С.В. Колосов

Программирование в среде Delphi

Допущено Министерством образования Республики Беларусь
в качестве учебного пособия
для студентов специальностей
«Автоматизированные системы обработки информации»
и «Автоматическое управление в технических системах»
учреждений, обеспечивающих получение
высшего образования

Минск БГУИР 2005

УДК 004.4 (075.8)
ББК 32.973–018.1 я 73
К 61

Р е ц е н з е н т ы:
и.о.зав. кафедрой «Вычислительная техника» БГАТУ,
канд.техн.наук, доц. А.И. Шакирин,
зав. кафедрой дискретной математики и алгоритмики БГУ,
д-р физ.-мат.наук, доц. В.М. Котов

Колосов С.В.

К 61 Программирование в среде Delphi: Учеб. пособие. – Мн.: БГУИР, 2005. – 166 с.: ил.

ISBN 985–444–650–6

Учебное пособие раскрывает основы визуального программирования в среде Delphi. Оно включает в себя 33 темы. Первые 16 тем посвящены освоению элементов языка Object Pascal и приемам программирования алгоритмов при решении типовых задач. В следующих 17 темах изложены основы объектно-ориентированного программирования, состав библиотек классов и компонентов Delphi, возможности межпрограммного взаимодействия, COM-технологии, работа с базами данных и некоторые другие, важные для практики аспекты программирования.

УДК 004.4 (075.8)
ББК 32.973–018.1 я 73

ISBN 985–444–650–6

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Основу данного учебного пособия составляет курс лекций по программированию, читаемый автором студентам первого курса специальности «Автоматизированные системы обработки информации» БГУИР. Этот курс предполагает наличие у студентов только школьной подготовки по информатике. Он базируется на системе визуального программирования Delphi, которая работает под управлением операционной системы Windows. Основу Delphi составляет язык программирования Object Pascal, который изначально был разработан Н. Виртом в начале 60-х годов прошлого века специально как язык обучения программированию. От всех других языков программирования его отличают строгость в определении всех переменных и констант, модульность программирования, широкие возможности в создании собственных структур данных, использование объектно-ориентированного программирования, отсутствие машинно-ориентированных конструкций. Корпорация Borland, которая является родоначальником Delphi, с самого начала сделала ставку на визуальное объектно-ориентированное программирование с предоставлением возможности работы с любыми базами данных. В настоящее время система программирования Delphi ни в чем не уступает по своим возможностям таким языкам программирования, как C++, C#, Visual C++, C–Builder, Visual Basic и др.

1. ИСТОРИЯ РАЗВИТИЯ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ, СИСТЕМЫ СЧИСЛЕНИЯ И ЕДИНИЦЫ ИНФОРМАЦИИ

1.1. История развития вычислительной техники

Первым счетным инструментом, который изобрел человек, был абак. Он появился еще в V веке до нашей эры. Существовали разные виды абака – греческий, римский, китайский, японский и т.д. Один из его вариантов представлял собой специальную доску с песком, по которому проводились линии и на них размещались по позиционному принципу какие–нибудь предметы, например камушки или палочки. И сегодня еще можно увидеть русские счеты, которые иногда используют пожилые бухгалтеры.

Следующий этап в развитии вычислительной техники связан с именем шотландского математика Джона Непера, который изобрел в 1614 г. логарифмы. Логарифмы позволили заменить умножение и деление сложением и вычитанием. Еще и сегодня можно найти в магазинах логарифмические линейки. Однако они обладают не очень высокой точностью вычислений – всего до третьего знака числа.

Первые идеи механизировать вычислительный процесс появились в XVII веке. Вначале такая машина была описана Вильгельмом Шикардом, потом Леонардо да Винчи. Однако первая действующая механическая суммирующая машина была построена Блезом Паскалем в 1642 г. Затем появилось много вариантов механических вычислительных машин. Они создавались Лейбницом, Еленой Якобсон из Несвижа, русским математиком П.Л. Чебышевым и др. В середине прошлого века в бухгалтериях можно было увидеть механические

счетные машинки «Феликс» с колесиками для ввода чисел и боковой ручкой для выполнения арифметических операций.

Идея полностью автоматизировать вычислительный процесс принадлежит англичанину Чарльзу Бэббиджу. В 1834 г. он изобрел универсальную вычислительную машину с программным управлением, которую назвал аналитической. Она должна была состоять из четырех блоков. В первом блоке должны были храниться исходные числа, промежуточные результаты и команды управления. Он называл этот блок складом. В современном компьютере – это оперативная память. Второй блок назывался мельницей, здесь выполнялись операции над числами, сейчас этот блок называют арифметическим устройством. Третий блок – блок управления последовательностью операций, сейчас это – блок управления. Четвертый блок – для ввода исходных данных и печати результатов. Бэббиджу не хватило средств на постройку своей машины, его идеи остались только на бумаге.

Следующим этапом было создание электромеханических машин для вычислений с помощью перфокарт, которые получили название счетно-аналитических. В 1896 г. для переписи населения США были использованы перфокарточные машины Германа Холлерита. Фирма, в которой работал Холлерит, впоследствии была преобразована в широко известную фирму ИБМ.

В 1941 г. немецкий инженер К. Цузе построил первую универсальную машину с программным управлением на базе электромагнитных реле Ц-3. Она состояла из 2 600 реле, а программа вводилась с помощью двухдорожечной перфоленты. В США аналогичная машина «Марк–1» была построена по проекту Горварда Айкена только в 1944 г. Первая советская релейная машина РВМ-1 была создана в 1956 г. инженером Н.И. Бессоновым. Она содержала 5 500 реле и могла выполнять 50 сложений или 20 умножений в секунду.

