных типов

Во-первых, следует всегда помнить, что в операторе присваивания

<имя переменной>:=<выражение>;

результат выражения (арифметического, логического) должен соответствовать типу переменной a, т.е. здесь нельзя смешивать типы. Исключение составляет возможность присваивать переменной вещественного типа результат целого типа.

Арифметические выражения

Арифметические выражения строятся из числовых констант, переменных, стандартных функций и операций над ними. Для обозначения операций используются символы: + сложение, - вычитание, * умножение, и / деление.

В арифметическом выражении принят следующий приоритет операций:

1) вычисление значений стандартных функций;
2) умножение и деление;
3) сложение и вычитание.

Порядок выполнения операций изменяется с помощью скобок.

Стандартные функции

Таблица 3.3.

Вызов функции	Тип аргумента	Тип результата	Назначение функции
Abs(x)	Целый/веществ	Тип аргум.	Абсолютное значение x
Pi		Веществ.	Значение числа π
Sin(x)	Целый/веществ	Веществ.	Синус x
Cos(x)	Целый/веществ	Веществ.	Косинус x
Arctan(x)	Целый/веществ	Веществ.	Арктангенс x
Sqrt(x)	Целый/веществ	Веществ.	Квадрат. корень из x, x>0
Sqr(x)	Целый/веществ	Тип аргум.	Значение квадрата x
Exp(x)	Целый/веществ	Веществ.	Значение показ. функ e^x
Ln(x)	Целый/веществ	Веществ.	Натур. логарифм x, x>0
Trunc(x)	Вещественный	LongInt	Целая часть значения x
Frac(x)	Вещественный	Веществ.	Дробная часть значения x
Int(x)	Вещественный	Веществ.	Целая часть значения x
Round(x)	Вещественный	LongInt	“Правильное” округление x до ближайшего целого
Random		Веществ.	Сл. число из диап.0<=…<1
Random(x)	Word	Word	Сл. число из диап.0<=…<x
Odd(x)	Целый	Логический	Возвращает True, если x–нечет. (x–целое)
Succ(x)	Целый	Целый	Возвр. след. за x значение в перечисляемом типе
Pred(x)	Целый	Целый	Возвр. предыдущее знач. x в перечисляемом типе
Chr(x)	Целый(Byte)	Символьный	Возвр. симв. ASCII кода x
Ord(x)	Символьный	Целый(Byte)	Возвр.ASCII код символа x

Inc(x) Dec(x) Inc(x, n) Dec(x, n)	Целый Целый Целый Целый	Целый Целый Целый Целый	Увеличивает знач. x на 1 Уменьшает знач. x на 1 Увеличивает знач. x на N Уменьшает знач x на N
A Div B A Mod B	Целочисленное деление A на B. Возвращает целую часть частного, дробная часть отбрасывается Восстанавливает остаток, полученный при выполнении целочисленного деления. A и B должны быть целого типа		

Отметим, что в тригонометрических функциях аргумент должен быть задан только в радианах.

Как видно из таблицы 3.3 в Pascal имеются стандартные функции для вычисления только трех тригонометрических функций. Для вычисления остальных необходимо использовать известные математические соотношения:

$$tg(x) = sin(x)/\cos(x);$$

$$arcsin(x) = arctg(\sqrt{x/(1-x^2)});$$

$$\arccos(x) = \pi/2 - arcsin(x);$$

$$arcctg(x) = \pi/2 - arctg(x);$$

Для вычисления логарифма с основанием a используется соотношение

$$\log_a(x) = ln(x)/ln(a);$$

Так как в Pascal отсутствует операция возведения в степень, то для вычисления выражения x^y используется известное математическое соотношение:

$$x^y = e^{y\cdot\ln x} = exp(y\cdot ln(x));$$

Но таким образом нельзя возвести в целую степень отрицательное число. Это можно сделать с использованием операторов цикла.

Над переменными ***целого типа*** определенными, например, как:

Var m, n, i, k : integer;

наряду с операцией присваивания возможны следующие целочисленные арифметические операции и логические отношения:

Табл.3.4.

Код операции	операция	пример	Логические отношения
+	Сложить	m+n	m>n
-	Отнять	m-n	m<n
*	Умножить	m*n	m=n
Div	Разделить	m div n	m>=n
Mod	Остаток от деления	m mod n	m<=n
			m<>n

а также функции и процедуры:

Табл.3.5.

Функции		процедуры	
abs(i)	| i |	Dec(i)	i:=i-1
sqr(i)	i^2	Dec(i,k)	i:=i-k
odd(m)	возвращает True,если m–нечетное и False, если четное	Inc(i)	i:=i+1
chr(n)	возвращает символ по его коду	Inc(i,k)	i:=i+k

При выполнении действий с целыми числами, если операнды имеют различные целые типы, то тип результата будет соответствовать типу, который включает в себя оба операнда. Например, при действиях с *ShortInt* и *Word* общим будет тип *Integer.*

Над переменными *вещественного типа* определенными, например, как:

Var x, y, a, b : extended;

возможны операции (+ - *) и логические отношения, те же, что и для целых. Для деления используется /. Результат операции 5/2, будет действительным, равным 2.5, в отличие от операции *div.* Результат следующих функций является целым числом: *Round(x)* - округление до ближайшего целого, *Trunc(x)* - усечение дробной части.

Логические выражения

Логические выражения строятся из логических констант и переменных, операций отношения и логических операций. В операциях отношения могут участвовать арифметические и логические выражения, а также символьные данные. Результатом логического выражения является значение *True* (истина) или *False* (ложь).

Операции отношения : < (меньше), > (больше), = (равно), <= (меньше или равно), >= (больше или равно), < > (не равно).

Логические операции:

Not --> НЕ - логическое отрицание;
And --> И - логическое умножение;
Or --> ИЛИ - логическое сложение.
Xor --> исключительное ИЛИ.

В логических выражениях операции выполняются слева направо с соблюдением следующего приоритета:

1) Not;
2) *, /, Div, Mod, And, Shr, Shl ;
3) +, -, Or, Xor;
4) =, <>, <, >, <=, >=, in.

К переменным *символьного типа,* определенным, например, как:

Var ch : char;

кроме операции присваивания, применима функция *Ord(ch),* которая возвращает порядковый номер символа в кодовой таблице. Обратная к ней функция *Chr(<порядковый номер символа >)* возвращает символ по его порядковому номеру. Для кодировки в *Windows* используется кодировка *ANSI (American National Standart Institute)*. Функция *Pred(ch)* возвращает символ, предшествующий символу *ch* в кодовой таблице. Функция *Succ(ch)* возвращает символ, следующий за символом *ch* в кодовой таблице .

В виду такой упорядоченности над символами допустимы логические операции отношения, например: *ch > 'a' , 'ф' <> ch, '!' <= ch.*

ЛЕКЦИЯ 4. ПРОГРАМИРОВАНИЕ РАЗВЕТВЛЯЮЩИХСЯ АЛГОРИТМОВ

4.1. Понятие разветвляющегося алгоритма

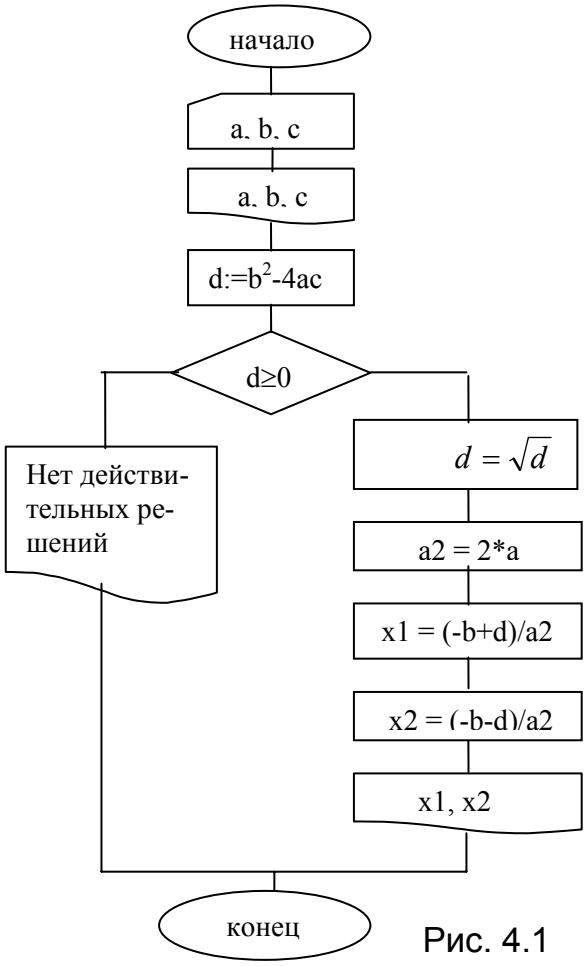


Рис. 4.1

Алгоритм называется разветвляющимся, если он содержит несколько ветвей отличающихся друг от друга содержанием вычислений. Выход вычислительного процесса на ту или иную ветвь алгоритма определяется исходными данными задачи. На рис. 4.1. приведена схема разветвляющегося алгоритма решения квадратного уравнения $ax^2+bx+c=0$:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

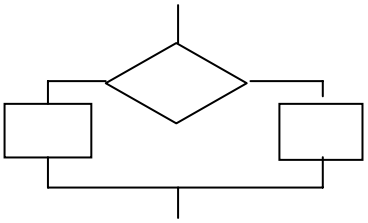
При выполнении алгоритма исследуется подкоренное выражение - дискриминант d уравнения. Печатается сообщение, что нет действительных корней, если $d<0$ и выводится пара корней $x1, x2,$ если $d \geq 0$.

Для программирования разветвлений в языке Pascal имеется два условных оператора ***if*** и ***Case*** **.**

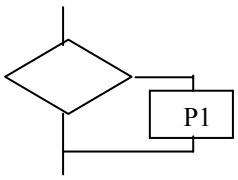
4.2. Оператор условия *if*

Этот оператор служит для разделения естественного порядка выполнения операторов программы на две ветви. Он может иметь одну из следующих форм записи:

if <условие> then P1
else P2;



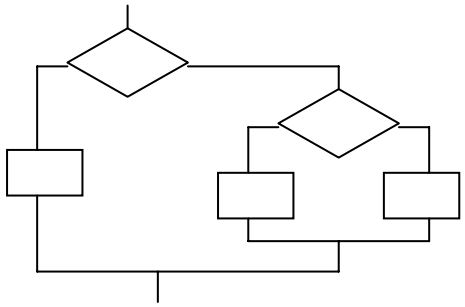
if <условие> then P1 ;



где *<условие>* записывается с помощью выражения логического типа (см. лекцию 3), *P1, P2* - операторы. Если условие выполняется, тогда выполняется оператор *P1,* иначе, в первом варианте, выполняется оператор *P2,* во втором – выполняется оператор следующий за *P1*. Ветвь ***then*** на схеме по умолчанию показывается справа. *Перед ключивым словом* ***else*** *не ставится точка с запятой.*

При вложенности операторов ***if*** каждое ***else*** соответствует тому ***then,*** которое непосредственно ему предшествует, например:

```
if <условие1> then
  if <условие2> then P1
      else P2
         else P3;
```



В условном операторе после ***then*** и ***else*** можно помещать только один оператор. Однако, часто необходимо в какой-то ветви выполнить группу операторов. В Паскале имеется возможность объединить группу операторов в один *составной оператор* с помощью ключевых слов ***begin ... end*** *(операторные скобки).* Такой составной оператор может быть размещен после ***then*** или ***else***.

Теперь мы готовы написать процедуру алгоритма, представленного на рис. 4.1:

```
Procedure <имя события>;
    var a,b,c,d,x1,x2:extended;
begin
  Read(a,b,c);
  Writeln(a,b,c);
  d:=sqr(b)-4*a*c;
  if d>=0 then
          begin
           d:=sqrt(d);
           a2:=2*a;
           x1:=(-b+d)/a2;
           x2:=(-b-d)/a2;
           writeln(x1,x2)
          end
    else
  writeln('нет корней');
  //конец оператора if
end;
```

В *Delphi* имя процедуры обработчика события, соответствует имени события, при возникновении которого она вызывается, например, щелчек мыши на кнопке, см. пп.2.3., 2.4.

Заметим, что здесь при реализации ввода и вывода мы для краткости используем операторы *read* и *writeln.* В Delphi обычно для ввода используют чтение данных из окон компанентов *Edit1, Edit2, Edit3*, расположенных на форме, а вывод осуществляется в окно *Memo1* (см. Лекцию 2).

При составлении сложных разветвляющихся алгоритмов необходимо стремиться, по возможности, минимизировать количество проверок условий. Так, например, для решения задачи о нахождении минимального из трех чисел: *m=min(x,y,z)* можно предложить следующие две схемы:

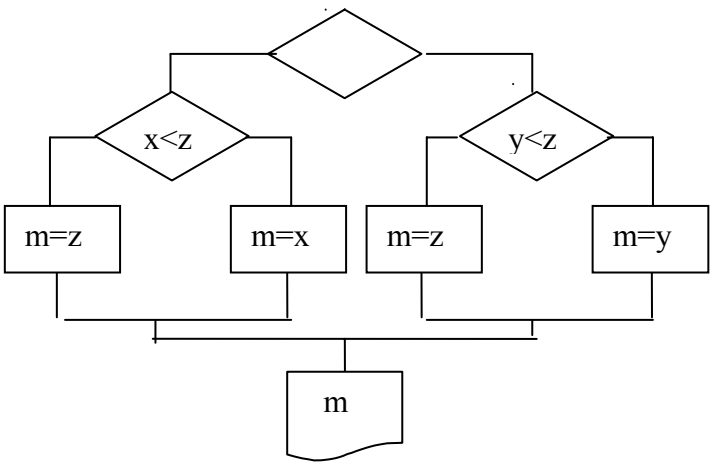


Рис. 4.2.

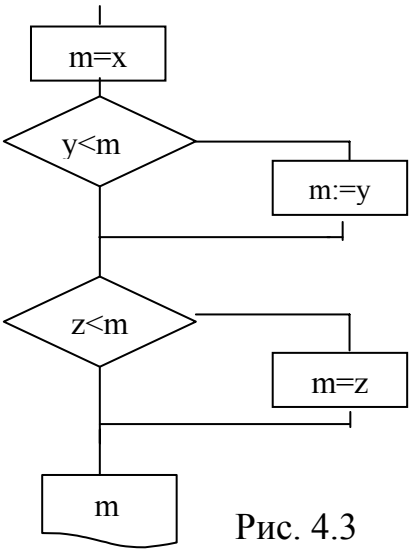


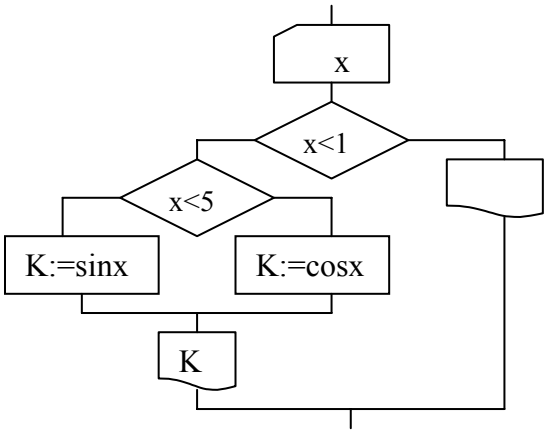
Рис. 4.3

```
. . .
m:=x;
if y<m then m:=y;
if z<m then m:=z;
writeln (m);
. . .
```

Обычно схему рис.4.2 составляют начинающие программисты. Как видно схема рис.4.3 более экономна, ее легко продолжить для 4-х, 5-ти чисел. Фрагмент программы ее реализующей приведен слева:

Приведем еще один поучительный пример составления схемы разветвляющегося алгоритма для решения следующей задачи. Вычислить и напечатать результат выражения:

$$K=\begin{cases}\cos x, & \textit{если}\ \ 1\le x<5;\\ \sin x, & \textit{если}\ \ 5\le x;\end{cases}$$, напечатать *«нет»*, если $x<1$.




```
Read(x);
if x<1 then writeln ('нет')
       else
   begin
   if x<5 then K:=cos(x)
                 else K:=sin(x);
   writeln (K);
   end;
```

Как видим, алгоритм составлен таким образом, что в каждом из двух операторов *if* проверяется только одно неравенство, хотя в условии 4 неравенства.

4.3. Оператор выбора *Case*

Этот оператор служит для разделения естественного порядка выполнения операторов на несколько (возможно более двух) ветвей в зависимости от значения, которое принимает выражение порядкового типа – (селектор). Напомним, что к порядковым типам относятся целые, символьные, логические, перечисляемые, интервальные.

Оператор Case имеет следующий вид:

```
case <селектор> of
<список 1>: P1;
<список 2>: P2;
      . . .
<список n>: Pn;
else Pe;
end;
Pk;
```

<список i> представляет перечисление тех значений переменной *<селектор>* при которых выполняется оператор *Pi.* После его выполнения будет выполняться *Pk.* Если значение селектора не попала ни в один *<список i>*, выполняется *Pe* (Заметим, что строка else может отсутствовать, тогда будет выполнен *Pk*).

С помощью оператора *Case* удобно программировать алгоритмы, имеющие возможность выбора из нескольких вариантов. Например, вычислить значение *y=f(x)* для одной из трех функций, *y=sinx, y=cosx, y=lnx* в зависимости от вводимого номера *Item* варианта *(0,1,2)*. можно так:

```
read (Item);
case Item of
  0:  y:=sin(x);
  1:  y:=cos(x);
  2:  y:=ln(x)
else Writeln('Item<>0,1,2');
end;
```

4.4. Некоторые возможности, предоставляемые Delphi для организации разветвлений

При создании программ в Delphi для организации разветвлений обычно используются компоненты в виде кнопок – переключателей *CheckBox* и *RadioGroup* [1]. Состояние таких кнопок визуально отражается на форме. Кнопка типа C*heckBox*, помещенная на форму, позволяет пользователю с помощью щелчка мышью на ней изменить ее состояние (*включена-выключена*) и соответственно изменить значение связанной с ней переменной *CheckBox.Checked* булевского типа (*true-false*), что может быть использовано в операторе *if* внутри программы, например:

```
if CheckBox.Checked then P1
                    else P2;
```

Группа кнопок типа *RadioGroup,* помещенная на форму, позволяет пользователю организовать селектор, передающий в программу через переменную целого типа *RadioGroup.ItemIndex* номер включенной кнопки *(0,1,2…)* при остальных выключенных. Это может быть полезно использовано с помощью оператора *Case* внутри программы.

ЛЕКЦИЯ 5. СОСТАВЛЕНИЕ И ПРОГРАМИРОВАНИЕ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ

5.1. Понятие цикла

Под циклом понимается организованное повторение некоторой последовательности операторов. Практически все алгоритмы решения задач содержат циклически повторяемые участки. Цикл – это одно из фундаментальных понятий программирования.

Реализовать его в Паскале можно следующим образом

```
Label Me1, Me2;
...
Me1 : P1;
      P2;
       .
      Pn;
  Goto  Me1;
  Me2  :  Pk;
```

Здесь *Me1* – метка, которая описывается в разделе объявлений программы (имена меток могут быть цифрами), оператор *Goto Me1;* передает управление на первый повторяемый оператор, «помеченный» (через двоеточие) меткой. Для того, чтобы приведенный алгоритм не «зацикливался» до бесконечности требуется с помощью одного из повторяемых операторов организовать условие выхода из цикла, например, записав, оператор *Pn* так:

if <условие> Goto Me2;

Такая технология организации циклов была характерна для «старых» языков – Бейсик, Фортран. Однако, очень скоро были выявлены ее недостатки. Особенно неудобно оказалось составлять программы автоматизированной отладки и тестирования. Поэтому при разработке новых языков были введены специальные операторы, позволяющие организовать циклы без использования меток. В языке Pascal имеется три таких оператора: ***Repeat, While, For***, а оператор ***Goto M*** оставлен для тех, кто традиционно привык к старой технологии. Все эти операторы организуют цикл и организуют проверку условия выхода из него.

5.2. Оператор *Repeat...Until*

Данный оператор организует проверку выхода из цикла после каждого выполнения всех повторяемых операторов. Схема и реализация цикла с помощью оператора ***Repeat*** имеет вид :

```
repeat
  P1;
  P2;

  Pn;
until <условие>;
```

Здесь <*условие*> записано с помощью выражения логического типа.

Оператор ***Repeat*** имеет следующие характерные особенности: он организует выполнение повторяемых операторов P1, P2, …,Pn по крайней мере один раз; хотя бы один повторяемый оператор должен влиять на значение «условие», (чтобы не было зацикливания) и это отслеживается на этапе трансляции: если не обнаруживается такого влияния, выдается сообщения об ошибке. Циклическй процесс прекращается при *<условие>=True.*

Проиллюстрируем использование данного оператора на примере решения следующей задачи:

Вычислить и напечатать таблицу значений заданной функции *y=f(x)* на интервале *[a,b]* с шагом *h.*

Для организации вычислений здесь введена переменная *x.* Вначале ей присваивается значение левого конца интервала *a,* вычисляется и печатается

```
x:=a;
repeat
  y:=f(x);
 //  печать x, y
  x:=x+h;
until x>b;
```

значения функции при *x=a.* Затем значение переменной *x* увеличивается на величину шага *h,* после чего проверяется условие выхода значения переменной *x* за пределы интервала *(x>b)*. При достижении этого условия происходит выход из цикла, т.е. управление передается на оператор, следующий за *until.* Как видим значение переменной *x,* входящей в условие, изменяется при каждом повторении, поэтому на этапе трансляции ошибок не будет. Однако, это не гарантирует от зацикливания в случае, если по ошибке ввода переменная *h* примет нулевое значение. Произойдет эффект «зависания» задачи на неопределенное время.

5.3. Оператор *While...do*

Иногда удобнее организовать проверку условия выхода из цикла не в конце, а перед выполнением первого повторяемого оператора. Например, в выше приведенной задаче, если из-за ошибки ввода *a* окажется больше чем

*b,*то значение функции вычислять и печатать не следует. Оператор *While* позволяет это сделать.

Схема и реализация цикла с помощью оператора *While* имеет вид:

While <условие> do Begin P1; . . Pn; End;

Если повторяется только один оператор, то скобки *begin...end;* можно не писать. Аналогично, как и в операторе *Repeat,* хотя бы один повторяемый оператор должен влиять на значение «условия». Циклическй процесс прекращается при *<условие>=False.*

В качестве примера использования оператора *While* рассмотрим решение следующей задачи:

Вычислить сумму квадратов целых чисел в диапазоне от *m* до *n:*

$$S = \sum_{i=m}^{n} i^2 = m^2 + (m+1)^2 + ... + n^2;$$

если *n<m* тогда сумма не вычисляется и *S=0*.

Фрагмент программы решения этой задачи имеют вид:

```
s:=0;    i:=m;
while i<=n do
        begin
        s:=s+sqr(i);
        i:=i+1;
        end;
```

При вычислении суммы вводится переменная, в которой она будет накапливаться. В данном примере это переменная с именем *S* (заметим, что имя удобно сохранить такое, которое введено при описании задачи). Вначале эта переменная, очищается (*S=0*), а затем при каждом проходе цикла в эту переменную добавляется очередное вычисленное значение ($S=S+i^2$).Иногда, для экономии вычислений вначале вместо (*S=0*) можно в *S* занести первый член суммы.

Отметим одну общую особенность решения двух вышеприведенных задач. При организации цикла используется так называемая *«переменная цикла» x* или *i*. Вначале цикла этой переменной присваивается начальное значение *(x=a)* или *(i=m)*; внутри цикла при каждом повторении её значение изменяется на определенную величину-*шаг переменной цикла (h* или *i)*; в условии выхода из цикла проверяется, не вышла ли переменная цикла за пределы интервала *(x>b)* или *(i>n)*. Следует заметить, что в большинстве решаемых задач циклы организуются с использованием именно такой переменной цикла.

5.4. Оператор *For...do*

Довольно часто встречается ситуация, когда переменная цикла относится к целому типу (в общем случае к порядковому) и её значение изменяется на единицу. Для организации таких циклов введен оператор ***For.*** Его реализация с помощью оператора *For* имеет вид:

For i:=m to n do begin P1; P2;…end;

Здесь *i, m, n* - переменные целого типа (в общем случае порядкового). Заметим, что на месте *m* и *n* могут быть выражения целого (порядкового) типа. При *n<m* операторы *P1,P2,…* не выполняются ни разу. Если повторяемый оператор только один тогда скобки *begin … end;* можно опустить.

В операторе *For* действия *(i:=m, i≤n, i:=i+1)* организуются автоматически, поэтому фрагмент схемы программы, реализующей решение задачи , имеет более экономный, чем при использовании оператора *While* вид:

```
S:=0;
For i:=m to n do n  s:=s+sqr(i);
```

Имеется разновидность оператора *For*, в которой организуется изменение переменной цикла по убыванию *(i:=i-1)*:

```
For i:=n  downto m do
    Begin  P1; P2; . . . ; Pn; end;
```

В этом операторе $m \geq i \geq n$. Если *m<n* то операторы *P1...Pn* не выполняются.

Приведем пример использования этого оператора в случае, когда переменная цикла имеет тип *Char* (один из порядковых).

Вывести все символы латинского алфавита в обратном порядке от z до *a* в компанент *Memo1:*

```
Var c:char;
. . .
For c:='z' downto 'a' do
      Memo1.lines.add(c+' ');
```

Для более гибкой организации циклов в состав *Object Pascal* включены два оператора:

Break – реализует немедленный выход из цикла, передавая управление оператору, стоящему сразу за концом оператора цикла;

Continue – обеспечивает досрочное завершение очередного прохода цикла, передавая управление на конец цикла.

5.5. Вложенные циклы

Вложенность циклов имеет место тогда, когда внутри повторяемой части операторов необходимо организовать цикл. Проиллюстрируем это на примере решения следующей задачи.

Рассмотрим функцию *S(x,n),* представляющую собой сумму ряда

$S(x,n)=\sum_{i=1}^{n}\frac{x+1}{i}$. Требуется вычислить таблицу значений этой функции на интервале *[a, b]* с шагом *h=(b-a)/m*. Задаются *a, b, n* и *m* – количество разбиений интервала.

Анализ показывает, что решение этой задачи представляет собой комбинацию уже известных нам алгоритмов .

Фрагмент программы имеет вид:

```
h:=(b-a)/m;
x:=a;
repeat
s:=0;
for i:=1 to n do
        s:=s+(x+1)/i;
memo1.lines.add(floattostr(x)+’  ‘+floattostr(s)); // Вывод x,y в memo1
until x>b+0.0000001;
```

Следует отметить, что при каждом приращении переменной цикла *x* накапливается погрешность, связанная с тем, что величина *h* из-за округлений вычислена не точно. Поэтому возможна ситуация, когда после *m+1* шага значение *x* не равно *b* и превышает *b* на некоторую незначительную величину, имеющую порядок погрешности округлений. В результате, если условие выхода из цикла записано как *x>b,* то значение функции в точке *b* не вычисляется и не печатается. Для того, чтобы в точке *b* функция всегда вычислялась в приведенном алгоритме используется условие *x>b+0.0000001*.

5.6. Примеры некоторых часто встречающихся циклических алгоритмов

Вычисление заданного члена рекуррентной последовательности

Последовательность чисел $a_0,a_1,a_2...a_n$ называется рекуррентной первого порядка, если каждый следующий член выражается через один предыдущий по определенному правилу φ: $a_k=\varphi(a_{k-1})$. Поэтому для того, чтобы найти все члены последовательности достаточно задать начальный a_0.

Фрагмент программы вычислений члена с номером n, при заданном a_0 , имеют вид:

```
a:=a0;          for k:=1 to n do   a:=f(a);
```

Заметим, что для реализации алгоритма понадобилось не *n,* а только две переменных *a0* и *a.*

Последовательность $a_к$ сходится к некоторому пределу *a*, если при $k \to \infty$, $|a_k - a| \to 0$. При увеличении *k* так же стремится к нулю разность между соседними членами последовательности $|a_k - a_{k-1}| \to 0$. Поэтому для нахождения предела последовательности *a* с заданной точностью ε используется следующий алгоритм:

```
a:=a0;   it:=0;
repeat
   w:=a;  it:=it+1;
    a:=f(a);
until (abs(a-w)<eps) or (it>100);
```

Здесь *it* – счетчик количества повторений, который служит как для их подсчета, так и страхования от зацикливания алгоритма, если последовательность не сходится к конечному пределу.

Вычисления сумм с использованием рекуррентной последовательности

В практике вычислений часто приходится находить всевозможные суммы. Использование рекуррентности позволяет значительно упростить вычисления. Представим задачу суммирования в общем виде как $S = \sum_{k=0}^{n} a_k$, где $a_0, a_1, \ldots a_к \ldots a_n$, последовательность, каждый член которой представляет некоторую функцию от *k*. Предположим, что из последовательности $a_к$ удается выделить рекуррентную последовательность $c_к$: $c_k = \varphi(c_{k-1})$, представив $a_к = c_к \cdot b_k$, где b_k функция от номера *k* – *f(k)*. В этом случае программа вычисления суммы имеет вид:

```
c:=c0          s:=c*f(0);
 for k:=1 to n do
        begin
          c:=φ(c);
          s:=s+c*f(k);
        end;
```

ЛЕКЦИЯ 6. ОТЛАДКА ПРОГРАММ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Закон программирования: «В каждой вновь написанной программе всегда найдется, по крайней мере, одна ошибка». Написать сложную программу без ошибок удается редко. Мастерство программиста определяется умением составить верный алгоритм, быстро отладить и протестировать программу его реализующую, т.е. :

а) выявить и исправить все ошибки;

б) убедиться в правильности алгоритма.

Итак, вы составили алгоритм, написали программу, запустили *(F9)* и … обнаружили ошибки!

6.1. Ошибки на этапе компиляции

Эти самые простые, их вылавливает компилятор, он останавливается в том месте, где обнаружил ошибку и внизу окна текста выдает сообщение о ее характере .

Например:

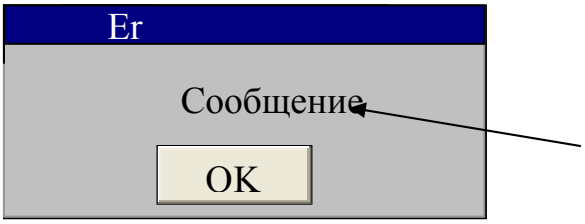
-«Несоответствие типа i:=2.5;»

-«Переменная нигде не используется»

Компилятор часто выдает сообщения, указывая на совершенно правильную строку, но обычно это результат ошибки в предыдущей строке. Постарайтесь исправить все замечания компилятора, даже если они несущественны. Здесь полезно знать, что если программа содержит модули, то при нажатии *(F9)* перекомпилируются только те из них, в которых что-то менялось. В режиме *Project/ Build Project* происходит перекомпиляция всех модулей, входящих в проект. Это бывает полезно отменить некоторые опции отладки для повышения эффективности результирующего модуля.

6.2. Ошибки на этапе выполнения

Итак, ваш проект откомпилировался и готов к работе. Вы вводите исходные данные, запускаете программу, но компьютер упорно не хочет выполнять вашу программу. Он останавливается и выдает какое-то сообщение:



Бывает так: вы в окне компанента *Edit* набрали *0.256,* а надо *0,256.* Появляется сообщение *«EConvertError»* (Ошибка преобразования строки в другие типы данных).

Или, если в тексте программы выполняются последовательно операторы:

y:=0;

a:=x+1/y;

то появляется сообщение *«EZeroDivide»* (Деление на нуль действительных чисел).

Когда компьютер встречает подобную ситуацию, он анализирует, что же это такое. У него в памяти заложено порядка сотни таких стандартных ситуаций. При возникновении одной из них, компьютер прекращает выполнение и выдает соответствующее сообщение.

Такие ситуации называются исключительными *(Exceptions)*. Обычно, при нажатии 

вам будет показано место, где это произошло.

Получив такое сообщение, нужно выяснить, что привело к такой ошибке.

Обычно для анализа ошибки желательно получить дополнительную информацию. Самый простой способ- это пошаговое исполнение с просмотром результатов, используя клавиши (*F7* – выполнить операторы, находящиеся в строке курсора, *F8* – тоже, что и *F7*, только без захода в процедуры, *F4* – выполнить все операторы до строки, на которую указывает курсор).

Для этого щелкните мышью напротив строки, на которой следует остановиться, при этом строка выделяется красным цветом, а в программу занесется *точка прерывания*. После этого запускается программа. Она остановится в этом месте. Подведя и зафиксировав курсор напротив переменной, можно посмотреть, чему она равна.

Более удобно делать просмотр значений с помощью окна наблюдения *(Watch List)*. *Мастер оценки выражений* вам его покажет, если подвести курсор к переменной и нажать *Ctrl +F5*.

Кроме окна *Watch List* для отладки используется также окно отладки инспектора *Run/Inspect.*

Добавление еще одной переменной в окно происходит установкой на нее курсора и нажатием нажать *Ctrl+F5*. Можно выделить целое выражение и аналогично добавить его в *Watch List.*

В процессе отладки полезно отключать оптимизацию транслируемого кода в окне *Project/Options/Compiler/Optimization.* Выполнить *Project / Build Project.*

После отладки сделать обратное!!!

6.3. Понятие исключительной ситуации

Под ***исключительной ситуацией*** понимается такое состояние, возникающее при выполнении некоторых действий программы (например: деление

на ноль, попытка открыть несуществующий файл, выход индекса за пределы массива и т.п.), при котором требуется выполнить определенные операции для продолжения ее работы или корректного завершения.

При работе в среде Delphi возникновение одной из вышеназванных или других подобных ситуаций, которых насчитывается более 50, обычно приводит к полной остановке выполнения программы с указанием причины, что не всегда удобно. Каждая такая ситуация имеет свое имя и тип. Для повышения надежности программы и защиты ее от преждевременного завершения в Delphi разработчику предоставлена возможность с помощью определенных операторов «перехватить» различные исключительные ситуации и организовать выполнение определенных операций при их возникновении.

При отладке программы, использующей обработку исключительных ситуаций предварительно необходимо ***отменить*** стандартную реакцию среды Delphi на эти ситуации. Для этого надо в главном меню Delphi отключить опцию ***Stop on Delphi Exceptions*** находящуюся в ***Tools Debbuger Options*** на закладке ***Language Exceptions***.

6.4. Защищенные блоки

Для перехвата исключительных ситуаций и описания реакций на их возникновение в Pascal предусмотрены операторы организации защищенного блока ***try...end;*** двух видов ***(Except и finally)***:

```
...
try          //попытаться выполнить
<последовательность защищенных операторов>
except       //обработчики исключительных ситуаций:
on <тип искл. ситуации 1> do <оператор–обработчик 1>;
...
On <тип искл. ситуации k> do <оператор–обработчик k>;
else         //может отсутствовать
<операторы выполняемые если перехваченная ситуация не обнаружена
среди типов ситуаций 1÷k>
end;
<следующий оператор>;
...
```

Здесь между ***except*** и ***else*** приводится список нескольких стандартных типов ситуаций между ключевыми словами ***on...do*** и соответствующих каждому из них операторов-обработчиков, выполняемых, если возникает эта ситуация.

```
...
try
<последовательность защищенных операторов>
finally
<последовательность операторов, которые выполняются всегда,
независимо от того перехвачена ситуация или нет>
```

end;
<*следующий оператор****>***
...

Эти блоки отличаются лишь способом обработки перехваченной в <*последовательности защищенных операторов*> исключительной ситуации. Если все операторы в этой последовательности выполнялись без возникновения исключительной ситуации, то в блоке ***Except*** управление передается на <*следующий оператор*>, в блоке ***finally*** на оператор, следующий за словом ***finally***. Если при выполнении одного из защищенных операторов возникла исключительная ситуация, то следующие за ним защищенные операторы пропускаются, причем в блоке ***finally*** управление передается опять на оператор, следующий за словом ***finally***. В блоке же ***Exception*** управление передается тому оператору-обработчику, <*тип искл. ситуации*> которого соответствует возникшей ситуации, и после его выполнения - <*следующему оператору*>. Если в списке операторов-обработчиков не обнаружен тип возникшей ситуации, то управление передается операторам, стоящим между ***else...end***. Если при этом область ***else*** отсутствует, то выполняется стандартная обработка ситуации, предусмотренная в среде Delphi. Следует помнить, что поиск нужного обработчика осуществляется с начала списка вниз до первого, соответствующего. Поэтому, если в списке имеется несколько типов ситуаций, соответствующих возникшей, то выполняется обработчик встретившийся первый.

Если для составителя программы важен лишь сам факт возникновения исключительной ситуации, то возможна следующая конструкция:

...
try
<*последовательность защищенных операторов****>***
except
<*операторы, которые выполняются при возникновении ИС****>***
End;
...

которая отличается от блока ***finally*** тем, что область <*операторы…*> здесь выполняется только в случае возникновения исключительной ситуации.

Защищенные блоки могут вкладываться друг в друга на любую глубину, причем в любой области, где возможно вставить оператор ***try...end***.

6.5. Некоторые стандартные типы исключительных ситуаций

Тип. Искл. ситуаций	Исключительная ситуация
EAbort	Любая исключительная ситуация
EArrayError	Ошибка при операциях с массивами (например, индекс выходит за пределы массива).
EConvertError	Ошибка преобразования строки в другие типы данных.
EDivByZero	Целочисленное деление на нуль.
EintOverFlow	Переполнение при операции с целыми числами включить {$Q+}.
EZeroDivide	Деление на нуль действительных чисел.
EOverFlow	Переполнение при работе с действительными числами.
EassertionFiled	Намеренная ситуация генерируемая с помощью процедуры Assert {$C+}.

6.6. Инициирование собственных исключительных ситуаций

Возникновение исключительной ситуации может быть инициировано самим разработчиком программы для обработки некоторых «своих» ситуаций. Для этого имеется три оператора.

Во-первых, процедура ***Abort*** вставленная в нужном месте раздела *try* генерирует ситуацию типа ***EAbort***. В отличие от *Break* и *Exit*, она позволяет, например, осуществить выход из глубоко вложенных циклов и процедур.

Во-вторых, процедура *Assert (B:Boolean; [const st:String]),* генерирует исключительную ситуацию типа *EAssertionFailed,* если результат логического выражения *B=false*, при этом возможна выдача в диалоговое окно сообщения, помещенного в *St* (этот аргумент может отсутствовать).

В-третьих, ключевое слово

raise <тип исключения**>. create (**‘текст сообщения’**),**

генерирует исключительную ситуацию указанного типа и выдает сообщение.

6.7. Примеры фрагментов программ

В дополнение к этой теме приведем две функции, позволяющие организовать экстренный диалог и управлять ходом вычислений. В *Windows* имеется набор *API*-функций (*Application Program Interface* - интерфейс прикладных программ).

При обработке исключительных ситуаций удобно использовать процедуру ***ShowMessage(**'Выводимый текст'**),*** которая выводит иконку с текстом. Если она появилась, то будет до конца работать.

Для обеспечения диалога применяется функция:

MessageDlg(**'текст'**,mtInformation,[mbok,mbNo,…],0):word,

которая, кроме сообщения на иконке имеет заголовок *Inf* (таких стандартных заголовков имеется 9) и еще от одной до 11 кнопок, т.е. главное то, что реализуется остановка и ожидание нажатия кнопки!!! При нажатии на одну из них функция выдает номер соответствующей кнопки (*mrOk, mrNo,* ...), что можно полезно использовать например с помощью оператора **case**.

```
K:=MessageDlg ('значениех<0', mtInformation,[mbok,mbNo,…],0);
Case k of
MrOk: <>;
MrNo: <>;
MrYes: <>;
End;
```

Пример перехвата переполнения:

```
...
try f:=1;
 for i:=2 to 1000 do f:=f*i;
except
 on EintOverFlow do
  ShowMessage('переполнение при i='+intTostr(I)');
end;
...
```

Пример обработки файла:

```
...
   st:=Edit1.text;
   Assign(f,st);
   try reset(f); except rewrite(f); end;
try
...
```

Работа с файлом:

```
...
finally
  Flush (f); // запись в файл содержимого буфера
  closefile(f);
end;
```

Программа не будет остановлена, файл будет закрыт.

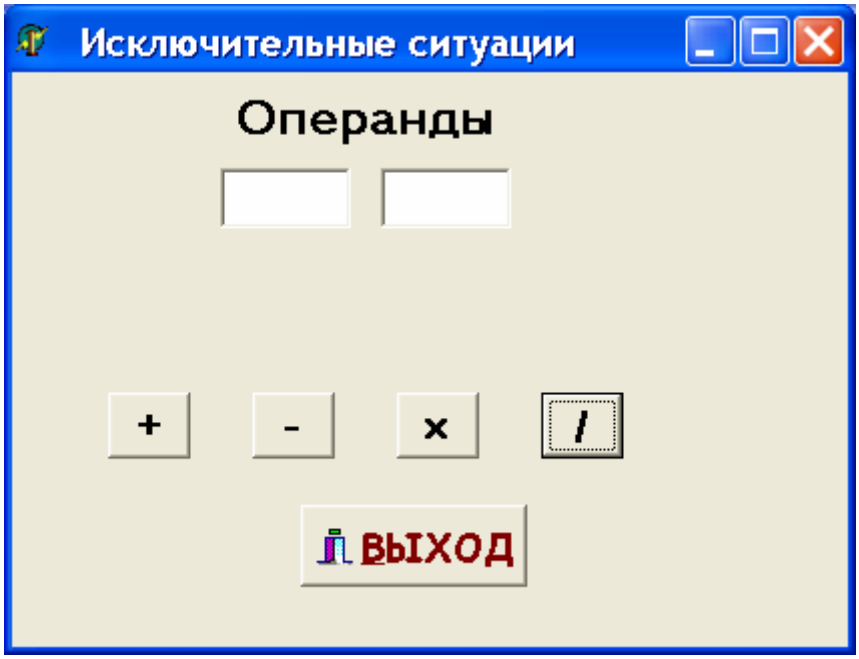
Пример перехвата деления на нуль внутри цикла

```
...
x:=a;
repeat
try
  y:=sin(x)/cos(x);
except
  on EZeroDivide do begin y:=0;
MessageDlg('при x='+FloattoStr(x),mtError? [mbOk? mbNO], 0) of case:
MrNo: exit;
MrYes: y:=0;
End;
end;
  Writeln(x,y); x=x+h;
Until x>b+h/2;
end;
```

Пример создания собственной исключительной ситуации:

```
...
try
if x=0 then raise Eabort.Create ('x=0');
  Assert(x>0,'x отрицательный');
y:=sgrt(x)·z+5/x;
...
except
  on EAssertionFiled do exit;
  on Eabort do Exit;
end;
```

Пример программы с обработкой исключительных ситуаций. Простейший калькулятор:




```
unit Unit1;
                interface
uses  Windows,  Messages,  SysUtils,  Variants,  Classes,
Graphics, Controls, Forms, Dialogs, StdCtrls, Buttons;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    BitBtn1: TBitBtn;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
```

```
             implementation

{$R *.dfm}
{$R+}

procedure TForm1.FormCreate(Sender: TObject);
begin
 Edit1.Clear;
 Edit2.Clear;
end;

procedure TForm1.Button1Click(Sender: TObject);
   var a,b,c:extended;    //    /
begin
  try
  a:=StrToFloat(Edit1.text);
  b:=StrToFloat(Edit2.text);
  c:=a/b;
  label2.Caption:=FloatToStrf(c,fffixed,8,2);
  except
  on EZeroDivide do MessageDlg('Нельзя делить на нуль',
      mtError,[mbOk],0);
  on Econverterror do MessageDlg('Проверьте'+
     ' значения в edit',mtError,[mbOk],0);
  end;
end;

procedure TForm1.Button4Click(Sender: TObject);
   var a,b,c:byte;    //  +
begin
  try
   a:=StrToInt(Edit1.text);
   b:=StrToInt(Edit2.text);
   c:=a+b;
   label2.Caption:=IntToStr(c);
  except
  on Erangeerror do MessageDlg('Результат выходит за'+
  '         пределы         диапазона         допустимых
значений',mtError,[mbOk],0);
  on Econverterror do MessageDlg('Проверьте'+
  ' значения в edit',mtError,[mbOk],0);
  end;

end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
    var a,b,c:byte;   // -
 begin
   try
   a:=StrToInt(Edit1.text);
   b:=StrToInt(Edit2.text);
   c:=a-b;
   label2.Caption:=IntToStr(c);
  except
   on Erangeerror do MessageDlg('Результат выходит за'+
     '        пределы        диапазона        допустимых
значений',mtError,[mbOk],0);
   on Econverterror do MessageDlg('Проверьте'+
     ' значения в edit',mtError,[mbOk],0);
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
   var a,b,c:byte;  //  *
begin
 try
   a:=StrToInt(Edit1.text);
   b:=StrToInt(Edit2.text);
   c:=a*b;
   label2.Caption:=IntToStr(c);
  except
   on ERangeerror do MessageDlg('Результат выходит за'+
      ' пределы диапазона допустимых значений',
         mtError,[mbOk],0);
   on EConverterror do MessageDlg('Проверьте'+
      ' значения в edit',mtError,[mbOk],0);
  end;
end;

end.
```

ЛЕКЦИЯ 7. СОСТАВЛЕНИЕ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ МАССИВОВ

7.1. Понятие массива

Математики для удобства записи различных операций часто используют индексированные переменные: векторы, матрицы, тензоры. Так, вектор $\vec{c}$ представляется набором чисел (c_1, c_2,…, c_n), называемых его компонентами, причем каждая компонента имеет свой номер, который принято обозначать в виде индекса. Матрица A – это таблица чисел (a_{ij}, $i=1,..m$, $j=1..n$), где i –номер строки, j – номер столбца. Операции над матрицами и векторами обычно имеют короткую запись, которая обозначает определенные порой сложные действия над их индексными компонентами. Например, произведения двух векторов записывается как

$$\vec{c} \cdot \vec{b} = \sum_{i=1}^{n} c_i \, b_i$$

Произведение матрицы на вектор

$$\vec{b} = A \cdot \vec{c}, \quad b_i = \sum_{j=1}^{n} a_{ij} \cdot c_j ;$$

Произведение двух матриц

$$D = A \cdot G, \quad d_{ij} = \sum_{k=1}^{n} a_{ik} \cdot q_{kj} ;$$

Введение индексированных переменных в языках программирования также позволяет значительно облегчить реализацию многих сложных алгоритмов, связанных с обработкой массивов однотипных данных.

В языке Pascal для этой цели имеется структурированный тип переменных – ***массив***. Переменные типа массив вводятся с помощью ключевого слова ***array*** следующим образом:

Type <имя типа>=array[m..n] of <тип элементов>;
Var b,c:<имя типа>;

Здесь *m* и *n* должны быть константами, *b, c* – массивы.

В программе доступ к каждому элементу массива осуществляется по индексу заключенному в квадратные скобки, например:

*b[j]:=5; c[k]:=b[i]+c[j*2+k];*

причем значение индексов не должно выходить за границы диапазона *(m≤j≤n)* указанного в описании. Чаще всего используются индексы целого типа, хотя возможны индексы порядкового типа (например, *Char*).

Программист по желанию может определить массивы с произвольным количеством индексов, например:

```
Const
        C1:array[1..3] of real (1.5,6.8,7);
        C2:array[1..2,1..3] of word ((1,2,3),(4,5,6));
type
        vek=array[1..10] of real;
        mat1=array[-5..5] of vek;
        mat2=array[-5..5,1..10] of real;
        mas3=array[0..2] of mat2;
Var
        a,z:vek;
        b:mat1;
        c:mat2;
        d:mas3;
Begin
        ...
        a[i]:=0.2;
        b[j,i]:=a[i];   c[j,i]:=5;
        d[k,j,i]:=a[i]+b[j,i];
        ...
```

здесь $1 \le i \le 10$, $-5 \le j \le 5$, $0 \le k \le 2$.

Заметим, что массивы *b* и *c* имеют одинаковую структуру, хотя их типы вводятся по разному. Однотипные массивы (как целое) могут участвовать только в операциях отношения *«равно», «не равно»* и в операторе присваивания. Заметим, что из вышеприведенных однотипными являются *a* и *z,* массивы *b* и *c* не однотипные, хотя и имеют одинаковую структуру.

Например:

выражение *a*=z возвращает значения ***true,*** если все элементы массивов *a* и *z* одинаковы и ***false*** в противном случае;

выражение *a*< >z возвращает значение ***true,*** если они отличаются хотя бы в одном элементе;

a:=*z*; всем элементам массива a присваиваются значения соответствующих элементов массива *z*.

7.2. Некоторые возможности ввода-вывода в Delphi

При вводе массивов с большим количеством элементов, обычно исходные данные (элементы) готовятся заранее с помощью текстового редактора в отдельном текстовом файле, и затем программа вводит их из этого файла. При необходимости изменить некоторые исходные элементы производят редакцию текстового файла, после чего снова запускают программу. Проиллюстрируем это на примере следующей задачи:

Ввести из файла и распечатать в виде таблицы в окно *Memo* двухмерный массив *A* размерности $m \times n$.

Вначале с помощью встроенного текстового редактора подготовим файл исходных данных, например следующего вида:

```
4 5      значения m и n
0.1 0.2 0.8 1.2 5.6 – первая строка A
1.8 2.1 4.2 6.3 2.5 – вторая строка A
0.8 0.2 0.6 1.2 1.4 – третья строка A
4.1 7.3 5.8 3.2 1.5 – четвертая строка A
```

и запишем его под именем *Prg.dat* используя в меню ***File*** опцию ***Save as***.

Теперь фрагмент программы выглядит так

```
Type mat=array[1..10,1..10] of extended;
Var a:mat;
    Lr:TextFile;//описание текстового файла;
    St:String;
begin
  AssignFile(Lr,'Prg.dat');
  Reset(Lr);
  Readln(Lr,m,n);//чтение  m и n из 1-й строки файла
    for i:=1 to m do
      begin
       st:='';
       for j:=1 to n do
         begin
         Read(Lr,a[i,j]);  //чтение i-й строки
         st:=st+FloatToStrF(a[i,j],fffixed,6,1);
         end;
      Readln(Lr);//перевод курсора на новую строку
файла
      Memo1.Lines.Add(st);//Вывод строки в окно Memo
      end;
```

В Delphi имеется возможность более изящно организовать ввод/вывод двухмерных и одномерных массивов с отображением их на форме. Для этого используют специальный компонент *StringGrid* (находится в меню *Additional),* который предназначен для отображения информации в виде двухмерной (одномерной) таблицы, каждая ячейка которой представляет собой окно однострочного редактора (аналогично окну *Edit*). После помещения этого компонента на форму в программе появляется возможность работать с двухмерным массивом вида

StringGrid1.Cells:array[0..ColCount-1,0..RowCount-1] of string;

Значения ***ColCount*** и ***RowCount,*** определяющие количество столбцов и строк в отображаемой таблице, задают с помощью инспектора объектов или вычисляют внутри программы. Каждый элемент массива *Cells* представляет собой строку текста, содержимое которой отображается в соответствующей ячейке на форме. Поэтому, если мы в ячейку на форме с помощью клавиатуры наберем любой текст, он тут же помещается в ячейку *Cells* и наоборот, если в процессе работы программы элементу массива *Cells[i, j]* присвоить строку текста, то она отобразится в соответствующей ячейке на форме.

В результате задача о вводе вышеописанного массива *A,* и его вывода на форму после преобразований решается следующим образом:

```
m:=4; n:=5;
StringGrid1.ColCount:=n;
StringGrid1.RowCount:=m;
   ...
Procedure Tform1.Button5Click(Sender:Tobject);
Begin
for i:=1 to m do //Ввод матрицы из таблицы
 for j:=1 to n do //на форме в массив A
  a[i,j]:=StrToFloat(StringGrid1.Cells[j-1,i-1]);
     //используется преобразование текстовой
     //записи числа в его значение типа extended.
    // Некоторые действия с массивом A
     ...
for i:=1 to m do//Вывод массива A в таблицу на форме
for j:=1 to n do
StriGrid1.Cells[j-1,i-1]:=FloatToStrF(a[i,j],
fffixed,8,3);
   // преобразование действительного   числа в текст
end;
```

Здесь в каждую ячейку будет выведено число с тремя десятичными знаками. Заметим, что в массиве *Cells* первый индекс соответствует номеру столбца в таблице отражаемой на форме, второй – номеру строки (нумерация строк и столбцов начинается с нуля), в массиве же *A* наоборот, первый индекс трактуется как номер строки, второй – номер столбца в соответствующей матрице.

7.3. Примеры часто встречающихся алгоритмов работы с массивами

***Сумма n элементов одномерного массива*:**

S:=0; for i:=1 to n do S:=S+a[i];

Сумма модулей всех элементов матрицы:

```
S:=0;
for i:=1 to m do
  for j:=1 to n do
    S:=S+abs(b[i,j]);
```

Произведение диагональных элементов квадратной матрицы:

```
p:=1;
For i:=1 to n do
   p:=p*b[i,i];
```

Вычислить значение полинома степени n при заданном x

Используем рекуррентную схему, для чего представим полином в виде $P(x)=a_0+x(a_1+x\cdot(a_2+x(a_3+...x(a_{n-2}+x(a_{n-1}+x\cdot a_n))..)$.

Фрагмент программы имеет вид:

```
read(x,n,a);
p:=a[n];
for k:=n-1 down to 0  do
        p:=p*x+a[k];
Writeln(p);
```

Нахождение максимального элемента одномерного массива:

```
max:=a[1];
for i:=2 to n do
        if a[i]>max then max=a[i];
Writeln(max);
```

Найти номер максимального элемента:

```
m:=1;
for i:=2 to n do
        if a[i]>a[m] then m:=i;
Writeln(m,a[m]);
```

Транспонирование одномерного массива из n элементов:

Требуется в массиве ($a_1, a_2, ..., a_{n-1}, a_n$) переставить элементы в обратном порядке, т.е. получить ($a_n, a_{n-1}, ..., a_2, a_1$). При этом не использовать дополнительный массив.

```
readln(n, a⃗);
        i:=1; j:=n;
        repeat
          r:=a[i];  a[i]:=a[j];  a[j]:=r;
          inc(i); dec(j);
        until i>=j;
        writeln(n, a⃗);
```

В данном алгоритме для перестановки местами двух элементов в массиве привлекается дополнительная переменная (ячейка памяти) *r*, тип которой совпадает с типом элементов массива $\vec{a}$. Такой прием перестановки довольно часто используется в алгоритмах обработки массивов.

Транспонировать двухмерный массив относительно главной диагонали:

В квадратной матрице *A* размерности *n×n* элементы главной диагонали имеют одинаковые индексы ($a_{i,i}$, $i=1, n$). Каждому элементу, стоящему выше главной диагонали ($a_{i,j}$, $1 \le i \le n-1,\ i+1 \le j \le n$), соответствует симметричный ему элемент, стоящий ниже главной диагонали ($a_{j,i}$, $1 \le i \le n-1,\ i+1 \le j \le n$). Транспонировать матрицу – это значит переставить местами эти элементы, например

$$\begin{bmatrix} 1 & 5 & 4 \\ 4 & 3 & 2 \\ 8 & 9 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 4 & 8 \\ 5 & 3 & 9 \\ 4 & 2 & 7 \end{bmatrix}$$

```
for i:=1 to n-1 do
  for j:=i+1 to n do
        begin
          r:=a[i,j];
          a[i,j]:=a[j,i];
          a[j,i]:=r;
        end;
```

В матрице A размерности m×n переставить местами две строки с номерами к1 и к2:

```
Readln(k1,k2);
for j:=1 to n do
        begin
          r:=a[k1,j];
          a[k1,j]:=a[k2,j];
          a[k2,j]:=r;
        end;
```

Сортировка одномерного массива в порядке возрастания (убывания) элементов. Метод пузырька:

```
Readln(n, a);
for i:=2 to n do
  for j:=n downto i do
    if a[j-1]>a[j] then
          begin
           r:=a[j];
           a[j]:=a[j-1];
           a[j-1]:=r;
         end;
Writeln(a);
```

Для сортировки по убыванию необходимо заменить знак > на < .

ЛЕКЦИЯ 8. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ

8.1. Статическое и динамическое распределение оперативной памяти

Все команды и данные программы во время ее выполнения размещаются в определенных ячейках оперативной памяти. При этом часть данных размещается в ячейки памяти еще на этапе компиляции и в процессе работы программы их адреса относительно начала программы не изменяются. Такое размещение данных и команд называется статическим и соответствующие этим данным переменные называются ***статическими переменными***.

В языке Pascal возможна также организация ***динамического размещения данных***, при котором под некоторые данные и программы память выделяется непосредственно во время выполнения по мере надобности, а после решения требуемой задачи память освобождается для других данных. Соответствующие таким данным переменные называются ***динамическими переменными***.

8.2. Понятие указателя

Для организации динамического распределения памяти используются переменные специального типа – указатели, которые обеспечивают работу непосредственно с адресами ячеек памяти. Под каждую переменную типа указатель отводится ячейка объемом 4 байта, в которой можно поместить адрес любой переменной.

Вводятся указатели следующим образом:

```
Type
      Pk=^mun;
      Pi=^array[1..20] of integer;
      Pb=^byte;
      Ps=^String [20];
Var
      p,q:pointer;
      a,b:Pk;
      i:Pi
      k:Pb;
      S1,S2:Ps;
```

Здесь *p, q* – ***нетипизированные указатели***; *a, b, i, k, S1, S2* – ***типизированные указатели***, т.е. они указывают, что, например, в ячейках начиная с адреса ***a*** размещается переменная указанного типа *Pk*.

Значениями указателей являются адреса переменных, размещенных в памяти. Значение одного указателя можно передать другому с помощью оператора присваивания, например:

```
p:=q;        a:=b;
```

При этом надо помнить, что в операторе присваивания типизированные указатели должны быть одного типа.

Используя нетипизированные указатели, можно передать адрес между указателями разного типа, например, так:

p:=i; k:=p;

С типизированными указателями можно работать как с обычными переменными следующим образом:

*S1^:=**'Иванов'**; // по адресу S1 разместить строку*
i^[11]:=88;
k^:=25;
m:=i^[9]+k^; // m – целого типа

Указатели одного типа можно сравнить на предмет равенства = и неравенства < >, например: ***if a=b then ...*** или ***if a<>b then...***

Следует заметить, что такие действия возможны лишь после того, как самим указателям *S1, i, k, a, b* будут присвоены конкретные значения (адреса).

Адрес указателю можно присвоить следующим образом:

Если *m, n* – обычные статические переменные, то ее адрес можно получить с помощью специальной функции

p:=Addr(m); *a:=Addr(n);*

Очистка адреса из указателя осуществляется с помощью специальной функции nil: ***p:=nil; a:=nil,*** при этом довольно часто используется проверка условия ***if p<>nil then …***

8.3. Наложение переменных

Использование указателей позволяет «накладывать» переменные разных типов друг на друга, интерпретируя по-разному данные, расположенные по некоторому адресу. Например:

```
Var Ch:Char;
      k:^byte;
      . . .
   k:=Addr(ch);
   Ch:='A';
   Write(Ch,k^);
```

Будет выведен сам символ 'A' и его номер в кодовой таблице 65.

Следующий пример иллюстрирует наложение одномерного массива на двухмерный:

```
Var a:^array[1..4] of integer;
      b:array[1..2,1..2] of integer;
      . . . .
      a:=Addr(b);
      for i=1 to 4 do a^[i]:=i;
      Write('b[1,2]=',b[1,2]);   // будет выведено:  b[1,2]=2
```

Такое наложение может быть полезно использовано, например, при вводе матрицы иногда удобнее использовать один индекс, а при вычислениях работать с двумя индексами.

8.4. Динамическое распределение памяти

Вся свободная от программ память компьютера представляет собой массив байтов, который называется ***кучей***. Когда возникает необходимость использования программой дополнительной памяти, это осуществляется одной из процедур *New* или *GetMem*.

Процедура *New(a:<типизированный указатель>);*находит в куче свободный участок памяти, размер которого позволяет разместить тип данных **`a`** и присваивает указателю **`a`** значение адреса первого байта этого участка. После этого данный участок памяти закрепляется за программой и с ним можно работать через возникшую в программе переменную **`a^`**. Такие переменные называются ***динамическими***. После того, как необходимость работы с этой переменной отпала, данный участок памяти освобождается с помощью процедуры

Dispose(a);

При работе как с типизированными, так и с нетипизированными указателями аналогичные действия выполняют процедуры

GetMem(P:pointer; size:Word);
FreeMem(P:pointer; size:Word);

здесь *size* - количество байтов выделяемой памяти, начиная с адреса, помещаемого в указатель *P*. Следует помнить, что память под указатель выделяется 8-байтными порциями. В результате возможна нежелательная фрагментация.

8.5. Организация динамических массивов

Обычно динамическое выделение и освобождение памяти используется при работе с массивами данных.

С помощью процедур ***Getmem*** и ***Freemem*** можно создавать **массивы с изменяемым размером – динамические массивы**. Для этого определим тип указателя на массив с небольшим размером, а затем выделим памяти столько, сколько необходимо:

```
Type vek=array[1..2] of <тип элементов>;
Var  a:^vek;   //указатель на массив
    mt:word;
    . . .
mt:=sizeof(<тип элемента>);  // определяем сколько байт
            // требуется для размещения одного элемента
Read(n);
GetMem(a,mt*n);//выделяем память под n элементов массива
for i:=1 to n do Read(a^[i]); // Задаем значение
        // элементов массива
```

. . .
*FreeMem(a,mt*n);//освобождаем память*

При работе с такой программой необходимо отключать проверку выхода индекса за пределы массива и внимательно следить за тем, чтобы индекс не вышел за пределы выделенной памяти.

Начиная с версии Delphi 4, в Object Pascal введены динамические массивы, не требующие указания границ массивов:

Var a: array of extended; // Одномерный динамический массив

b: array of array of integer; // Двумерный динамический массив

Распределение памяти и указание границ индексов по каждому измерению осуществляется в ходе выполнения программы путем ***инициализации*** массива с помощью функции *SetLength(имя динамического массива, длина).* В ходе выполнения оператора *SetLength(a, n)* одномерный динамический массив *a* будет инициализирован, т.е. для него будет отведена память достаточная для размещения *n* переменных типа *extended.* Нижняя граница индекса всегда будет равна *0,* поэтому верней границей индекса станет *n-1*. В многомерных массивах сначала устанавливается длина первого измерения, затем второго, третьего и т. д. Поэтому инициализация двумерного динамического массива b (матрицы, содержащей *n* строк и *m* столбцов) производится следующим образом:

SetLength(b, n);
For k := 0 to n-1 do SetLength(b[k], m);

Так как длина каждой строки задается отдельным оператором, то она может быть разной, например:

SetLength(b, n);
For k := 0 to n-1 do SetLength(b[k], k+1); // Треугольная матрица
или:
SetLength(b, n);
SetLength(b[0], 8); // Длина 1-й строки равна 8
SetLength(b[1], 2); // Длина 2-й строки равна 2
SetLength(b[2], 4); // Длина 3-й строки равна 4

Имя динамического массива является указателем. Поэтому после окончания работы с массивом необходимо освободить память с помощью функции *Finalize(<имя динамического массива>)* или оператором

<имя динамического массива> := nil;

Процедуру *SetLength()* в процессе выполнения программы можно вызывать произвольное количество раз. Каждый вызов приводит к изменению длины массива, причем содержимое массива сохраняется. Если при вызове *SetLength()* длина массива увеличивается, то добавленные элементы заполня-

ются произвольными значениями, так называемым *мусором.* Если длина массива уменьшается, то содержимое отброшенных элементов теряется. Для работы с динамическими массивами *Object Pascal* можно использовать функции *Low(), High(), Copy().*

Функции *Low(<имя динамического массива>)* и *High(<имя динамического массива>)* возвращают наименьшее и наибольшее значения индекса динамического массива, т. е. *0* и *длина-1* соответственно. Для **пустого** массива возвращаемое функцией *High()* значение равно *-1*. Функция *copy()* возвращает заданную часть массива и имеет вид:

Copy(<имя динамического массива>, <начальное значение индекса>, <количество копируемых элементов>);

Примеры работы с динамическими массивами приведены в лекции «Списки на основе динамических массивов».

ЛЕКЦИЯ 9. ПОДПРОГРАММЫ И БИБЛИОТЕКИ

9.1. Понятие подпрограммы

Каждый, кто сталкивается с компьютером, с удивлением обнаруживает, что компьютер знает и умеет выполнить многое из того, что умеет человек. Так, он, оказывается, умеет играть в шахматы, обыграет вас в карты, не хуже врача поставит диагноз, даст юридическую консультацию, поможет советом изобретателю новой техники, решит сложную математическую задачу, и еще много чего, о чем вы даже не подозреваете. Спрашивается, почему компьютер все это знает? Вот вы тоже кое-что знаете, почему? Потому что вас учили: с момента рождения, в вашу память закладывали различные навыки. У компьютера, (кстати, как и у человека) имеется два уровня памяти – оперативная и долговременная. Компьютер тоже учат. Уже при рождении в его долговременную память закладывают разные знания в виде программ, которые в нужный момент вызываются в оперативную память и подаются на процессор для выполнения тех или иных действий.

Для того, чтобы учить компьютер, имеется мощное средство – библиотека стандартных подпрограмм, которую каждый программист может дополнять по своему желанию если освоит способ ее организации. Подпрограмма является базовым элементом при программировании.

При решении многих задач, часто требуется производить вычисления по одним и тем же алгоритмам при различных исходных данных. Для облегчения программирования в языках программирования такие алгоритмы оформляются в виде подпрограмм.

Подпрограмма – это последовательность операторов, оформленная таким образом, что ее можно вызвать по имени из любой программы, которую вы составляете, передать ей определенные параметры и, получив, требуемый результат, продолжить вычисления.

9.2. Описание подпрограмм

В языке Паскаль имеется два типа подпрограмм:

Процедуры (Procedure) и функции (Function).

Оформление программ в виде процедур и функций производится в соответствии со структурой на Рис. 9.1., где

<имяП>, <имяФ> - имя подпрограммы;

<список формальных параметров> - содержит перечисление имен с указанием их типов, например:

*May (x,y : extended; var m, k : integer; const s : string**);***

Через эти параметры осуществляется обмен информацией между подпрограммой и той программой, которая к ней обращается.

В отличие от процедуры, для имени функции *<имяФ>* указан ее тип *<типФ>*, а через ключевое слово ***Result*** имени присваивается результат вычислений, имеющий тип *<типФ>*.

Структура подпрограммы – процедуры

```
Procedure <имяП>(<список формальных параметров>);
        const . . .
        type . . .
        var . . .
        procedure ИП(. . .); //  раздел описаний локальных
        begin                //  констант, типов, переменных
        . . .                //  и подпрограмм
        end;
        function ИФ(. . .):ТФ;
        begin
        . . .
        end;
begin
. . .           //   раздел выполняемых операторов процедуры
end;
```

Структура подпрограммы – функции

```
Function <имяФ>(<список формальных параметров>):<типФ>;
        Const . . .
        Type . . .              //      раздел описаний локальных
        var . . .               //      констант, типов, переменных и
        Procedure ИП(. . .);    //      подпрограмм
        begin
        . . .
        End;
        Function ИФ(. . .):ТФ;
        begin
        . . .
        End;
                begin
                . . .    // раздел выполняемых операторов функции
                Result:= <результат>;
                end;
```

Рис 9.1.

Вызов подпрограммы – процедуры из какой либо программы производится оператором вида:

имяП(<список фактических параметров>);

Здесь *<список фактических параметров>* содержит перечисление имен данных в вызывающей программе, которые соответствуют списку формальных параметров, причем каждый фактический параметр должен иметь тот же тип, что и соответствующий ему формальный. Следует отметить, что список формальных параметров может отсутствовать.

Вызов подпрограммы – функции, в отличие от процедуры, может быть осуществлен разными способами, например:

y:=имяФ(<список фактических параметров>);

Здесь переменная y должна иметь тип *<типФ>*, или

*s:=3*имяФ(<сп.факт.пар.1>)+9.45/имяФ(<сп.факт.пар.2>);*

Здесь вызов функции производится внутри арифметического выражения с разными фактическими параметрами.

Рекомендуется оформлять подпрограмму в виде функции в том случае, когда результатом является значение одной переменной.

9.3. Передача данных между подпрограммой и вызывающей ее программой

Имеется два способа передачи данных между подпрограммой и той программой, из которой производится ее вызов – через глобальные параметры и через формальные параметры.

Передача данных через глобальные параметры

Параметры, которые введены и описаны внутри подпрограмм, называются ***локальными***, т.к. они локализированы внутри той подпрограммы, в которой они описаны, и как бы «невидимы» снаружи (из других подпрограмм). В то же время, параметры, которые введены и описаны в программе до описания подпрограммы, называются ***глобальными*** по отношению к этой подпрограмме, т.к. они «видимы» как внутри неё, так и в программе её вызывающей. Это позволяет использовать глобальные параметры для передачи данных.

Для примера передачи данных через глобальные параметры рассмотрим следующую программу - обработчик события нажатия кнопки:

```
Procedure TForm1.Button1Click(Sender:Tobject);
    Var z,a,b:integer;
    Procedure Pw;
        Var z:integer;
        begin
            z:=a*b;
            writeln('zPw=',z);
```

```
    end;//конец Pw
begin
  z:=1; a:=2; b:=4;
  Pw;  //вызов процедуры
  Writeln('zb=',z);
end;
```

Переменные *a* и *b* являются локальными в процедуре *TForm1.Button1 Click* но они же глобальные по отношению к процедуре *Pw*, т.к. описаны выше её описания. Процедура *Pw* «видит» переменные *a* и *b*, поэтому она напечатает *zPw=8.*

Переменная z здесь описана как внутри *Pw* так и снаружи. В этом случае под переменную *z* будет отведено две разных ячейки памяти, одна – под *z,* которая снаружи *Pw,* другая – под *z* внутри *Pw*. В результате внешняя z как бы становится «невидимой», т.е. внутри *Pw* она не изменится и второй оператор печати выведет *zb=1.*

Передача данных через формальные параметры

Формальные параметры могут быть трех разновидностей:

Параметры-значения, параметры - переменные, параметры - константы.

Параметры-значения описываются следующим образом:

имя(a,b:Tun1; c,d,e:Tun2; ...)

Для каждого формального параметра – значения транслятор внутри подпрограммы резервирует дополнительные ячейки памяти в соответствии с типом параметра. При вызове подпрограммы, происходит пересылка фактического параметра в эти ячейки памяти, после чего выполняется подпрограмма. При этом значение ячейки, где находился сам фактический параметр, не изменяется. Этот механизм обеспечивает, как, защищенность фактического параметра, так и его универсальность, т.е. то, что фактическим параметром может быть константа, переменная или арифметическое выражение.

Например:

```
Var x,u,z real;
Function sqxy(x,y:real):real;
      begin
            if x<0 then x:=0;
            if y<0 then y:=0;
            Result:=sqrt(x)+sqrt(y);
      end;
begin
      ...
      x:=-0.5;      u:=4;
```

```
        z:=sqxy(x,u);//вызов функции
        write(' x=',x,' u=',u,' z=',z);
        z:=sqxy(sin(u)+x,1.86);
        ...
end;
```

Здесь внутри функции *(x, y)* – формальные параметры, а в программе *(x, u)* и *(sin(u)+x, 1.86)* фактические.

При работе данной программы будет напечатано

x= -0.5 u=4 z=2

хотя внутри подпрограммы будет вычислено *x=0*.

Недостатком формального параметра-значения является необходимость дублирования ячеек памяти в вызывающей и в вызываемой программах, что в случае, например, параметра-массива приводит к неоправданным затратам памяти.

Параметры – константы описываются следующим образом:

```
имя(Const a,b:Tun1;  Const c,d,e:Tun2;  ...)
```

В этом случае фактическим параметром может быть переменная или константа. Для такого формального параметра новой ячейки не отводится, а при вызове подпрограммы в неё передается адрес ячейки фактического параметра, но внутри запрещены все его изменения.

Параметры – переменные описываются следующим образом:

```
Имя(Var a,b:Tun1;  Var c,d,e:Tun2;  ...)
```

В этом случае фактическим параметром может быть только имя переменной. При вызове подпрограмм передается адрес ячейки переменной, в которой находится фактический параметр, и все действия производятся над этой ячейкой. Поэтому после работы подпрограммы в ячейке фактического параметра при необходимости будет находиться результат.

Например:

```
type vek = array[1..10] of integer;
procedure sab(n:byte;  const a,b:vek;  var s:integer);
        Var i:byte;
        begin
          s:=0;
          for i:=1 to n do
                s:=s+a[i]*b[i];
        end;
        ...
        Var x,y:vek;
            sk:integer;
```

```
begin
        Read(x,y);//ввод массивов
        sab(5,x,y,sk);
        Write(sk);
end;
```

В процедуре *sab* массивы *a*, *b* описаны как константы, а переменная *s* описана как параметр-переменная. Это позволило для массивов *x*, *y* сэкономить память, а через параметр-переменную *s* вывести результат. Ячейку, выделяемую под параметр *n,* экономнее переслать саму, чем через её адрес, поэтому для нее используется параметр-значение.

9.4. Оформление подпрограмм в библиотечный модуль

Набор подпрограмм, которые могут быть использованы при разработке целого ряда программ удобно оформить в виде отдельных тематических библиотек. В разные библиотеки обычно собираются подпрограммы алгоритмов решения задач по определенной теме, например: вычисления всевозможных арифметических функций, обработка массивов и матриц, решение уравнений и др. Для организации таких библиотек в Pascal введены модули.

Модуль (Unit) – это автономно компилируемая программная единица, структура которой представлена на Рис. 9.2. Модуль состоит из *интерфейсной части,* т.е. доступной для других программ, и *исполняемой части,* которая скрыта. В интерфейсной части располагаются необходимые константы, переменные и все заголовки подпрограмм, помещенных в данную библиотеку. В исполняемой части располагаются описания констант, переменных и подпрограмм.

Структура модуля

```
Unit <имяМ>;

        interface
Uses имя1,имя2,. . .
Const . . .
type . . .          //  описания, «видимые» из  программ в которых
Var . . .             //  подключен данный модуль
Procedure имяП(. . .);
Function имяФ(. . .):типФ;

        implementation    //   Исполняемая часть
Uses имя3,...;
Const . . .
type  . . .
Var . . .             // описания, которые используются
Procedure ИПУ(. . .);  // только внутри библиотечного   модуля
```

```
        begin
        . . .
        end;

Procedure ИП;
        Begin
   . . .
        ИПV(. . .);
        . . .           //  описание подпрограмм, заголовки которых
        end;            //  помещены в интерфейсной части
Function ИФ;
        begin
        . . .
        end;

      Initialization // Инициирующая часть, например:
Var Lw:textfile;      // описание действий, которые
Assign(Lw,'Modul'); //  выполняются в начале работы программы
Rewrite(Lw);

      Finalization    // Завершающая часть, например:
Closefile(Lw);      //  описание действий, которые выполняются
                        // в момент завершения работы программы
end.
```

Рис.9.2.

Кроме того, модуль может содержать *инициирующую* и *завершающую* части, которые, однако, редко используются.

Заголовок модуля состоит из слова ***Unit*** и следующего за ним имени модуля, которое служит для связи с другими модулями и основной программой. Имя модуля должно совпадать с именем файла на диске, в который помещается исходный текст модуля. Подключение модулей к разрабатываемой программной единице осуществляется с помощью оператора

Uses имяМ1 , имяМ2, ... ;

который должен стоять вначале раздела описаний, т.е. сразу после заголовков ***Program***, ***Interface*** или ***Implementation*** (см. Рис. 9.2).

После подключения модуля в разрабатываемой программе становятся доступными все конструкции, описанные в интерфейсной части модуля.

Для создания нового модуля в Delphi следует в меню ***File*** выбрать *File/New* и затем, в открывшемся репозитории, выбрать пиктограмму ***Unit*** или ***Form*** (во втором случае создается скелет модуля со связанным с ним окном). После этого модуль следует наполнить содержанием, записать, исполь-

зуя ***Save As*** меню ***File*** и добавить в проект основной программы, используя опцию ***Add to Project*** меню ***Project*** [1].

9.5. Примеры подпрограмм, оформленных в отдельные библиотечные модули

Модуль для работы с массивами

```
Unit RabMas;
            interface
const mmax=10;
type vek=array[1..nmax] of extended;
     mat=array[1..nmax] of vek;

//Процедура умножения двух квадратных (n×n) матриц A и B
Procedure Pmat(const A,B:mat; Var C:mat; n:Word);

    // Процедура вывода матрицы A в текстовый файл Lw
Procedure Vivod(const A:mat; n,m:Word; Var Lw:TextFile);

                implementation
Procedure Pmat;  //  повторение заголовка без параметров
    Var i,j,k:Word;
    begin
    for i:=1 to n do
      for j:=1 to n do
        begin
        c[i,j]:=0;
        for k:=1 to n do
            c[i,j]:=c[i,j]+a[i,k]*b[k,j];
        end;
    end;

Procedure Vivod;  // повторение заголовка
     Var i,j:Word;
    begin
    for i:=1 to n do
        begin
        for j:=1 to m do
            Write(Lw,a[i,j]:8:4);
        Writeln(Lw);
        end;
    end;

    end.
```

Пример программы, использующей модуль RabMas

```
Unit P1;
      Interface
. . .
    Implementation
Uses RabMas;
    ...
Procedure TForm1.Button1Click(Sender:Tobject);
Var x,y,z:mat;
    n:word;    f:textfile;
begin
    n:=. . .
    x[i,j]:= . . .//вычисления или ввод матриц x, y
    y[i,j]:= . . .
    Assignfile(f,'Mass.out');
    Rewrite(f);
    Writeln(f,'Полученная матрица:');
    Pmat(x,y,z,n);      //   вычисления z=x·y
    Vivod(z,n,n,f);     //   вывод z в файл f
    Closefile(f);
End;
```

Модуль для выдачи таблицы значений функций

```
Unit Tab;
            interface
 type Tfun = function(x:extended):extended;
 Procedure Tabf(a,b:extended; n:word; f:Tfun;
         var Lw:textFile);

           implementation
Procedure Tabf;
    Var x,h:extended;
        i:word;
   begin
    h:=(b-a)/n;   x:=a;
    for i:=0 to n do
        begin
          writeln(Lw, x:6:2, f(x):8:4);
          x:=x+h;
        end;
     end;
   end.
```

Пример программы, использующей модуль Tab

```
Unit unit1;
                interface
  ...
             implementation
Uses Tab;
Var M:word;
  //    Вычисление  s(x) = Σ_{n=0}^{M} (-1)^n cos(nx)/n!
Function Sum(x:extended):extended,
     Var s,w:extended;
         n:word;
     begin
     w:=1;    s:=1;
     for n:=1 to M do
         begin
         w:=-w/n;
         s:=s+cos(n*x)*w;
         end;
      result:=s;
     end;

Procedure TForm1.Button1Click(Sender:Tobject);
    Var f:textFile;
begin
     Assignfile(f,'P2.out');
     Rewrite(f);    M:=8;
     writeln(f,'  x          sum(x)  ');
     Tabf(0,2,5,sum,f);
     CloseFile(f);
end;
```

В результате работы данной программы в текстовом файле P2.out будет распечатана таблица:

x	**sum(x)**
0.00	**0.0508**
0.40	**0.3254**
0.80	**0.7122**
1.20	**0.8676**
1.60	**1.8711**
2.00	**2.6998**

ЛЕКЦИЯ 10. ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ ТИПА МНОЖЕСТВА

10.1. Понятие множества

В математике под множеством понимается неупорядоченный набор различных однотипных элементов. Весь набор элементов $\Omega=\{\omega_1, \omega_2, ... \omega_n\}$ называется пространством элементов. Различные наборы, например:

$A=\{\omega_2, \omega_4, \omega_1, \omega_5\}$, $B=\{\omega_4, \omega_1, \omega_3\}$, ... называется множествами.

Пустое множество не содержит ни одного элемента.

Для работы с множествами в Pascal введен специальный тип переменных – ***set of***:

Type <имя типа> = set of <базовый тип>;

Var a, b, c : <имя типа>;

здесь *<базовый тип>* - любой порядковый тип, кроме *word, integer, longint*, т.е. *(перечисляемый, интервальный, char, byte, boolean)*.