Появление электронных ламп в 40-х годах прошлого столетия позволило совершить огромный скачок в повышении быстродействия вычислительных машин. Первую электронную вычислительную машину (ЭВМ) построили в США под руководством Дж.В. Моучли и Д.П. Эккарта. Она называлась ЭНИАК и содержала около 18 000 электронных ламп и 1 500 реле. Умножение чисел выполнялось уже за 2,8 миллисекунды. Правда, такая машина потребляла 150 кВт и работала не более одного часа в сутки, так как из 18 000 ламп какая–нибудь да выходила из строя и обслуживающий персонал постоянно менял блоки машины в поисках неисправности. Она занимала очень большую площадь и в ней одновременно гудели сотни вентиляторов, охлаждая ламповые блоки машины. В бывшем СССР первая малая электронно-счетная машина (МЭСМ) была создана под руководством академика С.А. Лебедева в 1950 г. Затем были разработаны ЭВМ – БЭСМ, «Стрела», «Урал» и др.

Следующим этапом стала замена ламп на полупроводниковые приборы. При этом резко сократилось потребление энергии и значительно возросла надежность ЭВМ. На Западе основным производителем таких машин стала американская фирма ИБМ. В СССР в 1963 г. появилась ЭВМ БЭСМ-6, обладающая скоростью 1 млн операций в секунду. В то время наша страна лишь незначительно отставала от США по производительности ЭВМ. На

Западе и в США в то время стала быстро развиваться микроэлектроника, и на ее основе появились микросхемы.

В 1969 г. в СССР была принята концепция единой серии ЭВМ – ЕС ЭВМ, в основу которой были положены аналоги американских микросхем фирм ИБМ и ИНТЕЛ. Переход на новую технологию у нас происходил очень сложно. Например, в Минске были простроены два завода: «Интеграл» – для производства микросхем, и Машиностроительный завод им. Орджоникидзе – для сборки ЭВМ. Первые ЭВМ этой серии ЕС-1020 обладали производительностью всего 20 тыс. операций в секунду. Лишь к 90-м годам прошлого века стали выпускаться ЭВМ ЕС-1060 производительностью около 10 млн операций в секунду. Это были большие машины, они занимали целый зал и состояли из нескольких шкафов памяти, питания, процессора и т.д. В США в это время уже стали появляться персональные ЭВМ, которые располагались на столе и обладали более высокими скоростными параметрами по сравнению с ЕС ЭВМ.

Параллельно с созданием универсальных ЭВМ шла разработка супер–ЭВМ для военных целей. Если в прошлом веке суперЭВМ обладали скоростью порядка 1 млрд операций в секунду, то теперь скорость таких машин увеличилась на три порядка. Это единичные ЭВМ, которые включают в себя тысячи процессоров и стоят очень дорого – миллионы долларов, но они определяют возможности общества в прогнозировании погоды, разработке новых технологий и решении очень сложных задач.

В настоящее время персональные ЭВМ обладают тактовой частотой около 4 ГГц, оперативной памятью более 1 Гбайта и при решении линейных задач практически не уступают суперЭВМ.

1.2. Системы счисления

Системы счисления бывают позиционные, когда каждый разряд числа имеет определенный вес, и знаковые, когда значение числа обозначают определенными знаками (римская система чисел). Мы будем рассматривать только позиционные системы счисления. В них в каждом разряде числа может быть только один символ.

Двоичная система счисления

Для записи двоичных чисел используются только два знака 0 и 1. Все вычислительные машины используют двоичную систему при своей работе. На выходе любого устройства имеется или низкий потенциал (0) или высокий (1). На рис.1.1 показан процесс переключения устройства из одного состояния в другое. Ранее были попытки использования десятичной системы счисления или троичной, но устройства на их основе оказались очень ненадежными.

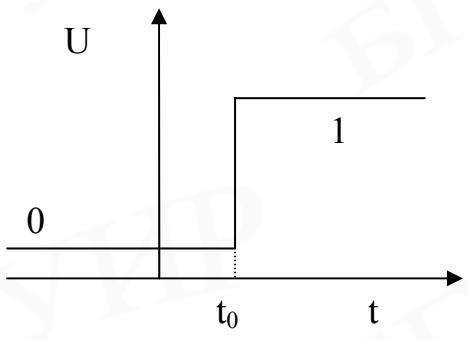


Рис.1.1

Для перевода числа из любой системы

счисления в десятичную можно использовать следующую формулу:

$$C_{10} = \sum_i a_i \cdot b^i ,$$

где a_i – разрядные коэффициенты; b – основание системы счисления; i – номер разряда.

Первый целый разряд числа имеет номер 0, дробные разряды нумеруются отрицательными числами. Например, следующее двоичное число можно перевести в десятичное таким образом:

номера разрядов 43210

двоичное число **$11011_{(2)}$**

$C_{10}=1\cdot2^0+1\cdot2^1+0\cdot2^2+1\cdot2^3+1\cdot2^4=27_{(10)}$.

Для перевода десятичного числа в любую другую систему счисления нужно: целую часть числа сначала делить на основание системы счисления, пока остаток от деления не станет меньше основания, затем результат деления опять делится на основание и так до тех пор, пока результат последнего деления не станет меньше основания. Результат последнего деления дает старший разряд числа, а остатки от предыдущих делений – остальные разряды числа. Дробную часть десятичного числа и последующих результатов умножений нужно умножать на основание системы счисления. Целые части результатов умножений и дадут требуемую дробь.

Например, переведем десятичное число C_{10}=25,35 в двоичное. Сначала целую часть этого числа будем делить на основание системы счисления – 2.

$25_{(10)}$	2		
1	12	2	
	0	6	2
		0	3	2
			1	**1**

Целая часть числа будет равна $C_{(2)}$=$11001_{(2)}$. Дробную часть числа сначала будем умножать на основание системы счисления – 2, а затем дробные части результатов умножения опять умножать на основание системы счисления.

0,	35
*	2
0,	7
*	2
1,	4
*	2
0,	8
*	2
1,	6

Целые части результатов умножений дадут следующую двоичную дробь – 0,0101…$_{(2)}$.

Попробуем произвести сложение и вычитание двоичных чисел.