Максимальное количество элементов в множестве – 256.

Примеры описаний множеств:

```
type   bukva=set of ('a'..'z');
       simv=set of char;
       cifra=set of byte;

Var    a, b : bukva;
       c, d : simv;
       e, g : cifra;
Begin
       c:=['u', 'v', 'z'];   // Задать множество из трех букв
       e:=[1,2,0];           // Задать множество из 3 чисел
       g:=e+[5];             // Добавить в множество E элемент 5
       d:=[ ];               // Пустое множество
End;
```

10.2. Операции над множествами

К множествам применимы операции:

Объединение множеств (+): $D=A+B=\{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5\}$, множество D состоит из элементов, входящих или в A или в B.

Разность множеств (-): $E=A-B=\{\omega_2, \omega_5\}$, множество E состоит из элементов множества A, не входящих в B.

Пересечение множеств (*): $F=A*B=\{\omega_1, \omega_4\}$, множество F состоит из элементов, одновременно входящих в A и в B.

Равенство (=): *A=B,* результат равен *true,* если множества *A* и *B* состоят из одних и тех же элементов.

Неравенство (<>): *A <> B,* результат равен *true* в противном случае.

Подмножество (<=): *A<=B,* результат равен *true,* если все элементы множества *A* являются элементами множества *B*.

Надмножество (>=): *A>=B,* результат равен *true,* если все элементы множества *B* являются элементами множества *A*.

Принадлежности (in): *ω* in *A,* результат равен *true*, если элемент *ω* входит в множество *A*.

Кроме того, эти операции дополняют две процедуры:

***Include (s, i)*;** - добавляет в множество *s* элемент *i*;

Exclude (s, i); - исключает из множества *s* элемент *i*.

Элемент *i* должен быть базового типа. Эти операции выполняются значительно быстрее, чем их эквиваленты *s:=s+[i]; s:=s - [i];*

10.3. Примеры работы с множествами

1. Ввод *n* элементов множества:

```
Var  a : set of char;
     s : char;
     n : word;
begin
  a:=[ ];   // пустое множество
  for i:=1 to n do
        begin
          read(s);
          a:=a+[s];
        end;
end.
```

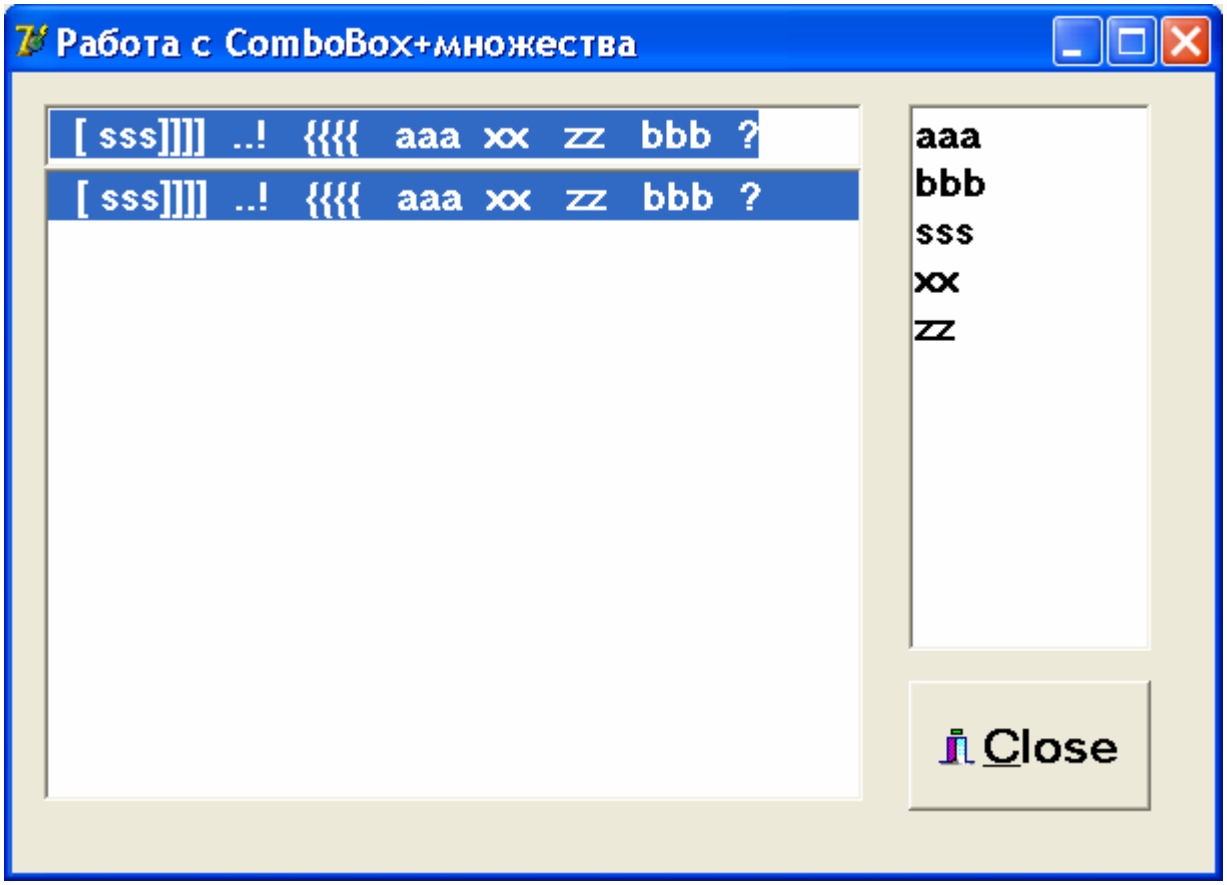
2. Распечатать содержимое множества:

```
Var  b : set of 1..100;
     k : byte;
begin
  . . .
  for k:=1 to 100 do
       if  k in b  then Writeln(k);
end.
```

3. Использование множеств в ряде случаев позволяет в более компактном виде записать проверку условия, например, вместо оператора:

```
if  (k=5) or  (k=1)  or  (k=8)  or (k=12)  then . . .
    записать:     if  k  in  [5, 1, 8, 12]   then . . .
```

4. Выделить отдельные слова из строки символов и вывести их в алфавитном порядке. Пример программы приведен ниже.




```
unit Unit1;
                        interface
uses Windows,Messages,SysUtils,Variants,Classes,
Graphics,Controls,Forms,Dialogs,StdCtrls,Buttons;
type
   TForm1 = class(TForm)
   ComboBox1: TComboBox;
   BitBtn1: TBitBtn;
   Memo1: TMemo;
   procedure FormActivate(Sender: TObject);
   procedure ComboBox1KeyPress(Sender: TObject; var
Key: Char);
    procedure ComboBox1Click(Sender: TObject);
        private
        { Private declarations }
    public
        { Public declarations }
  end;

var  Form1: TForm1;
```

```
implementation

{$R *.dfm}

procedure TForm1.FormActivate(Sender: TObject);
begin
   combobox1.setfocus;
   memo1.Clear;
end;

procedure TForm1.ComboBox1KeyPress(Sender: TObject;
      var Key: Char);
begin
  if key=#13 then
     begin
     combobox1.items.add( combobox1.text);
     combobox1.text:='';
     end;
end;

procedure TForm1.ComboBox1Click(Sender: TObject);
   const   c : set of char = [' ', '.', ',', ':',
';', '(', ')', '?', '!' ,'[', ']', '{', '}'];
                                                  //
список разделителей
         var s : string;
               n ,k, i : integer;
                    a : array[1..100] of string;
 begin
 s:= combobox1.text;       memo1.clear;
 s:=s+' ';     n:=0;
 while s <> '' do
   begin  // удаление начальных разделителей
   while  (s  <>  '')  and  (s[1]  in  c)  do  de-
lete(s,1,1);
   if s <> '' then
      begin
      for k:=1 to length(s) do
          if s[k] in c then break;
      n:=n+1;
      a[n]:=copy(s,1,k-1);
      delete(s,1,k);
      end;
   end;
```

```
            // Сортировка по алфавиту
     for i:=1 to n-1 do
       for k:=1 to n-i do
          if a[k]>a[k+1] then
              begin
              s:=a[k];
              a[k]:=a[k+1];
              a[k+1]:=s;
              end;
     for k:=1 to n do   memo1.lines.add(a[k]);
   end;
end.
```

ЛЕКЦИЯ 11. ИСПОЛЬЗОВАНИЕ СТРОКОВЫХ ДАННЫХ

11.1. Зачем нужны строки

Уже в первой лабораторной работе мы столкнулись с понятием строки, когда использовали «окно однострочного редактора» *Edit*, «окно многострочного редактора» *Memo*. И мы уже знаем, что *строка – это последовательность символов.* В системе Delphi имеется удобный ввод и отображение информации из окон (или в окна) этих редакторов в виде строк. Т.е. вводится информация в виде *строки символов* и результат отображается в виде *строки символов*. Вы уже почувствовали, что работе со строками отводится важное место в языках программирования. Так было не всегда. Первые версии языков программирования, как и Фортран, и Бэйсик, были ориентированы на разработку научных вычислительных программ. Понятие строки, заключенной в апострофы, (например, *'x='*), использовалось лишь для наглядного представления выводимой информации. Однако, с широким распространением ПЭВМ появилась насущная потребность во всевозможных программах обработки текстов. С другой стороны, с развитием компьютерной техники и языков высокого уровня, возникло понимание того, что можно создать язык высокого уровня, позволяющий писать программы обработки и редактирования текстов, компиляторы для новых языков.

Как решить эту проблему? Казалось бы, просто: ввести возможность работы с символами, что было и сделано. В языке Pascal, а затем и в Си был введен тип *char*, с которым вы знакомы. Массивы из символов

`a: array [1..N] of char;` в принципе решают проблему обработки символьной информации. Однако, массив оказался не очень удобной конструкцией в основном из-за фиксированной структуры и сложности выполнения операций над ним. В результате творческих поисков в языках появился новый изящный тип переменных: *строковый*. Для того, чтобы овладеть навыками работы с этим типом необходимо знать ответы на следующие вопросы.

Как описываются переменные строкового типа ?

Какие операции над переменными этого типа возможны?

Как решать задачи с использованием строковых переменных?

11.2. Описание переменных строкового типа *«Короткие строки»*

В стандартном Pascal, вплоть до Delphi 1 использовались только «короткие строки», длина которых ограничена 255 символами. В последующих версиях Delphi они описываются следующим образом:

Var sk:string[N]; N≤255 или

Var sk:shortstring; - короткая строка максимальной длины, эквивалент
Var sk : string[255];

Рассмотрим структуру памяти коротких строк. На этапе трансляции под переменную `sk` будет отведено *N+1* байтов. После выполнения оператора *`sk:='abc';`* в этих ячейках символы будут размещаться следующим образом:

	0	1	2	3	4			N
Sk:	3	a	b	c	не	испо	льзо	ваны

sk:='d';

в этих ячейках будет:

	0	1	2				N
Sk:	1	d	не	испо	льзо	ва	ны

При работе со строковой переменной имеется возможность обращаться к отдельным символам строки, как элементам одномерного массива sk.

`Ch:=sk[i];  1≤i≤N,  (var ch:char;)`

«Длинные строки»

При внимательном рассмотрении можно увидеть, что «короткие строки» нерационально используют память и ограничены (<=255). В эту строку не поместишь достаточно большой текст, что бывает неудобно. В последних версиях Delphi введены так называемые «длинные строки». Для их описания используется ключевое слово *string:*

`var sd: string;`

«Длинные строки» являются динамическими переменными (указателями), память для которых (до 2 Гбайт) выделяется по мере надобности на этапе выполнения программы.

«Широкие строки»

Они отличаются от «длинных строк» только тем, что каждый символ строки кодируется не одним, а двумя байтами памяти (*ANSI* кодировка позволяет работать только с 256 символами, что явно недостаточно, например, при работе с китайским или японским алфавитом). Для описания «широких строк» используется ключевое слово *WideString:*

var ss: WideString;

Нуль – терминальные строки

Строковые переменные этого типа представляют собой цепочки символов, ограниченные символом конца строки #0. Для их описания используется ключевое слово *PChar:*

var sc: PChar;

Необходимость в нуль - терминальных строках возникает только при прямом обращении к *API* - функциям ОС. При работе с компонентами Delphi в основном используются более удобные короткие и длинные строки.

11.3. Основные операции над переменными строкового типа

Над данными строкового типами допустимы операции присваивания (:=), сцепления (+) и операции отношения (=, <>, >, <, >=, <=).

Операция сцепления используется для объединения нескольких строк в одну. Операции отношения проводят сравнение двух строковых операндов. Сравнение строк проводится посимвольно, слева направо, до первого несовпавшего символа и та строка считается больше, в которой первый несовпавший символ имеет больший кодовый номер.

Например:

'ax' < 'bcd' потому что 'a' < 'b'

'Попов' > 'Панин' потому что 'о'>'а'

11.4. Некоторые процедуры и функции обработки строк

В Delphi имеется широкий набор специальных процедур и функций обработки строк, с помощью которых можно запрограммировать всевозможные редакторы и обработчик текстов. Приведем здесь некоторые из них.

Процедуры:

`Delete(S:String; pz,n:integer);` - удаляет из строки s n символов, начиная с позиции pz, например:

```
S:='florex';
Delete(S,4,2);     // результат: S='flox'
```

`Insert(Sv,S:string; pz:integer);` - вставляет строку *Sv* внутрь строки *S,* начиная с позиции *pz*, например:

```
S:='это символы';
Insert('не те ',S,5);  // результат: S='это не те символы'.
```

`Str(x[:ширина:десятич. знаков]; S:String);` преобразует число x целого или действительного типа в строковое представление, например:

```
x:=25.4e-1;   k:=86;
Str(x:8:4,S);  // результат: S='  2.5400'
Str(x,S);      // результат: S='2,540000e+00'
Str(k:3,S);    // результат: S=' 86'
```

`Val(S:string; x:(действит. или целое); Kod:integer);` -

преобразует строковое представление числа в числовое в соответствии с типом переменной *x, Kod* – код ошибки, *Kod = 0* – успешно, или *Kod=номеру позиции* в строке где обнаружен символ не соответствующий числовому представлению, например:

```
S1:='2e-5'; S2:='468'; S3='0..256'
```

```
Val(S1,X,kod);   // результат: x=2·10-5,   kod = 0
Val(S2,K,kod);   // результат: k=486,      kod = 0
Val(S3,X,kod);    // результат: x не изменяется,   kod=3
```

Функции:

`Copy(S,pz,n)` - выделяет из строки *S* подстроку из *n* символов, начиная с позиции *pz,* при этом *S* сохраняется, например:

```
S:='d:\pas\prog.pas';
Sp:=Copy(S,8,4);      // результат Sp='prog'.
```

`Length(S)` - *определяет текущую длину строки S.*

`pos(Sp,S)` - определяет номер позиции, начиная с которой внутри строки *S* расположена подстрока *Sp,* (если внутри *S* нет подстроки *Sp*, тогда результат=0), например:

```
pz:=pos('pas',S); // для вышеприведенной S  pz=4.
```

`strToFloat(S)` - переводит строковое представление в действительное число (аналог процедуры *Val*).

`strToInt(S)` - переводит строковое представление в целое число.

`FloatToStrF(x,fffixed,8,4)` - переводит действительное число в строковое представление с форматом *x:8:4*, (аналог процедуры *Str*).

`IntToStr(m)` - переводит целое число в строковое представление, причем длина строки равна количеству десятичных цифр с учетом знака.

11.5. Примеры алгоритмов обработки строк

Примеры использования функций преобразования строк для реализации ввода/вывода приведены в Лекции 2.

Задача 1. Заменить в строке букву ‘н’ на ‘ф’.

```
function zam(s:string):string;
        var i: word;
    begin
    for i:=1 to length (s) do
        if s[i]='н' then s[i]:='ф';
     result:=s;
    end;
```

Задача 2. Отсортировать массив строк, содержащий список сотрудников, по алфавиту.

```
for i:=1 to n-1 do
    for j:=1 to n-i do
        if A[j]>A[j+1] then
                begin
                s:=A[j];
```

```
A[j]:=A[j+1];
A[j+1]:=s;
end;
```

Задача 3. Посчитать количество цифр в строке.

```
Function Kol(s:string): byte;
    var i:word;
  begin
    result:=0;
    for i:=1 to length(s) do
     if s[i] in ['0'..'9'] then inc(result);
  end;
```

Задача 4. Подсчитать количество различных символов в строке.

```
 Function Kol(S:string):Word;
    Var   a:set of char;
       C:char;
       i,m:Word;
    begin
     m:=0; a:=[];
     for i:=1 to Length(S) do
       begin
        C:=S[i];
        if not(C in a) then
       begin
        a:=a+[C];
        m:=m+1;
       end;
       end;
    Result:=m;
    end;
```

ЛЕКЦИЯ 12. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ЗАПИСЕЙ

12.1. Понятие записи

Удобство использования структурированных типов, когда под одним именем объединяется целый набор данных, было продемонстрировано ранее на примере массивов, строк и множеств. Указанные типы объединяют под одним именем одинаковые данные. В отличие от них записи позволяют объединить под одним именем разнотипные данные, что открывает более широкие возможности для составления программ.

Запись – это структура данных, в которой под одним именем объединены переменные, в общем случае разного типа, которые называются ***полями***.

Запись в Delphi описывается в общем случае, следующим образом:

```
Type
      Tzap=record
       a,b,c: Typ1;        // Разделы фиксированной части
       e,f: Typ2;          // записи
            ...
       z,x,y,u: Typn;
       case byte of
        1:(d:Typ01;        // Разделы вариантной части записи
           g:Typ02);
           ...
        m:(p,q:Typ1m)
      end;
Var Z1,Z2:Tzap;
```

Здесь *Z1, Z2* – записи, *a, b, . . , p, q* – поля этих записей. Все поля в одной записи должны иметь различные имена. Записи в общем случае имеют фиксированную и вариантную части. Вариантная часть открывается предложением ***case*** *<перечисляемый тип>* ***of***, и всегда последняя. В ней указывается несколько вариантов полей, каждый из которых заключен в скобки, перед которыми стоит значение константы, имеющей тип, указанный в предложении case. Всем вариантам отводится одна и та же область памяти, объем которой равен максимальному из объемов вариантов полей. Возможны записи, имеющие только фиксированную часть (отсутствует вариантная часть) или имеющие только вариантную часть (отсутствует фиксированная часть).

Запись - константа вводится следующим образом:

```
Type Point=Record x,y:real; end;
Const w:Point=(x:1.5; y:8.4);
```

12.2. Операции над записями

Для однотипных записей допускается оператор присваивания:

Z1:=Z2;

при выполнении которого всем полям записи *Z1* присваиваются значения соответствующих полей записи *Z2*.

К каждому полю записи можно получить доступ через составное имя, представляющее собой разделенные точкой имя записи и имя поля (составное имя), например: если *Typ1=real:*

Z1.a:=25.86; Z2.b:=Z1.a;

Если поле, в свою очередь, имеет тип записи (вложенность записей), тогда для конкретизации потребуется составное имя с двумя точками. Например, если *Typ2* определяется как

```
Type
    Typ2=record
           Re,im:extended;
         end;
```

то, полям *re* и *im* можно присвоить некоторое значение следующим образом:

Z1.e.re:=5.1; Z1.e.im:=0.8;

Чтобы упростить доступ к полям, используется оператор присоединения

```
With <имя записи> do
    begin
      <операции с полями записи>
    end;
```

Например, присвоить значения полям *re* и *im* можно и так:

```
With Z1.e do begin re:=5.1;  im:=0.8; end;
```

То обстоятельство, что все варианты записи помещены в одном месте памяти позволяет использовать эффект наложения друг на друга переменных различных типов.

В нижеприведенном примере происходит наложение одномерного массива на двумерный:

```
Type  zap=record
        Case boolean do
          true: (a:array[1..4] of integer);
          false:(b:array[1..2,1..2] of integer);
        end;
Var z:zap;
          ...
    for i:=1 to 4 do z.a[i]:=i;
    Writeln(z.b[2,1]);    // Будет выведено число 3.
```

12.3. Использование записей для работы с комплексными числами

Комплексные числа представляют собой расширенное понятие действительных чисел. Исторически они возникли при решении алгебраических уравнений. Например, решение уравнения $x^2 - x + 1 = 0$:

$$x_{1,2} = \frac{1 \pm \sqrt{1-4}}{2} = \frac{1 \pm \sqrt{-3}}{2} = \frac{1 \pm \sqrt{3} * \sqrt{-1}}{2}$$

приводит к выражению вида $u + \sqrt{-1} * v$. Оказывается, к аналогичным выражениям приводит решение многих задач.

Обозначив $\sqrt{-1} = i$, получим $i^2 = -1 < 0$. Но мы знаем, что квадрат обычного числа >0. Выход из этой ситуации нашли, введя понятие «мнимая единица» и обозначив ее $i = \sqrt{-1}$. В результате выражение $\dot{a} = u + \sqrt{-1} * v$ принимает вид $\dot{a} = u + iv = a_{re} + ia_{im}$ и называется комплексным числом (числом, имеющим действительную (a_{re}) и мнимую (a_{im}) части). Точка сверху отличает его от действительного.

Геометрическая интерпретация комплексного числа - точка с координатами (a_{re}, a_{im}) на комплексной плоскости позволяет также однозначно определить комплексное число через ***модуль*** (расстояние от начала координат до точки) и ***фазу*** комплексного числа (угол между радиус – вектором точки и положительным направлением оси абцисс). Таким образом, комплексное число можно записать двумя способами:

$\acute{\alpha} = a_{re} + ia_{im} = ae^{j\varphi} = a(\cos\varphi + i\sin\varphi)$,

$a = \sqrt{a_{re}^2 + a_{im}^2}$, $\varphi = \operatorname{arctg}(a_{im} / a_{re}) + 2k\pi$,

$a_{re} = a*\cos\varphi$, $a_{im} = a*\sin\varphi$.

Операции над комплексными числами

Рассмотрим два числа $\acute{\alpha}$ и $\acute{c}$:

$\acute{\alpha} \pm \acute{c} = (a_{re} \pm c_{re}) + i(a_{im} \pm c_{im})$

$\acute{\alpha} * \acute{c} = (a_{re} + ia_{im})(c_{re} + ic_{im}) = (a_{re}c_{re} - a_{im}c_{im}) + i(a_{re}c_{im} + a_{im}c_{re})$

$\acute{\alpha} / \acute{c} = ((a_{re}c_{re} + a_{im}c_{im}) + i(a_{im}c_{re} - a_{re}c_{im}))/(c^2_{re} + c^2_{im})$

В языке Pascal отсутствуют специальные встроенные операции с комплексными числами. Однако этот недостаток легко можно компенсировать, если дополнить свою библиотеку специальным модулем *(Unit Cmplx)* в котором все операции под комплексными числами и функциями комплексной переменной оформлены в виде набора функций и процедур. Начальный фрагмент такого модуля приведен ниже. При желании, его легко дополнить всеми необходимыми для решения задач функциями. В этом модуле тип комплексных чисел – *complex* вводится, как запись, имеющая два поля с именами *re* и *im*. Первое поле трактуется как действительная часть комплексного числа, второе - как мнимая часть.

```
Unit Cmplx;
                Interface
Type Complex=record re,im:extended; end;
function Addc(x,y:Complex):Complex;
function Mulc(x,y:Complex):Complex;
function Divc(x,y:Complex):Complex;

               Implementation
function Addc;  // x+y
     begin
     Addc.re:=x.re+y.re;
     Addc.im:=x.im+y.im;
     end;

function Mulc;   // x*y
     begin
     Mulc.re:=x.re*y.re-x.im*y.im;
     Mulc.im:=x.re*y.im+x.im*y.re;
     end;

function Divc;    // x/y
       var z:extended;
     begin
     z:=sgr(y.re)+sgr(y.im);
     Divc.re:=(x.re*y.re+x.im*y.im)/z;
     Divc.im:=(x.im*y.re-x.re*y.im)/z;
     end;
end.
```

Предположим, что требуется ввести два комплексных числа *a, b* и рассчитать $u=(a+b)/(a \cdot b)$.

Фрагмент вычислений с комплексными числами при использовании этого модуля выглядит следующим образом:

```
Uses Cmplx;
  Var a,b,u:Complex;
begin
     a.re:=1.2; a.im:=0.8;
     b.re:=0.5; b.im:=0;
     u:=Divc(Addc(a,b),Mulc(a,b));
     memo1.lines.add(floattostr(u.re)+'    '+
           floattostr(u.im));  // вывод в memo1
end;
```

ЛЕКЦИЯ 13. ИСПОЛЬЗОВАНИЕ ФАЙЛОВ

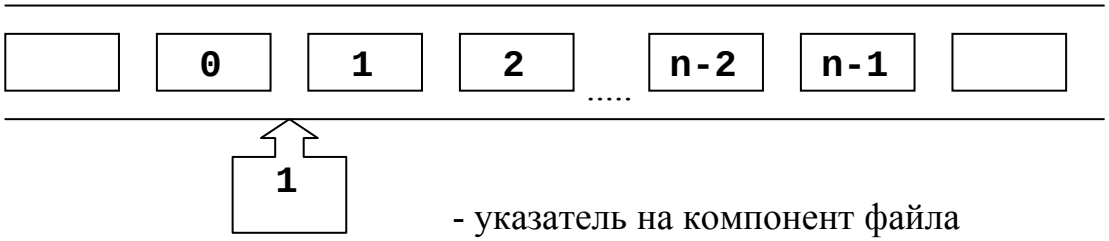
13.1. Понятие файла

В программировании слово *файл* встречается очень часто. Вы с этим понятием познакомились уже на первой лекции и знаете, что файлы предназначены для размещения данных на внешних носителях (обычно магнитных дисках), и последующей работы с этими размещенными данными. В языке Pascal для организации и последующей работы с файлами предусмотрен специальный файловый тип переменных. Операциями над переменными файлового типа соответствуют определенные действия над внешними носителями (дисками, магнитными лентами, принтерами).

Переменная файлового типа, или коротко файл, в языке Pascal представляет последовательность однотипных компонент, соответствующих последовательности записей на внешнем носителе. В отличие от массива, количество компонент заранее не оговаривается, и компоненты файла не имеют индексов. Файловые переменные в Delphi вводятся следующим образом:

```
Var
    ft1 ,ft2 : File of <тип компонент>;     // типизированные файлы
    Lw, Lr : TextFile;                      // текстовые
    f1, f2 : File;                          // нетипизированные
```

Объясняя принципы работы с файлами, можно для наглядности считать, что каждый файл записан на некоторой магнитной ленте, как это показано на следующем рисунке:



- указатель на компонент файла

n – количество записанных компонент.

Указатель определяет положение магнитной головки магнитофона, с помощью которой осуществляется покомпонентная запись или чтение информации. В начале файла записана информация о файле *BOF* (*Begin of File*), его имя, тип, длина и т.д., в конце файла помещается признак конца файла *EOF (End of File)*. Если файл пуст, то *BOF* и *EOF* совмещены, а указатель установлен в нуль. Если файл не пуст, то указатель совмещен либо с началом некоторой компоненты и его значение равно номеру этой компоненты (нуме-

рация начинается с нуля), либо указатель совмещен с признаком конца и его значение равно количеству компонент.

13.2. Операции над файлами

13.2.1. Типизированные файлы

Пусть *f* – имя типизированного файла, а переменные *x, y, z* имеют тот же тип, что и его компоненты.

```
Type
   Typ = < тип компонента файла f >;
Var
   f:file of Typ;
     x,y,z:Typ;
```

Начинается работа над файлами с процедур открытия файла:

```
AssignFile(f, <имя файла>:String);
Reset(f);  или  Rewrite(f);
```

Процедура *AssignFile()* устанавливает соответствие между файловой переменной *f* и *<именем файла>* на внешнем носителе. Процедуры *Reset(f)* и *Rewrite(f)* инициируют (подготавливают) файл для работы. При этом, если файл на внешнем носителе отсутствует, то следует использовать оператор *Rewrite(f)*, который создает новый файл с именем *<имя файла>*. Если требуется работать с уже существующим файлом, то необходимо использовать оператор *Reset(f)*. После выполнения процедур открытия файла указатель всегда установлен в начало файла (на компонент с номером 0). Если открытие производится оператором *Rewrite(f)*, то признак конца файла совмещен с началом (т.е. файл пуст). Если перед этим в файле имелась некоторая информация, то она стирается. Запись значений переменных в файл производится покомпонентно с помощью оператора

```
Write(f,x);
Write(f,y,z);
```

После записи каждой переменной значения указатель увеличивается на единицу, что соответствует его перемещению к следующему компоненту файла. Если перед записью указатель находился напротив признака конца файла, то при записи каждой переменной признак конца смещается на длину этой записи и количество компонентов в файле возрастает на единицу.

Количество компонентов, записанных в файле, можно определить с помощью функции `FileSize(f)`.

Чтение значений переменных из компонентов файла производится с помощью оператора

```
Read(f,x);
Read(f,y,z);
```

При чтении каждой переменной указатель увеличивается на единицу. Если производится попытка чтения при указателе, установленном на конец файла, то происходит останов программы по ошибке чтения.

Распознать ситуацию, когда указатель установлен на конец файла, можно с помощью функции `Eof(f)`, возвращающей значение `True`, если достигнут конец файла, и `False`, в противном случае.

Организовать чтение с проверкой этого условия можно, например, следующим образом:

```
if not Eof(f) then Read(f,x);
```

При необходимости чтения или записи заданного компонента файла нужно предварительно указатель установить на номер этого компонента. Это делается с помощью процедуры `Seek(f,<номер компонента>)`.

Например, при выполнении следующей группы операторов

```
Seek(f,2);
Read(f,x);
Write(f,y);
```

Переменная *x* будет прочитана из компонента с номером 2 (третьего), а переменная y запишется в компонент с номером 3 (четвертый).

Текущее положение указателя можно узнать с помощью функции `FilePos(f)`.

После окончания работы с файлом его следует обязательно закрыть с помощью процедуры

```
CloseFile(f).
```

Если этот оператор отсутствует, то из-за специфики обмена данными с файлом, часть информации, помещенной в файл, может быть утеряна.

13.2.2. Текстовые файлы

Для удобства хранения текстовой информации, т.е. представленной последовательностью строк символов, в языке Паскаль введены файловые переменные текстового типа (см. выше *Lw, Lr)* или просто текстовые файлы. Текстовые файлы можно представлять эквивалентным типизированным файлом, компонентами которого являются символы. Однако в текстовом файле последовательность символов разбита на строки различной длины, т.е. в конце каждой строки ставится специальный признак *EOLN (End of LiNe)*, а в конце файла признак конца файла *EOF*. При просмотре такого файла текстовым редактором на экране возникает естественная «книжная» страница текста, которую затем можно отредактировать. При работе с текстовым файлом следует помнить, что в отличие от типизированного, после открытия файла процедурой *Reset (Lr)* разрешается только чтение

```
Read(Lr,a,b);
Readln(Lr,c);
Readln(Lr);
```

После открытия файла процедурой *Rewrite (Lw)* разрешается только запись в него, при этом предыдущий файл стирается. Для сохранения предыдущей информации в текстовом файле его следует открывать процедурой *Append (Lw)*. В этом случае указатель устанавливается на конец файла и последующие записи добавляют новые строки в конец файла.

Для записи используются операторы:

```
Write(Lw,a,b);
Writeln(Lw,c:8,'переменная d=',d:10:2);
```

Для текстовых файлов введены модификации операторов ввода/вывода *Readln, Writeln,* которые осуществляют переход на следующую строку после чтения или записи указанных в операторе переменных, при этом оператор *Writeln* записывает символ конца строки *EOLN*.

Заметим, что тип переменных *a, b, c* здесь может быть любого скалярного типа. При выводе широко используются форматирование и автоматически происходит преобразование чисел в их строковые представления. При вводе, строковые представления чисел, разделенные пробелами автоматически преобразуются в числовые представления в соответствии с типом переменных.

13.2.3. Нетипизированные файлы

Описываются: `var f:File;`

Открываются процедурами `Reset(f,size)` или `Rewrite(f,size)`, где `Size` – размер блока в байтах.

Тип компонентов этих файлов заранее не оговорен и считается последовательностью байт. Поэтому их можно использовать для обработки файлов любого типа, а так же для организации высокоскоростного обмена между дисками и памятью.

13.3. Подпрограммы работы с файлами

`AssignFile(var F; FileName: string)` - связывает файловую переменную *F* и файл с именем *FileName.*

`Reset(var F[: File; RecSize: word])` - открывает существующий файл. При открытии нетипизированного файла *RecSize* задает размер элемента файла.

`Rewrite(var F[: File; RecSize: word])` - создает и открывает новый файл.

`Append(var F: TextFile)` - открывает текстовой файл для дописывания текста в конец файла.

`Read(F,v1[,v2,…vn])` - чтение значений переменных начиная с текущей позиции для типизированных файлов и строк для текстовых.

`Write(F,v1[,v2,…vn])` - запись значений переменных начиная с текущей позиции для типизированных файлов и строк для текстовых.

`CloseFile(F)` - закрывает ранее открытый файл.

Rename(var F; NewName: string) - переименовывает неоткрытый файл любого типа.

Erase(var F) - удаляет неоткрытый файл любого типа.

Seek(var F; NumRec: Longint) - для нетекстового файла устанавливает указатель на элемент с номером *NumRec.*

Truncate(var F) - урезает файл, начиная с текущей позиции.

IoResult: integer - код результата последней операции ввода-вывода.

FilePos(var F): longint - для нетекстовых файлов возвращает номер текущей позиции. Отсчет ведется от нуля.

FileSize(var F): longint - для нетекстовых файлов возвращает количество компонентов в файле.

Eoln(var F: TextFile): boolean - возвращает *True*, если достигнут конец строки.

Eof(var F)): boolean - возвращает *True*, если достигнут конец файла.

SeekEoln(var F: TextFile): boolean – возвращает *True*, если пройден последний значимый символ в строке или файле, отличный от пробела или знака табуляции.

SeekEof(var F: TextFile): boolean - то же, что и *SeekEoln,* но для всего файла.

13.4. Компоненты *OpenDialog* и *SaveDialog*

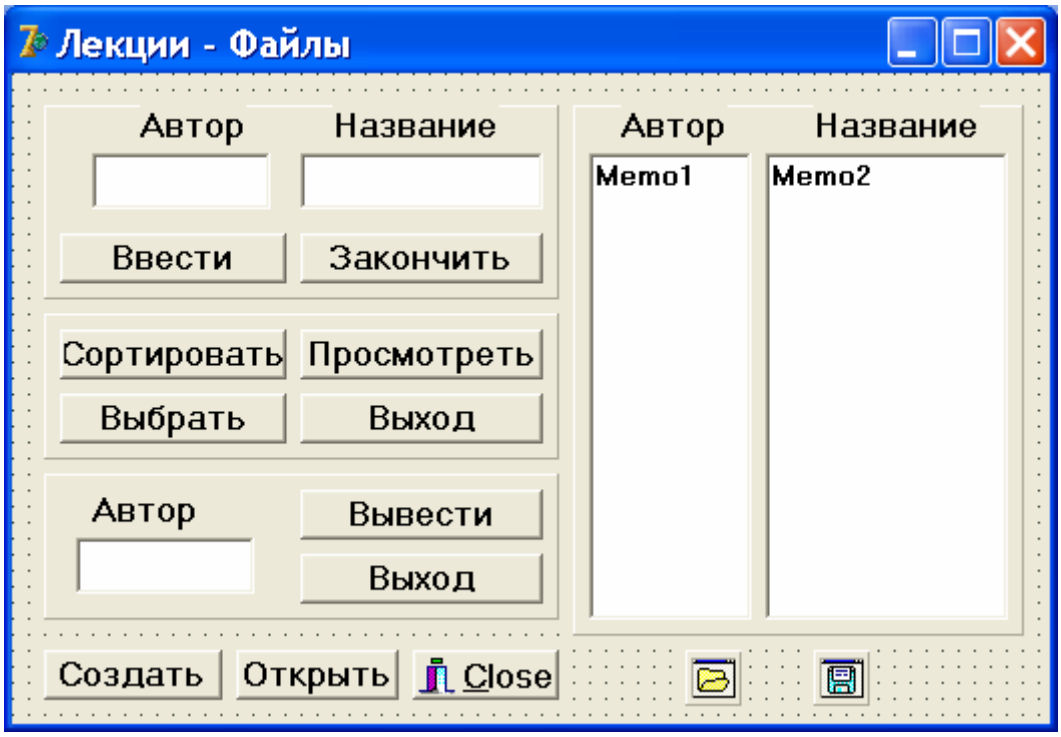
Для удобства работы с файлами в Delphi имеются два специальных компонента, предназначенных для выбора требуемого файла на диске через удобное окно просмотра. Результатом этого выбора является имя и маршрут к файлу, помещаемому в переменную строкового типа

OpenDialog1.FileName или *SaveDialog1.FileName*

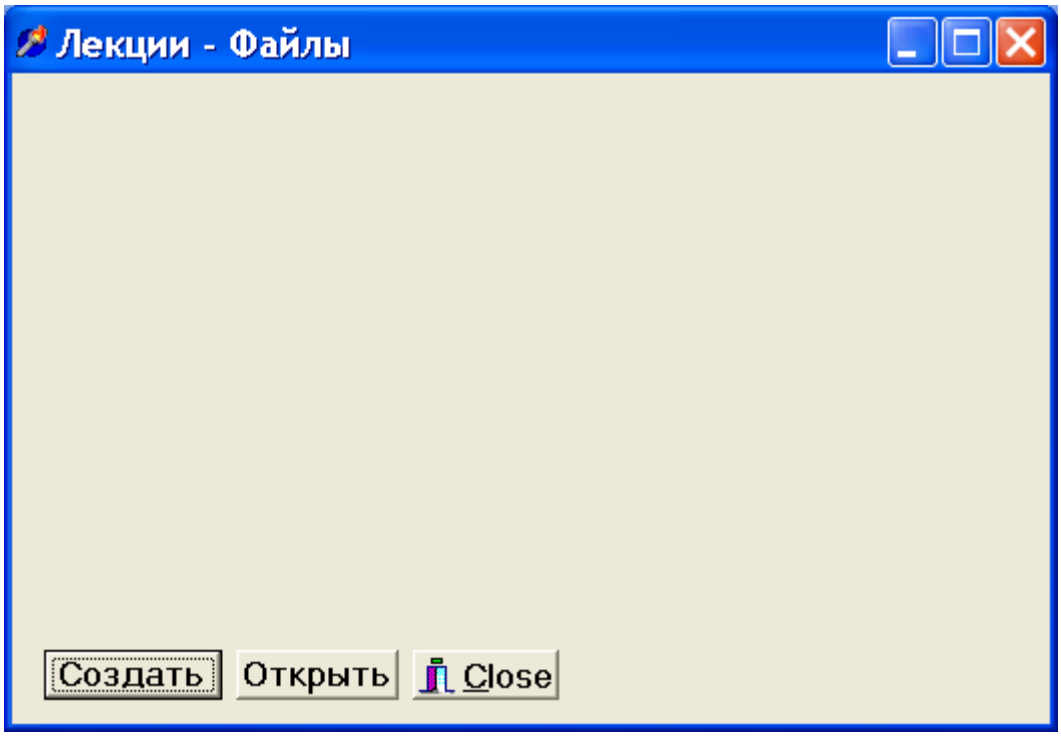
которые затем используются в процедуре *AssignFile()*.

Компоненты *OpenDialog* и *SaveDialog* находятся на странице *DIALOGS.* Все компоненты этой страницы являются невизуальными, т.е. не видны в момент работы программы. Поэтому их можно разместить в любом удобном месте формы. Оба рассматриваемых компонента имеют идентичные свойства и отличаются только внешним видом. После вызова компонента появляется диалоговое окно, с помощью которого выбирается имя программы и путь к ней. В случае успешного завершения диалога имя выбранного файла и маршрут поиска содержатся в свойстве *FileName.* Для фильтрации файлов, отображаемых в окне просмотра, используется свойство *Filter,* а для задания расширения файла, в случае, если оно не задано пользователем, – свойство *DefaultExt.* Для того, чтобы файл автоматически записывался, например, с расширением dat, в свойстве *DefaultExt* компонета *SaveDialog* запишем требуемое расширение – *.dat* (для текстового файла – *.txt*). Если необходимо изменить заголовок диалогового окна, используется свойство *Title.*

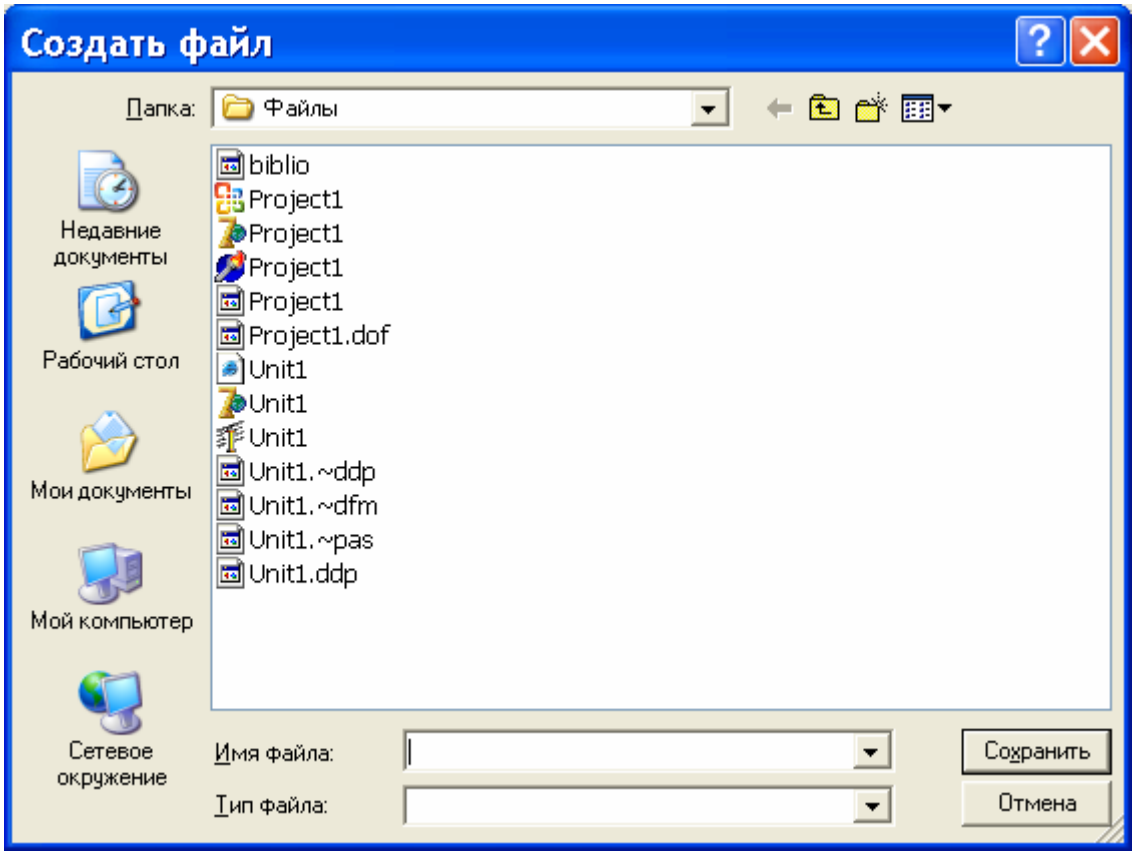
Пример: Создать файл, содержащий список книг. Предусмотреть возможность сортировки книг по алфавиту по фамилии автора и вывод, как всего списка, так и списка книг заданного автора.



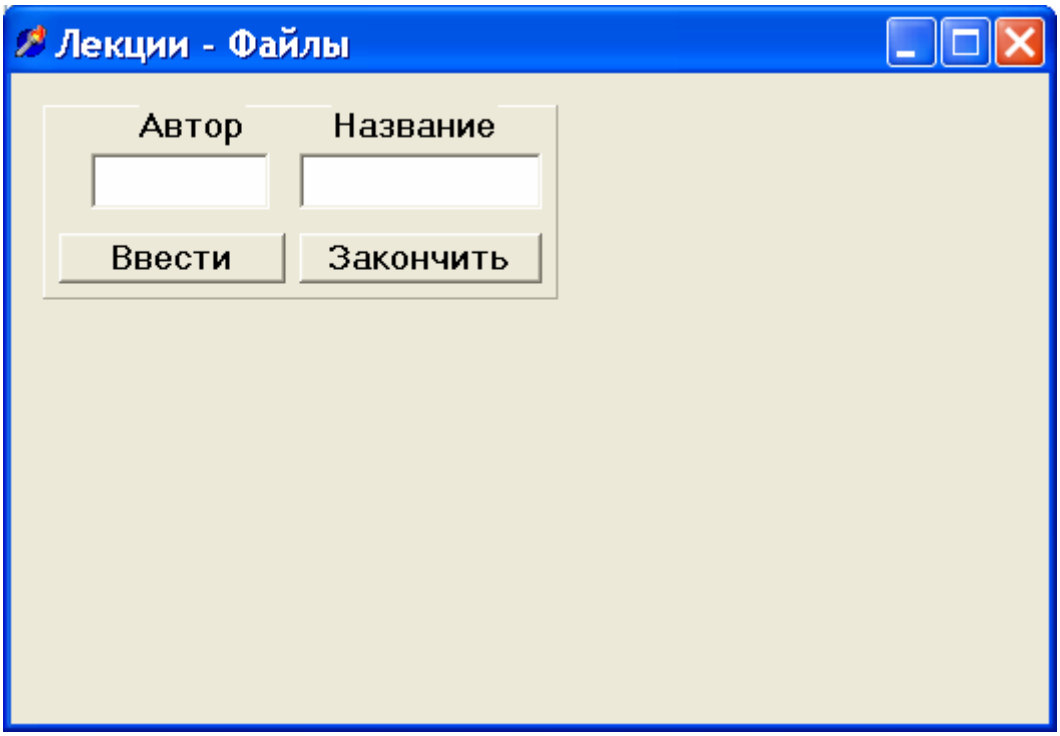
Общий вид формы



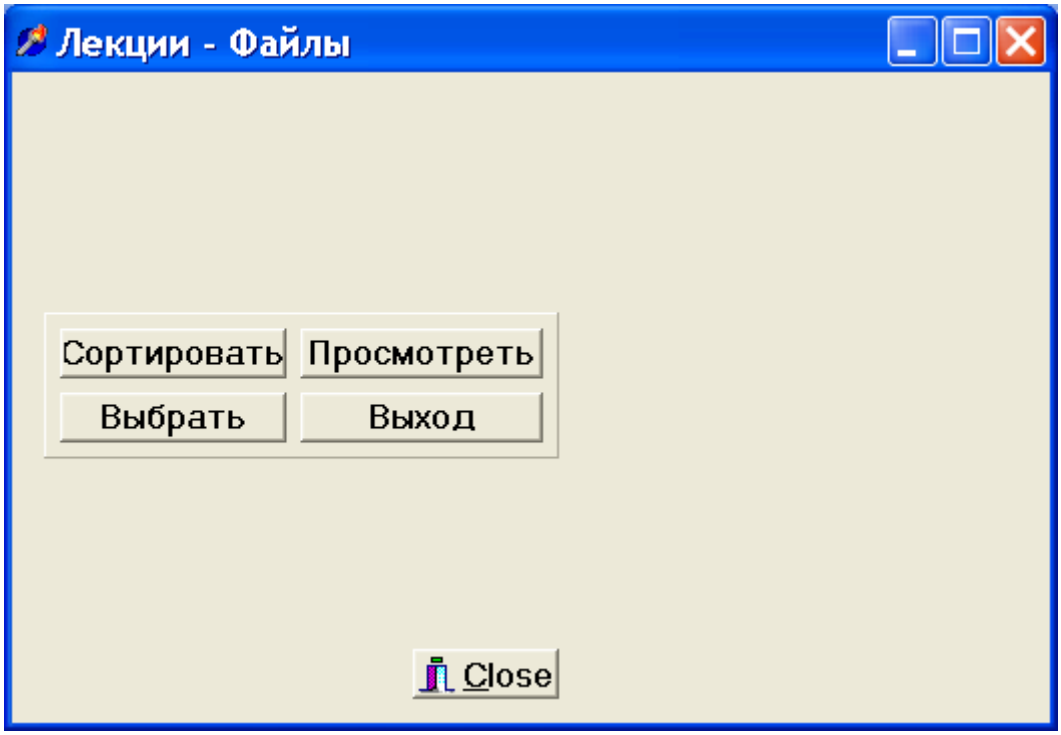
Вид формы после запуска программы



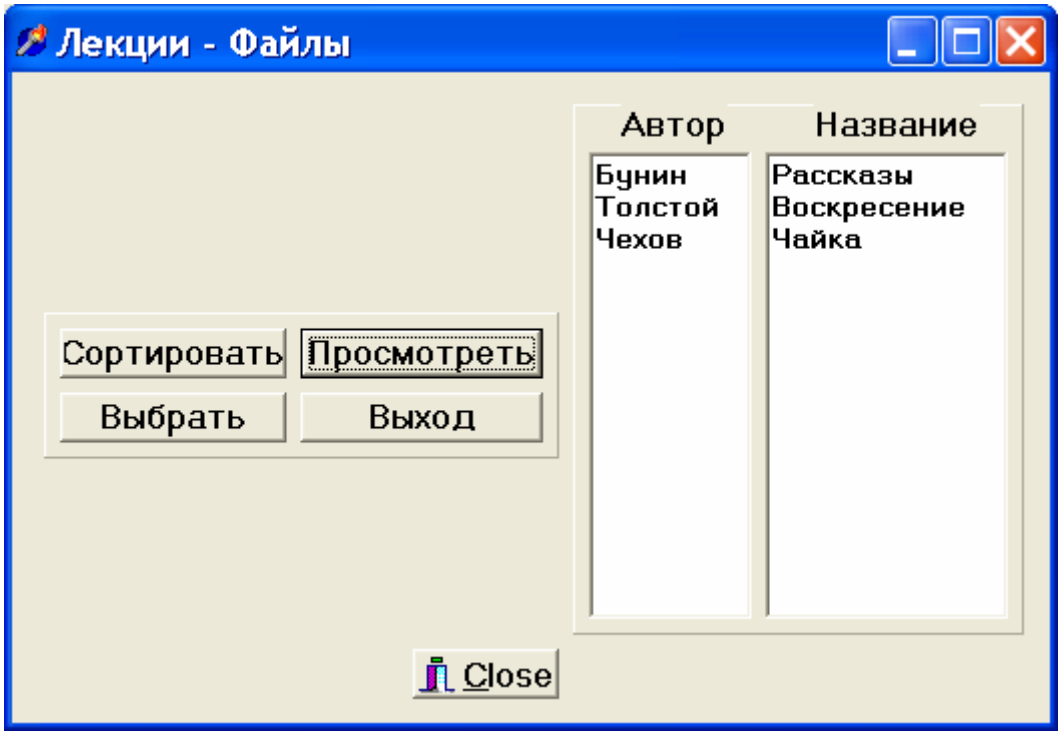
Вид формы после нажатия кнопки «Создать».



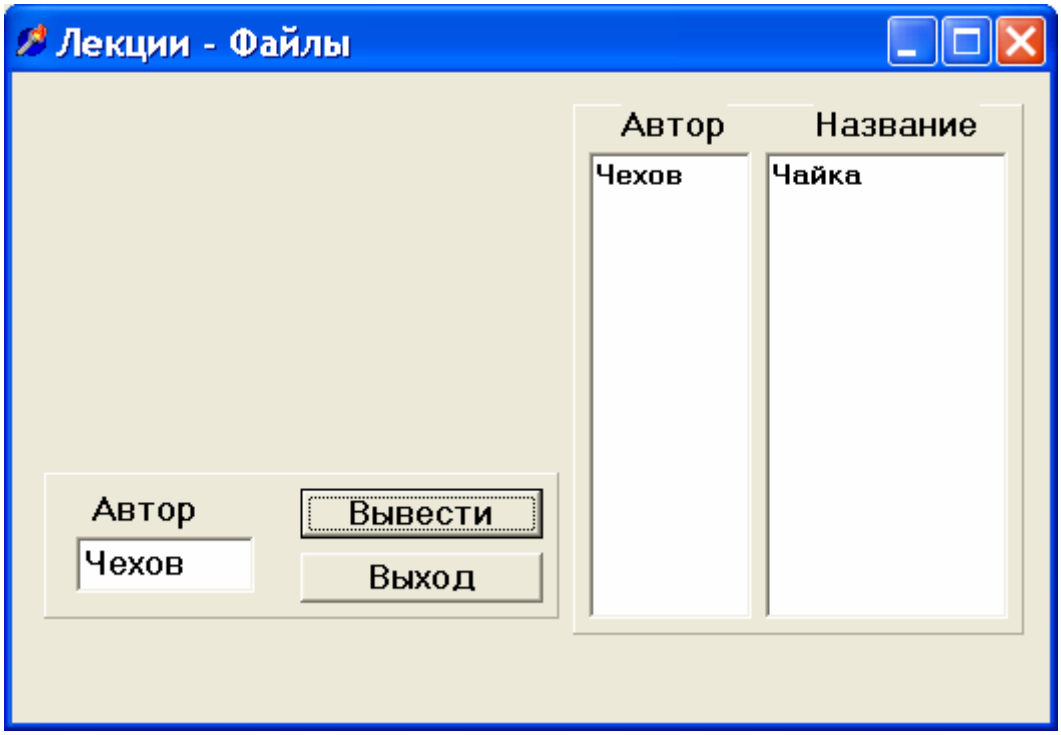
Вид формы после задания имени файла.



Вид формы после нажатия кнопки «Закончить».



Вид формы после нажатия кнопки «Просмотреть».



Вид формы после нажатия кнопки «Выбрать», задании фамилии интересующего автора и нажатия кнопки «Вывести».

Листинг 13.1

```
unit Unit1;
                interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Con-
trols, Forms, Dialogs,StdCtrls, ExtCtrls, Buttons;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    BitBtn1: TBitBtn;
    Panel1: TPanel;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Button3: TButton;
    Button4: TButton;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    Panel2: TPanel;
    Button5: TButton;
    Button6: TButton;
    Button7: TButton;
```

```
    Button8: TButton;
    Panel3: TPanel;
    Button9: TButton;
    Button10: TButton;
    Edit3: TEdit;
    Label3: TLabel;
    Panel4: TPanel;
    Label4: TLabel;
    Label5: TLabel;
    Memo1: TMemo;
    Memo2: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button8Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);
    procedure Button9Click(Sender: TObject);
    procedure Button10Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
type book=record
       avt:string[20];
       naz:string[30];
       end;
var
  Form1: TForm1;
  f:file of book;
  w:book;
  a:array[1..100] of book;
  fname:string;
  n,i,k:integer;

implementation

{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   panel1.Hide;  panel2.Hide;
   panel3.Hide;  panel4.Hide;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin     //  Создать
  SaveDialog1.Title:='Создать файл';
  if SaveDialog1.Execute then
      begin
      fname:=SaveDialog1.FileName;
      AssignFile(f,fname);
      Rewrite(f);
      end;
  Panel1.Show;
  Button1.Hide;   Button2.Hide;  BitBtn1.Hide;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin          // Ввести
   w.avt:=edit1.text;
   w.naz:=edit2.text;
   write(f,w);
   edit1.clear;  edit2.clear;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin                  // Закончить
  CloseFile(f);
  panel1.Hide;
  panel2.show;
  BitBtn1.show;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin   //  Сортировать
   reset(f);  n:=0;
   while not eof(f) do
     begin
     n:=n+1;
     read(f,a[n]);
     end;
   closeFile(f);
```

```
   for i:=1 to n-1 do
     for  k:=1 to n-i do
       if a[k].avt > a[k+1].avt then
         begin w:=a[k]; a[k]:=a[k+1]; a[k+1]:=w; end;
   rewrite(f);
   for i:=1 to n do  write(f,a[i]);
   closeFile(f);
end;

procedure TForm1.Button6Click(Sender: TObject);
begin   //  Просмотреть
   panel4.Show;  memo1.clear;  memo2.clear;
   reset(f);
   while not eof(f) do
     begin
     read(f,w);
     memo1.lines.add(w.avt);
     memo2.lines.add(w.naz);
     end;
   closeFile(f);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin   // Открыть
  OpenDialog1.Title:='Открыть файл';
  if OpenDialog1.Execute then
      begin
      fname:=OpenDialog1.FileName;
      AssignFile(f,fname);
      Reset(f);
      end;
  Panel2.Show;
  Button1.Hide;   Button2.Hide;  BitBtn1.Hide;
end;

procedure TForm1.Button8Click(Sender: TObject);
begin   //  Выход из panel2
 Button1.show;   Button2.show;  BitBtn1.show;
 panel2.Hide;
end;

procedure TForm1.Button7Click(Sender: TObject);
begin   //  Выбрать
   panel2.Hide;   panel3.Show;   bitbtn1.Hide;
```

```
   memo1.clear;    memo2.clear;
end;

procedure TForm1.Button9Click(Sender: TObject);
begin      //  Вывести
  memo1.clear;  memo2.clear;
  reset(f);
   while not eof(f) do
     begin
     read(f,w);
     if w.avt=edit3.text then
          begin
          memo1.lines.add(w.avt);
          memo2.lines.add(w.naz);
          end;
     end;
   closeFile(f);
end;

procedure TForm1.Button10Click(Sender: TObject);
begin    //  Выход из panel3
  panel3.Hide;    panel4.Hide;
  Button1.show;   Button2.show;   BitBtn1.show;
end;

end.
```

ЛЕКЦИЯ 14. ПРОГРАММИРОВАНИЕ С ОТОБРАЖЕНИЕМ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ

Как происходит отображение на экране

Экран дисплея устроен таким образом, что любое изображение формируется из набора светящихся точек, получивших название **пиксель**.

Этот процесс отображения осуществляется с помощью специальной, довольно сложной микросхемы, называемой графическим **адаптером.** Современные компьютеры комплектуются цветным графическим адаптером. Работа графического адаптера осуществляется под управлением специальной программы - драйвера. Адаптер содержит порты ввода-вывода информации, оперативную память, в которой помещается таблица, содержащая информацию о каждом светящемся на экране пикселе (его координате, цвете, яркости). Левый верхний угол экрана имеет нулевые координаты пикселя. Разрешающая возможность графического адаптера определяется размерами одного пикселя (обычно 800x600), но может быть и больше. В результате рисование осуществляется по клеткам. Такое поле из клеточек, называется канва или холст. В *Windows* пользователю для рисования предоставляется окно, в котором он осуществляет рисование с помощью средств, предоставляемых системой Delphi.

14.1. Как рисуются изображения

Нарисовать картинку в среде Delphi можно на многих компонентах (например на форме, на *TPaintBox*), однако наиболее удобно использовать компонент *TImage* (страница *Additional*). Нарисованную в *Image1* картинку можно перенести в отчет, используя процедуру `ClipBoard.Assign(Image1.Picture)` (модуль *Clipbrd*). Для рисования используют класс *TCanvas*, который является свойством многих компонентов, и представляет собой прямоугольный холст в виде матрицы из пикселей и набор инструментов для рисования на нем. Каждый пиксель имеет координату *(x, y)*, где *x* – порядковый номер пикселя, начиная от левой границы холста, а *y* – порядковый номер пикселя, начиная от верхней границы холста. Левый верхний угол холста имеет координату *(0, 0)*, а нижний правый *(Image1.Width-1, Image1.Height-1)*.

Основные ***свойства*** класса *TCanvas*

`Property Pen : TPen;` – карандаш. В свою очередь, имеет свойства *Color* - цвет , *Width* - толщина и *Style* – стиль (*psSolid* – сплошной, *psDash* – штриховой, *psDot* – пунктирный, *psClear* – отсутствие линии и др.).

`Property Brush : TBrush;` – кистью. Это свойство определяет фон заполнения замкнутых фигур. В свою очередь, имеет свойства *Color* - цвет и *Style* – стиль (*bsSolid* – сплошной, *bsCross* – сетка, *bsClear* – отсутствие фона и др.).

`Property Font : TFont;` – шрифт. В свою очередь, имеет свойства *Color* - цвет, *Size* - размер и *Style* – стиль (*fsBold* – жирный, *fsitalic* – курсив и др.).

Некоторые ***методы*** класса *TCanvas*

`Procedure Ellipse(X1, Y1, X2, Y2: Integer)` – рисует эллипс в охватывающем прямоугольнике *(X1, Y1), (X2, Y2)* и заполняет внутреннее пространство эллипса текущей кистью.

`Procedure LineTo (X, Y: Integer)` – рисует линию от текущего положения пера до точки *(X. У)*.

`Procedure MoveTo(X, Y: Integer)` – перемещает карандаш в точку *(X, У)* без вычерчивания линий.

`Procedure Polygon (Points: array of TPoint)` – рисует многоугольник по точкам, заданным в массиве *Points*.

Например: `Canvas.Polygon([Point(x1, y1), Point(x2, y2), Point(x3, y3]);` Конечная точка соединяется с начальной и многоугольник заполняется кистью. Для вычерчивания без заполнения используется метод `Polyline.`

`Procedure Rectangle (X1, Y1, X2, Y2: Integer)` – рисует и заполняет прямоугольник *(X1, У1), (X2, У2)*. Для вычерчивания без заполнения используется `FrameRect` или `PolyLine.`

`Procedure TextOut (X, Y: Integer; const Text: String)` – выводит текстовую строку *Text* так, чтобы левый верхний угол прямоугольника, охватывающего текст, располагался в точке *(X, У)*.

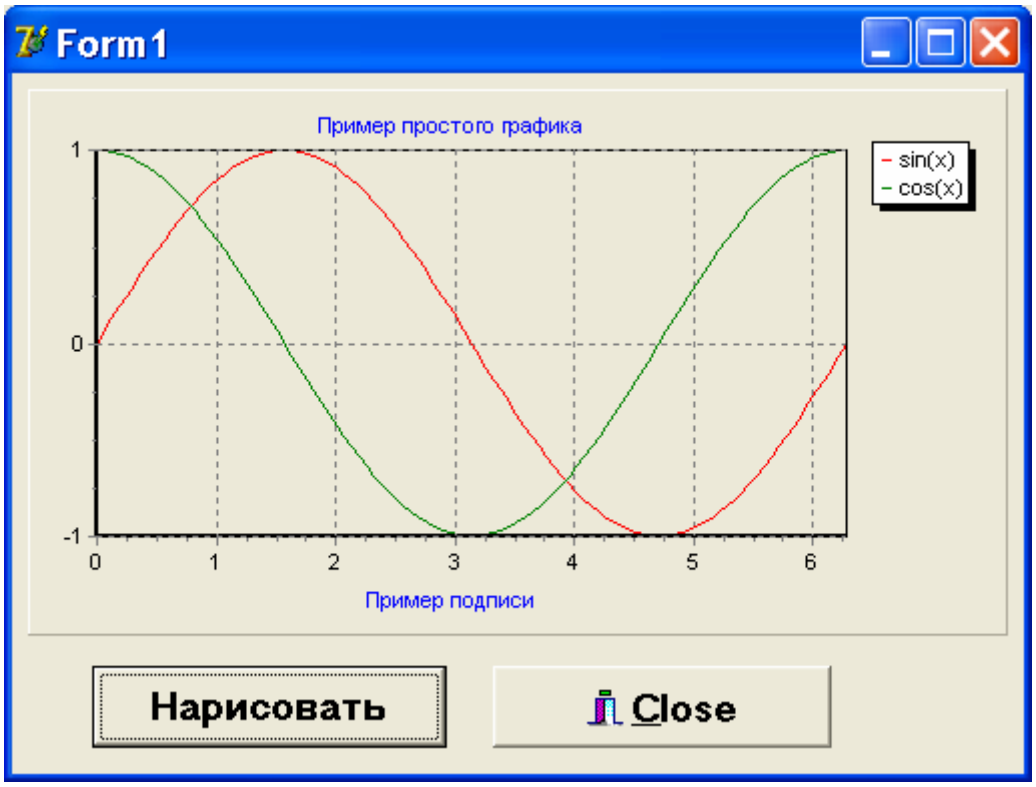
14.2. Построение графиков с помощью компонента *Chart*

Обычно результаты расчетов представляются в виде графиков и диаграмм. Среда Delphi имеет мощный пакет стандартных программ вывода на экран и редактирования графической информации, который реализуется с помощью визуально отображаемого на форме компонента *Chart*.

Построение графика (диаграммы) производится после вычисления таблицы значений функции *y=f(x)*. Полученная таблица передается в специальный двумерный массив *Chart**I**.SeriesList[**k**]* (***k** – номер графика (0,1,2,...)*) компонента *Chart* с помощью метода *AddXY*. Компонент *Chart* осуществляет всю работу по отображению графиков, переданных в объект *Chart**I**.SeriesList[**k**]:* строит и размечает оси, рисует координатную сетку, подписывает название осей и самого графика, отображает переданную таблицу в виде всевозможных графиков или диаграмм. При необходимости, с помощью встроенного редактора *EditingChart* компоненту *Chart* передаются данные о толщине, стиле и цвете линий, параметрах шрифта подписей, шагах разметки координатной сетки и другие настройки. В процессе работы программы изменение парамет-

ров возможно через обращение к соответствующим свойствам компонента *Chart.* Так, например, свойство *Chart1.BottomAxis* содержит значение максимального предела нижней оси графика. Перенести график в отчет можно через буфер обмена, используя процедуру *Chart**I**.CopyToClipboardMetafile(True).*

Примеры. Построить графики функций *sin(x)* и *cos(x)* с помощью компонента *Chart.* Текст и форма проекта приведены ниже.



Листинг 14.1

```
unit Unit1;
                    interface
uses  Windows,Messages,SysUtils,Variants,Classes,
  Graphics,Controls,Forms,Dialogs,StdCtrls,TeEngine,
       Series, ExtCtrls, TeeProcs, Chart, Buttons;
type
  TForm1 = class(TForm)
    Chart1: TChart;
    Series1: TLineSeries;
    Series2: TLineSeries;
    Button1: TButton;
    BitBtn1: TBitBtn;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
```

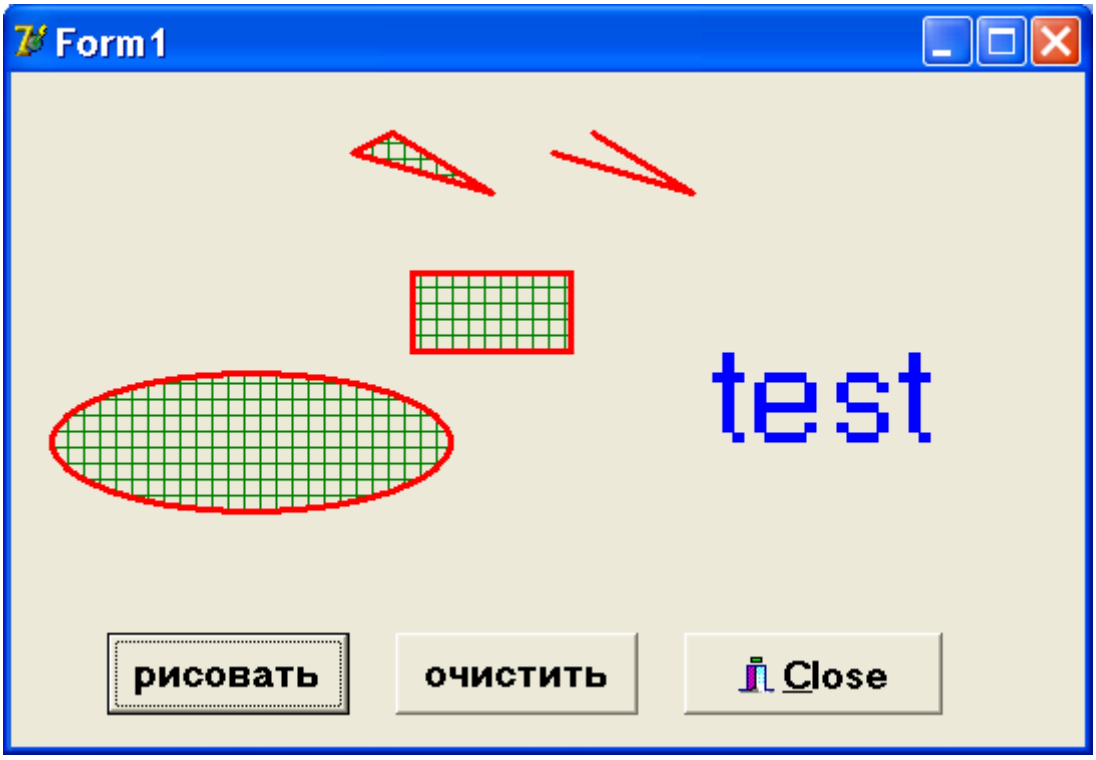
```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  f:file of extended;
implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
  var k:integer;  // Нарисовать
      x,y,z:extended;
begin
 for k:=0 to 100 do
   begin
   x:=0.02*pi*k;  y:=sin(x);  z:=cos(x);
   series1.AddXY(x,y,'',clRed);
   series2.AddXY(x,z,'',clGreen);
   end;
end;
end.
```

Вывести на форму некоторые простейшие фигуры и текст.



Листинг 14.2

```
unit Unit1;
                    interface
uses
  Windows,Messages,SysUtils,Variants,Classes,Graphics,
Controls,Forms,Dialogs,StdCtrls,Buttons;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    BitBtn1: TBitBtn;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
with form1.Canvas do
  begin
  pen.Width:=3;
  pen.color:=clred;
  brush.style:=bsCross;
  brush.Color:=clgreen;
  polygon([point(190,30), point(240,60),
  point(170,40)]);
  polyline([point(290,30), point(340,60),
  point(270,40)]);
  rectangle(200,100,280,140);
  ellipse(20,150,220,220);
  font.color:=clblue;
  font.Size:=50;
  textout(350,120,'test');
end;

end;


procedure TForm1.Button2Click(Sender: TObject);
begin    // Очистить
refresh;
end;

end.
```

ЛЕКЦИЯ 15. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ РЕКУРСИИ

15.1. Понятие рекурсии

Рекурсивным называется описание объекта, частично состоящее и определяемое с помощью самого описываемого объекта. Рекурсия – это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения обращается сама к себе.

Классический пример: вычисление факториала *n! = 1*2*3…(n-1)*n.*

Рекурсивное описание: *n! = (n-1)! * n, 0! = 1.*

Рекурсивная функция:

```
Function fak(n:word):extended;
   begin
    if n=0 then result:=1
           else result:=fak(n-1)*n
   end;
```

Примечание: Тип результата вычислений *(extended)* выбран из-за большего диапазона допустимых значений быстрорастущей функции *n!.*

Нерекурсивная функция:

```
Function fak(n:word):extended;
         Var k:word;
   begin
     result:=1;
         if n>1 then  for k:=2 to n do
              result:=result*k;
   end;
```

При выполнении рекурсивной подпрограммы осуществляется многократный переход от текущего уровня организации алгоритма к нижнему уровню последовательно, до тех пор, пока не будет получено тривиальное решение задачи (в вышеприведенном примере, *n=0*). Рекурсивная форма записи алгоритма обычно выглядит изящнее итерационной и дает более компактный текст программы, но при выполнении работает медленнее и может вызвать переполнение программного стека (исключительная ситуация *EStackOverFlow*), т. к. при каждом переходе к следующему уровню (рекурсивной активации подпрограммы) происходит создание и запоминание всех ее

локальных и формальных параметров. В результате, после *n-й* активации в памяти будет находиться список из *n+1* комплектов таких параметров.

Рекурсивный вызов может быть ***прямым***, как в вышеприведенном примере, и ***косвенным***. В этом случае подпрограмма обращается к себе опосредованно, путем вызова другой подпрограммы, в которой содержится обращение к первой.

```
          // Опережающее описание
Procedure rex(<список параметров 1>); Forward;

Procedure fox(<список параметров 2>);
  begin
     ...
     rex(<список параметров 1>);
     ...
  end;

Procedure rex;
  begin
     ...
     fox(<список параметров 2>);
     ...
  end;
```

В этом примере обращение к процедуре ***rex()*** записано раньше, чем ее описание, что недопустимо в Pascal. Для разрешения этой ситуации используется **опережающее описание** с помощью стандартной директивы ***Forward.***

Внимание. *Для предотвращения зацикливания рекурсивной подпрограммы необходимо предусмотреть* ***обязательный*** *выход на тривиальное решение, т. е. на ветвь, не содержащую обращение подпрограммы к самой себе.*

15.2. Примеры рекурсивных вычислений

Найти максимальный элемент в массиве используя метод деления массива пополам $max\ (a_1 \ldots a_n) = max\ (\ max\ (\ a_1 \ldots a_{n/2}),\ max\ (\ a_{n/2+1} \ldots a_n)\ )$

```
type vek=array[1..50] of extended;

function   maxR(x:vek;  m,n:integer):extended  ;
   var  k:integer;
 begin
```

```
  if m=n then result:=x[m]
         else begin   k:=(m+n) div 2;
  if  maxR(x,m,k)>maxR(x,k+1,n)
         then result:=maxR(x,m,k)
         else result:=maxR(x,k+1,n);
 end;
end;
```

Найти максимальный элемент в массиве используя очевидное соотношение $max\ (a_1 \ldots a_n) = max\ (\ max(a_1 \ldots a_{n-1}),\ a_n)$

```
function maxRn(x:vek;  n:integer):extended;
 begin
  if n=1 then result:=x[1]
    else if maxRn(x,n-1)>x[n] then result:=maxRn(x,n-1)
                              else result:=x[n];
 end;
```

ЛЕКЦИЯ 16. ПОИСК И СОРТИРОВКА МАССИВОВ

16.1. Организация работы с базами данных

Для создания и обработки всевозможных баз данных широко применяются массивы записей. Обычно база данных накапливается и хранится на магнитном диске. К ней часто приходится обращаться, обновлять, перегруппировывать. Работа с базой может быть организована двумя способами:

1. Вносить изменения и осуществлять поиск можно прямо на диске, используя специфическую технику работы с записями на файлах, при этом временные затраты на обработку данных (поиск, сортировку) значительно возрастают, но нет ограничений на оперативную память.

2. В начале работы вся база (или ее необходимая часть) считывается в массивы записей и обработка производится в оперативной памяти, что значительно сокращает ее время, однако требует затрат оперативной памяти. Наиболее частыми операциями при работе с базами данных являются «поиск» и «сортировка». При этом алгоритмы решения этих задач существенно зависят от того, организованы записи в массивы или размещены на диске. Обычно запись содержит некое ключевое поле (ключ), по которому ее находят среди множества других аналогичных записей. В зависимости от решаемой задачи ключом может служить, например, фамилия, номер расчетного счета или адрес. Основное требование к ключу в задачах поиска состоит в том, чтобы операция проверки на равенство была корректной. Поэтому, например, в качестве ключа не следует выбирать действительное число, т.к. из-за всегда возможной ошибки округления поиск нужного ключа может оказаться безрезультатным, хотя этот ключ в массиве имеется.

16.2. Поиск в массиве записей

Задача поиска требуемого элемента в массиве записей *a[i], i = 1..n* заключается в нахождении индекса *i,* удовлетворяющего условию *a[i].k = isk.* Здесь поле записи *k* выступает в качестве ключа, *isk* – искомый ключ. После нахождения *i* обеспечивается доступ ко всем другим полям найденной записи *a[i].*

Линейный (последовательный) поиск используется, когда нет никакой дополнительной информации о разыскиваемых данных. Он представляет собой последовательный перебор массива до обнаружения требуемого ключа или до конца, если ключ не обнаружен:

```
i:=1;
while (i<=n) and (a[i].k<>isk) do  i:=i+1;
if i=n+1 then write('элемент не найден')
         else write('индекс искомого элемента= ',i);
```

Видно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли уменьшить затраты на поиск? Единственная возможность – попытаться упростить логическое выражение с помощью введения вспомогательного элемента - *барьера*, который предохраняет от перехода за пределы массива. Для этого добавим ***в конец*** массива элемент с искомым ключом. Количество проверок на каждом шаге уменьшается (одна, а не две) т.к. нет необходимости проверки выхода за пределы массива, элемент с искомым ключом обязательно будет найден до выхода за пределы массива.

```
a[n+1].k := isk;    i:=1;
while a[i].k <> isk do  i:=i+1;
if i=n+1 then write('элемент не найден' )
         else write('индекс искомого элемента= ',i);
```

Поиск делением пополам (бинарный поиск) используется, когда данные упорядочены по возрастанию ключа *k*, т.е. *a[i]. k ≤ a[i+1]. k.* Основная идея – возьмем «средний» (*m-й*) элемент. Если *a[m].k < isk,* то все элементы $i \le m$ можно исключить из дальнейшего поиска, если *a[m]. k≥ isk,* то можно исключить все $i > m$:

```
i:=1;   j:=n;
 while i < j do
   begin
   m := ( i+j ) div 2;
   if a[m].k<isk then i:=m+1  else j:=m;
   end;
if a[i].k=isk then write('индекс искомого элемента=',i)
             else write('элемент не найден');
```

В этом алгоритме отсутствует проверка внутри цикла совпадения *a[m].k=isk*. На первый взгляд это кажется странным, однако тестирование показывает, что в среднем выигрыш от уменьшения количества проверок превосходит потери от нескольких «лишних» вычислений до выполнения условия *i=j*.

16.3. Сортировка массивов

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа. Цель сортировки – облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов – это не вводить дополнительных массивов, т.е. все перестановки элементов должны выполняться «на том же месте» в исходном массиве. Сортировку массивов принято называть **внутрен-**

ней в отличие от сортировки файлов (списков), которую называют **внешней**.

Методы внутренней сортировки классифицируются по времени их работы. Хорошей мерой эффективности может быть число сравнений ключей - *C* и число пересылок элементов - *P*. Эти числа являются функциями *C(n), P(n)* от числа сортируемых элементов *n*. **Быстрые** (но сложные) алгоритмы сортировки требуют (при $n \to \infty$) порядка *n log n* сравнений, **прямые** (простые) методы - n^2.

Прямые методы коротки, просто программируются. Быстрые, усложненные, методы требуют меньшего числа операций, но эти операции обычно сами более сложны, чем операции прямых методов, поэтому для достаточно малых n ($n \leq 50$) прямые методы работают быстрее. Значительное преимущество быстрых методов (в $n/\log(n)$ раз) начинает проявляться при $n \gtrsim 100$.

Среди простых методов наиболее популярны:

1) **Метод прямого обмена** (пузырьковая сортировка).

2) **Метод прямого выбора**.

3) Сортировка с помощью **прямого (двоичного) включения.**

4) **Шейкерная** сортировка (модификация пузырьковой).

Улучшенные методы сортировки [3]:

1) **Метод Д. Шелла** , усовершенствование метода прямого включения.

2) Сортировка с помощью дерева, метод **HeapSort**, Д. Уильямсон.

3) Сортировка с помощью разделения, метод **QuickSort**, Ч. Хоар, улучшенная версия пузырьковой сортировки. На сегодняшний день это самый эффективный метод сортировки. Сравнение методов сортировок показывает, что при $n>100$ наихудшим является метод пузырька, метод QuickSort в 2-3 раза лучше, чем HeapSort, и в 3-7 раз, чем метод Шелла.

Рассмотрим алгоритмы и реализацию некоторых методов.

16.3.1. Метод пузырька

В алгоритме сортировки методом пузырька сравниваются два соседних элемента. Если они расположены в неправильной последовательности, то выполняется перестановка этих элементов. Сортировка осуществляется путем многократного прохождения по списку элементов. При сортировке по возрастанию элементы с малыми значениями поднимаются вверх в начало списка, подобно пузырькам воздуха в воде. Процедура пузырьковой сортировки имеет вид

```
type vec = array[1..100] of extended;
     ind = array[1..100] of integer;
```

```
Procedure PuzSort(var a:vec; n:integer);
  var k,kol:integer;  //  Метод Пузырька
      w:extended;
      p:boolean;
 begin
   kol:=1;
   repeat
     p:=true;
     for k:=1 to n-kol do
         if a[k]>a[k+1] then
                  begin
                  w:=a[k];
                  a[k]:=a[k+1];
                  a[k+1]:=w;
                  p:=false;
                  end;
     inc(kol);
   until p;
 end;
```

В общем случае для сортировки требуется *n–1* проход по массиву. Чтобы исключить ненужные проходы, если массив уже полностью или частично отсортирован, используется не цикл с заданным количеством повторений (*for kol := 1 to n – 1 do*), а оператор repeat с флажком *p*.