Сложение:

$$\begin{array}{cccccc} & \Leftarrow & \Leftarrow & \Leftarrow & \Leftarrow & \Leftarrow \\ & 0 & 1 & 1 & 0 & 0 & 1 \\ + & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Здесь стрелочками показан перенос единицы в старший разряд числа, когда сумма разрядных чисел достигает или превышает основание системы счисления.

Вычитание:

$$\begin{array}{cccccc} & & \Rightarrow & & & \\ -1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 \end{array}$$

Здесь стрелочкой показан заем единицы из старшего разряда, которая для данного разряда равна основанию системы счисления.

Шестнадцатеричная система счисления

В шестнадцатеричной системе счисления для разрядных чисел используются не только 10 арабских цифр от 0 до 9, но и буквы латинского алфавита: A–10, B–11, C–12, D–13, E–14, F–15. Например, переведем следующее шестнадцатеричное число в десятичное.

номера разрядов 2 1 0

шестнадцатеричное число **$1AE_{(16)}$**

$C_{10}=14\cdot16^0+10\cdot16^1+1\cdot16^2=430_{(10)}$

Теперь переведем десятичное число в шестнадцатеричное:

$475_{(10)}$	16	
32	29	16
155	16	**1**
144	**13**	
11		

В итоге последний результат деления и остатки от деления дадут следующее шестнадцатеричное число: $1DB_{(16)}$.

Можно очень легко переводить шестнадцатеричные числа в двоичные и обратно, минуя десятичную систему счисления. Следует только иметь в виду, что один шестнадцатеричный разряд числа полностью соответствует четырем двоичным разрядам.

Например, переведем шестнадцатеричное число в двоичное:

$$A8E_{(16)}=1010\ 1000\ 1110_{(2)}.$$

1.3. Единицы информации

Наименьшей единицей информации в ЭВМ является 1 бит, который соответствует одному двоичному разряду числа. Например, двоичное число $11001_{(2)}$ содержит 5 бит информации, так как оно состоит из пяти двоичных разрядов и количество информации не зависит от значения разрядов числа.

При передаче информации наименьшей единицей информации является

1 байт, который соответствует 8 двоичным разрядам, или 8 битам.

В одном байте можно хранить 256 различных двоичных чисел, так как $11111111_{(2)}=255_{(10)}$ и плюс нулевая комбинация двоичных разрядов. Обычно в одном байте хранится информация об одном символе. Например, слово «студент» будут занимать в памяти ЭВМ 7 байт.

Далее следуют единицы информации:

1 килобайт = 1024 байта = 1 Кбайт,

1 мегабайт = 1024 Кбайт = 1 Мбайт,

1 гигабайт = 1024 Мбайт = 1 Гбайт,

1 терабайт = 1024 Гбайт = 1 Тбайт.