16.3.2. Метод прямого выбора

Сортировка осуществляется путем многократного прохождения по списку элементов. На каждом (*k-ом*) проходе находится минимальный элемент с *к-го* по *n- ый* элементы, который затем переставляется с *к-ым* элементом. Процедура этой сортировки имеет вид

```
Procedure PramSort(var a:vec; n:integer);
   var k,i,m:integer;
       w:extended;
 begin
   for k:=1 to n-1  do
      begin
      m:=k;
      for i:=k+1 to n do
         if a[i]<a[m] then m:=i;
      w:=a[m]; a[m]:=a[k]; a[k]:=w;
      end;
 end;
```

16.3.3. Метод Шелла

Алгоритм Шелла намного эффективнее, чем метод пузырька. Сначала сравниваются отдаленные, а затем близкорасположенные элементы. Переменная *kol* содержит интервал, разделяющий сравниваемые элементы. Начальное значение *kol* равно половине количества элементов. В процессе сортировки значение *kol* уменьшается в два раза на каждом проходе, пока не начнет выполнятся сравнение соседних элементов, как в методе пузырька. Процедура сортировки Шелла имеет вид

```
Procedure ShellSort(var a:vec;   n:integer);
   var k,kol:integer;    //Метод Шелла
       w:extended;
       p:boolean;
  begin
    kol:=n div 2;  //  Зазор
    repeat
       repeat
          p:=true;
          for k:=1 to n-kol do
              if a[k]>a[k+kol] then
                     begin
                       w:=a[k];
                       a[k]:=a[k+kol];
                       a[k+kol]:=w;
                       p:=false;
                     end;
       until p;
       kol:=kol div 2;
     until kol=0;
   end;
```

Полезный совет: перестановка элементов со сложными типами данных – довольно длительный процесс. Поэтому рекомендуется вместо перестановки самих данных переставлять индексы. Такой прием используется практически во всех коммерческих приложениях. В этом случае процедура сортировки Шелла имеет вид

```
type ind=array[1..100] of integer;

Procedure ShellSortInd(a:vec; n:integer; var nom:ind);
   var k,kol,w:integer;
       p:boolean;
  begin
    for k:=1 to n do nom[k]:=k;
```

```
    kol:=n div 2;  //  Зазор
    repeat
      repeat
         p:=true;
         for k:=1 to n-kol  do
            if a[nom[k]]>a[nom[k+kol]] then
                 begin
                   w:=nom[k];
                   nom[k]:=nom[k+kol];
                   nom[k+kol]:=w;
                   p:=false;
                 end;
      until p;
      kol:=kol div 2;
    until kol=0;
 end;
```

16.3.4. Метод Хоара (Hoare)

В алгоритме Хоара сначала выбирается так называемое опорное значение. Затем элементы со значением, меньше опорного, переносятся влево, а со значением, большим или равным ему, - вправо. Таким образом, на первом шаге алгоритм Хоара делит элементы на два раздела: со значениями, меньшими опорного, и со значениями, большими или равными ему. Затем подпрограмма, рекурсивно вызывая сама себя, продолжает разбивку разделов. В каждом разделе выбирается новое опорное значение, и элементы раздела переставляются влево и вправо. Процесс разбивки на разделы рекурсивно продолжается до тех пор, пока размер раздела достигнет одного или двух элементов. Эффективность метода зависит от объема данных и выбора метода опорного сечения. В приведенном ниже алгоритме в качестве опорного выбирается средний элемент раздела. Процедура сортировки Хоара имеет вид

```
Procedure QuickSort(var a:vec; low,high:integer);
   var l,r:integer;      //   Метод Хоара
       op,w:extended;
  begin
    op:=a[(low+high) div 2];  //  Опорный элемент
    // Перенос элементов, меньших опорного, влево, а
больших - вправо
    l:=low;   r:=high;
    repeat
      while (l<=high) and (a[l]<op) do inc(l);
      while (r>=low)  and (a[r >op) do dec(r);
      if l<=r then begin
```

```
                  w:=a[l]; a[l]:=a[r]; a[r]:=w;
                  inc(l);   dec(r);
                  end;
  until l>r;
  if r>low  then QuickSort(a,low,r);
  if l<high then QuickSort(a,l,high);
end;
```

ЛЕКЦИЯ 17. РАБОТА СО СПИСКАМИ НА ОСНОВЕ ДИНАМИЧЕСКИХ МАССИВОВ

17.1. Работа со списками

В практике программирования довольно часто встречается задача обработки набора однотипных данных, количество которых меняется в зависимости от ситуации. Их надо разместить в оперативной памяти и по мере надобности с ними работать: добавлять новые данные в список, освобождаться от старых данных, находить нужную информацию. Показательным примером является задача обслуживания очереди заказов на покупку товара. Данные могут содержать информацию о заказчике: ФИО, адрес, финансовые возможности и др. По мере появления новых заказчиков их дописывают в конец очереди, по мере поступления товара и обслуживания заказчиков из начала очереди данные об этих заказчиках стираются. При решении подобных задач программисты вводят понятие списка.

Список – это последовательность однотипных элементов (данных), с которыми надо работать в оперативной памяти: добавлять новые элементы в группу, удалять использованные, сортировать, находить нужные. В процессе работы список может возрастать и уменьшаться. Простейшая форма организации списка – это массив данных. Данные, размещенные в массиве, имеют свой номер (индекс), что придает эффект «видимости» каждому элементу. Для организации работы со списками на основе динамического массива необходимы процедуры добавления нового элемента в список на заданную позицию, удаление элемента с заданным номером, процедуры поиска и сортировки. Две последние операции рассмотрены ранее.

17.2. Добавление нового элемента в список на заданную позицию

При добавлении нового элемента в список необходимо выделить память, на один элемент большую существующей, затем скопировать старый список в новый раздел памяти, сдвинув элементы от заданной позиции до конца списка на одну позицию правее и добавить туда новый элемент.

Для организации добавления элемента *avs* в позицию *nvs* создадим следующую процедуру:

```
Type Tspis=integer; //Описание базового типа массива
var  a:array of Tspis;//Указатель на динамический массив

procedure addvec(avs:Tspis;   nvs:integer);
 begin  // Вставка avs на nvs-ю позицию
  setlength(a,high(a)+2);  // Новая длина массива
  for k:=high(a)-1 downto nvs do a[k+1]:=a[k]; //Сдвиг
```

```
  a[nvs]:=avs;     //      Вставка
end;
```

Напомним, что при задании нового размера массива процедурой setlength() старые значения элементов массива сохраняются.

17.3. Удаления элемента с заданным номером

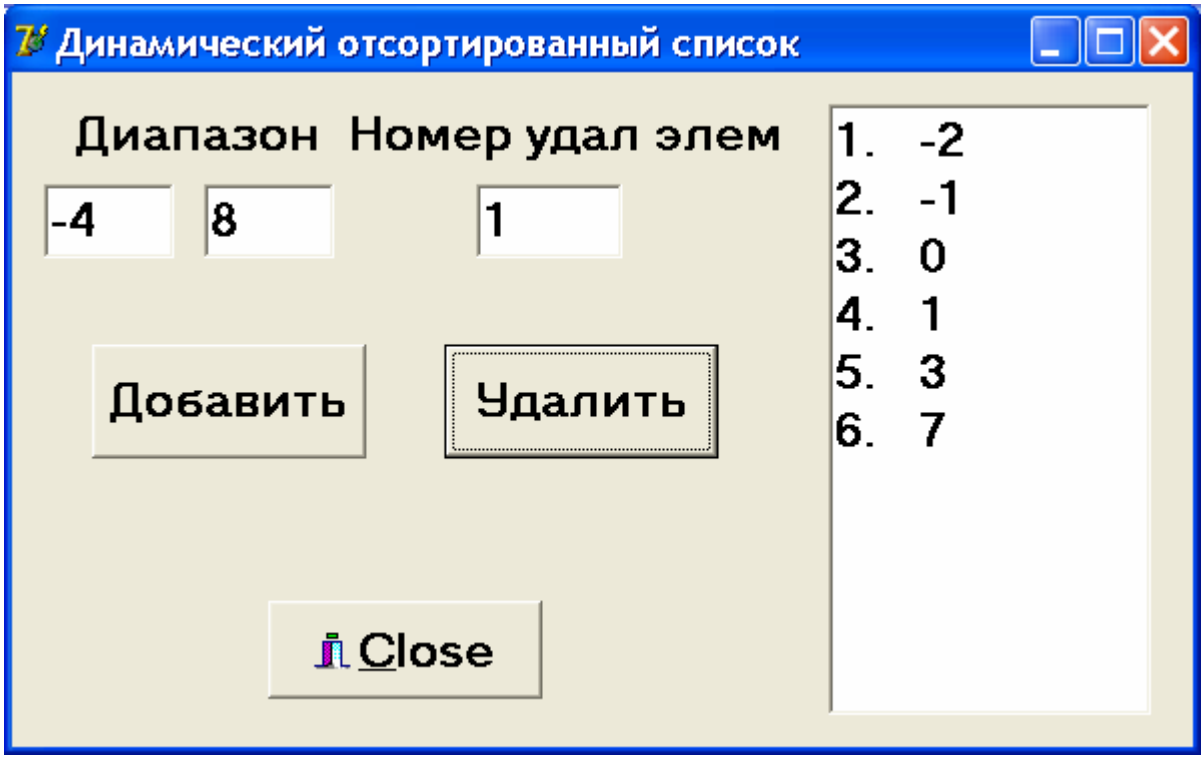
Процедура удаления элемента с номером *nyd* тоже требует, кроме выделения новой памяти и копирования, еще и «схлопывания», т.е. сдвига элементов с *nyd+1-го* до конца списка на одну позицию влево, чтобы заполнить освободившуюся *nyd-ю* позицию. Процедура имеет вид;

```
procedure delvec(nyd:integer);
  begin
    for k:=nyd+1 to high(a) do a[k-1]:=a[k]; // Сдвиг
    setlength(a,high(a));   //  Новая длина массива
  end;
```

Это простая схема хорошо работает для небольших списков, но у нее есть существенный недостаток: при каждом добавлении приходится менять размер выделенной памяти и, что еще хуже, сдвигать часть элементов на одну позицию вправо или влево.

17.4. Пример программы

Ниже приведен пример программы создания списка на основе динамического массива. При нажатии кнопки *Button1* в список добавляется новый элемент со случайным значением в интервале *min÷max* (компоненты *Edit1, Edit2*) на позицию, не нарушающую сортировки. Таким образом, последовательно нажимая кнопку *Button1,* формируем отсортированный список. При нажатии кнопки *Button2* из списка удаляется элемент с номером, задаваемым в компоненте *Edit3.*



Листинг 17.1

```
unit Unit1;
                    interface
uses Windows,Messages,SysUtils,Variants,Classes,
   Graphics,Controls,Forms,Dialogs,Buttons,StdCtrls;
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Button2: TButton;
    Memo1: TMemo;
    BitBtn1: TBitBtn;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Edit2Change(Sender: TObject);
    procedure Edit3Change(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;
type Tspis=integer;  //Описание типа динам. массива
var
  Form1: TForm1;
  a:array of Tspis; //Указатель на динам. массив
  k:integer;
              implementation

{$R *.dfm}

procedure addvec(avs:Tspis; nvs:integer);
begin // Вставка avs на nvs-ю позицию
  setlength(a,high(a)+2);// Новая длина массива
  for k:=high(a)-1 downto nvs do a[k+1]:=a[k];
  a[nvs]:=avs;
end;

procedure delvec(nyd:integer);
begin // Удаление элемента с nyd-ой позиции
  for k:=nyd+1 to high(a) do a[k-1]:=a[k];
  setlength(a,high(a));// Новая длина массива
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  if (edit1.text='') or (edit2.text='') then
          button1.enabled:=false;
  if edit3.text='' then button2.enabled:=false;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  if (edit1.text<>'') and (edit2.text<>'') then
          button1.enabled:=true;
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
  if (edit1.text<>'') and (edit2.text<>'') then
          button1.enabled:=true;
end;
```

```
procedure TForm1.Edit3Change(Sender: TObject);
begin
  if edit3.text<>'' then button2.enabled:=true;
end;

procedure TForm1.Button1Click(Sender: TObject);
     var min,max,n:integer;   // Добавить
           w:Tspis;
begin
  min:=strtoint(edit1.text);
  max:=strtoint(edit2.text);
  randomize;
  w:=min+random(max-min+1); // Новый элемент
  if high(a)=-1 then
    begin setlength(a,1); a[0]:=w; end;//если список
пуст
                else
      begin
      n:=high(a)+1;
      for k:=0 to high(a) do//Находим позицию вставки
         if w<a[k] then begin n:=k; break;  end;
      addvec(w,n);  // Вставка нового элемента
      end;
  memo1.clear;
  for k:=0 to high(a) do
  memo1.Lines.Add(inttostr(k+1)+'.'+inttostr(a[k]));
end;

procedure TForm1.Button2Click(Sender: TObject);
  var n:integer;  //  Удалить
begin
  n:=strtoint(edit3.Text)-1;
  if n>high(a) then
     begin
     edit3.Clear;  button2.enabled:=false;  exit;
     end;
  delvec(n);  //  Удаление n-го элемента
  memo1.clear;
  for k:=0 to high(a) do
  memo1.Lines.Add(inttostr(k+1)+'.'+inttostr(a[k]));
end;


procedure TForm1.BitBtn1Click(Sender: TObject);
```

```
begin
  a:=nil; // Освобождение памяти
end;

end.
```

ЛЕКЦИЯ 18. СВЯЗАННЫЕ СПИСКИ НА ОСНОВЕ РЕКУРСИВНЫХ ДАННЫХ

18.1. Что такое стек и очередь

Ранее мы рассмотрели простейшую форму организации списка – на основе динамических массивов данных. Недостатком таких списков является то, что при добавлении или удалении элементов в список приходится копировать динамические массивы и менять размер выделенной памяти, чтобы перестроить массив, сдвигая на одну позицию все элементы расположенные правее места вставки (удаления). Более эффективная организация списка, не требующая копирования динамических массивов, может осуществляться на основе рекурсивного типа данных. Рекурсивный тип данных при своем описании допускает обращение к самому себе. В Pascal рекурсивный тип данных представляет собой запись, содержащую набор полей для хранения информационной части элемента и одного (односвязный) или двух (двухсвязный список) полей, представляющих собой указатели (т. е. адреса) на соседний или соседние элементы списка.

Связанный список – это структура данных в виде списка, элементы которого связаны друг с другом с помощью указателей. Наибольшее распространение получили две формы списка – стек и очередь.

Стек – это список с одной точкой входа. Данные добавляются в список и удаляются из него только с одной стороны списка (вершины стека). Таким образом, реализуется принцип – «последним вошел – первым вышел». Наглядный пример – трубка, запаянная с одного конца и заполняемая шариками, имеющими диаметр, равный внутреннему диаметру трубки. Первый вставляемый шарик всегда будет доставаться последним.

Очередь – это список с двумя точками входа. С одной стороны последовательности данные добавляются в список (в конец очереди), с другой стороны списка данные удаляются из него (из начала очереди). Таким образом, реализуется принцип – «первым вошел – первым вышел». Наглядный пример – трубка, открытая с обеих сторон, заполняемая шариками, имеющими диаметр, равный внутреннему диаметру трубки. С одной стороны трубки шарики добавляются в нее, с другой вынимаются.

18.2. Понятие рекурсивных данных и однонаправленные списки

Рекурсивный тип данных для односвязных списков имеет вид:

```
Type Tinf=integer; //Описание типа информационной части
     TSel=^Sel;  //Описание типа указателя на элемент
     Sel=record    // односвязных списков
           inf:Tinf;
```

```
    a:TSel;
  end;
```

Тип *Tinf* содержит описание информационной части элемента списка, которая в каждом конкретном случае своя. В данном случае и при последующем изложении материала будет использована эта простейшая структура информационной части. ***Не следует забывать переопределить её в той или иной конкретной программе.***

Как видим, описание типа *TSel* содержит внутри обращение к самому себе *(a:TSel)*, т.е. оно рекурсивно. При этом *a* является указателем на ячейку памяти точно такой же структуры. В поле *inf* размещается информационная часть элемента списка, причем, по крайней мере, в одном из полей *inf*, а иногда и в отдельном поле записи расположены сведения, по которым производится поиск или сортировка требуемой информации. Это поле будем обозначать *key:Tkey*, а сведения называть ***ключом***.

С помощью такого рекурсивного типа организуются однонаправленные ***связанные списки*** следующим образом: элементы списка размещаются в ячейках памяти типа *TSel*, причем в поле **a** каждой ячейки помещается адрес следующей за ней ячейки (рис. 18.1.):

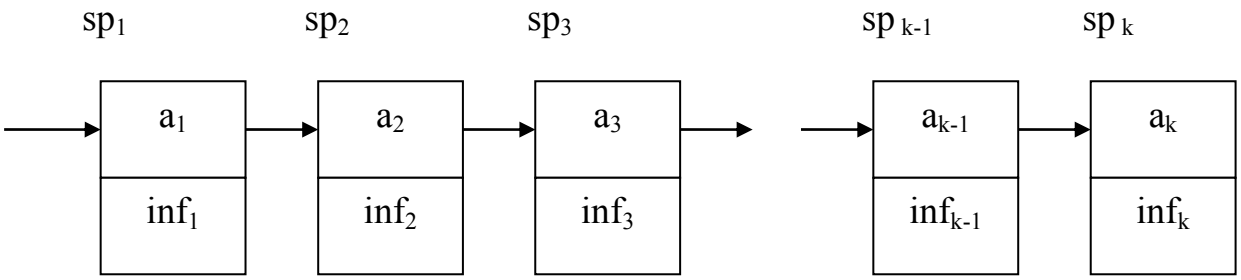


Рис 18.1.

Все ячейки динамически размещаются в куче по адресам $sp_1, sp_2, \ldots, sp_k$. При такой организации списков очень просто удалять или вставлять новые ячейки. Так, чтобы удалить ячейку с адресом sp_2, в адресную часть предшествующей ячейки нужно занести адрес последующей ячейки и освободить память, занимаемую удаляемой ячейкой (*sp1^.a:=sp3; dispose(sp2);* см. рис 18.2.).

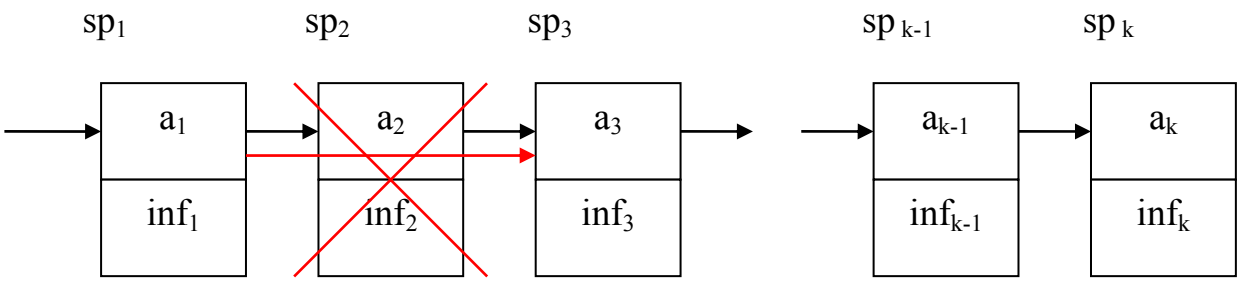


Рис 18.2.

Для добавлении новой ячейки между ячейками с адресами sp_1 и sp_2 необходимо создать новую ячейку, занести в нее информационную часть, в адресную часть предшествующей ячейки занести адрес новой ячейки, а в адресную часть новой ячейки адрес последующей ячейки (*new(sp); sp^.inf:=inf; sp1^.a:=sp; sp^.a:=sp2;* см. рис. 18.3.).

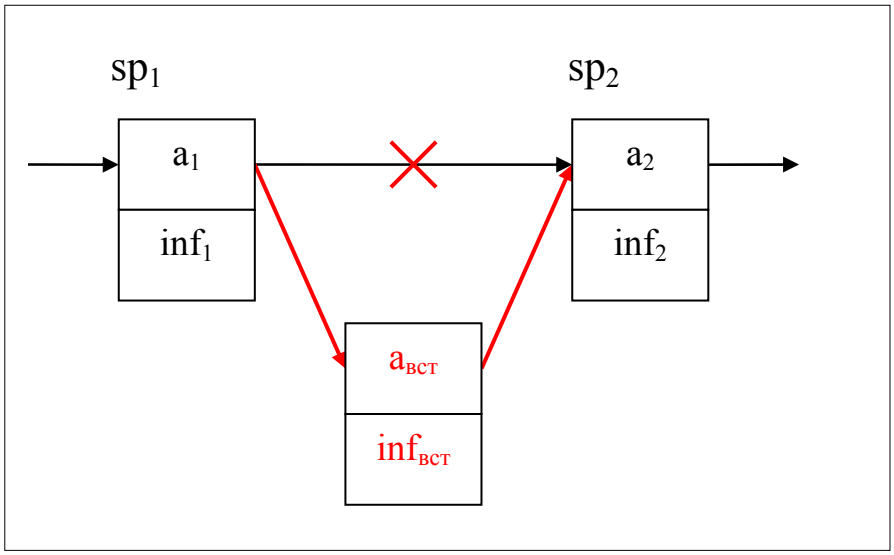


Рис. 18.3.

Адресная часть последней ячейки *spk^.a:=nil;* Для стека задается вершина стека, равная адресу 1-й ячейки (в обозначениях рис. 18.1.) *w:=sp1;*, а для односвязной очереди адрес начала очереди *w1:=sp1;* и адрес конца очереди (адрес последней ячейки) *wk:=spk;* .

Пример. Создать стек, содержащий целые числа, и найти количество положительных элементов стека

```
Type
  Tinf=integer; // Описание типа информационной части
  TSel=^Sel;
  Sel=record
      inf:Tinf;
      a:TSel;
   end;
```

```
 var k,n,kol:integer;
     w,t:TSel;
begin
  n:=strtoint(edit1.Text); // Количество элементов стека
  w:= nil;  Randomize;      //  Вершина стека
  for k:=1 to n do
    begin
    New(t); //  Создаем новый элемент
    t^.inf:=random(101)-50;//Запись информационной части
    t^.a:=w; // В адресную часть заносим прежнюю вершину
    w:=t;     //  Задаем новую вершину
    end;
  ListBox1.Clear;
  t:=w;  kol:=0;
  while t<>nil do
    begin
    k:=t^.inf;//Читаем информ. часть текущего элемента
    ListBox1.items.add(inttostr(k));
    if k>0 then inc(kol);
    t:=t^.a; // Задаем адрес следующего элемента
    end;
  label1.Caption:='Кол. полож. элементов стека = '
                +inttostr(kol);
end;
```

Для работы с односвязными списками необходимо создать, например, модуль *List1S,* содержащий описание рекурсивного типа и набор специализированных процедур.

18.3. Процедуры для работы со стеками

Добавить новый элемент в стек

```
procedure AddStek(var w:TSel; inf:Tinf);
     var t:TSel;
 begin
   New(t);       //  Создать новый элемент
   t^.inf:=inf; // Запись информационной части
   t^.a:=w; // В адресную часть заносим прежнюю вершину
   w:=t;     //  Задаем новую вершину
 end;
```

Прочитать *n-й* элемент стека (от вершины)

```
procedure ReadStek(w:TSel; n:integer; var inf:Tinf);
   var i:integer;
```

```
       t:TSel;
 begin
  if w=nil then exit;
  t:=w;        // Текущий адрес=вершине
  for i:=1 to n do
    begin
      inf:=t^.inf;
      t:=t^.a;  // Задаем адрес следующего элемента
    end;
 end;
```

Прочитать и удалить последний элемент стека (1-й от вершины)

```
procedure DelLast(var w:TSel; var inf:Tinf);
   var t:TSel;
 begin
  if w=nil then  exit;   // Стек пуст
  inf:=w^.inf;
  t:=w;          // Запоминаем адрес вершины
  w:=w^.a;//За новую вершину стека берем адрес след элем
  dispose(t); // Освобождаем память
 end;
```

Удалить стек

```
procedure DelStek(var w:TSel);
   var inf:Tinf;
  begin
    while w<>nil do DelLast(w,inf);
  end;
```

Вывод стека в ListBox

```
procedure WrtStek(w:Tsel; LS:TListBox);
    var t:Tsel;
 begin
  LS.clear;
  t:=w;     // Текущий адрес = вершине стека
  while t<>nil do
    begin
      LS.Items.Add(intToStr(t^.inf));
      t:=t^.a;     // Задаем адрес след элемента
    end;
 end;
```

Перестановка адресов двух соседних элементов, следующих за элементом с адресом *sp*

```
procedure RevAfter(sp:TSel);
      var t:TSel;
  begin
    t:=sp^.a^.a;
    sp^.a^.a:=t^.a;
    t^.a:=sp^.a;
    sp^.a:=t;
  end;
```

Обмен информации между элементом с адресом *sp* и следующим за ним

```
Procedure Revinf(sp:TSel);
     var inf:Tinf;
  begin
    inf:=sp^.inf;
    sp^.inf:=sp^.a^.inf;
    sp^.a^.inf:=inf;
end;
```

Пузырьковая сортировка стека перестановкой адресов

```
procedure SortAfter(var w:TSel);
     var p,t:Tsel;
         inf:Tinf;
 begin
   if (w=nil)or(w^.a=nil) then exit;//Пуст или 1 элемент
   AddStek(w,0);   // Добавляем пустой элемент
   t:=nil;
   repeat
     p:=w;
     while p^.a^.a<>t do
       begin
       if p^.a^.inf>p^.a^.a^.inf then RevAfter(p);
       p:=p^.a;
       end;
     t:=p^.a;
   until w^.a^.a=t;
   DelLast(w,inf);  // Удаляем пустой элемент
 end;
```

Пузырьковая сортировка стека обменом информации

```
procedure Sortinf(w:TSel);
     var p,t:Tsel;
begin
  if (w=nil)or(w^.a=nil) then exit;//Пуст или 1 элемент
```

```
  t:=nil;
  repeat
    p:=w;
    while p^.a<>t do
      begin
      if p^.inf>p^.a^.inf then Revinf(p);
      p:=p^.a;
      end;
    t:=p;
  until w^.a=t;
end;
```

18.4. Процедуры для работы с односвязными очередями

Добавление нового элемента в начало очереди

```
procedure AddBeg(var w1,wk:TSel; inf:Tinf);
    var t:TSel;
begin
  if w1=nil then begin  // Если очередь пуста
                   new(w1);
                   w1^.inf:=inf;
                   w1^.a:=nil;
                   wk:=w1;
                   end
              else begin
                   new(t);
                   t^.inf:=inf;
                   t^.a:=w1;
                   w1:=t;
                   end;
end;
```

Добавление нового элемента в конец очереди

```
procedure AddEnd(var w1,wk:TSel; inf:Tinf);
   var t:TSel;
begin
  if wk=nil then begin    // Если очередь пуста
                 new(wk);
                 wk^.inf:=inf;
                 wk^.a:=nil;
                 w1:=wk;
                 end
            else begin
                 new(t);
```

```
                    t^.inf:=inf;
                    t^.a:=nil;
                    wk^.a:=t;
                    wk:=t;
                    end;
 end;
```

Добавление нового элемента в середину стека или очереди после элемента с адресом *sp*

```
procedure AddAfter(sp:TSel; inf:Tinf);
     var t:TSel;
 begin
   if sp^.a=nil then exit; //Нельзя вставлять в конец
   new(t);
   t^.inf:=inf;
   t^.a:=sp^.a;
   sp^.a:=t;
 end;
```

Чтение и удаление элемента из начала очереди

```
procedure DelBeg(var w1,wk:TSel; var inf:Tinf);
    var t:tsel;
 begin
    if w1=nil then exit;  // Если очередь пуста
    inf:=w1^.inf;
    t:=w1;
    w1:=w1^.a;  // новое начало - адрес след элемента
    dispose(t);  // Освобождаем память
    if w1=nil then wk:=nil;
 end;
```

Чтение и удаление элемента из середины стека или очереди после элемента с адресом *sp*

```
procedure DelAfter(sp:TSel; var inf:Tinf);
    var t:tsel;
  begin   // Нельзя удалять последний  элемент
    if (sp^.a=nil) or (sp^.a^.a=nil) then exit;
    t:=sp^.a;         //  Адрес удаляемого элемента
    inf:=t^.inf;      //  Читаем информацию
    sp^.a:=t^.a;//Задаем адрес эл. стоящего за удаляемым
    dispose(t);     // Освобождаем память
  end;
```

Удалить очередь

```
procedure DelSpis(var w1,wk:TSel);
    var a:Tinf;
 begin
   while w1<>nil do DelBeg(w1,wk,a);
 end;
```

Процедура сортировки очереди слиянием двух отсортированных очередей *SortSl()* требует двух вспомогательных процедур: слияния двух отсортированных очередей в одну (отсортированную) *slip()* и разбиения очереди на две *div1s():*

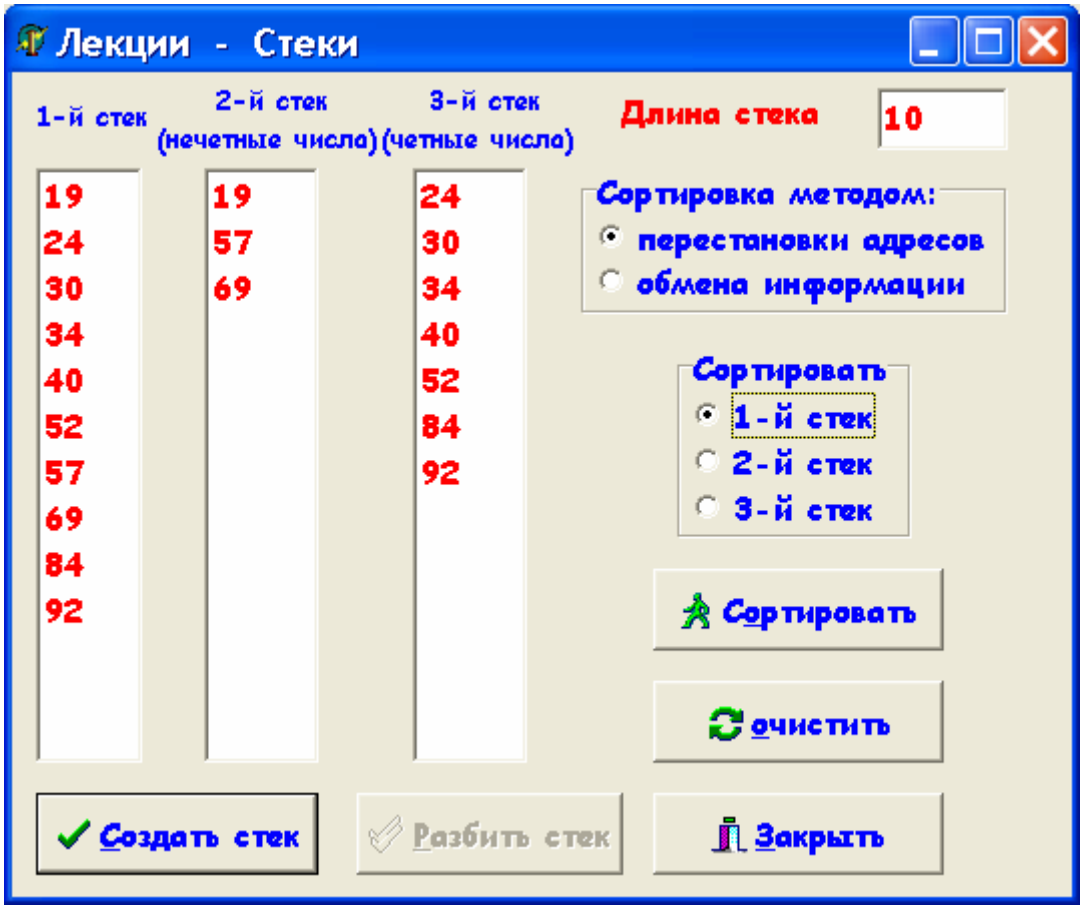
```
procedure slip (var sq1,sqk,sr1,srk,sp1,spk:TSel);
     var  infq,infr:Tinf;
 begin
   sp1:=nil;   spk:=nil;
   while (sq1<>nil) and (sr1<>nil) do
      begin
      DelBeg(sq1,sqk,infq);
      DelBeg(sr1,srk,infr);
      if infq<infr then begin
                        AddEnd(sp1,spk,infq);
                        AddBeg(sr1,srk,infr);
                        end
                   else begin
                        AddEnd(sp1,spk,infr);
                        AddBeg(sq1,sqk,infq);
                        end;
      end;
   while sq1<>nil do
      begin
      DelBeg(sq1,sqk,infq);
      AddEnd(sp1,spk,infq);
      end;
   while sr1<>nil do
      begin
      DelBeg(sr1,srk,infr);
      AddEnd(sp1,spk,infr);
      end;
 end;

procedure div1s(var sp1,spk,sq1,sqk,sr1,srk:TSel);
    var inf:Tinf;
 begin
```

```
   sr1:=nil;  srk:=nil;
   sq1:=nil;  sqk:=nil;
   while sp1<>nil do
      begin
      DelBeg(sp1,spk,inf);
      AddEnd(sq1,sqk,inf);
      if sp1<>nil then begin
                       DelBeg(sp1,spk,inf);
                       AddEnd(sr1,srk,inf);
                       end;
      end;
 end;

procedure SortSl(var w1,wk:TSel);
    var sq1,sr1,sqk,srk:TSel;
 begin
   if w1<>wk then
      begin
      div1s(w1,wk,sq1,sqk,sr1,srk);
      sortsl(sq1,sqk);
      sortsl(sr1,srk);
      slip(sq1,sqk,sr1,srk,w1,wk);
      end;
 end;
```

Пример программы по созданию стека из случайных целых чисел, формированию из него двух стеков, содержащих четные и нечетные числа, предусматривающую сортировку методами обмена информации и адресов.



Листинг 18.1

```
unit Unit1;
                              interface
uses Windows,Messages,SysUtils,Variants,Classes,
   Graphics,Controls,Forms,Dialogs,StdCtrls,ExtCtrls,
      Buttons;
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    ListBox2: TListBox;
    ListBox3: TListBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    RadioGroup1: TRadioGroup;
    RadioGroup2: TRadioGroup;
    BitBtn3: TBitBtn;
    BitBtn4: TBitBtn;
    BitBtn5: TBitBtn;
```

```
    Edit1: TEdit;
    Label4: TLabel;
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
    procedure BitBtn3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
    procedure BitBtn5Click(Sender: TObject);
    procedure RadioGroup2Click(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var  Form1: TForm1;
                          implementation
uses List1S;
var w1,w2,w3:TSel;
{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
  begin
   bitbtn2.enabled:=false;  //  Разбить
   bitbtn3.enabled:=false;  //  Сортировать
   bitbtn5.Enabled:=false;  //  Очистить
  end;

procedure TForm1.BitBtn1Click(Sender: TObject);
    var k,n : integer;  // Создать стек
begin
  n:=strtoint(edit1.Text);
  ListBox1.Clear;
  ListBox2.Clear;
  ListBox3.Clear;
  w1:=nil;  Randomize;
  for k:=1 to n do  AddStek(w1,Random(100));
  WrtStek(w1,ListBox1);
  bitbtn2.enabled:=true; //  Разбить
  bitbtn3.enabled:=true; //  Сортировать
  bitbtn5.Enabled:=true; //  Очистить
end;
```

```
procedure TForm1.BitBtn2Click(Sender: TObject);
   var  t:TSel; //  Разбить стек
        k:integer;
begin
  ListBox2.Clear;  ListBox3.Clear;
  t:=w1;  w2:=nil;  w3:=nil;
  while t<>nil do
     begin
     k:=t^.inf; //Читаем информ. часть текущего элемента
     if  odd(k) then  AddStek(w2,k)
                else  AddStek(w3,k);
     t:=t^.a; // Задаем адрес следующего элемента
     end;
  WrtStek(w2,ListBox2);
  WrtStek(w3,ListBox3);
  bitbtn2.enabled:=false;  //  Разбить
end;

procedure TForm1.BitBtn3Click(Sender: TObject);
begin    //  Сортировать
  case RadioGroup2.ItemIndex of
    0: begin
         case RadioGroup1.ItemIndex of
           0: SortAfter(w1);
           1: Sortinf(w1);
         end;
         WrtStek(w1,ListBox1);
       end;
    1: begin
         case RadioGroup1.ItemIndex of
           0: SortAfter(w2);
           1: Sortinf(w2);
         end;
         WrtStek(w2,ListBox2);
       end;
    2: begin
         case RadioGroup1.ItemIndex of
           0: SortAfter(w3);
           1: Sortinf(w3);
         end;
         WrtStek(w3,ListBox3);
       end;
  end;
    bitbtn3.enabled:=false;
```

```
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
  begin
    bitbtn3.enabled:=true;
  end;

procedure TForm1.RadioGroup2Click(Sender: TObject);
  begin
   bitbtn3.Enabled:=true;
  end;

procedure TForm1.BitBtn5Click(Sender: TObject);
  begin                 // Очистить
   listbox1.Clear;
   listbox2.Clear;
   listbox3.Clear;
   DelStek(w1);// Стирание стека
   DelStek(w2);// Стирание стека
   DelStek(w3);// Стирание стека

   bitbtn2.Enabled:=false;
   bitbtn3.Enabled:=false;
   bitbtn5.Enabled:=false;
  end;

procedure TForm1.BitBtn4Click(Sender: TObject);
begin      // Close
   DelStek(w1);    // Стирание стека
   DelStek(w2);   // Стирание стека
   DelStek(w3);   // Стирание стека
end;
end.
```

18.5. Работа с двухсвязными очередями

Двухсвязные списки организуются, когда требуется просматривать список, как в прямом, так и в обратном направлениях. Эта проблема легко решается, если ввести рекурсивный тип с двумя адресными ячейками:

```
Type Tinf=integer; //Описание типа информационной части
     Tseld=^seld; //Описание типа  указателя на элемент
     seld=record     //   двухсвязных списков
            inf:Tinf;
            a,b:Tseld;
          end;
```

Структура двусвязного списка, схемы удаления внутреннего элемента списка и вставки нового элемента в середину списка приведены соответственно на рис. 18.4.-18.6. ниже.

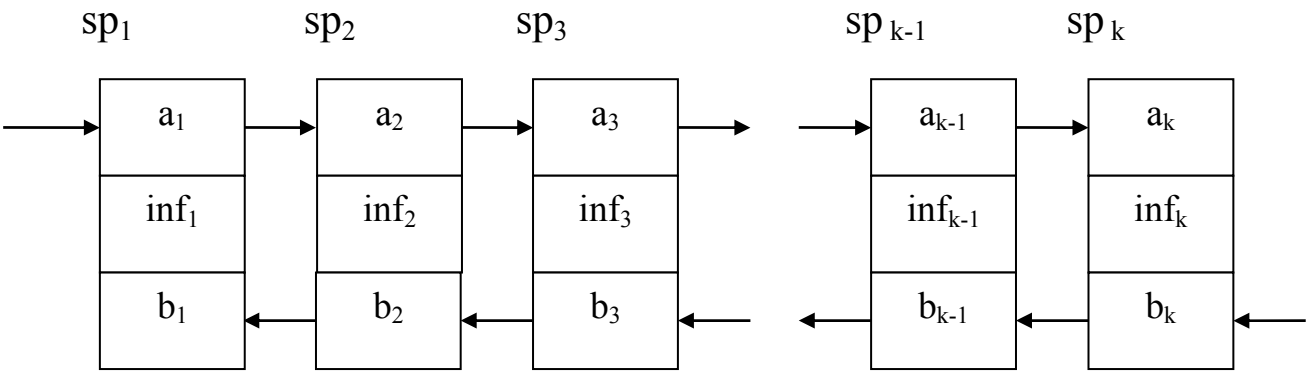


Рис. 18.4.

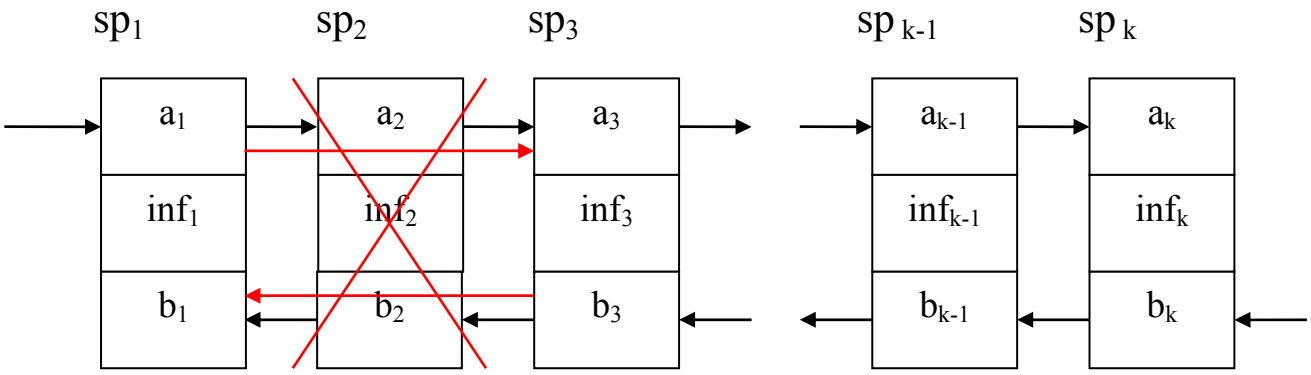


Рис18.5.

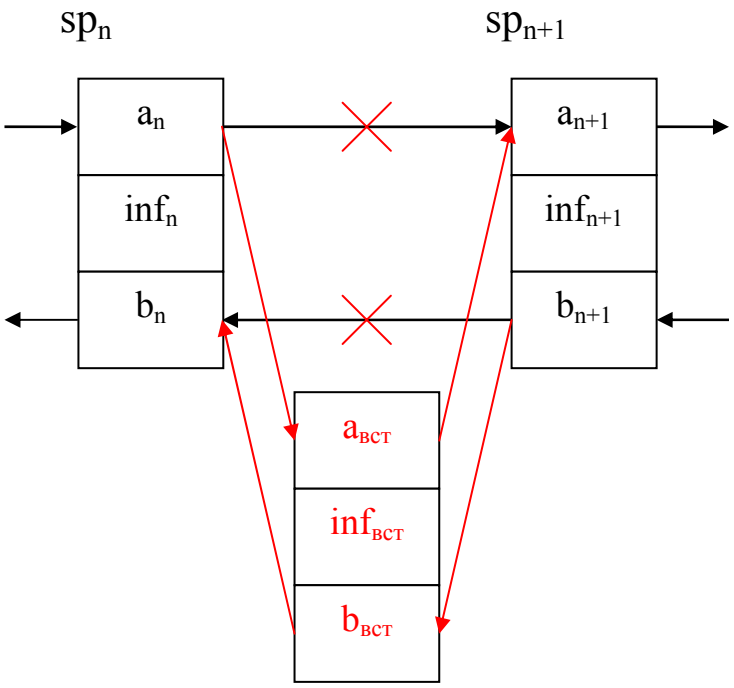


Рис. 18.6.

18.6. Процедуры для работы с двусвязными очередями

Для работы с двухсвязными списками необходимо создать, например, модуль *List2S*, содержащий описание рекурсивного типа и набор специализированных процедур.

Вставка нового элемента в начало очереди

```
Procedure AddBegD(var w1,wk:Tseld; inf:Tinf);
   var t:Tseld;
begin
  New(t);
  t^.inf:=inf;
  t^.b:=nil;
  if w1=nil then begin t^.a:=nil; w1:=t; wk:=t; end
            else begin t^.a:=w1; w1^.b:=t; w1:=t; end;
end;
```

Вставка нового элемента в конец очереди

```
Procedure AddEndD(var w1,wk:Tseld; inf:Tinf);
   var t:Tseld;
begin
  New(t);
  t^.inf:=inf;
  t^.a:=nil;
  if wk=nil then begin t^.b:=nil; w1:=t; wk:=t; end
            else begin t^.b:=wk; wk^.a:=t; wk:=t; end;
end;
```

Вставка нового элемента после элемента с адресом *sp*

```
Procedure AddAfterD(sp:Tseld; inf:Tinf);
   var t:Tseld;
begin        // Нельзя добавлять в конец
  if (sp=nil) or (sp^.a=nil) then exit;
  New(t);
  t^.inf:=inf;
  t^.a:=sp^.a;
  t^.b:=sp;
  sp^.a:=t;
  t^.a^.b:=t;
end;
```

Вставка нового элемента перед элементом с адресом *sp*

```
Procedure AddBefD(sp:Tseld; inf:Tinf);
   var t:Tseld;
begin    // Нельзя добавлять в начало
  if (sp=nil) or (sp^.b=nil) then exit;
  New(t);
  t^.inf:=inf;
  t^.a:=sp;
  t^.b:=sp^.b;
  sp^.b^.a:=t;
  sp^.b:=t;
end;
```

Чтение и удаление элемента с начала очереди

```
Procedure DelBegD(var w1,wk:Tseld; var inf:Tinf);
   var t:Tseld;
begin
  if w1=nil then exit;
  inf:=w1^.inf;
  t:=w1;
```

```
  if w1=wk then begin w1:=nil; wk:=nil; end
           else begin w1^.a^.b:=nil; w1:=w1^.a; end;
  Dispose(t);
end;
```

Чтение и удаление элемента с конца очереди

```
Procedure DelEndD(var w1,wk:Tseld; var inf:Tinf);
   var t:Tseld;
begin
  if wk=nil then exit;
  inf:=wk^.inf;
  t:=wk;
  if w1=wk then begin w1:=nil; wk:=nil; end
           else begin wk^.b^.a:=nil; wk:=wk^.b; end;
  Dispose(t);
end;
```

Чтение и удаление внутреннего элемента очереди с адресом *sp*

```
Procedure DelD(sp:Tseld; var inf:Tinf);
     begin   // Нельзя удалять первый или последний
  if (sp=nil) or (sp^.a=nil) or (sp^.b=nil) then exit;
  inf:=sp^.inf;
  sp^.b^.a:=sp^.a;
  sp^.a^.b:=sp^.b;
  Dispose(sp);
end;
```

Удаление очереди

```
Procedure DelSpisD(var w1,wk:Tseld);
   var inf:Tinf;
begin
  while w1<>nil do DelBegD(w1,wk,inf);
end;
```

Вывод в *Memo* с начала очереди

```
procedure WrtBegD(w1:Tseld; s:TMemo);
   var t:Tseld;
begin
   t:=w1;   s.clear;
   while  t<>nil  do
      begin
      s.lines.add(IntToStr(t^.inf));
      t:=t^.a;
      end;
```

```
end;
```

Вывод в *Memo* с конца очереди

```
procedure WrtEndD(wk:Tseld; s:TMemo);
   var t:Tseld;
begin
   t:=wk;  s.clear;
   while  t<>nil  do
      begin
      s.lines.add(IntToStr(t^.inf));
      t:=t^.b;
      end;
end;
```

Процедура сортировки очереди слиянием двух отсортированных очередей *SortSlD()* требует двух вспомогательных процедур: слияния двух отсортированных очередей в одну (отсортированную) *slipD()* и разбиения очереди на две *div2s():*

```
procedure slipD (var sq1,sqk,sr1,srk,sp1,spk:TSeld);
     var  infq,infr:Tinf;
 begin
   sp1:=nil;   spk:=nil;
   while (sq1<>nil) and (sr1<>nil) do
     begin
     DelBegD(sq1,sqk,infq);
     DelBegD(sr1,srk,infr);
     if infq<infr then begin
                       AddEndD(sp1,spk,infq);
                       AddBegD(sr1,srk,infr);
                       end
                  else begin
                       AddEndD(sp1,spk,infr);
                       AddBegD(sq1,sqk,infq);
                       end;
     end;
   while sq1<>nil do
     begin
     DelBegD(sq1,sqk,infq);
     AddEndD(sp1,spk,infq);
     end;
   while sr1<>nil do
     begin
     DelBegD(sr1,srk,infr);
```

```
     AddEndD(sp1,spk,infr);
     end;
 end;

procedure div2s(var sp1,spk,sq1,sqk,sr1,srk:TSeld);
    var inf:Tinf;
 begin
   sr1:=nil;  srk:=nil;
   sq1:=nil;  sqk:=nil;
   while sp1<>nil do
      begin
      DelBegD(sp1,spk,inf);
      AddEndD(sq1,sqk,inf);
      if sp1<>nil then begin
                       DelBegD(sp1,spk,inf);
                       AddEndD(sr1,srk,inf);
                       end;
      end;
 end;

procedure sortslD(var w1,wk:TSeld);
    var sq1,sr1,sqk,srk:TSeld;
  begin
    if w1<>wk then
         begin
         div2s(w1,wk,sq1,sqk,sr1,srk);
         sortslD(sq1,sqk);
         sortslD(sr1,srk);
         slipD(sq1,sqk,sr1,srk,w1,wk);
         end;
  end;
```

Пример программы. Создать два двухсвязных списка из случайных целых чисел. Заменить элементы между максимальным и минимальным 1-го списка вторым списком. Создать новый список из удаленных элементов. Отсортировать результирующий список..


```
unit Unit1;
                                interface
uses   Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs, ExtCtrls, StdCtrls,Grids,
Buttons, List2S;
type  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    BitBtn1: TBitBtn;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Button4: TButton;
    Memo1: TMemo;
    Memo2: TMemo;
    Memo3: TMemo;
    Memo4: TMemo;
    Memo5: TMemo;
    Button5: TButton;
    Memo6: TMemo;
    Label1: TLabel;
```

```
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
   Form1: TForm1;
   w11, w12, w13, wk1, wk2, wk3 : Tseld;

             implementation

 {$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  memo1.Clear;    memo2.Clear;    memo3.Clear;
  memo4.Clear;    memo5.Clear;    memo6.Clear;
  w11:=nil;   wk1:=nil;   w12:=nil;   wk2:=nil;
  w13:=nil;   wk3:=nil;        Randomize;
  button4.enabled:=false;
end;

procedure TForm1.Button1Click(Sender: TObject);
  begin   //  добавить в 1-й список
     AddEndD(w11,wk1,Random(100));
     WrtBegD(w11,memo1); // вывод 1-го списка
    button3.Enabled:=true;
  end;

procedure TForm1.Button2Click(Sender: TObject);
  begin //  Добавить во 2-й список
     AddEndD(w12,wk2,Random(100));
     WrtBegD(w12,memo2); // вывод 2-го списка
  end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
   var t,tmin,tmax:Tseld;  // Выполнить
       i,min,max:integer;
       inf:Tinf;
 begin
   t:=w11;   tmin:=t;   tmax:=t;
   i:=1;     min:=1;    max:=1;
   while  t<>nil  do
    begin
    if t^.inf<tmin^.inf then begin tmin:=t; min:=i; end;
    if t^.inf>tmax^.inf then begin tmax:=t; max:=i; end;
    t:=t^.a;   inc(i);
    end;
  if min=max then
     begin
       ShowMessage('Заново создать 1-й список');
       button3.Enabled:=false;
       exit;
     end;
  if min>max then
      begin t:=tmin; tmin:=tmax; tmax:=t; end;
  if abs(max-min)>1 then
    begin
      t:=tmin^.a;
      while t<>tmax do
      begin//Удаление элем. между min и max в 1-м списке
      DelD(t,inf);
      AddEndD(w13,wk3,inf);  //Вставка их в 3-й список
      t:=tmin^.a;
      end;
    end;
 WrtBegD(w11,memo3); // вывод промежуточного 1-го списка
 WrtBegD(w13,memo4); // вывод вырезанного списка
 t:=tmin;
 while w12<>nil do
    begin
      DelBegD(w12,wk2,inf);   // Вставка 2-го списка
      AddAfterD(t,inf);  // между min и max 1-го списка
      t:=t^.a;
    end;
  WrtBegD(w11,memo5);   // вывод итогового списка
  button4.enabled:=true;
end;
```

```
procedure TForm1.Button4Click(Sender: TObject);
    begin                 // Сортировать
      sortslD(w11,wk1);
      WrtBegD(w11,memo6);
   end;

procedure TForm1.Button5Click(Sender: TObject);
   begin   // Очистить
     memo1.Clear;    memo2.Clear;    memo3.Clear;
     memo4.Clear;    memo5.Clear;    memo6.Clear;
     DelSpisD(w11,wk1);  // Удаление очереди
     DelSpisD(w12,wk2);  // Удаление очереди
     DelSpisD(w13,wk3);  // Удаление очереди
  end;

procedure TForm1.BitBtn1Click(Sender: TObject);
   begin  // Close
      DelSpisD(w11,wk1);  // Удаление очереди
      DelSpisD(w12,wk2);  // Удаление очереди
      DelSpisD(w13,wk3);  // Удаление очереди
   end;

end.
```

ЛЕКЦИЯ 19. АЛГОРИТМЫ РЕШЕНИЯ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

19.1. Основные понятия и определения

Выделяют четыре основные задачи линейной алгебры: решение СЛАУ, вычисление определителя матрицы, нахождение обратной матрицы, определение собственных значений и собственных векторов матрицы.

Задача отыскания решения СЛАУ с n неизвестными - одна из наиболее часто встречающихся в практике вычислительных задач, так как большинство методов решения сложных задач основано на сведении их к решению некоторой последовательности СЛАУ.

Обычно СЛАУ записывается в виде

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \ ; \ 1 \le i \le n, \quad \text{или в матричном виде } A\vec{x} = \vec{b},$$

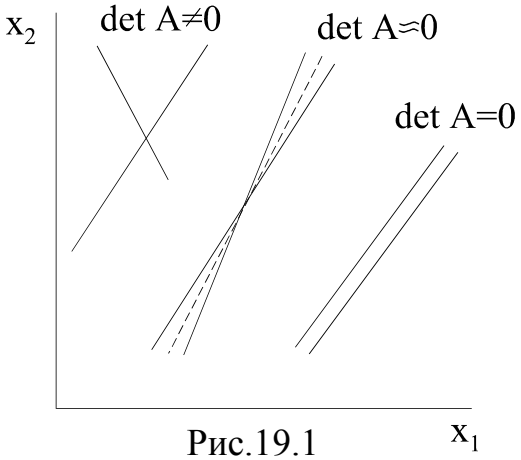


Рис.19.1

$$A = \begin{bmatrix} a_{11}a_{12}...a_{1n} \\ a_{21}a_{22}...a_{2n} \\ ... \\ a_{n1}a_{n2}...a_{nn} \end{bmatrix}; \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix}; \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix}. \quad (19.1)$$

Здесь A и $\vec{b}$ заданы, требуется найти $\vec{x}^*$, удовлетворяющий (19.1).

Известно, что если определитель матрицы $|A| \neq 0$, то СЛАУ имеет единственное решение. В противном случае решение либо отсутствует (если $\vec{b} \neq 0$), либо имеется бесконечное множество решений (если $\vec{b} = 0$). При решении систем, кроме условия $\det|A| \neq 0$, важно чтобы задача была ***корректной***, т.е. чтобы при малых погрешностях правой части $\Delta\vec{b}$ и (или) коэффициентов Δa_{ij} погрешность решения $\Delta\vec{x}^*$ также оставалась малой. Признаком некорректности, или плохой обусловленности, является близость к нулю определителя матрицы.

Плохо обусловленная система двух уравнений геометрически соответствует почти параллельным прямым (рис.19.1). Точка пересечения таких прямых (решение) при малейшей погрешности коэффициентов резко сдвигается. Обусловленность (корректность) СЛАУ характеризуется числом $\chi = \|A\| \cdot \|A^{-1}\| \ge 1$. Чем дальше χ от 1, тем хуже обусловлена система. Обычно при $\chi > 10^3$ система некорректна и требует специальных методов решения -

методов регуляризации. Приведенные ниже методы применимы только для корректных систем.

Методы решения СЛАУ делятся на прямые и итерационные.

Прямые методы дают в принципе точное решение (если не учитывать ошибок округления) за конечное число арифметических операций. Для хорошо обусловленных СЛАУ небольшого порядка $n \le 200$ применяются практически только прямые методы.

Наибольшее распространение среди прямых методов получили **метод Гаусса** для СЛАУ общего вида, его модификация для трехдиагональной матрицы - **метод прогонки** и **метод квадратного корня** для СЛАУ с симметричной матрицей.

Итерационные методы основаны на построении сходящейся к точному решению $\vec{x}*$ рекуррентной последовательности векторов $(\vec{x}^o, \vec{x}^1, \vec{x}^2, ..., \vec{x}^k \xrightarrow[k\to\infty]{} \vec{x}^*$. Итерации выполняют до тех пор, пока норма разности $\delta_k = \left\| \overrightarrow{x}^{k} - \overrightarrow{x}^{k-1} \right\| = \max_i \left| x_i^k - x_i^{k-1} \right| \le \varepsilon$. (ε- заданная малая величина).

Итерационные методы выгодны для систем большого порядка *n>100*, а также для решения плохо обусловленных систем. Многообразие итерационных методов решения СЛАУ объясняется возможностью большого выбора рекуррентных последовательностей, определяющих метод. Наибольшее распространение среди итерационных методов получили одношаговые **методы простой итерации и Зейделя** с использованием релаксации.

Для контроля полезно найти невязку полученного решения $\vec{x}$:

$$\Delta = \max_{1\le k\le n} \left| b_k - \sum_{i=1}^{n} a_{ki} x_i \right|;$$

если Δ велико, то это указывает на грубую ошибку в расчете.

Ниже приведено описание алгоритмов указанных методов решения СЛАУ.

19.2. Прямые методы решения СЛАУ

19.2.1. Метод Гаусса

Метод основан на приведении с помощью преобразований, не меняющих решение, исходной СЛАУ (19.1) с произвольной матрицей к СЛАУ с верхней треугольной матрицей вида

$$\begin{aligned} a'_{11}x_1 + a'_{12}x_2 + ... + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + ... + a'_{2n}x_n &= b'_2 \\ ..... & \\ a'_{nn}x_n &= b'_n \end{aligned} \qquad (19.2)$$

Этап приведения к системе с треугольной матрицей называется ***прямым ходом метода Гаусса*.**

Решение системы с верхней треугольной матрицей (19.2), как легко видеть, находится по формулам, называемым ***обратным ходом метода Гаусса*:**

$$x_n = b'_n / a'_{nn};\ x_k = \frac{1}{a'_{kk}}\left[b'_k - \sum_{i=k+1}^{n} a'_{ki} x_i \right],\ k = n-1,\ n-2,\ ...,\ 1.. \qquad (19.3)$$

Прямой ход метода Гаусса осуществляется следующим образом: вычтем из каждого *m*-го уравнения *(m=2,3,..,n)* первое уравнение, умноженное на a_{m1} / a_{11}, и вместо *m*-го уравнения подставим полученное. В результате в матрице системы исключаются все коэффициенты 1-го столбца ниже диагонального. Затем, используя 2-е полученное уравнение, аналогично исключим элементы второго столбца *(m=3,4,..,n)* ниже диагонального и т.д. Такое исключение называется ***циклом метода Гаусса***. Проделывая последовательно эту операцию с расположенными ниже k-го уравнениями *(k=1, 2 ,..., n-1),* мы приходим к системе вида (19.2). При указанных операциях решение СЛАУ не изменяется.

На каждом *k*-м шаге преобразований прямого хода элементы матриц изменяются по ***формулам прямого хода метода Гаусса***:

$$\begin{aligned} a_{mi} &= a_{mi} - a_{ki}\frac{a_{mk}}{a_{kk}}, \qquad k = 1,\ n-1, \quad i = k,\ n; \\ b_m &= b_m - b_k \frac{a_{mk}}{a_{kk}}, \qquad m = k+1,\ n. \end{aligned} \qquad (19.4)$$

Элементы a_{kk} называются главными. Заметим, что если в ходе расчетов по данному алгоритму на главной диагонали окажется нулевой элемент $a_{kk} = 0$, то произойдет сбой в ЭВМ. Для того чтобы избежать этого, следует каждый цикл по *k* начинать с перестановки строк: среди элементов *k*-го столбца a_{mk}, $k \le m \le n$ находят номер *p* главного, т.е. наибольшего по модулю, и меняют местами строки *k* и *p*. Такой выбор главного элемента значительно повышает устойчивость алгоритма к ошибкам округления, т.к. в формулах (19.4) при этом производится умножение на числа a_{mk} / a_{kk} меньшие единицы и ошибка, возникшая ранее, уменьшается.

19.2.2. Метод прогонки

Многие задачи (например, решение дифференциальных уравнений 2-го порядка) приводят к необходимости решения СЛАУ с трехдиагональной матрицей:

$$\begin{vmatrix} q_1 & r_1 & 0 & 0 & .. & 0 & 0 & 0 \\ p_2 & q_2 & r_2 & 0 & .. & 0 & 0 & 0 \\ 0 & p_3 & q_3 & r_3 & .. & 0 & 0 & 0 \\ .. & .. & .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & 0 & .. & p_{n-1} & q_{n-1} & r_{n-1} \\ 0 & 0 & 0 & 0 & .. & 0 & p_n & q_n \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ .. \\ x_{n-1} \\ x_n \end{vmatrix} = \begin{vmatrix} d_1 \\ d_2 \\ d_3 \\ .. \\ d_{n-1} \\ d_n \end{vmatrix} \qquad (19.5)$$

или коротко эту систему записывают в виде

$$\begin{aligned} & q_1 x_1 + r_1 x_2 = d_1 \\ & p_i x_{i-1} + q_i x_i + r_i x_{i+1} = d_i \\ & p_n x_{n-1} + q_n x_n = d_n \quad , \\ & 2 \le i \le n-1. \end{aligned} \qquad (19.6)$$

В этом случае расчетные формулы метода Гаусса значительно упрощаются. После исключения поддиагональных элементов в результате прямого хода метода Гаусса и последующего деления каждого уравнения на диагональный элемент систему (19.5) можно привести к виду

$$\begin{vmatrix} 1 & -\xi_1 & 0 & .. & 0 & .. & 0 & 0 \\ 0 & 1 & -\xi_2 & .. & 0 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & 0 & .. & 1 & -\xi_{n-1} \\ 0 & 0 & 0 & .. & 0 & .. & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ .. \\ x_{n-1} \\ x_n \end{vmatrix} = \begin{vmatrix} \eta_1 \\ \eta_2 \\ .. \\ \eta_{n-1} \\ \eta_n \end{vmatrix} \qquad (19.7)$$

При этом ***формулы прямого хода*** для вычисления ξ_i, η_i, как нетрудно получить, имеют вид

$$\begin{aligned} & \xi_1 = -r_1 / q_1; \qquad \eta_1 = d_1 / q_1; \\ & \xi_i = -r_i /(q_i + p_i \xi_{i-1}); \quad \eta_i = (d_i - p_i \eta_{i-1})/(q_i + p_i \xi_{i-1}); \\ & i = 2,3,...,n-1. \end{aligned} \qquad (19.8)$$

Когда такое преобразование (прямой ход) сделано, ***формулы обратного хода*** метода Гаусса получаются в виде

$$\begin{aligned} & x_n = (d_n - p_n \eta_{n-1})/(q_n + p_n \xi_{n-1}); \\ & x_i = \xi_i x_{i+1} + \eta_i \ , \\ & i = n-1,\ n-2,\ ...,\ 1. \end{aligned} \qquad (19.9)$$

Расчетные формулы (19.8), (19.9) получили название ***«метод прогонки»***. Достаточным условием того, что в формулах метода прогонки не произойдет деления на нуль и расчет будет устойчив относительно погрешностей округления, является выполнение неравенства $|q_i| \ge |p_i| + |r_i|$ (хотя бы для одного *i* должно быть строгое неравенство).

19.2.3. Метод квадратного корня

Предназначен для решения СЛАУ с симметричной матрицей. Этот метод основан на представлении такой матрицы в виде произведения трех матриц: $A = S^T \cdot D \cdot S,$ где D - диагональная с элементами $d_i = \pm 1$; S - верхняя треугольная ($s_{ik} = 0,$ если $i>k,$ причем $s_{ii} > 0$), S^T - транспонированная нижняя треугольная. Матрицу S можно по аналогии с числами трактовать как корень квадратный из матрицы A, отсюда и название метода.

Если S и D известны, то решение исходной системы $A \cdot \vec{x} = S^T \cdot D \cdot S \cdot \vec{x} = \vec{b}$ сводится к последовательному решению трех систем - двух треугольных и одной диагональной:

$$S^T \cdot \vec{z} = \vec{b};\ \ D\vec{y} = \vec{z};\ \ S\vec{x} = \vec{y}\,. \qquad (19.10)$$

Здесь $\vec{z} = D\,S\,\vec{x},\ \ \vec{y} = S\,\vec{x}\,.$

Решение систем (19.10) ввиду треугольности матрицы S осуществляется по формулам, аналогичным обратному ходу метода Гаусса:

$$y_1 = b_1 / s_{11} d_1;\quad y_i = (b_i - \sum_{k=1}^{i-1} d_k y_k s_{ki}) / s_{ii} d_i;\ \ i = 2,\ 3\ ,...,\ n;$$

$$x_n = y_n / s_{nn};\quad x_i = (y_i - \sum_{k=i+1}^{n} s_{ik} x_k) / s_{ii};\ \ i = n-1,\ n-2,\ ...,\ 1.$$

Нахождение элементов матрицы S (извлечение корня из А) осуществляется по рекуррентным формулам:

$$d_k = sign(a_{kk} - \sum_{i=1}^{k-1} d_i \left| s_{ik} \right|^2);$$

$$s_{kk} = \sqrt{\left| a_{kk} - \sum_{i=1}^{k-1} d_i \left| s_{ik} \right|^2 \right|}; \qquad (19.11)$$

$$k = 1, 2, \ldots, n;$$

$$s_{kj} = (a_{kj} - \sum_{i=1}^{k-1} d_i s_{ik} s_{ij}) / (s_{kk} d_k);$$

$$j = k+1,\ k+2,\ \ldots,\ n.$$

В этих формулах сначала полагаем $k=1$ и последовательно вычисляем $d_1 = sign(a_{11});\ \ s_{11} = \sqrt{|a_{11}|}$ и все элементы первой строки матрицы $S\ \ (s_{1j}, j>1)$, затем полагаем $k=2$, вычисляем s_{22} и вторую строку s_{1j} для $j>2$ и т.д.

Метод квадратного корня почти вдвое эффективнее метода Гаусса, т.к. полезно использует симметричность матрицы.

Функция **sign(x)** возвращает -1 для всех $x<0$ и $+1$ для всех $x>0$.

19.3. Итерационные методы решения СЛАУ

19.3.1. Метод простой итерации

В соответствии с общей идеей итерационных методов исходная система (19.1) должна быть приведена к виду, разрешенному относительно $\vec{x}$:

$$\vec{x} = G\vec{x} + \vec{c} = \varphi(\vec{x}), \qquad (19.12)$$

где G - матрица; $\vec{c}$ - столбец свободных членов. При этом решение (19.12) должно совпадать с решением (19.1). Затем строится рекуррентная последовательность первого порядка в виде

$$\vec{x}^k = \varphi(\vec{x}^{k-1}) = G\vec{x}^{k-1} + \vec{c}, \quad k = 1, 2, \ldots .$$

Для начала вычислений задается некоторое начальное приближение $\vec{x}^0$ (например, $x_1^0 = 1, \ldots, x_n^0 = 1$), для окончания - некоторое малое ε.

Получаемая последовательность будет сходиться к точному решению, если норма матрицы $\|G\| < 1$.

Привести исходную систему к виду (19.12) можно различными способами, например,

$$\vec{x} = \vec{x} + \alpha(A\vec{x} - \vec{b}) = (E + \alpha A)\vec{x} - \alpha\vec{b} = G\vec{x} + \vec{c};$$

Здесь E - единичная матрица; α - некоторый параметр, подбирая который, можно добиться, чтобы $\|G\| = \|E + \alpha A\| < 1$.

В частном случае, если исходная матрица A имеет преобладающую главную диагональ, т.е. $|a_{ii}| > \sum\limits_{\substack{k=1 \\ k \neq i}}^{n} |a_{ik}|$, то преобразование (19.1) к (19.12) можно осуществить просто, решая каждое i-е уравнение относительно x_i. В результате получим следующую рекуррентную формулу:

$$x_i^k = \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{k-1} \right) / a_{ii} = \sum_{j=1}^{n} g_{ij} x_j^{k-1} + c_i . \qquad (19.13)$$

$$g_{ij} = -a_{ij} / a_{ii}; \quad g_{ii} = 0; \quad c_i = b_i / a_{ii} .$$

19.3.2. Метод Зейделя

Метод Зейделя является модификацией метода простой итерации. Суть его состоит в том, что при вычислении очередного приближения $x_i^k (2 \le i \le n)$ в формуле (19.13) используются вместо $x_1^{k-1}, \ldots, x_{i-1}^{k-1}$ уже вычисленные ранее $x_1^k, \ldots, x_{i-1}^k$, т.е. (19.13) преобразуется к виду

$$x_i^k = \sum_{j=1}^{i-1} g_{ij} x_j^k + \sum_{j=i+1}^{n} g_{ij} x_j^{k-1} + c_i . \qquad (19.14)$$

Такое усовершенствование позволяет ускорить сходимость итераций почти в два раза. Кроме того, данный метод может быть реализован на ЭВМ без привлечения дополнительного массива, так как полученное новое x_i^k сразу засылается на место старого.

19.3.3. Понятие релаксации

Методы простой итерации и Зейделя сходятся примерно так же, как геометрическая прогрессия со знаменателем $\|G\|$. Если норма матрицы G близка к 1, то сходимость очень медленная. Для ускорения сходимости используется метод релаксации. Суть его в том, что полученное по методу простой итерации или Зейделя очередное значение x_i^k пересчитывается по формуле

$$x_i^k = \omega x_i^k + (1 - \omega) x_i^{k-1} \qquad (19.15)$$

Здесь $0 < \omega \le 2$ - параметр релаксации.

Если $\omega < 1$ - *нижняя релаксация,* если $\omega > 1$ - *верхняя релаксация.* Параметр ω подбирают так, чтобы сходимость метода достигалась за минимальное число итераций.

ЛЕКЦИЯ 20. АППРОКСИМАЦИЯ ФУНКЦИЙ

20.1. Зачем нужна аппроксимация функций?

В окружающем нас мире все взаимосвязано, поэтому одной из наиболее часто встречающихся задач является установление характера зависимости между различными величинами, что позволяет по значению одной величины определить значение другой. Математической моделью зависимости одной величины от другой является понятие функции *y=f(x)*.

В практике расчетов, связанных с обработкой экспериментальных данных, вычислением *f(x),* разработкой вычислительных методов встречаются следующие две ситуации:

1. Как установить вид функции *y=f(x),* если она неизвестна? Предполагается при этом, что задана таблица ее значений $\left\{ \left(x_i,\ y_i\right),\ i=1,\ m \right\}$, которая получена либо из экспериментальных измерений, либо из сложных расчетов.

2. Как упростить вычисление известной функции *f(x)* или же ее характеристик $(f'(x), \max f(x))$, если *f(x)* слишком сложная.

Ответы на эти вопросы даются теорией аппроксимации функций, ***основная задача*** которой состоит в нахождении функции *y=φ(x)*, близкой (т.е. аппроксимирующей) в некотором нормированном пространстве к исходной функции *y=f(x)*. Функцию *φ(x)* при этом выбирают такой, чтобы она была максимально удобной для последующих расчетов.

Основной подход к решению этой задачи заключается в том, что *φ(x)* выбирается зависящей от нескольких свободных параметров $\vec{c} = (c_1,\ c_2,\ ...,\ c_n)$, т.е. $y = \varphi(x) = \varphi(x,\ c_1,\ ...,\ c_n) = \varphi(x,\ \vec{c})$, значения которых подбираются из некоторого условия близости *f(x)* и *φ(x)*.

Обоснование способов нахождения удачного вида функциональной зависимости $\varphi(x,\ \vec{c})$ и подбора параметров $\vec{c}$ составляет задачу ***теории аппроксимации функций*.**

В зависимости от способа подбора параметров $\vec{c}$ получают различные ***методы аппроксимации***, среди которых наибольшее распространение получили ***интерполяция*** и ***среднеквадратичное приближение,*** частным случаем которого является ***метод наименьших квадратов.***

Наиболее простой, хорошо изученной и нашедшей широкое применение в настоящее время является ***линейная аппроксимация*,** при которой выбирают функцию $\varphi(x,\ \vec{c})$, линейно зависящую от параметров $\vec{c}$, т.е. в виде обобщенного многочлена:

$$\varphi(x,\ \vec{c}) = c_1\varphi_1(x) + ... + c_n\varphi_n(x) = \sum_{k=1}^{n} c_k\varphi_k(x). \qquad (20.1)$$

Здесь $\{\varphi_1(k), ..., \varphi_n(x)\}$ - известная система линейно независимых (базисных) функций. В качестве $\{\varphi_k(x)\}$ в принципе могут быть выбраны любые элементарные функции. Например: тригонометрические, экспоненты, логарифмические или комбинации таких функций. Важно, чтобы система базисных функций была *полной*, т.е. обеспечивающей аппроксимацию *f(x)* многочленом (20.1) с заданной точностью при $n \to \infty$.

Приведем хорошо известные и часто используемые системы. При интерполяции обычно используется система линейно независимых функций $\{\varphi_k(x) = x^{k-1}\}$. Для среднеквадратичной аппроксимации удобнее в качестве $\varphi_k(x)$ брать ортогональные на интервале *[-1, 1]* многочлены Лежандра:

$$\{\varphi_1(x) = 1;\ \varphi_2(x) = x;\ \varphi_{k+1}(x) = [(2k+1)x\varphi_k(x) - k\varphi_{k-1}(x)],\ \ k = 2,3,\ ...,\ n\};$$

$$\int_{-1}^{1} \varphi_k(x) \cdot \varphi_l(x)\,dx = 0;\ \ k \neq l\,.$$

Заметим, что если функция $f(x)$ задана на отрезке *[a, b],* то при использовании этой системы необходимо предварительно осуществить преобразование координат $x' = \left(x - \frac{b+a}{2}\right)\frac{2}{b-a}$, приводящее интервал $a \le x \le b$ к интервалу $-1 \le x' \le 1$.

Для аппроксимации периодических функций используют ортогональную на *[a, b]* систему тригонометрических функций $\left\{\varphi_k(x) = \cos\left(2k\pi\frac{x-a}{b-a}\right),\ \psi_k(x) = \sin\left(2k\pi\frac{x-a}{b-a}\right)\right\}$. В этом случае обобщенный многочлен (20.1) записывается в виде: $y = \sum_{k=1}^{n} c_k\varphi_k(x) + d_k\psi_k(x)$.

20.2. Что такое интерполяция

Интерполяция является одним из способов аппроксимации функций. Суть ее состоит в следующем. В области значений x, представляющей некоторый интервал *[a, b]*, где функции *f(x)* и *φ(x)* должны быть близки, выбирают упорядоченную систему точек (узлов) $x_1 < x_2 < ... < x_n$, (обозначим $\vec{x} = (x_1, ..., x_n)$), число которых равно количеству искомых параметров $c_1, c_2, ..., c_n$. Далее, параметры $\vec{c}$ подбирают такими, чтобы функция

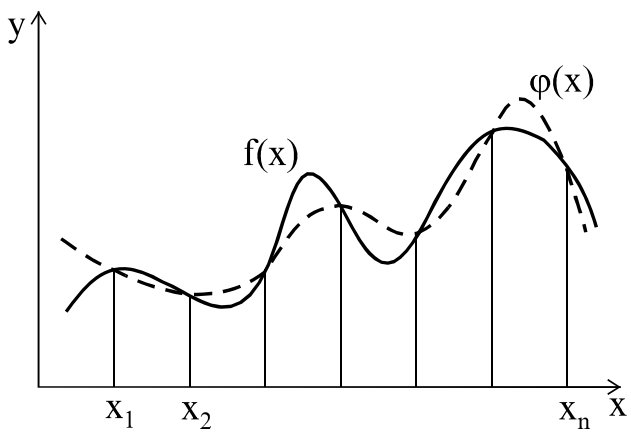


Рис. 20.1

$\varphi(x, \vec{c})$ совпадала с *f(x)* в этих узлах, $\varphi(x_i, \vec{c}) = f(x_i)$, $i = 1 \ldots n$ (см. рис. 20.1), для чего решают полученную систему *n* алгебраических, в общем случае нелинейных, уравнений.

В случае линейной аппроксимации (20.1) система для нахождения коэффициентов $\vec{c}$ линейна и имеет следующий вид:

$$\sum_{k=1}^{n} c_k \varphi_k(x_i) = f_i; \;\; i = 1, \; 2, \; ..., \; n; \;\; f_i = f(x_i), \tag{20.2}$$

Система базисных функций $\{\varphi_k(x)\}$, используемых для интерполяции, должна быть **чебышевской**, т.е. такой, чтобы определитель матрицы системы (20.2) был отличен от нуля и, следовательно, задача интерполяции имела единственное решение.

Для большинства практически важных приложений при интерполяции наиболее удобны обычные алгебраические многочлены, ибо они легко обрабатываются.

Интерполяционным многочленом называют алгебраический многочлен степени *n-1*, совпадающий с аппроксимируемой функцией в выбранных *n* точках.

Общий вид алгебраического многочлена

$$\varphi(x, \vec{c}) = P_{n-1}(x) = c_1 + c_2 x + c_3 x^2 + ... + c_n x^{n-1} = \sum_{k=1}^{n} a_k x^{k-1}. \tag{20.3}$$

Матрица системы (20.2) в этом случае имеет вид

$$G = \begin{bmatrix} 1 & x_1 & .. & x_1^{n-1} \\ 1 & x_2 & .. & x_2^{n-1} \\ .. & .. & .. & .. \\ 1 & x_n & .. & x_n^{n-1} \end{bmatrix}; \qquad |G| = \prod_{n \ge k > m \ge 0} (x_k - x_m) \tag{20.4}$$

и ее определитель (это определитель Вандермонда) отличен от нуля, если точки x_i разные. Поэтому задача (20.2) имеет единственное решение, т.е. для заданной системы различных точек существует единственный интерполяционный многочлен.

Погрешность аппроксимации функции *f(x)* интерполяционным многочленом степени *n-1*, построенным по *n* точкам, можно оценить, если известна ее производная порядка *n,* по формуле

$$\varepsilon = \| f(x) - P_{n-1}(x) \|_C \le \sqrt{\frac{2}{(n-1)\pi}} \left\| \frac{d^n f(x)}{dx^n} \right\|_C (h/2)^n, \; h = \max_i | x_i - x_{i-1} |. \tag{20.5}$$

Из (20.5) следует, что при $h \to 0$ порядок погрешности *p* при интерполяции алгебраическим многочленом равен количеству выбранных узлов $p=n$. Величина ε может быть сделана малой как за счет увеличения *n,* так и уменьшения *h.* В практических расчетах используют, как правило, многочлены не-

высокого порядка *(n ≤ 6)*, в связи с тем, что с ростом *n* резко возрастает погрешность вычисления самого многочлена из-за погрешностей округления.

20.3. Многочлены и способы интерполяции

Один и тот же многочлен можно записать по-разному, например, $1-2x+x^2=(x-1)^2$. Поэтому в зависимости от решаемых задач применяют различные виды представления интерполяционного многочлена и способы интерполяции.

Наряду с общим представлением (20.3) наиболее часто в приложениях используют интерполяционные многочлены в форме Лагранжа и Ньютона. Их особенность в том, что не надо находить параметры $\vec{c}$, так как многочлены в этой форме прямо записаны через значения таблицы $\{(x_i,\ y_i)\ \ i=1\ \ldots\ n\}$.