2. СТРУКТУРА ПЕРСОНАЛЬНОГО КОМПЬЮТЕРА И ОПЕРАЦИОННЫЕ СИСТЕМЫ

2.1. Структура персонального компьютера

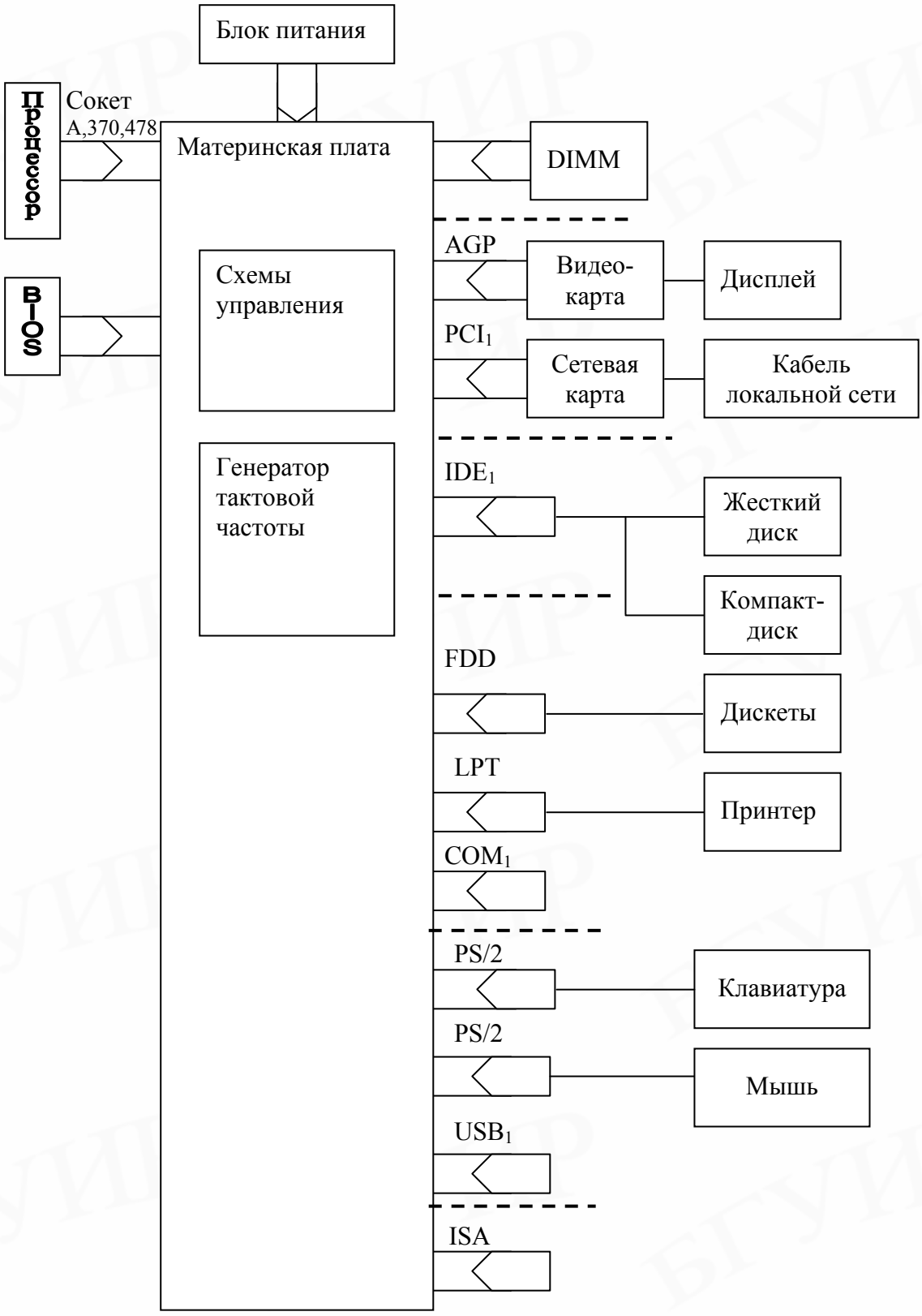


Рис.2.1. Структурная схема персонального компьютера

Здесь приняты следующие обозначения:

1. BIOS (Basic Input – Output System) – базовая система ввода-вывода. Она поставляется в виде отдельной микросхемы с постоянным запоминающим устройством (ПЗУ), в котором хранятся стандартные программы ввода-вывода и настройки материнской платы.
2. DIMM (Dual Inline Memory Module) – двойной встроенный модуль оперативной памяти – микросхема с 68 контактами, таких схем на материнской плате может быть несколько и объем такой памяти может достигать 2 Гбайт.
3. AGP (Accelerated Graphic Port) – ускоренный графический порт, к которому обычно подключается дисплей.
4. PCI (Peripheral Component Interconnect) – шина подключения внешних устройств, их обычно бывает несколько.
5. IDE (Integrated Dual Channel Enhenced) – объединенный двойной расширенный канал, к нему могут подключаться жесткие диски и устройства управления компакт-дисками, обычно таких каналов 2.
6. FDD (Floppy Dual Disk) – двойной канал для подключения устройств управления дискетами.
7. LPT (Line Printer Type) – параллельный порт для подключения принтера.
8. COM – последовательный порт для подключения внешних устройств, например мыши, таких портов обычно 2.
9. PS/2 – порт для подклячения клавиатуры и мыши.
10. USB (Universal Serial Bas) – универсальный последовательный порт, к нему может, например, подключаться устройство переносной памяти – Flash Drive.
11. ISA (Industry Standard Architecture) – устаревший порт для подключения внешних устройств.

Персональные компьютеры в настоящее время бывают настольные или портативные (Notebook). Тактовая частота работы процессора может превышать 3 ГГц, оперативная память обычно превышает 256 Мбайт. Дисплеи обладают разрешением 600 на 800 точек или 1 024 на 768 точек, причем видеопамять достигает 128 Мбайт и применяются специальные видеоускорители для отображения трехмерных объектов. К персональному компьютеру может быть подключено множество внешних устройств, таких, как жесткие диски с объемом памяти более 100 Гбайт, компакт-диски с объемом памяти 640 Мбайт, DVD диски, дискеты с объемом памяти 1,4 Мбайт, сканеры графической информации, принтеры всевозможных типов, сетевые карты, модемы, мышь, клавиатура, цифровая видеокамера и т.д.

2.2. Операционные системы

Все многообразие программ, используемых на современном компьютере, называется программным обеспечением (ПО). Программы, составляющие ПО, можно разделить на три группы: системное ПО, системы программирования, прикладное ПО. Ядром системного ПО является операционная система (ОС).

ОС – это неотъемлемая часть ПО, управляющая техническими средствами компьютера. Операционная система – это комплекс программ, которые выполняют функции посредника между пользователем и компьютером.

ОС, выполняя роль посредника, служит двум целям:

а) эффективно использовать компьютерные ресурсы;

б) предоставлять удобный интерфейс взаимодействия пользователя с компьютером.

В качестве ресурсов компьютера обычно рассматривают:

а) время работы процессора;

б) адресное пространство основной памяти;

в) оборудование ввода-вывода;

г) файлы, хранящиеся во внешней памяти.

Операционная система – это неотъемлемая часть вычислительного комплекса. Основными функциямя ОС являются управление:

а) процессами (распределяет ресурс – процессорное время);

б) памятью (распределяет ресурс – адресное пространство основной памяти);

в) устройствами (распределяет ресурс – оборудование ввода-вывода);

г) данными (распределяет ресурс – данные или файлы).

Одной из первых ОС для персонального компьютера была ДОС – дисковая операционная система фирмы Microsoft. Она изначально предполагала хранение ОС на диске. Работу диска можно функционально представить следующим образом:

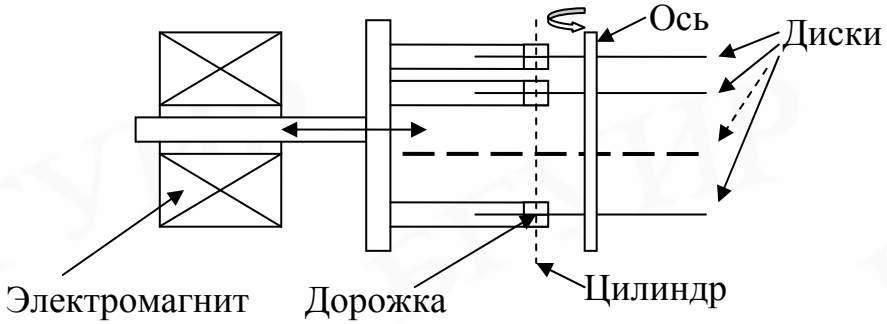


Рис.2.2. Схема дисковода

Здесь на оси закреплено несколько дисков с магниточувствительным покрытием. На подвижной системе, которая может перемещаться с помощью электромагнита только в радиальном направлении вдоль поверхности дисков, закреплены магнитные головки, по одной на каждую поверхность дисков. Все диски вращаются с какой-то угловой частотой Ω, и магнитные головки плавают на воздушной подушке у поверхности дисков. Каждое фиксированное радиальное положение магнитных головок образует как бы цилиндр, а каждая магнитная головка – свою дорожку на этом цилиндре. Каждая дорожка при записи информации разбивается на сектора обычно длиной 512 байт.