20.3.1. Интерполяционный многочлен Ньютона

$$N_{n-1}(x_T)=f_1+\sum_{k=1}^{n-1}(x_T-x_1)(x_T-x_2)\ldots(x_T-x_k)\Delta_k \qquad (20.6)$$

Здесь x_T - текущая точка, в которой надо вычислить значение многочлена, Δ_k - разделенные разности порядка *k,* которые вычисляются по следующим рекуррентным формулам:

$$\Delta_1 f(x_i x_{i+1})=\frac{f_i-f_{i+1}}{x_i-x_{i+1}};\quad \Delta_2 f(x_i x_{i+1} x_{i+2})=\frac{\Delta_i f(x_i x_{i+1})-\Delta_1 f(x_{i+1} x_{i+2})}{x_i-x_{i+2}};\quad \ldots$$

20.3.2. Линейная и квадратичная интерполяция

Иногда при интерполяции по заданной таблице из *m>3* точек применяют квадратичную *n=3* или линейную *n=2* интерполяцию. В этом случае для приближенного вычисления значения функции *f(x)* в текущей точке x_T находят в таблице ближайший к этой точке *i*-узел из общей таблицы, строят интерполяционный многочлен Ньютона первой или второй степени по формулам

$$N_1(x_T)=f_{i-1}+(x_T-x_{i-1})\frac{f_i-f_{i-1}}{x_i-x_{i-1}};\quad x_{i-1}\le x_T\le x_i; \qquad (20.7)$$

$$N_2(x_T)=N_1(x_T)+(x_T-x_{i-1})(x_T-x_i)\frac{\left(\frac{f_{i-1}-f_i}{x_{i-1}-x_i}\right)-\left(\frac{f_i-f_{i+1}}{x_i-x_{i+1}}\right)}{x_{i-1}-x_{i+1}};\quad x_{i-1}\le x_T\le x_{i+1},$$

и за значение *f(x)* принимают $N_1(x)$ (*линейная интерполяция*) или $N_2(x)$ (*квадратичная интерполяция*)

20.3.3. Интерполяционный многочлен Лагранжа

$$L_{n-1}(x_T) = \sum_{k=1}^{n} f_k \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x_T - x_i}{x_k - x_i} \quad . \tag{20.8}$$

Произведение $\prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x_T - x_i}{x_k - x_i}$ выбрано так, что во всех узлах, кроме *k-го*, оно обращаются в нуль, а в *k-м* узле оно равно единице:

$$\prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x_T - x_i}{x_k - x_i} = \begin{cases} 1, & при \quad x_T = x_k \\ 0, & при \quad x_T \neq x_k. \end{cases}$$

Поэтому из выражения (20.8) видно, что $L_{n-1}(x_i) = f_i$.

20.3.4. Интерполяция общего вида, использующая прямое решение системы (20.2) методом Гаусса

Следует отметить, что ввиду громоздкости многочлены Ньютона и Лагранжа уступают по эффективности расчета многочлену общего вида (20.3), если предварительно найдены коэффициенты $\vec{c}$.

Поэтому, когда требуется производить многократные вычисления многочлена, построенного по одной таблице, оказывается выгодно вначале один раз найти коэффициенты $\vec{c}$ (решив систему линейных алгебраических уравнений) и затем использовать формулу (20.3). Коэффициенты $\vec{c}$ находят прямым решением системы (20.2) с матрицей (20.4), затем вычисляют его значения по экономно программируемой формуле (алгоритм Горнера)

$$P_{n-1}(x) = c_1 + x(c_2 + \ldots \; x(c_{n-2} + x(c_{n-1} + xc_n) \; \ldots) \, . \tag{20.9}$$

20.4. Среднеквадратичная аппроксимация

Суть среднеквадратичной аппроксимации заключается в том, что параметры $\vec{c}$ функции $\varphi(x, \vec{c})$ подбираются такими, чтобы обеспечить минимум квадрата расстояния между функциями *f(x)* и $\varphi(x, \vec{c})$, т.е. из условия

$$\min_{c_1,\ldots,c_n} \left\| f(x) - \varphi(x, \vec{c}) \right\|_{L_2} . \tag{20.12}$$

В случае линейной аппроксимации (20.1) задача (20.12) сводится к решению СЛАУ для нахождения необходимых коэффициентов $\vec{c}$:

$$\sum_{k=1}^{n} (\varphi_i \varphi_k)_{L_2} \cdot c_k = (f, \varphi_i)_{L_2} ; \quad i = 1,\ldots,n. \tag{20.13}$$

Здесь $(\varphi_i \varphi_k)_{L_2}$, $(f, \varphi_i)_{L_2}$ - скалярные произведения в L_2.

Матрица системы (20.13) симметричная, и ее следует решать методом квадратного корня.

Особенно просто эта задача решается, если выбрана ***ортогональная система функций*** $\{\varphi_k(x)\}$, т.е. такая, что

$$(\varphi_i \varphi_k) = \begin{cases} 0, & i \neq k \\ \|\varphi_k\|^2, & i = k. \end{cases}$$

Тогда матрица СЛАУ (20.13) диагональная и параметры $\vec{c}$ находятся по формулам

$$c_k = \frac{(f,\ \varphi_k)}{\|\varphi_k\|^2}.$$ В этом случае представление (20.1) называется ***обобщенным рядом Фурье***, а c_k называются коэффициентами Фурье.

20.4.1. Метод наименьших квадратов

МНК является частным случаем среднеквадратичной аппроксимации. При использовании МНК аналогично задаче интерполяции в области значений *x*, представляющей некоторый интервал *[a, b],* где функции *f(x)* и *φ(x)* должны быть близки, выбирают систему различных точек (узлов) x_1 , ..., x_m, число которых обычно больше, чем количество искомых параметров c_1, ...,c_n , $m \geq n$. Далее, потребовав, чтобы сумма квадратов невязок во всех узлах была минимальна (см. рис. 20.2):

$$\min_{\vec{c}} \sum_{j=1}^{m} [f(x_j) - \varphi(x_j, \vec{c})]^2 = \min_{\vec{c}} \sum_{j=1}^{m} \delta_j^2 = \min_{\vec{c}} \delta(\vec{c}), \qquad (20.13)$$

находим из этого условия параметры c_1, ...,c_n .

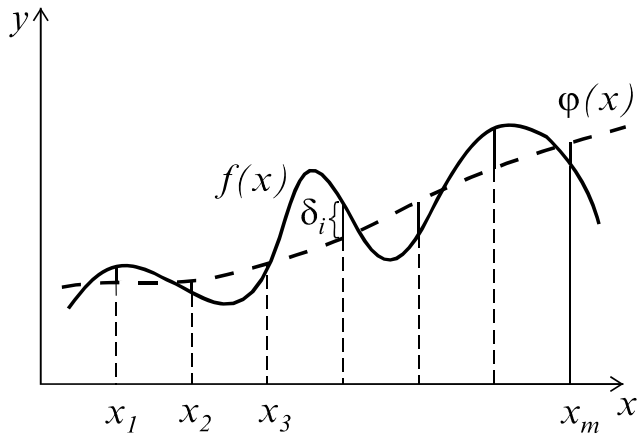


Рис. 20.2

В общем случае эта задача сложная и требует применения численных методов оптимизации. Однако в случае линейной аппроксимации (20.1), составляя условия минимума суммы квадратов невязок во всех точках $\delta(\vec{c})$ (в точке минимума все частные производные должны быть равны нулю):

$$\frac{\partial\delta(c_1, c_2, ..., c_n)}{\partial c_i} = 0, \quad i = 1, 2, ..., n, \qquad (20.14)$$

получаем систему n линейных уравнений относительно n неизвестных $c_1, ..., c_n$ следующего вида:

$$\sum_{k=1}^{n} (\vec{\varphi}_i\ \vec{\varphi}_k) c_k = (\vec{f},\ \vec{\varphi}_i),\ \ i = 1, ..., n \qquad \text{или } G\vec{c} = \vec{b}\,. \qquad (20.15)$$

Здесь $\vec{\varphi}_i = \left(\varphi_i(x_1),\ \varphi_i(x_2),\ ...\ ,\ \varphi_i(x_m)\right)$, $\vec{f} = \left(f_1,\ ...\ ,\ f_m\right)$ - векторы таблицы функций. Элементы матрицы G и вектора $\vec{b}$ в (20.15) определяются выражениями

$$\left.\begin{aligned} g_{ik} &= (\vec{\varphi}_i\ \ \vec{\varphi}_k) = \sum_{j=1}^{m} \varphi_i(x_j)\varphi_k(x_j), \\ b_i &= (\vec{f},\ \vec{\varphi}_i) = \sum_{j=1}^{m} f_j\, \varphi_i(x_j) \end{aligned}\right\} \text{ скалярные произведения векторов.}$$

Система (20.15) имеет симметричную матрицу G и решается методом квадратного корня.

При аппроксимации по МНК обычно применяют такие функции $\{\varphi_i\}$, которые используют особенности решаемой задачи и удобны для последующей обработки.

ЛЕКЦИЯ 21. ВЫЧИСЛЕНИЕ ПРОИЗВОДНЫХ И ИНТЕГРАЛОВ

21.1. Формулы численного дифференцирования

Формулы для расчета производной $d^m f / dx^m$ в точке x получаются следующим образом. Берется несколько близких к x узлов $x_1, x_2, ..., x_n$ ($n \geq m+1$), называемых ***шаблоном*** (точка x может быть одним из узлов). Вычисляются значения $f_i = f(x_i)$ в узлах шаблона, строится интерполяционный многочлен Ньютона и после взятия производной от этого многочлена $d^m P_{n-1} / dx^m$ получается выражение приближенного значения производной (формула численного дифференцирования) через значения функции в узлах шаблона: $d^m f / dx^m \approx \Lambda_m^n[f] = d^m P_{n-1} / dx^m$.

При $n=m+1$ формула численного дифференцирования не зависит от положения точки x внутри шаблона. Т.к. m-я производная от полинома m-й степени $P_m(x)$ есть константа, такие формулы называют ***простейшими формулами*** численного дифференцирования.

Анализ оценки погрешности вычисления производной

$$\varepsilon = \max_{x_1<x<x_n} \left| \frac{d^m f}{dx^m} - \Lambda_m^n[f] \right| \leq \frac{\max_x \left| f^{(m)}(x) \right|}{(n-m)} \max_i \left| x - x_i \right| \leq C h^{n-m}. \qquad (21.1)$$

$$h = \max \left| x_i - x_{i-1} \right|;\ C = const,\ \ n \geq m+1$$

показывает, что погрешность минимальна для значения x в центре шаблона и возрастает на краях. Поэтому узлы шаблона, если это возможно, выбираются симметрично относительно x. Заметим, что порядок погрешности при $h \to 0$ равен $n-m \geq 1$, и для повышения точности можно либо увеличивать n, либо уменьшать h (последнее более предпочтительно).

Приведем несколько важных формул для равномерного шаблона.

$$\frac{df}{dx} \approx \frac{dP_1}{dx} = \Lambda_1^2[f(x)] = \frac{f_2 - f_1}{h};\quad x_1 \leq x \leq x_2. \qquad (21.2)$$

Простейшая формула (21.2) имеет второй порядок погрешности в центре интервала и первый по краям.

$$\frac{df}{dx} \approx \frac{dP_2}{dx} = \Lambda_1^3[f(x)] = \frac{f_2 - f_1}{h} + (2x - x_1 - x_2)\frac{f_1 - 2f_2 + f_3}{2h^2} \qquad (21.3)$$

Эта формула имеет второй порядок погрешности во всем интервале $x_1 \leq x \leq x_3$ и часто используется для вычисления производной в крайних точках таблицы, где нет возможности выбрать симметричное расположение узлов. Заметим, что из (21.3) получается три важные формулы второго порядка точности

$$\frac{df(x_2)}{dx} = \Lambda_1^3[f(x_2)] = \frac{f_3 - f_1}{2h}; \qquad (21.4)$$

$$\frac{df(x_1)}{dx} = \Lambda_1^3[f(x_1)] = -\frac{3f_1 - 4f_2 + f_3}{2h}; \tag{21.5}$$

$$\frac{df(x_3)}{dx} = \Lambda_1^3[f(x_3)] = \frac{f_1 - 4f_2 + 3f_3}{2h}; \tag{21.6}$$

Для вычисления второй производной часто используют следующую простейшую формулу:

$$\frac{d^2 f}{dx^2} \approx \frac{d^2 P_2}{dx^2} = \Lambda_2[f(x)] = \frac{f_1 - 2f_2 + f_3}{h^2}; \qquad x_1 \le x \le x_3 \tag{21.7}$$

которая имеет второй порядок погрешности в центральной точке x_2.

21.2. Формулы численного интегрирования

Формулы для вычисления интеграла $U = \int_a^b f(x)dx$ получают следующим образом. Область интегрирования *[a, b]* разбивают на *m* малых отрезков с шагом $h = (b-a)/m$. Значение интеграла по всей области равно сумме интегралов на отрезках $\int_a^b f(x)dx = \sum_{i=1}^{m} \int_{x_{i-1}}^{x_i} f(x)\,dx$, где $x_i = a + ih.$ Выбирают на каждом отрезке $[x_{i-1}, x_i]$ 1-5 узлов и строят для каждого отрезка интерполяционный многочлен соответствующего порядка. Вычисляют интеграл от этого многочлена на отрезке. В результате получают выражение интеграла (формулу численного интегрирования) через значения подынтегральной функции в выбранной системе точек. Такие выражения называют ***квадратурными формулами.***

21.2.1. Формула средних

Формула средних получается, если на каждом i-ом отрезке $[x_{i-1}, x_i]$ взять один центральный узел $x_{i-1/2} = x_i - h/2,$ соответствующий середине отрезка. Функция на каждом отрезке аппроксимируется многочленом нулевой степени (константой) $P_0(x) = f(x_{i-1/2}).$ В этом случае получим:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_{i-1}}^{x_i} P_0(x)dx = h\sum_{i=1}^{m} f_{i-1/2} = \Sigma_{cp} f\,. \tag{21.8}$$

Погрешность формулы средних имеет второй порядок по *h:*

$$\varepsilon_{cp} = \max\left|\int_a^b f(x)dx - \Sigma_{cp} f\right| \le \left|\frac{h^2}{24}\int_a^b f''(x)dx\right|. \tag{21.9}$$

21.2.2. Формула трапеций

Формула трапеций получается при аппроксимации функции *f(x)* на каждом отрезке $[x_{i-1}, x_i]$ интерполяционным многочленом первого порядка, т.е. прямой, проходящей через точки (x_{i-1}, f_{i-1}), (x_i, f_i). Площадь криволинейной фигуры заменяется площадью трапеции с основаниями f_{i-1}, f_i и высотой *h*

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_{i-1}}^{x_i} P_1(x)dx = h\sum_{i=1}^{m} \frac{f_{i-1}+f_i}{2} = h\left[\frac{f_0+f_m}{2} + \sum_{i=1}^{m-1} f_i\right] = \Sigma_{m\,\text{р}} f \quad (21.10)$$

Погрешность формулы трапеций в два раза больше, чем погрешность формулы средних:

$$\varepsilon_{m\,\text{р}} = \max\left|\int_a^b f(x)dx - \Sigma_{m\,\text{р}} f\right| \le \left|-\frac{h^2}{12}\int_a^b f''(x)dx\right|. \quad (21.11)$$

21.2.3. Формула Симпсона

Формула Симпсона получается при аппроксимации функции *f(x)* на каждом отрезке $[x_{i-1}, x_i]$ интерполяционным многочленом второго порядка (параболой) с узлами x_{i-1}, $x_{i-1/2}$, x_i. После интегрирования параболы получаем

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_{i-1}}^{x_i} P_2(x)dx = \frac{h}{6}\sum_{i=1}^{m} (f_{i-1} + 4f_{i-1/2} + f_i) = \Sigma_{cu} f \quad (21.12)$$

После приведения подобных членов формула (21.12) приобретает удобный для программирования вид:

$$\sum_{cu} f = \frac{h}{3}\cdot\left[\frac{f_0 + f_m}{2} + 2\sum_{i=1}^{m} f_{i-1/2} + \sum_{i=1}^{m-1} f_i\right]$$

Погрешность формулы Симпсона имеет четвертый порядок по *h*:

$$\varepsilon_{cu} = \max\left|\int_a^b f(x)dx - \Sigma_{cu} f\right| \le \left|\frac{h^4}{2880}\int_a^b f^{(4)}(x)dx\right|. \quad (21.13)$$

21.2.4. Формулы Гаусса

При построении предыдущих формул в качестве узлов интерполяционного многочлена выбирались середины и (или) концы интервала разбиения. При этом оказывается, что увеличение количества узлов не всегда приводит к уменьшению погрешности (сравни формулы средних и трапеций), т.е. за счет удачного расположения узлов можно значительно увеличить точность. ***Суть методов Гаусса*** с *n* узлами состоит в таком расположении этих *n* узлов интерполяционного многочлена на отрезке $[x_{i-1}, x_i]$, при котором достигается минимум погрешности квадратурной формулы. Детальный анализ показывает, что узлами, удовлетворяющими такому усло-

вию, являются нули ортогонального многочлена Лежандра *n-й* степени. Так, для *n=1* один узел должен быть выбран в центре. Следовательно, метод средних является **методом Гаусса с одним узлом**.

Для *n=2* узлы на отрезке $[x_{i-1}, x_i]$ должны быть выбраны следующим образом:

$$x_i^{1,2} = x_{i-1/2} \pm \frac{h}{2} \cdot 0.5773502692$$

и соответствующая **формула Гаусса с двумя узлами** имеет вид

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{m}[f(x_i^1) + f(x_i^2)]. \qquad (21.15)$$

Порядок погрешности этой формулы при $h \to 0$ - четвертый, т.е. такой же, как у метода Симпсона, хотя используется только два узла!

Для n=3 узлы на отрезке $[x_{i-1}, x_i]$ выбираются следующим образом:

$$x_i^0 = x_{i-1/2}, \qquad x_i^{1,2} = x_i^0 \pm \frac{h}{2} \cdot 0.7745966692$$

и соответствующая **формула Гаусса с тремя узлами** имеет вид

$$\int_a^b f(x)dx \approx \frac{h}{18}\sum_{i=1}^{m}\left[5f(x_i^1) + 8f(x_i^0) + 5f(x_i^2)\right]. \qquad (21.16)$$

Порядок погрешности этой формулы при $h \to 0$ - шестой, т.е. значительно выше, чем у формулы Симпсона, практически при одинаковых затратах на вычисления. Следует отметить, что формулы Гаусса, особенно широко применяются для вычисления несобственных интегралов специального вида, когда подынтегральная функция имеет достаточно высокие производные.

ЛЕКЦИЯ 22. МЕТОДЫ РЕШЕНИЯ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

22.1. Как решаются нелинейные уравнения

Математической моделью многих физических процессов является функциональная зависимость *y=f(x)*. Поэтому задачи исследования различных свойств функции *f(x)* часто возникают в инженерных расчетах. Одной из таких задач является нахождение значений *x*, при которых функция *f(x)* обращается в нуль, т.е. решение уравнения *f(x)=0.* (22.1)

Точное решение удается получить в исключительных случаях, и обычно для нахождения корней уравнения применяются численные методы. Решение уравнения (22.1) при этом осуществляется в два этапа:

1. Приближенное определение местоположения, характер и выбор интересующего нас корня.

2. Уточнение выбранного корня с заданной точностью ε.

На рис. 22.1 представлены три наиболее часто встречающиеся ситуации:

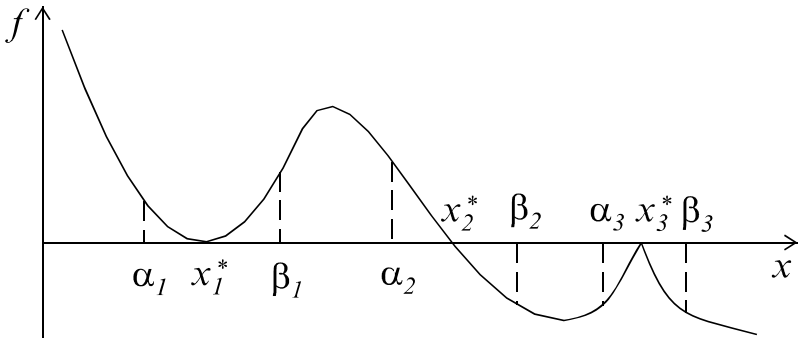


Рис.22.1

а) кратный корень: $f'(x_1^*) = 0, \quad f(\alpha_1) \cdot f(\beta_1) > 0;$

б) простой корень: $f'(x_2^*) \neq 0, \quad f(\alpha_2) \cdot f(\beta_2) < 0;$

в) вырожденный корень: $f'(x_3^*)$ не существует, $f(\alpha_3) \cdot f(\beta_3) > 0$.

Как видно из рис. 22.1, в случаях а) и в) значение корня совпадает с точкой экстремума функции и для нахождения таких корней рекомендуется использовать методы поиска минимума функции, описанные в лекции 23.

На втором этапе вычисление значения корня с заданной точностью осуществляется одним из итерационных методов. При этом, в соответствии с общей методологией *m-шагового итерационного метода*, на интервале $[\alpha, \beta]$, где находится интересующий нас корень x, выбирается m начальных значений $x_0, x_1, \ldots, x_{m-1}$ (обычно $x_0=\alpha$, $x_{m-1}=\beta$), после чего последовательно находятся члены $(x_m, x_{m+1}, \ldots, x_{n-1}, x_n)$ рекуррентной последовательности *порядка m* по правилу $x_k = \varphi(x_{k-1}, \ldots, x_{k-m})$ до тех пор, пока $|x_n - x_{n-1}| < \varepsilon$. Последнее значение x_n выбирается в качестве приближенного значения корня $(x^* \approx x_n)$, найденного с погрешностью ε .

Многообразие методов определяется возможностью большого выбора законов φ. Наиболее часто используемые на практике методы описаны ниже.

22.2. Итерационные методы уточнения корней

22.2.1. Метод простой итерации

Очень часто в практике вычислений встречается ситуация, когда уравнение (22.1) записано в виде разрешенном, относительно x:

$$x = \varphi(x). \qquad (22.2)$$

Заметим, что переход от записи уравнения (22.1) к эквивалентной записи (22.2) можно сделать многими способами, например, положив

$$\varphi(x) = x + \psi(x) f(x), \qquad (22.3)$$

где $\psi(x)$ - произвольная, непрерывная, знакопостоянная функция (часто достаточно выбрать $\psi=const$). В этом случае корни уравнения (22.2) являются также корнями (22.1), и наоборот. Исходя из записи (22.2) члены рекуррентной последовательности в методе простой итерации вычисляются по закону

$$x_k = \varphi(x_{k-1}), \quad k = 1, 2, \ldots . \qquad (22.4)$$

Метод является одношаговым, и для начала вычислений достаточно знать одно начальное приближение $x_0 = \alpha$ или $x_0 = \beta$ или $x_0 = (\alpha + \beta)/2$.

Условием сходимости метода простой итерации, если $\varphi(x)$ дифференцируема, является выполнение неравенства $|\varphi'(\xi)| < 1$, для любого

$$\xi \in [\alpha, \beta], \quad x^* \in [\alpha, \beta]. . \qquad (22.5)$$

Максимальный интервал $[\alpha, \beta]$, для которого выполняется неравенство (22.5), называется *областью сходимости*. При выполнении условия (22.5) метод сходится, если начальное приближение x_0 выбрано из области сходимости. При этом *скорость сходимости погрешности* $\varepsilon_k = |x^* - x_k|$ к нулю вблизи корня приблизительно такая же, как у геометрической прогрессии $\varepsilon_k \approx \varepsilon_{k-1} q$ со знаменателем $q \cong |\varphi'(x^*)|$, т.е. чем меньше q, тем быстрее сходимость, и наоборот. Поэтому при переходе от (22.1) к (22.2) функцию $\psi(x)$ в (22.3) выбирают так, чтобы выполнялось условие сходимости (22.5) для как

можно большей области *[α, β]* и с наименьшим q. Удачный выбор этих условий гарантирует эффективность расчетов.

22.2.2. Метод Ньютона

Этот метод является модификацией метода простой итерации и часто называется *методом касательных*. Если *f(x)* имеет непрерывную производную, тогда, выбрав в (22.3) $\psi(x) = 1/f'(x)$, получаем эквивалентное уравнение $x = x - f(x)/f'(x) = \varphi(x)$, в котором $q = \varphi'(x^*) \equiv 0$. Поэтому скорость сходимости рекуррентной последовательности метода Ньютона

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})} = \varphi(x_{k-1}) \qquad (22.6)$$

вблизи корня очень большая, погрешность очередного приближения примерно равна квадрату погрешности предыдущего $\varepsilon_k \cong |\varphi''(x^*)|\varepsilon_{k-1}^2$.

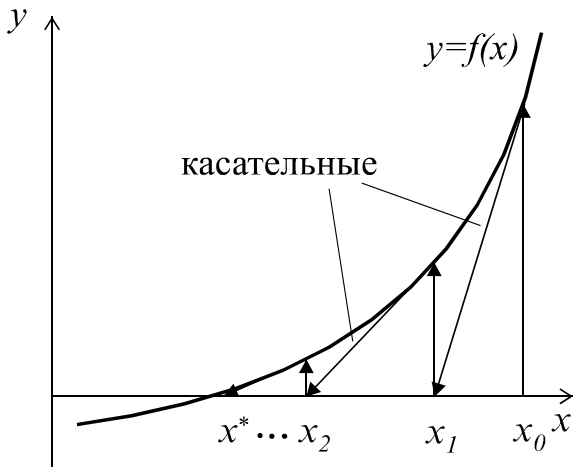


Рис. 22.2

Из (22.6) видно, что этот метод одношаговый *(m=1)* и для начала вычислений требуется задать одно начальное приближение x_0 из *области сходимости*, определяемой неравенством $|f(x) \cdot f''(x)| < [f'(x)]^2$. Метод Ньютона получил также второе название *метод касательных* благодаря геометрической иллюстрации его сходимости, представленной на рис. 22.2. Этот метод позволяет находить как простые, так и кратные корни. Основной его недостаток - малая область сходимости и необходимость вычисления производной *f'(x)*.

22.2.3. Метод секущих

Данный метод является модификацией метода Ньютона, позволяющей избавиться от явного вычисления производной путем ее замены приближенной формулой (22.2). Это эквивалентно тому, что вместо касательной на рис. 22.2 проводится секущая. Тогда вместо процесса (22.6) получаем

$$x_k = x_{k-1} - \frac{f(x_{k-1})\,h}{f(x_{k-1}) - f(x_{k-1} - h)} = \varphi(x_{k-1}). \qquad (22.7)$$

Здесь h - некоторый малый параметр метода, который подбирается из условия наиболее точного вычисления производной.

Метод одношаговый *(m=1),* и его условие сходимости при правильном выборе h такое же, как у метода Ньютона.

22.2.4. Метод Вегстейна

Этот метод является модификацией предыдущего метода секущих. В нем предлагается при расчете приближенного значения производной по разностной формуле использовать вместо точки $x_{k-1} - h$ в (22.7) точку x_{k-2}, полученную на предыдущей итерации. Расчетная формула метода Вегстейна:

$$x_k = x_{k-1} - \frac{f(x_{k-1})(x_{k-1} - x_{k-2})}{f(x_{k-1}) - f(x_{k-2})} = \varphi(x_{k-1}, x_{k-2}) \qquad (22.8)$$

Метод является двухшаговым (*m=2*), и для начала вычислений требуется задать два начальных приближения x_0, x_1. Лучше всего $x_0 = \alpha$, $x_1 = \beta$. Метод Вегстейна сходится медленнее метода секущих, однако, требует в два раза меньшего числа вычислений *f(x)* и за счет этого оказывается более эффективным.

Этот метод иногда называется улучшенным методом простой итерации и в применении к записи уравнения в форме (22.2) имеет вид

$$x_k = x_{k-1} - \frac{x_{k-1} - \varphi(x_{k-1})}{1 - \dfrac{\varphi(x_{k-1}) - \varphi(x_{k-2})}{x_{k-1} - x_{k-2}}} \qquad (22.9)$$

22.2.5. Метод парабол

Предыдущие три метода (Ньютона, секущих, Вегстейна) фактически основаны на том, что исходная функция *f(x)* аппроксимируется линейной зависимостью вблизи корня и в качестве следующего приближения выбирается точка пересечения аппроксимирующей прямой с осью абсцисс. Ясно, что аппроксимация будет лучше, если вместо линейной зависимости использовать квадратичную. На этом и основан один из самых эффективных методов - ***метод парабол***. Суть его в следующем: в окрестности корня задаются три начальные точки x_0, x_1, x_2, ($f(x_0) \cdot f(x_2) < 0$, $x_1 = (x_0 + x_2)/2$), в этих точках рассчитываются три значения функции $f(x)$: f_0, f_1, f_2 и строится интерполяционный многочлен второго порядка, который удобно записать в форме

$$P_2(x) = a(x - x_2)^2 + b(x - x_2) + c = az^2 + bz + c. \qquad (22.10)$$

Коэффициенты этого многочлена вычисляются по формулам:

$z = x - x_2$; $\quad z_0 = x_0 - x_2$; $\quad z_1 = x_1 - x_2$; $\quad c = f_2$;

$$a = \frac{(f_0 - f_2)/z_0 - (f_1 - f_2)/z_1}{z_0 - z_1}; \quad b = (f_0 - f_2)/z_0 - a z_0. \qquad (22.11)$$

Полином (22.10) имеет два корня:

$$z_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/(2a).,$$

из которых выбирается наименьший по модулю z_m и рассчитывается следующая точка $x_m = x_2 + z_m$. Если выполняется условие $|x_m - x_1| < \varepsilon$, то за значение корня принимается x_m, в противном случае одна из крайних точек заменяется на точку x_m: если $x_m < x_1$ *то* $x_0 = x_m$, $f_0 = f(x_m)$, иначе $x_2 = x_m$, $f_2 = f(x_m)$ и процесс повторяется.

22.2.6. Метод деления отрезка пополам

Все вышеописанные методы могут работать, если функция *f(x)* является непрерывной и дифференцируемой вблизи искомого корня. В противном случае они не гарантируют получение решения. Для разрывных функций, а также. если не требуется быстрая сходимость, для нахождения *простого корня* на интервале $[\alpha, \beta]$ применяют надежный метод деления отрезка пополам. Его алгоритм основан на построении рекуррентной последовательности по следующему закону: в качестве начального приближения выбираются границы интервала, на котором имеется один простой корень $x_0 = \alpha$, $x_1 = \beta$, далее находится середина интервала $x = (x_0 + x_1)/2$, после чего отбрасывается половинка интервала, не содержащая корня. Очередная точка x выбирается как середина нового, в два раза меньшего, интервала. В результате получается следующий алгоритм метода деления отрезка пополам:

1. Вычисляем $f_0 = f(x_0)$, $f_1 = f(x_1)$.
2. Вычисляем $x = (x_0 + x_1)/2$, $f_m = f(x)$.
3. Если $f_0 \cdot f_m > 0$, то $x_0 = x$, $f_0 = f_m$, иначе $x_1 = x$, $f_1 = f_m$.
4. Если $|x_1 - x_0| > \varepsilon$, то повторять с пункта 2.
5. Вычисляем корень $x^* = (x_0 + x_1)/2$.

За одно вычисление функции интервал уменьшается вдвое, то есть скорость сходимости невелика, однако метод устойчив к ошибкам округления и всегда сходится.

ЛЕКЦИЯ 23. МЕТОДЫ ОПТИМИЗАЦИИ

23.1. Постановка задач оптимизации, их классификация

Трудно назвать область деятельности, где не возникали бы задачи оптимизационного характера. Это, например, задачи определения наиболее эффективного режима работы различных систем, задачи организации производства, дающего наибольшую возможную прибыль при заданных ограниченных ресурсах или ограничении на количество товара, которое может поглотить рынок и т.д.

Постановка каждой задачи оптимизации включает в себя моделирование рассматриваемой ситуации с целью получения математической функции, которую необходимо минимизировать, а также определения ограничений, если таковые существуют и выбора подходящей процедуры для осуществления минимизации функции.

Задача оптимизации заключается в выборе среди элементов множества X *(множества допустимых решений*) такого решения, которое было бы с определенной точки зрения наиболее предпочтительным. В дальнейшем понятие решения отождествляется с вектором (точкой) *n*-мерного евклидова пространства R^n. В соответствии с этим допустимое множество X представляет собой некоторое подмножество пространства R^n, т. е. $X \subset R^n$, а *целевая функция* (а также *критерий качества* или *критерий оптимальности*) $f(\vec{x})$ – это функция n переменных $x_1, x_2, ..., x_n$. Сравнение решений по предпочтительности осуществляется с помощью целевой функции $f(\vec{x})$, которую формулируют таким образом, чтобы наиболее предпочтительному решению $\vec{x}^{(0)}$ соответствовал минимум целевой функции:

$$f(\vec{x}^{(0)}) = \min_{\vec{x} \in X} f(\vec{x}) \qquad (23.1)$$

При этом решение $\vec{x}^{(0)}$ называют *оптимальным* (точнее говоря, *минимальным*) , а значение $f(\vec{x}^{(0)})$ - *оптимумом (минимумом*).

Существует два вида минимумов *: локальный и глобальный*. Говорят, что точка $\vec{x}^{(0)} \in X$ доставляет функции $f(\vec{x})$ на множестве X локальный минимум, если существует такая окрестность $U_\varepsilon(\vec{x}^{(0)}) \quad (\varepsilon > 0)$ точки $\vec{x}^{(0)}$, что неравенство $f(\vec{x}^{(0)}) \le f(\vec{x})$ справедливо для всех $\vec{x} \in X \cap U_\varepsilon(\vec{x}^{(0)})$. Глобальный минимум функции $f(\vec{x})$ доставляет точка $\vec{x}^{(0)} \in X$, для которой записанное выше неравенство выполняется при всех $\vec{x} \in X$.

Если множество допустимых значений $X = R^n$, то говорят о *задаче минимизации без ограничений*. В этом случае нужно найти такую точку $\vec{x}^{(0)}$, чтобы неравенство $f(\vec{x}^{(0)}) \le f(\vec{x})$ выполнялось для всех точек пространства R^n без ограничения. Задачу минимизации без ограничений называют также *задачей безусловной минимизации*. При этом для характеристики точки минимума и самого минимума добавляют прилагательное «безусловный».

Если $X \neq R^n$, то имеет место *задача минимизации с ограничениями*. В этом случае также говорят о *задаче условной минимизации*, о *точках условного минимума* и об *условном минимуме.*

Если допустимое множество X задано в виде

$$X = \left\{ \vec{x} \in R^n \mid g_j(\vec{x}) \le 0,\ j = 1,2,...,k;\ g_j(\vec{x}) = 0,\ j = k+1,...,m \right\} \qquad (23.2)$$

где все функции $g_j(\vec{x})$ определены на R^n, то говорят о *задаче математического программирования.* Среди задач этого класса различают *задачи с ограничениями типа неравенств* – когда множество X имеет вид (23.2) и $m = k$; *задачи с ограничениями типа равенств* – когда в (23.2) неравенства отсутствуют ($k = 0$), и *задачи со смешанными ограничениями* – когда в задании множества X встречаются как равенства, так и неравенства. Следует отметить, что ограничение типа равенства $g(\vec{x}) = 0$ всегда можно заменить двумя ограничениями типа неравенства $g(\vec{x}) = 0 \rightarrow g(\vec{x}) \le 0, \quad g(\vec{x}) \ge 0.$ С другой стороны, ограничение-неравенство всегда можно заменить эквивалентным ему ограничением-равенством, введя дополнительную переменную x_{n+1} $g(\vec{x}) \le 0 \ \rightarrow \ g(\vec{x}) + x_{n+1}^2 = 0.$

23.2. Методы нахождения минимума функции одной переменной

Задача нахождения минимума функции одной переменной *min f(x)* нередко возникает в практических приложениях. Кроме того, многие методы решения задачи минимизации функции многих переменных сводятся к многократному поиску одномерного минимума. Поэтому разработка новых, более эффективных одномерных методов оптимизации продолжается и сейчас, несмотря на кажущуюся простоту задачи.

Примечание. *В дальнейшем, если не будет особо оговорено, под минимумом функции будет подразумеваться локальный минимум.*

Нахождение минимума функции осуществляется в два этапа:

1. Приближенное определение местоположения минимума.

2. Вычисление точки минимума x_{min} с заданной точностью ε одним из нижеприведенных методов.

На первом этапе, задав некоторую начальную точку x_0, спускаются с заданным шагом h в направлении уменьшения функции и устанавливают интервал длиной $2h$, на котором находится минимум, из условия

$f(x_m - h) > f(x_m) < f(x_m + h)$. Для функции, изображенной на рис. 23.1, если

$A < x_0 < x_g$, будет выделен интервал *[a,b]* с локальным минимумом x_{min1}, а если $x_g < x_0 < B$ – с глобальным минимумом x_{min2}, т. е. тот, в области «притяжения» которого оказалась начальная точка x_0.

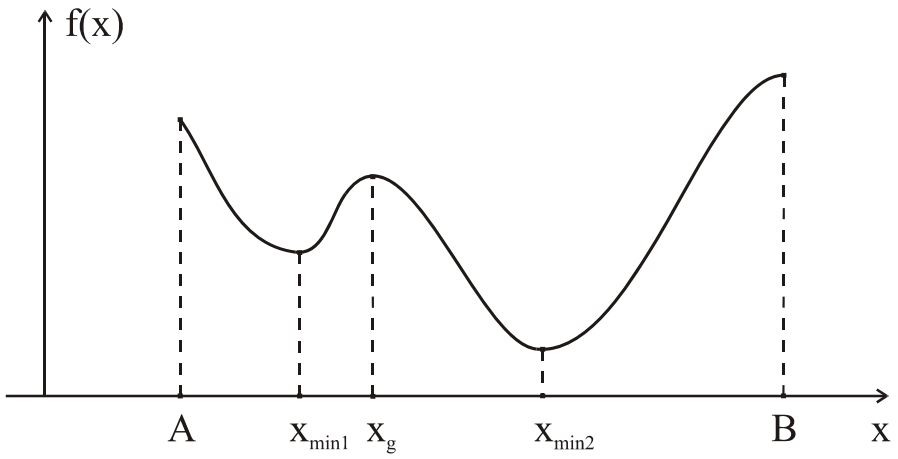


Рис. 23.1

Если на отрезке *[a,b]* функция *f(x)* унимодальна, т. е. она имеет на этом отрезке единственную точку минимума x_{min} и слева от этой точки является строго убывающей, а справа – строго возрастающей, то для вычисления точки минимума с заданной точностью могут использоваться нижеприведенные методы:

23.2.1. Метод деления отрезка пополам

Задаются интервал *[a, b]* и погрешность ε.

1. Вычисляется середина интервала *[a, b]:* $x = (a + b)/2$.
2. Отбрасывается половина интервала, не содержащая минимум:
   Если $f(x - \varepsilon) > f(x + \varepsilon)$, то $a = x$, иначе $b = x$.
3. Если $| b - a | > \varepsilon$, то повторяем с пункта 1.
4. Вычисляем $x_{\min} = (a + b)/2, \quad f_{\min} = f(x_{\min})$.

Этот метод прост в реализации, позволяет находить минимум разрывной функции, однако требует большого числа вычислений функции для обеспечения заданной точности.