В ДОС была принята файловая система FAT (File Allocation Table – таблица распределения файлов). На системном диске на нулевом цилиндре на нулевой дорожке в первом секторе размещается информация о формате диска и о месте положения на диске программы первоначальной загрузки. Далее располагаются две таблицы FAT, которые разбиваются на 12-, 16- или 32-битные ячейки, в них хранятся номера кластеров, образующих файлы. Кластер–

это наименьший объем памяти, отводимый для файла на диске. Для дискеты размер кластера обычно равен размеру сектора, а для больших дисков размер кластера может составлять несколько секторов. Файл – это набор данных на внешнем носителе информации, например на диске. Создаются две таблицы FAT для большей надежности и обеспечения возможности восстановления случайно удаленных файлов. Обычно информация об удаленных файлах стирается только из первой таблицы.

За таблицами FAT идет корневой каталог, в котором для каждого файла отводится блок в 32 байта, куда записываются информация об имени и расширении файла, его атрибуты, размер, дата создания и номер первого кластера файла. В этой ячейке таблицы FAT записывается следующий номер кластера файла и т.д., пока не появится специальная запись, означающая последний кластер файла. Все остальные каталоги диска представляют собой обычные файлы с длиной, кратной 32 байтам, и специальным атрибутом файла – каталогом. За корневым каталогом до конца диска следует область данных, куда и помещаются все файлы.

Второй основной особенностью ДОС был текстовый режим работы дисплея, когда весь экран разбивался на 25 строк по 80 символов в каждой строке. В видеопамяти для каждого символа экрана отводилось два байта: один для хранения кода символа, а второй – для атрибута. В атрибуте хранилась информация о цвете символа и цвете фона. Всего могло быть 16 цветов символа и 8 цветов фона. Один бит отводился для возможного мигания символа. ДОС впрямую не поддерживала графический режим работы дисплея.

Третья особенность заключалась в том, что каждой программе можно было выделять не более 640 Кбайт оперативной памяти и вся эта память была доступна каждой программе. Это приводило к очень неустойчивой работе самой ОС. Любая прикладная программа могла нарушить работу ДОС и, как следствие, остановить работу компьютера.

Следующей широко распространенной операционной системой стала **Windows** той же фирмы Microsoft. Эта система изначально должна была работать в графическом режиме, хотя в ней и сохранилась поддержка текстового режима работы дисплея. Основной режим работы этой ОС – защищенный режим, когда каждой программе выделяется своя область памяти и никакая другая программа не может туда ничего записать или прочитать. Это стало возможным в связи с появлением новых процессоров Intel 386, 486 и Пентиум, которые стали поддерживать такие режимы работы.

Основу работы системы Windows составляет процесс обработки сообщений, которыми обмениваются окна (отсюда и название Windows – окна). В этой ОС может быть до 65 535 различных видов сообщений и до 65 535 окон различных программ.

Начиная с версий Windows NT/2000/XP, эта ОС стала поддерживать наряду с файловой системой FAT файловую систему NTFS, которая была разработана для компьютеров, работающих в сети, где очень важное значение имеет защита информации от несанкционированного доступа.

Как и любая другая система, NTFS делит все просранство на кластеры. NTFS поддерживает почти любые размеры кластеров – от 512 байт до 64 Кбайт, стандартом же считается кластер размером 4 Кбайт.

Диск NTFS условно делится на две части. Первые 12 % диска отводятся под так называемую MFT-зону – пространство, в котором располагается метафайл MFT. Запись каких-либо данных в эту область невозможна. В свободную часть MFT-зоны ничего не записывается – это делается для того, чтобы самый главный, служебный файл (MFT) не фрагментировался при своем росте. Остальные 88 % диска представляют собой обычное пространство для хранения файлов:

Рис.2.3. Распределение дискового пространства

Свободное место диска включает в себя всё физически свободное пространство – незаполненные куски MFT–зоны туда тоже включаются. Механизм использования MFT-зоны таков: когда файлы уже нельзя записывать в обычное пространство, MFT-зона просто сокращается, освобождая таким образом место для записи файлов. При освобождении места в обычной области MFT зона может снова расшириться.

Каждый элемент NTFS системы представляет собой файл – даже служебная информация. Самый главный файл на NTFS называется MFT, или Master File Table (общая таблица файлов). Именно он размещается в MFT-зоне и представляет собой централизованный каталог всех остальных файлов диска и, как ни парадоксально, себя самого. MFT поделен на записи фиксированного размера (обычно 1 Кбайт), и каждая запись соответствует какому-либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны операционной системе – они называются метафайлами, причем самый первый метафайл – сам MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая фиксированное положение. Интересно, что вторая копия первых трех записей для надежности (они очень важны) хранится ровно посередине диска. Остальной MFT-файл может располагаться, как и любой другой файл, в произвольных местах диска – восстановить его положение можно с помощью его самого, «зацепившись» за самую основу – за первый элемент MFT.

Итак, у системы есть файлы и ничего кроме файлов. Что включает в себя это понятие на NTFS? Прежде всего обязательный элемент – запись в MFT, ведь, как было сказано ранее, все файлы диска упоминаются в MFT. В этом месте хранится вся информация о файле, за исключением собственно данных:

имя файла, размер, положение на диске отдельных фрагментов и т.д. Если для информации не хватает одной записи MFT, то используются несколько, причем не обязательно подряд.

Опциональный элемент – потоки данных файла. Во–первых, файл может не иметь данных – в таком случае на него не расходуется свободное место самого диска. Во-вторых, файл может иметь не очень большой размер. Тогда идет в ход довольно удачное решение: данные файла хранятся прямо в MFT, в оставшемся от основных данных месте в пределах одной записи MFT.

Довольно интересно обстоит дело и с данными файла. Каждый файл на NTFS в общем-то имеет несколько абстрактное строение – у него нет как таковых данных, а есть потоки (streams). Один из потоков имеет привычный нам смысл – данные файла. Но большинство атрибутов файла – тоже потоки! Таким образом, получается, что базовая сущность файла только одна – номер в MFT, а всё остальное опционально.

NTFS – отказоустойчивая система, которая вполне может привести себя в корректное состояние при практически любых реальных сбоях. Любая современная файловая система основана на таком понятии, как **транзакция** – действие, совершаемое целиком и корректно или не совершаемое вообще.

NTFS поддерживает автоматическое сжатие и шифрование файлов и разграничивает права доступа к файлам.

Приведем простенький пример работы с потоком файлов. Наберите в командной строке оператор:

Echo Stream–NTFS > test1.pas:stream

В результате будут создан файл test1 с расширением pas нулевой длины. Однако, если теперь набрать команду

More < test1.pas:stream ,

то мы увидим на экране строчку – Stream-TNFS. Это значит, что имя потока отделяется от имени файла двоеточием.

В настоящее время идет постоянное расширение возможностей операционной системы. Это касается работы в сети Интернет, создания распределенных приложений для работы с базами данных, использования представления данных на языке XML, технологии .NET FrameWork и т.д.

3. ОСНОВЫ АЛГОРИТМИЗАЦИИ И РАБОТА В DELPHI

3.1. Основы программирования

Программирование – это процесс создания программы, который может быть представлен для реальных приложений последовательностью следующих шагов:

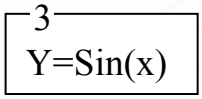
1. Формулирование требований к программе.
2. Разработка алгоритма.
3. Кодирование (запись алгоритма на языке программирования).
4. Отладка.
5. Тестирование.
6. Создание справочной системы.
7. Создание установочного диска.

Определение требований к программе – один из важнейших этапов, на котором подробно описывается исходная информация, формулируются требования к результату, определяется поведение программы в особых случаях (например при вводе неверных данных), разрабатываются диалоговые окна, обеспечивающие взаимодействие пользователя и программы.

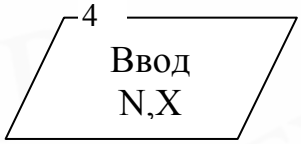
На этапе разработки алгоритма нужно определить последовательность действий, которые надо выполнить для получения результата. Если задача может быть решена несколькими способами и, следовательно, возможны различные варианты алгоритма решения, то программист, используя некоторый критерий, например скорость решения алгоритма, выбирает наиболее подходящее решение. Результатом этапа разработки алгоритма является подробное словесное описание алгоритма или его блок–схема. Для построения схемы алгоритма могут использоваться следующие графические элементы:

1. Начало и конец алгоритма

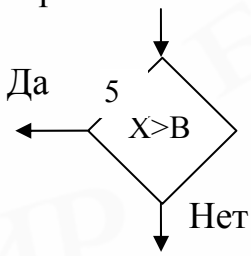
2. Линейный блок



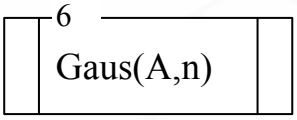
3. Ввод-вывод информации



4. Блок разветвления алгоритма



5. Вызов подпрограммы



6. Организация циклов

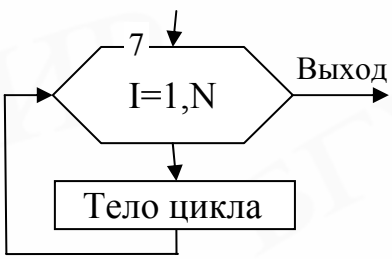


Рис.3.1. Графические элементы алгоритма

Следует помнить, что линии передачи управления между блоками схемы могут не иметь стрелок, если управление передается сверху – вниз или слева направо.

Приведем схему алгоритма для решения следующей задачи: необходимо рассчитать таблицу значений функции $y = \sum_{i=1,\ i\neq j}^{n} \sin(i \cdot x)/(i - j)$ для $a \le x \le b$. Шаг увеличения переменной x равен $h_x = (b - a)/10$.