23.2.2. Метод золотого сечения

Золотое сечение - это такое деление отрезка *[a, b]* на две неравные части при котором отношение большего отрезка ко всему интервалу равно отношению меньшего отрезка к большему. При этом имеет место следующее соотношение:

$$(b - x_1)/(b - a) = (x_1 - a)/(b - x_1) = 1 - \xi \cong 0.618, \quad \xi = (3 - \sqrt{5})/2 \cong 0.382.$$

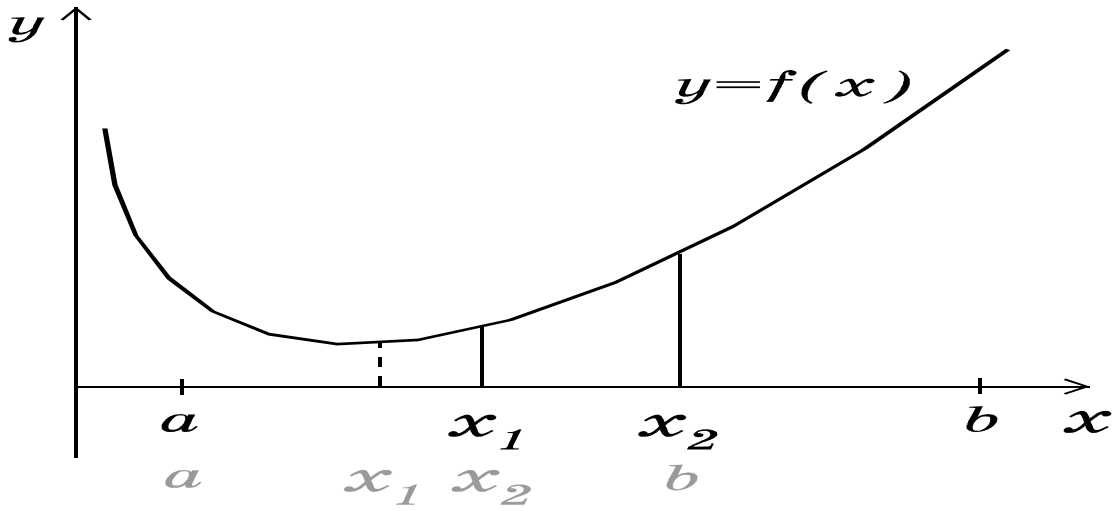


Рис. 23.2

О точке, которая расположена на расстоянии ξ длины от одного из концов отрезка, говорят, что она осуществляет *золотое сечение* данного отрезка. Каждый отрезок имеет две такие точки, расположенные симметрично относительно середины.

Алгоритм поиска минимума аналогичен вышеописанному методу деления пополам и отличается тем, что вначале точки x_1 и x_2 выбираются так, чтобы они осуществляли золотое сечение отрезка, и вычисляются значения функции в этих точках:

$$x_1 = a + \xi(b-a), \quad x_2 = b - \xi(b-a), \quad f_1 = f(x_1), \quad f_2 = f(x_2)$$

В последующем, после сокращения интервала путем отбрасывания неблагоприятной крайней точки, на оставшемся *отрезке уже имеется точка, делящая его в золотом отношении*, (точка x_1 на рис. 23.2) известно и значение функции в этой точке. Остается лишь выбрать ей симметричную и вычислить значение функции в этой точке для того, чтобы вновь решить, какую из крайних точек отбросить.

Алгоритм метода:

Задаются *a, b* и погрешность ε.

1. Вычисляются две точки золотого сечения
$x_1 = a + \xi(b-a), \quad x_2 = b - \xi(b-a), \quad f_1 = f(x_1), \quad f_2 = f(x_2)$.
2. Если $f_1 > f_2$, то $a = x_1, \quad x_1 = x_2, \quad f_1 = f_2, \quad x_2 = b - \xi(b-a), \quad f_2 = f(x_2)$,
иначе $b = x_2, \quad x_2 = x_1, \quad f_2 = f_1, \quad x_1 = a + \xi(b-a), \quad f_1 = f(x_1)$
3. Если $|b-a| > \varepsilon$, то повторить с пункта 2.
4. Вычисляется $x_{\min} = (a+b)/2, \quad f_{\min} = f(x_{\min})$.

При одинаковом количестве вычислений функции отрезок, на котором находится x_{min}, уменьшается быстрее, чем в методе деления пополам.

23.2.3. Метод Фибоначчи

На практике количество вычислений значений функции часто бывает ограничено некоторым числом *n* (тем самым ограничено и число шагов вычислений по методу золотого сечения; оно не превышает *n-1*). Метод Фибоначчи отличается от метода золотого сечения лишь выбором первых двух симметричных точек и формул их пересчета и гарантирует более точное приближение к точке x_{min} за *n -1* шаг, чем метод золотого сечения за то же количество шагов.

Согласно методу Фибоначчи, на нулевом шаге первые две симметричные точки вычисляют по формулам:

$$x_1^0 = a_0 + F_n (b_0 - a_0) / F_{n+2},$$
$$x_2^0 = b_0 - F_n (b_0 - a_0) / F_{n+2} = a_0 + F_{n+1} (b_0 - a_0) / F_{n+2},$$

где F_n, F_{n+1}, F_{n+2} – числа Фибоначчи , определяемые рекуррентной формулой

$$F_k = F_{k-1} + F_{k-2}, \quad k=3, 4, \ldots ; \qquad F_1 = F_2 = 1.$$

Запишем первые десять чисел Фибоначчи: $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$, $F_{10} = 55$.

В последующем, после сокращения интервала путем отбрасывания неблагоприятной крайней точки, одна из точек пересчитывается по одной из соответствующих формул

$$x_1^k = a_k + F_{n-k} (b_0 - a_0) / F_{n+2},$$
$$x_2^k = a_k + F_{n-k+1} (b_0 - a_0) / F_{n+2},$$

Выполняется *n - 1* шаг, при $k = 1, 2, \ldots, n-1$.

На последнем шаге две симметричные точки сливаются в одну, которая и принимается за точку минимума

$$x_{min} = x_1^{n-1}.$$

Погрешность вычисления точки минимума не превышает $(b_0 - a_0)/(2F_{n+2})$, т. е. за три вычисления функции *(n=2)* получают точку минимума с погрешностью, не превышающей 1/6 первоначального интервала, пять вычислений *(n=4)* – 1/16, девять *(n=8)*– 1/110.

Т.к. $\lim\limits_{n \to \infty} F_n / F_{n+2} = (3 - \sqrt{5})/2,$ то при достаточно больших n вычисления по методу Фибоначчи и золотого сечения начинаются практически из одной и той же пары симметричных точек.

Алгоритм метода:

Задаются *a, b* и число *n*.

1. Вычисляются $d = (b-a) / F_{n+2}$ и две симметричные точки

$$x_1 = a + F_n d, \quad x_2 = b - F_n d, \quad f_1 = f(x_1), \quad f_2 = f(x_2).$$

2. Если $f_1 > f_2$, то $a = x_1$, $x_1 = x_2$, $f_1 = f_2$, $x_2 = a + F_{n-k+1} d$, $f_2 = f (x_2)$,

иначе $b = x_2$, $x_2 = x_1$, $f_2 = f_1$, $x_1 = a + F_{n-k} d$, $f_1 = f (x_1)$.

пункт 2 повторяется *n-1* раз, при $k = 1, 2, ..., n-1$.

3. Вычисляется $x_{\min} = x_1,\ \ f_{\min} = f(x_{\min})$.

23.2.4. Метод последовательного перебора

Этот метод не требует предварительного определения местоположения точки минимума. Идея метода состоит в том, что, спускаясь из точки x_0 с заданным шагом h в направлении уменьшения функции, устанавливают интервал длиной $2h$, на котором находится минимум, который затем последовательно уточняют, повторяя спуск с последней точки, уменьшив шаг и изменив его знак, пока не будет достигнута заданная точность (некое подобие затухающего маятника). Алгоритм метода приведен ниже.

Задаются x_0, начальный шаг h, $(h>0)$ и погрешность ε.

1. Вычисляем $f_0 = f(x_0)$
2. Определяем направление убывания функции.
   Если $f(x_0 + \varepsilon) > f_0,$ то $h = -h.$
3. Из точки x_0 делается шаг $x_1 = x_0 + h$ и вычисляется $f_1 = f(x_1)$.
4. Если $f_1 < f_0$, то $x_0 = x_1,\ f_0 = f_1,$ и повторить с пункта 3.

5. В точке x_1 функция оказалась большей, чем в x_0, следовательно, мы перешагнули точку минимума. Организуем спуск в обратном направлении с меньшим шагом, например, $h = -h/4$ или $h = -h/10.$

6. Если $|h| > \varepsilon$, то повторить с пункта 3.

7. $x_{\min} = x_0,\ \ f_{\min} = f_0.$

Скорость сходимости данного метода существенно зависит от удачного выбора начального приближения x_0 и шага h. Шаг h следует выбирать как половину оценки расстояния от x_0 до предполагаемого минимума x_{min}.

23.2.5. Метод квадратичной параболы

Для нахождения точки минимума с заданной точностью задают 3 точки x_1, x_2, x_3 для которых выполняются условия $f(x_2)<f(x_1)$ *и* $f(x_2)<f(x_3)$. На этом интервале функцию аппроксимируют квадратичной параболой, минимум которой известен. Суть метода в следующем:

Для заданных трех точек x_1, x_2, x_3 вычисляются значения функции в них f_1, f_2, f_3. Через эти точки проводится квадратичная парабола

$$p(x - x_3)^2 + q(x - x_3) + r = pz^2 + qz + r,$$
$$z = x - x_3, \quad z_1 = x_1 - x_3, \quad z_2 = x_2 - x_3, \quad r = f_3, \qquad (23.3)$$
$$p = \frac{(f_1 - f_3)/z_1 - (f_2 - f_3)/z_2}{z_1 - z_2}, \qquad q = (f_1 - f_3)/z_1 - pz_1.$$

Парабола имеет минимум в точке $z_m = -q/(2p)$. Следовательно, можно аппроксимировать положение минимума функции значением $x_m = x_3 + z_m$ и, если точность не достигнута, следующий спуск производить, используя эту новую точку и две предыдущие, отбросив одну наихудшую точку. Получается последовательность x_{m1}, x_{m2}, x_{m3}, ... , сходящаяся к точке x_{min}.

Алгоритм метода можно записать следующим образом:

1. Задается точки x_1, x_2, x_3 и точность нахождения минимума ε.
2. Вычисляем $f_1 = f(x_1)$, $f_2 = f(x_2)$, $f_3 = f(x_3)$.
3. Вычисляем z_1, z_2, p, q, z_m по вышеприведенным формулам (23.3).
4. Вычисляем точку минимума параболы $x_m = x_3 + z_m$, $f_m = f(x_m)$.
5. Если $|x_m - x_2| < \varepsilon$, то $x_{\min} = x_m$ иначе переименовываем точки, отбрасывая наихудшую точку: если $x_m < x_2$ то $x_3 = x_2$, $f_3 = f_2$, $x_2 = x_m$, $f_2 = f_m$,

Иначе $x_1 = x_2$, $f_1 = f_2$, $x_2 = x_m$, $f_2 = f_m$.

и повторяем с пункта 3.

Данный метод сходится очень быстро и является одним из наилучших методов спуска. Следует однако отметить, что при очень малых ε расчет по приведенным здесь формулам для p и q вблизи минимума может привести к появлению большой погрешности из-за потери значащих цифр при вычитании близких чисел. Поэтому разные авторы предлагают свои эквивалентные формулы, счет по которым более устойчив. Кроме того, в алгоритм вносятся некоторые поправки, позволяющие предусмотреть различные неприятные ситуации - переполнение, деление на нуль, уход от корня.

23.2.6. Метод кубической параболы

Данный метод аналогичен предыдущему, но за счет использования аппроксимации кубической параболой имеет более высокую сходимость, *если функция допускает простое вычисление производной*. При его использовании выбираются две точки x_1 и x_2 ($f'(x_1) < 0$, $f'(x_2) > 0$), вычисляются значения функции f_1, f_2 и ее производной $D_1 = f'(x_1)$, $D_2 = f'(x_2)$. Затем через эти точки проводится кубическая парабола, коэффициенты которой определяются таким образом, чтобы совпадали значения функции и производных в точках x_1 и x_2 :

$$p(x-x_2)^3 + q(x-x_2)^2 + r(x-x_2) + s = pz^3 + qz^2 + rz + s = P(z),$$

$$z = x - x_2, \quad z_1 = x_1 - x_2,$$

$$P(z_1) = f_1, \quad P'(z_1) = D_1, \quad P(0) = f_2, \quad P'(0) = D_2.$$

Как нетрудно убедиться, коэффициенты параболы вычисляются по следующим формулам:

$$s = f_2, \quad r = D_2,$$

$$p = (D_1 + D_2 - 2(f_1 - f_2)/z_1)/z_1^2,$$
$$q = ((D_1 - D_2)/z_1 - 3pz_1)/2.$$

Известно, что кубическая парабола имеет минимум в точке

$$z_m = (-q + \sqrt{q^2 - 3pr})/(3p).$$

Поэтому приближенное положение минимума можно получить по формуле $x_m = x_2 + z_m$ и, если точность не достигнута, заменить одну из крайних точек точкой x_m и снова повторить процесс.

Алгоритм метода можно записать следующим образом:

1. Задается точки x_1, x_2, ($f'(x_1) < 0$, $f'(x_2) > 0$) и точность ε.
2. Вычисляем $f_1 = f(x_1)$, $f_2 = f(x_2)$, $D_1 = f'(x_1)$, $D_2 = f'(x_2)$, $x_p = (x_1 + x_2)/2$.
3. Вычисляем z_1, p, q, r, z_m, x_m по вышеприведенным формулам.
4. Если $|x_m - x_p| < \varepsilon$, то $x_{min} = x_m$.

В противном случае заменяем одну из крайних точек точкой x_m:

если $f'(x_m) < 0$, то $x_1 = x_m$, $f_1 = f(x_m)$, $D_1 = f'(x_m)$,

иначе $x_2 = x_m$, $f_2 = f(x_m)$, $D_2 = f'(x_m)$.

запоминаем полученную точку $x_p = x_m$ и снова повторяем с пункта 3.

ЛЕКЦИЯ 24. РЕШЕНИЕ ЗАДАЧИ КОШИ ДЛЯ ОБЫКНОВЕННЫХ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ

24.1. Задачи для обыкновенных дифференциальных уравнений

Обыкновенными дифференциальными уравнениями можно описать поведение системы взаимодействующих частиц во внешних полях, процессы в электрических цепях, закономерности химической кинетики и многие другие явления. Поэтому решение обыкновенных дифференциальных уравнений занимает одно из важнейших мест среди прикладных задач физики, электроники, экономики, химии и техники.

Конкретная прикладная задача может приводить к дифференциальному уравнению любого порядка или к системе таких уравнений. Известно, что произвольную систему дифференциальных уравнений любого порядка можно привести к некоторой эквивалентной системе уравнений первого порядка. Среди таких систем выделим класс систем, разрешенных относительно производной неизвестных функций

$$\begin{cases} \dfrac{du_1(x)}{dx} = f_1(x,\ u_1,\ ...,\ u_m) \\ \quad . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \\ \dfrac{du_m(x)}{dx} = f_m(x,\ u_1,\ ...,\ u_m) \end{cases} \qquad (24.1)$$

Обычно требуется найти решение системы $\vec{u}(x) = \{u_1(x),\ ...,\ u_m(x)\}$ для значений x из заданного интервала $a \le x \le b$.

Известно, что система (24.1) имеет бесконечное множество решений, семейство которых в общем случае зависит от m произвольных параметров $\vec{c} = \{c_1,\ ...,\ c_m\}$ и может быть записано в виде $\vec{u} = \vec{u}(x,\ \vec{c})$. Для определения значений этих параметров, т.е. для выделения одного нужного решения, надо наложить дополнительно m условий на функции $\vec{u} = \{u_1,\ ...,\ u_m\}$. В зависимости от способа постановки дополнительных условий можно выделить два основных типа задач, наиболее часто встречающихся на практике.

краевая (граничная) задача, когда часть условий задается на границе a (при $x=a$), остальные условия - на границе b (при $x=b$). Обычно это значения искомых функций на границах;

задача Коши (задача с начальными условиями), когда все условия заданы в начале отрезка в виде

$$u_1(a) = u_1^0;\ ...;\ u_m(a) = u_m^0. \qquad (24.2)$$

При изложении методов решения задачи Коши воспользуемся компактной записью задачи (24.1), (24.2) в векторной форме.

$$\frac{d\vec{u}}{dx} = \vec{f}(x,\vec{u}); \quad \vec{u}(a) = \vec{u}^0. \tag{24.3}$$

Требуется найти $\vec{u}(x)$ для $a \le x \le b$.

24.2. Основные положения метода сеток для решения задачи Коши

Чаще всего задача (24.3) решается методом сеток.

Суть метода сеток состоит в следующем:

В области интегрирования выбирается упорядоченная система точек $a = x_0 < x_1 < x_2 < ... < x_n = b,$ называемая *сеткой*. Точки x_i называют *узлами*, а $h_k = x_k - x_{k-1}$ - *шагом сетки*. Если $h_k = h = (b-a)/n$, сетка называется *равномерной*. Для *неравномерной* сетки обозначим $h = \max\limits_k h_k$. Для упрощения, в дальнейшем будим считать сетку равномерной. Решение $\vec{u}(x)$ ищется в виде таблицы значений в узлах выбранной сетки $\vec{u}^k = \vec{u}(x_k)$, для чего дифференциальное уравнение заменяется системой алгебраических уравнений, связывающих между собой значения искомой функции в соседних узлах. Такая система называется *конечно-разностной схемой*.

Имеется несколько распространенных способов получения конечно-разностных схем. Приведем здесь один из самых универсальных - *интегро-интерполяционный метод*.

Согласно этому способу для получения конечно-разностной схемы проинтегрируем уравнение (24.3) на каждом интервале $\left[x_{k-1},\, x_k\right]$ для *k=1,2,...,n*

$$\int\limits_{x_{k-1}}^{x_k} \frac{d\vec{u}}{dx} dx = \vec{u}^k - \vec{u}^{k-1} = \int\limits_{x_{k-1}}^{x_k} \vec{f}(x,\, \vec{u}(x))dx.$$

Переписав последнюю формулу в виде

$$\vec{u}^k = \vec{u}^{k-1} + \int\limits_{x_{k-1}}^{x_k} \vec{f}(x,\, \vec{u}(x))dx \tag{24.4}$$

получаем, что значение искомой функции в *к-ом* узле определяется через значение в предшествующем узле с поправкой, выраженной в форме интеграла. Аппроксимируя интеграл одной из квадратурных формул, получаем те или иные формулы относительно *приближенных* неизвестных значений искомой функции, которые в отличие от точных обозначим $\vec{y}^k \approx \vec{u}^k$.

Структура конечно-разностной схемы для задачи Коши такова, что она устанавливает закон рекуррентной последовательности $\vec{y}^{\,k} = \varphi\left(\vec{y}^{\,k-1}\right)$ для искомого решения $\vec{y}^0,\ \vec{y}^1,\ \vec{y}^2,\ ...,\ \vec{y}^n$. Поэтому, используя начальное условие задачи (24.2) и задавая $\vec{y}^0 = \vec{u}^0$, затем по рекуррентным формулам последовательно находят все $y^k,\ k = 1,\ ...,\ n.$

При замене интеграла приближенной квадратурной формулой вносится *погрешность аппроксимации* дифференциального уравнения разностным.

Говорят, что разностная схема *аппроксимирует* исходную дифференциальную задачу с порядком *p*, если при $h \to 0$ погрешность аппроксимации $\psi(h) \le Ch^p,\ C - const.$ Чем больший *порядок аппроксимации p*, тем выше *точность решения*:

$$\varepsilon(h) = \left\| \vec{y} - \vec{u} \right\| = \max_k \left| \vec{y}^k - \vec{u}^k \right|. \qquad (24.5)$$

Основная теорема теории метода сеток утверждает, что если схема устойчива, то при $h \to 0$ погрешность решения $\varepsilon(h)$ стремится к нулю с тем же порядком, что и погрешность аппроксимации:

$\varepsilon(h) \le C_0 \psi(h) \le C_0 \cdot C \cdot h^p$; здесь C_0 - константа устойчивости.

Неустойчивость обычно проявляется в том, что с уменьшением *h* решение $\vec{y}^k \to \infty$ при возрастании *k*, что легко устанавливается экспериментально с помощью просчета на последовательности сеток с уменьшающимся шагом *h, h/2, h/4…* Если при этом $\vec{y}^k \to \infty$, то метод неустойчив. Таким образом, если имеется аппроксимация и схема устойчива, то, выбрав достаточно малый шаг *h*, можно получить решение с заданной точностью при этом затраты на вычисления резко уменьшаются с увеличением порядка аппроксимации *p*, т.е. при большем *p* можно достичь той же точности, используя более крупный шаг *h*. Большое разнообразие методов обусловлено возможностью поразному выбирать узлы и квадратурные формулы для аппроксимации интеграла в (24.4).

24.2.1. Явная схема 1-го порядка (метод Эйлера)

Вычисляя интеграл в (24.4) по формуле левых прямоугольников получим

$$\vec{y}^k = \vec{y}^{k-1} + h \cdot f(x_{k-1}, \vec{y}^{k-1}),\quad k = 1,\ 2,\ ...,\ n. \qquad (24.6)$$

Задавая $\vec{y}^0 = \vec{u}^0$, с помощью (24.6) легко получить все последующие значения $\vec{y}^k$, $k = 1,\ 2,\ ...,\ n$, так как формула явно разрешается относительно $\vec{y}^k$. Погрешность аппроксимации $\psi(h)$ и соответственно точность $\varepsilon(h)$ имеют первый порядок в силу того, что формула левых прямоугольников на интервале $\left[x_{k-1},\ x_k\right]$ имеет погрешность первого порядка, а схема устойчива.

24.2.2. Неявная схема 1-го порядка

Вычисляя интеграл в (24.4) по формуле правых прямоугольников получим

$$\vec{y}^k = \vec{y}^{k-1} + h \cdot \vec{f}(x_k, \vec{y}^k),\quad k = 1,\ 2,\ ...,\ n. \qquad (24.7)$$

Эта схема явно не разрешена относительно $\vec{y}^k$, поэтому для получения $\vec{y}^k$ требуется использовать итерационную процедуру решения уравнения (24.7):

$$\vec{y}^{k,s} = \vec{y}^{k-1} + h \cdot \vec{f}(x_k,\ \vec{y}^{k,s-1});\quad s = 1,\ 2,\ \ldots \text{- номер итерации.}$$

За начальное приближение можно взять значение $\vec{y}^{k,0} = \vec{y}^{k-1}$ из предыдущего узла. Обычно, если *h* выбрано удачно, достаточно сделать 2-3 итерации для достижения заданной погрешности $\left\| y^{k,s} - y^{k,s-1} \right\| < \varepsilon$. Эффективность неявной схемы заключается в том, что у нее константа устойчивости C_0 значительно меньше, чем у явной схемы.

24.2.3. Неявная схема 2-го порядка

Вычисляя интеграл в (24.4) по формуле трапеций получим

$$\vec{y}^k = \vec{y}^{k-1} + \frac{h}{2} \cdot \left[\vec{f}(x_{k-1},\ \vec{y}^{k-1}) + \vec{f}(x_k,\ \vec{y}^k) \right]. \qquad (24.8)$$

Так как формула трапеций имеет второй порядок точности, то и погрешность метода имеет второй порядок.

Схема (24.8) явно не разрешена относительно $\vec{y}^k$, поэтому требуется итерационная процедура:

$$\vec{y}^{k,s} = \vec{y}^{k-1} + \frac{h}{2} \cdot \left[\vec{f}(x_{k-1},\ \vec{y}^{k-1}) + \vec{f}(x_k, \vec{y}^{k,s-1}) \right],\quad s = 1,\ 2,\ \ldots,\ \vec{y}^{k,0} = \vec{y}^{k-1}.$$

24.2.4. Схема предиктор-корректор (Рунге-Кутта) 2-го порядка

Вычисляя интеграл в (24.4) по формуле средних прямоугольников получим

$$\vec{y}^k = \vec{y}^{k-1} + h \cdot \vec{f}(x_{k-1/2},\ \vec{y}^{k-1/2}). \qquad (24.9)$$

Уравнение разрешено явно относительно $\vec{y}^k$, однако в правой части присутствует неизвестное значение $\vec{y}^{k-1/2}$ в середине отрезка $[x_{k-1},\ x_k]$. Для решения этого уравнения используют следующий способ. Вначале по явной схеме (24.6) рассчитывают $\vec{y}^{k-1/2}$ (предиктор):

$$\vec{y}^{k-1/2} = \vec{y}^{k-1} + \frac{h}{2} \cdot \vec{f}(x_{k-1},\ \vec{y}^{k-1}).$$

После этого рассчитывают $\vec{y}^k$ по (24.9) (корректор). В результате схема оказывается явной и имеет второй порядок. Заметим, что схема получается из схемы М3, если в ней выполнять только две итерации *(s=1, 2)*.

24.2.5. Схема Рунге-Кутта 4-го порядка

Вычисляя интеграл в (24.4) по формуле Симпсона получим

$$\vec{y}^{k} = \vec{y}^{k-1} + \frac{h}{6} \cdot [\vec{f}(x_{k-1},\ \vec{y}^{k-1}) + 4\vec{f}(x_{k-1/2},\ \vec{y}^{k-1/2}) + \vec{f}(x_k,\ \vec{y}^{k})]. \quad (24.10)$$

Ввиду того, что формула Симпсона имеет четвертый порядок, погрешность метода тоже имеет четвертый порядок.

Можно по-разному реализовать расчет неявного по $\vec{y}^{k}$ уравнения (24.10), однако наибольшее распространение получил следующий способ. Вычисляют предиктор

$$\vec{y}^{k-1/2,1} = \vec{y}^{k-1} + \frac{h}{2}(\vec{f}(x_{k-1},\ \vec{y}^{k-1}),$$

$$\vec{y}^{k-1/2,2} = \vec{y}^{k-1} + \frac{h}{2}\vec{f}(x_{k-1/2},\ \vec{y}^{k-1/2,1}),$$

$$\vec{y}^{k,1} = \vec{y}^{k-1} + h\vec{f}(x_{k-1/2},\ \vec{y}^{k-1/2,2}),$$

затем корректор по формуле

$$\vec{y}^{k} = \vec{y}^{k-1} + \frac{h}{6}[f(x_{k-1},\ \vec{y}^{k-1}) + 2\vec{f}(x_{k-1/2,}\ \vec{y}^{k-1/2,1}) +$$

$$+2\vec{f}(x_{k-1/2,}\ \vec{y}^{k-1/2,2}) + \vec{f}(x_k, \vec{y}^{k,1})].$$

24.3. Многошаговые схемы Адамса

При построении всех предыдущих схем для вычисления интеграла в правой части (24.4) использовались лишь точки в диапазоне одного шага $[x_{k-1},\ x_k]$. Поэтому при реализации таких схем для вычисления следующего значения $\vec{y}^{k}$ требуется знать только одно предыдущее значение $\vec{y}^{k-1}$. Такие схемы называют *одношаговыми*. Мы, однако, видели, что для повышения точности при переходе от узла x_{k-1} к узлу x_k приходилось использовать и значения функции $\vec{f}(x_{k-1/2,}\ \vec{y}^{k-1/2})$ внутри интервала $[x_{k-1},\ x_k]$. Схемы, в которых это используется (M4,M5,...), называют *схемами с дробными шагами*. В этих схемах повышение точности достигается за счет дополнительных затрат на вычисление функции $\vec{f}(x, \vec{y}\ )$ в промежуточных точках интервала $[x_{k-1},\ x_k]$.

Идея методов Адамса заключается в том, чтобы для повышения точности использовать уже вычисленные на предыдущих шагах значения $\vec{y}^{k-1}$, $\vec{y}^{k-2}$, $\vec{y}^{k-3}$, ... в ***нескольких*** предыдущих узлах.

Заменим в (24.4) подынтегральную функцию интерполяционным многочленом Ньютона вида

$$f(x) \approx f(x_{k-1}) + (x - x_{k-1})\frac{f(x_{k-1}) - f(x_{k-2})}{h} +$$
$$+(x - x_{k-1})(x - x_{k-2})\frac{f(x_{k-1}) - 2f(x_{k-2}) + f(x_{k-3})}{2h^2} + \ \ldots \ .$$

После интегрирования на интервале $[x_{k-1},\ x_k]$ получим *явную экстраполяционную схему* Адамса. (*Экстраполяцией* называется получение значений интерполяционного многочлена в точках, выходящих за крайние узлы сетки). В нашем случае интегрирование производится на интервале $[x_{k-1},\ x_k]$, а полином строится по узлам $x_{k-1},\ x_{k-2},\ x_{k-3}$.

Порядок аппроксимации схемы в этом случае определяется количеством использованных при построении полинома узлов (например, если используются $x_{k-1},\ x_k$, то схема второго порядка).

Если в (24.4)) подынтегральную функцию заменим многочленом Ньютона вида

$$f(x) \approx f(x_k) + (x - x_k)\frac{f(x_k) - f(x_{k-1})}{h} +$$
$$+(x - x_k)(x - x_{k-1})\frac{f(x_k) - 2f(x_{k-1}) + f(x_{k-2})}{2h^2} + \ldots ,$$

то после интегрирования получим *неявную интерполяционную схему Адамса*. Заметим, что неявная интерполяционная схема второго порядка совпадает со схемой (24.10).

24.3.1. Явная экстраполяционная схема Адамса 2-го порядка

$$\vec{y}^k = \vec{y}^{k-1} + \frac{h}{2}\cdot[3\vec{f}(x_{k-1},\ \vec{y}^{k-1}) - \vec{f}(x_{k-2},\ \vec{y}^{k-2})]. \qquad (24.11)$$

Схема двухшаговая, поэтому для начала расчетов необходимо найти $\vec{y}^1$ по методу (24.9), после чего $\vec{y}^2,\ \vec{y}^3$, ... вычислять по (24.11).

24.3.2. Явная экстраполяционная схема Адамса 3-го порядка

$$\vec{y}^k = \vec{y}^{k-1} + \frac{h}{12}\cdot[23\vec{f}(x_{k-1},\ \vec{y}^{k-1}) - 16\vec{f}(x_{k-2},\ \vec{y}^{k-2}) + 5\vec{f}(x_{k-3}, \vec{y}^{k-3}) \qquad (24.12)$$

Схема трехшаговая, поэтому для начала расчетов необходимо найти $\vec{y}^1,\ \vec{y}^2$ по методу (24.10), после чего $\vec{y}^3,\ \vec{y}^4$, ... вычислить по (24.12).

24.3.3. Неявная схема Адамса 3-го порядка

$$\vec{y}^k = \vec{y}^{k-1} + \frac{h}{12}\cdot[5\vec{f}(x_k,\ \vec{y}^k) + 8\vec{f}(x_{k-1},\ \vec{y}^{k-1}) - \vec{f}(x_{k-2},\ \vec{y}^{k-2})]. \qquad (24.13)$$

Так как схема двухшаговая, то для начала расчетов необходимо найти $\vec{y}^1$ по методу (24.10), после чего $\vec{y}^2$, $\vec{y}^3$, ... вычисляются по (24.13). Эта формула явно не разрешена относительно $\vec{y}^k$, поэтому для получения $\vec{y}^k$ требуется использовать итерационную процедуру решения уравнения (24.13)

$$\vec{y}^{k,s} = \vec{y}^{k-1} + \frac{h}{12}\cdot\left[5\vec{f}(x_k,\ \vec{y}^{k,s-1}) + 8\vec{f}(x_{k-1},\ \vec{y}^{k-1}) - \vec{f}(x_{k-2},\ \vec{y}^{k-2})\right]$$

Значение $\vec{y}^{k,0}$ следует расчитать по формуле (24.11):

$$\vec{y}^{k,0} = \vec{y}^{k-1} + \frac{h}{2}\cdot\left[3\vec{f}(x_{k-1},\ \vec{y}^{k-1}) - \vec{f}(x_{k-2},\ \vec{y}^{k-2})\right]$$

ПРИЛОЖЕНИЕ 1. ПРОЦЕДУРЫ И ФУНКЦИИ ДЛЯ ПРЕОБРАЗОВАНИЯ СТРОКОВОГО ПРЕДСТАВЛЕНИЯ ЧИСЕЛ

Для работы со строками применяются следующие процедуры и функции (в квадратных скобках указываются необязательные параметры).

ПОДПРОГРАММЫ ПРЕОБРАЗОВАНИЯ СТРОК В ДРУГИЕ ТИПЫ	
Function StrToFloat(St: String): Extended;	Преобразует символы строки St в вещественное число. Строка не должна содержать ведущих или ведомых пробелов
Function StrToInt(St: String): Integer;	Преобразует символы строки St в целое число. Строка не должна содержать ведущих или ведомых пробелов
Procedure Val(St: String; var X; Code: Integer);	Преобразует строку символов St во внутреннее представление целой или вещественной переменной X, которое определяется типом этой переменной. Параметр Code содержит ноль, если преобразование прошло успешно
Подпрограммы обратного преобразования	
Function FloatToStr(Value: Extended): String;	Преобразует вещественное значение Value в строку символов
Function FloatToStrF(Value: Extended; Format: TFloat-For-mat; Precision, Digits: Integer) : String;	Преобразует вещественное значение Value в строку символов с учетом параметров Precision и Digits (см. пояснения ниже)
Procedure Str(X [:width [:Decimals]]; var St: String);	Преобразует число X любого вещественного или целого типа в строку символов St; параметры Width и Decimals, если они присутствуют, задают формат преобразования
Правила использования параметров функции FloatToStrF	
Значение Format	Описание
fFfExponent	Научная форма представления с множителем eXX. Precision задает общее количество десятичных цифр мантиссы. Digits - количество цифр в десятичном порядке XX.
ffFixed	Формат с фиксированным положением разделителя целой и дробной частей. Precision задает общее количество десятичных цифр в представлении числа. Digits - количество цифр в дробной части. Число округляется с учетом первой отбрасываемой цифры: 3,14
ffGeneral	Универсальный формат, использующий наиболее удобную для чтения форму представления вещественного числа. Соответствует формату ffFixed, если количество цифр в целой части меньше или равно Precision, а само число - больше или равно 0,00001, в противном случае соответствует формату ffExponent: 3,1416

ffNumber	Отличается от ffFixed использованием символа - разделителя тысяч при выводе больших чисел (для русифицированной версии Windows таким разделителем является пробел)
ffCurrency	Денежный формат. Соответствует ffNumber, но в конце строки ставится символ денежной единицы (для русифицированной версии Windows - символы «р.»). Для Value = π*1000 получим: 3 141,60р

ПРИЛОЖЕНИЕ 2. МАТЕМАТИЧЕСКИЕ ФОРМУЛЫ

Язык Object Pascal имеет ограниченное количество встроенных математических функций (|x| → abs (x), Arctg(x) → ArcTan(x), e^x → Exp(x), $\sqrt{x}$ → Sqrt(x), x^2 → Sqr(x), Ln(x), Cos(x), Sin (x) и др.). Поэтому при необходимости использовать другие функции следует применять известные соотношения или модуль Math. В таблице приведены выражения наиболее часто встречающихся функций.

Функция	Соотношение	Модуль Math
$Log_a(x)$	$\frac{Ln(x)}{Ln(a)}$	$LogN(a, x)$
x^a	$e^{a*Ln(x)}$	$Power(x,a)$
$Tg(x)$	$\frac{Sin(x)}{Cos(x)}$	$Tan(x)$
$Ctg(x)$	$\frac{Cos(x)}{Sin(x)}$	$CoTan(x)$
$ArcSin(x)$	$ArcTg\left(\frac{x}{\sqrt{1-x^2}}\right)$	$ArcSin(x)$
$ArcCos(x)$	$\frac{\pi}{2} - ArcSin(x)$	$ArcCos(x)$
$ArcCtg(x)$	$\frac{\pi}{2} - ArcTg(x)$	
$Sh(x)$	$\frac{e^x - e^{-x}}{2}$	$Sinh(\mathbf{x})$
$Ch(x)$	$\frac{e^x + e^{-x}}{2}$	$Cosh(x)$

$Sign(x)$	1, если x>0; 0, если x=0; *-1, если x<0*	$Sign(x)$

ПРИЛОЖЕНИЕ 3. ТАБЛИЦЫ СИМВОЛОВ ASCII

Стандартная часть таблицы символов ASCII

КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С
0		16	►	32		48	0	64	@	80	P	96	`	112	p
1	☺	17	◄	33	!	49	1	65	A	81	Q	97	a	113	q
2	☻	18	↕	34	"	50	2	66	B	82	R	98	b	114	r
3	♥	19	‼	35	#	51	3	67	C	83	S	99	c	115	s
4	♦	20	¶	36	$	52	4	68	D	84	T	100	d	116	t
5	♣	21	§	37	%	53	5	69	E	85	U	101	e	117	u
6	♠	22	▬	38	&	54	6	70	F	86	V	102	f	118	v
7	•	23	↨	39	'	55	7	71	G	87	W	103	g	119	w
8	◘	24	↑	40	(	56	8	72	H	88	X	104	h	120	x
9	○	25	↓	41	)	57	9	73	I	89	Y	105	i	121	y
10	◙	26	→	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	27	←	43	+	59	;	75	K	91	[	107	k	123	{
12	♀	28	∟	44	,	60	<	76	L	92	\	108	l	124	|
13	♪	29	↔	45	-	61	=	77	M	93	]	109	m	125	}
14	♫	30	▲	46	.	62	>	78	N	94	^	110	n	126	~
15	☼	31	▼	47	/	63	?	79	O	95	_	111	o	127	⌂

Некоторые из вышеперечисленных символов имеют особый смысл. Так, например, символ с кодом 9 обозначает символ горизонтальной табуляции, символ с кодом 10 – символ перевода строки, символ с кодом 13 – символ возврата каретки.

Дополнительная часть таблицы символов

КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С
128	А	144	Р	160	а	176	▒	192	└	208	╨	224	р	240	Ё
129	Б	145	С	161	б	177	▒	193	┴	209	╤	225	с	241	ё
130	В	146	Т	162	в	178	▓	194	┬	210	╥	226	т	242	Є
131	Г	147	У	163	г	179	│	195	├	211	╙	227	у	243	є
132	Д	148	Ф	164	д	180	┤	196	─	212	╘	228	ф	244	Ї
133	Е	149	Х	165	е	181	╡	197	┼	213	╒	229	х	245	ї
134	Ж	150	Ц	166	ж	182	╢	198	╞	214	╓	230	ц	246	Ў
135	З	151	Ч	167	з	183	╖	199	╟	215	╫	231	ч	247	ў

136	И	152	Ш	168	и	184	╕	200	╚	216	╪	232	ш	248	°
137	Й	153	Щ	169	й	185	╣	201	╔	217	┘	233	щ	249	·
138	К	154	Ъ	170	к	186	║	202	╩	218	┌	234	ъ	250	·
139	Л	155	Ы	171	л	187	╗	203	╦	219		235	ы	251	√
140	М	156	Ь	172	м	188	╝	204	╠	220		236	ь	252	№
141	Н	157	Э	173	н	189	╜	205	═	221		237	э	253	¤
142	О	158	Ю	174	о	190	╛	206	╬	222		238	ю	254	■
143	П	159	Я	175	п	191	┐	207	╧	223		239	я	255	

В таблицах обозначение **КС** означает "код символа", а **С** – "символ".

ЛИТЕРАТУРА

1. Архангельский А.Я. Программирование в Delphi 7. – М.: ЗАО “Издательство БИНОМ”, 2003.
2. Фаронов В.В. Delphi 6. Учебный курс. -М.: Издатель Молгачева С.В., 2001.
3. 3.Дж.Гленн Брукшир. Введение в компьютерные науки. – «Вильямс» М, С-П, Киев. 2001.
4. А.К. Синицын, С.В. Колосов, А.А. Навроцкий и др. Программирование алгоритмов в среде Delphi. Лаб. практикум. Ч. 1. – Мн., БГУИР, 2004.
5. Колосов С. В. Программирование в Delphi. Учеб. пособие – Мн., БГУИР, 2005.
6. Калиткин Н.Н. Численные методы. -М.: Наука, 1978. -512 с.
7. Бахвалов Н.С. Численные методы. -М.: Наука, 1975. -632 с.
8. Демидович В.П. и др. Численные методы анализа. -М.: Физматгиз, 1963. -400 с.
9. Волков Е.А. Численные методы. -М.: Наука, 1982. -255 с.
10. Крылов В.И. и др. Вычислительные методы высшей математики. Т.1. -Мн.: Вышэйшая школа, 1972. -584 с.
11. Крылов В.И. и др. Вычислительные методы высшей математики. Т.2. -Мн.: Вышэйшая школа, 1975. -672 с.
12. Форсайт Дж. и др. Машинные методы математических вычислений -М.: Мир, 1980. -280 с.
13. Шуп Т. Решение инженерных задач на ЭВМ. -М.: Мир, 1982. -238 с.
14. Самарский А.А. Введение в численные методы. -М.: Наука, 1982.-272 с.
15. 12. Березин И.С., Жидков Н.П. Методы вычислений. Т.2. -М.: Физматгиз, 1970. -620 с.
16. 13. Б.Банди. Методы оптимизации. Вводный курс. -М.: Мир, 1989. -277 с.