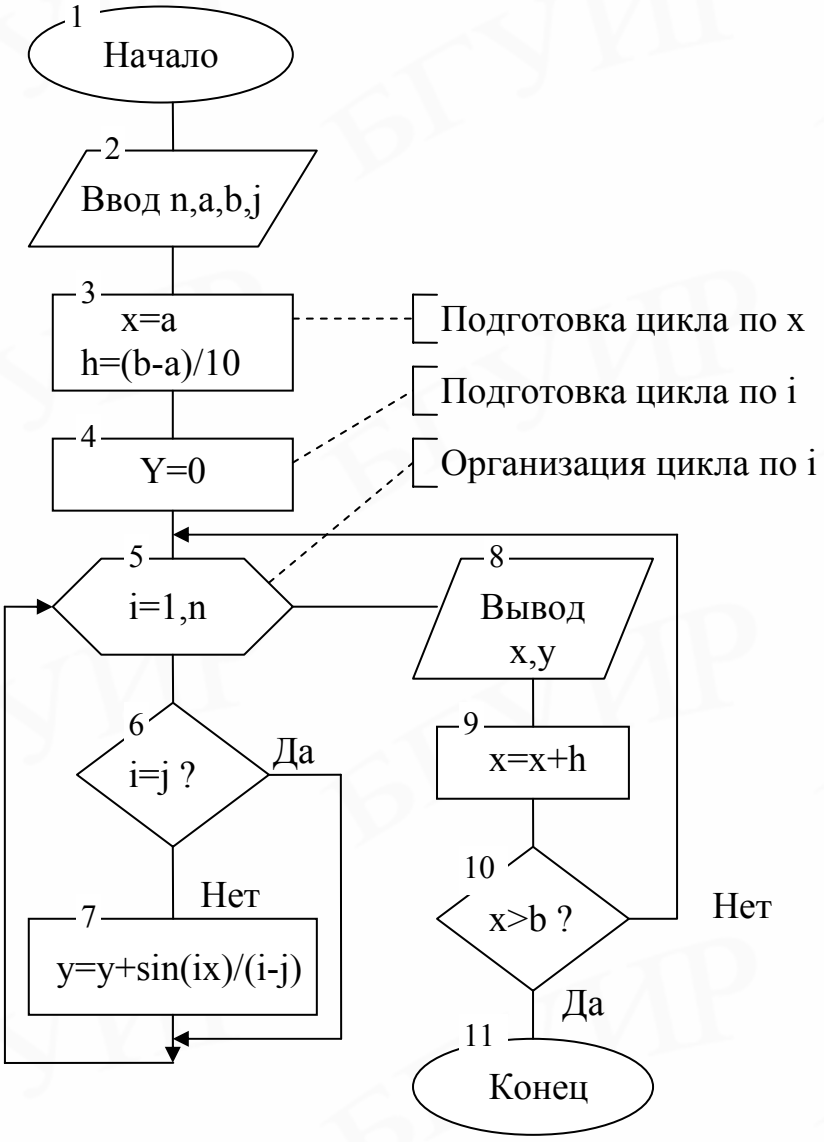


Рис.3.2. Пример схемы алгоритма

После того как определены требования к программе и составлен алгоритм решения, алгоритм записывается на выбранном языке программирования. В результате получается исходная программа.

Отладка — это процесс поиска и устранения ошибок. Практически ни одна программа не была написана сразу без ошибок. Ошибки в программе разделяют на две группы: синтаксические (ошибки в тексте) и алгоритмические. Синтаксические ошибки — наиболее легко устраняемые, большинство их выявляет транслятор на этапе компиляции. Алгоритмические ошибки обнару-

жить труднее. Этап отладки можно считать законченным, если программа правильно работает на одном-двух наборах входных данных.

Этап тестирования особенно важен, если вы предполагаете, что вашей программой будут пользоваться другие. На этом этапе следует проверить, как ведет себя программа на как можно большем количестве входных наборов данных, в том числе и на заведомо неверных.

Если предполагается, что программой будут пользоваться другие, то обязательно нужно создать справочную систему и обеспечить пользователю удобный доступ к справочной информации во время работы с программой. В современных программах справочная информация представляется в форме CHM- или HLP-файлов. Помимо справочной информации, доступ к которой осуществляется из программы во время ее работы, в состав справочной системы включают инструкцию по установке (инсталляции) программы, которую оформляют в виде Readme-файла в одном из форматов: TXT, DOC или HTM.

Для распространения программы необходимо будет создать установочный диск или CD-ROM для того, чтобы пользователь мог самостоятельно, без помощи разработчика, установить программу на свой компьютер. Обычно помимо самой программы на установочном диске находятся файлы справочной информации и инструкция по установке программы (Readme-файл). Следует учитывать, что современные программы, в том числе разработанные в Delphi, в большинстве случаев (за исключением самых простых программ) не могут быть установлены на компьютер пользователя путем простого копирования, так как для своей работы требуют специальных библиотек и компонентов, которых может и не быть у конкретного пользователя. Поэтому установку программы на компьютер пользователя должна выполнять специальная программа, которая помещается на установочный диск. Эта программа должна проверять права пользователя на установку конкретной программы или пакета программ. Как правило, установочная программа создает отдельную директорию для устанавливаемой программы, копирует в нее необходимые файлы и, если надо, выполняет настройку операционной системы путем внесения дополнений и изменений в реестр. Процесс создания установочного диска (CD-ROM) осуществляется обычно при помощи входящей в состав Delphi утилиты InstallShield Express.

3.2. Программирование в среде Delphi

Delphi – это среда визуального программирования на основе языка Object Pascal. Сам язык программирования Pascal был создан Н. Виртом в начале 60-х годов прошлого века специально как язык обучения программированию. От всех других языков программирования его отличает строгость в определении всех переменных и констант, модульность программирования, широкие возможности в создании собственных структур данных, использование объектно-ориентированного программирования, отсутствие машинно-ориентированных конструкций (например, как в Си “++i”). Корпорация Borland, которая является родоначальником Delphi, с самого начала сделала

ставку на визуальное объектно–ориентированное программирование с предоставлением возможности работы с любыми базами данных и опередила всех своих конкурентов на пару лет. В настоящее время система программирования Delphi ни в чем не уступает по своим возможностям таким языкам программирования, как C++, C#, Visual C++, C–Builder, Visual Basic и др.

Среда Delphi

После запуска Delphi на экране появляются пять окон (рис. 3.3):

- главное – Delphi 7**;**
- стартовой формы – Form1;
- редактора свойств объектов – Object Inspector;
- просмотра списка объектов – Object TreeView;
- редактора кода – Unitl.pas.

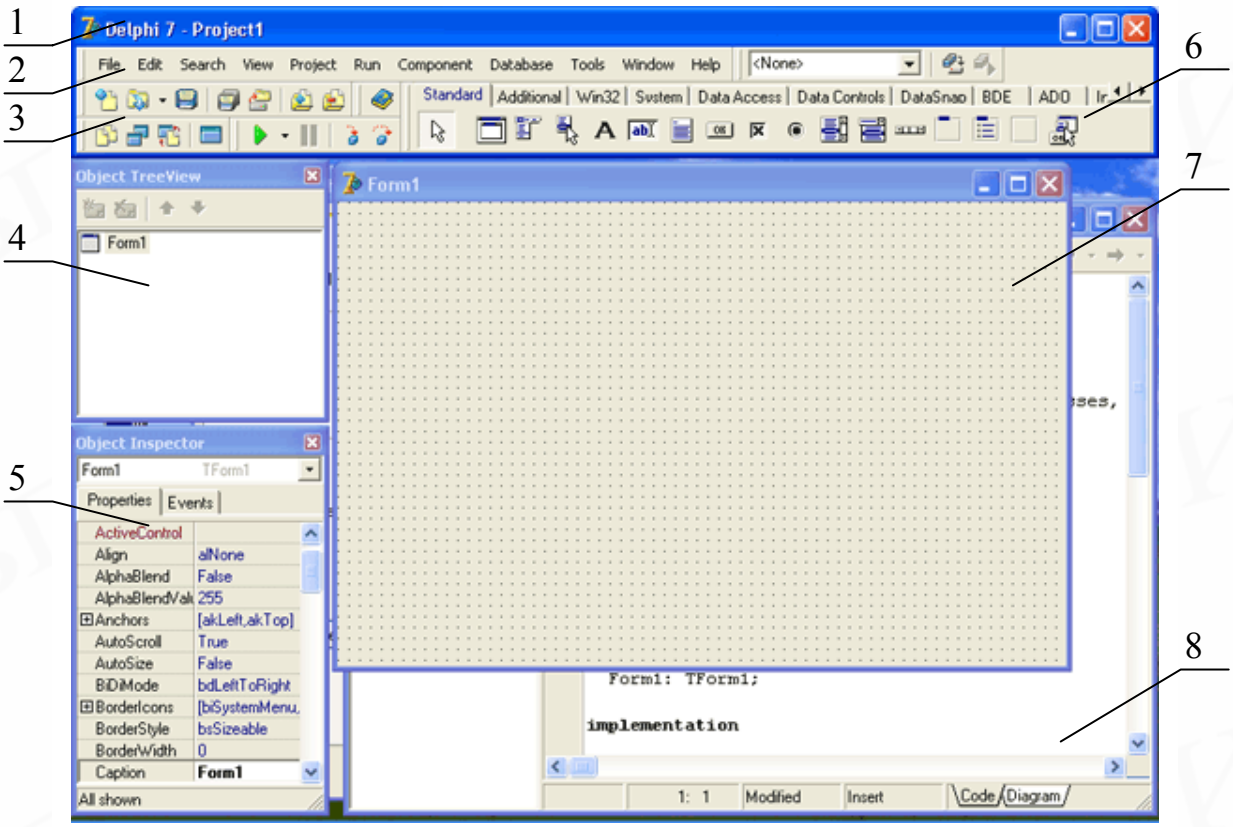


Рис. 3.3. Вид экрана после запуска Delphi:

1 – главное окно; 2 – основное меню; 3 – значки основного меню; 4 – окно просмотра дерева объектов; 5 – окно инспектора объектов; 6 – палитра компонентов; 7 – окно пустой формы; 8 – окно текста программы

В главном окне находятся меню команд, панели инструментов и палитра компонентов.

Окно стартовой формы **(Forml)** представляет собой заготовку главного окна разрабатываемого *приложения.*

Программное обеспечение принято делить на системное и прикладное. Системное программное обеспечение – это все то, что составляет операцион-

ную систему. Остальные программы принято считать прикладными. Для краткости прикладные программы называют приложениями.

Окно **Object Inspector** (см рис. 3.3) – предназначено для редактирования значений свойств и событий объектов. В терминологии визуального проектирования *объекты* – это диалоговые окна и элементы управления (поля ввода и вывода, командные кнопки, переключатели и др.). *Свойства объекта* – это характеристики, определяющие вид, положение и поведение объекта. Например, свойства Width и Height задают размер (ширину и высоту) формы, свойства Top и Left – положение формы на экране, свойство Caption – текст заголовка.

На страничке **Properties** перечислены *свойства* объекта и указаны их значения. На страничке **Events** перечислены *события*, на которые может реагировать объект, и здесь могут быть указаны методы обработки этих событий.

В окне редактора кода (см. рис. 3.3), которое можно увидеть, щелкнув мышью по этому окну, следует набирать текст программы. В начале работы над новым проектом это окно редактора кода содержит сформированный Delphi шаблон программы. При помещении любого компонента на форму текст программы автоматически дополняется описанием необходимых библиотек (раздел Uses) и типов переменных в классе TForm1. На начальном этапе обучения программированию в среде Delphi настоятельно не рекомендуется изменять имена компонентов и самому изменять содержимое классов.

Delphi для каждого приложения создает несколько файлов со следующими расширениями:

- ***.dpr** – файл описания проекта, где описываются все формы проекта (Project1.dpr);
- ***.pas** – файл модуля Unit, который является текстом программы для данной формы Form1 (Unit1.pas);
- ***.dfm** – файл описания формы и ее компонентов (Unit1.dfm). Он может храниться как в виде бинарного файла, так и в виде текстового файла;
- ***.res** – ресурсный файл, где хранятся значки, картинки, меню, константы, которые будут помещаться в форму (Project1.res);
- ***.dof** – файл настроек проекта (Project1.dof);
- ***.dcu** – результат трансляции модуля с расширением *.pas, т.е. текст модуля в машинных кодах;
- ***.exe** – результат редактирования программы, т.е. объединения всех модулей *.dcu в одну готовую к выполнению программу.

При выполнении лабораторных работ на дискете следует сохранять только файлы с расширениями *.dpr, *.pas, *.dfm и *.res. Остальные файлы являются рабочими и их можно не сохранять.

Следует иметь в виду, что Delphi поддерживает совместимость только снизу вверх, но не наоборот. Другими словами, программа, написанная в системе Delphi 5, будет работать и в системе Delphi 7, но не наоборот.

Пример написания простейшей программы в среде Delphi

Допустим, нам надо рассчитать простейшее выражение $y = \sin\left(\sqrt{|\cos(x)|}\right)$ для любого x. Для этого мы поместим на форму следующие компоненты: Label1 – для нанесения на форму текста «x=»; Edit1 – для ввода значения *x*; Memo1 – для вывода названия работы и результатов вычислений; Button1 – кнопку с текстом «Старт» для запуска процесса вычисления данной формулы.

Дважды щелкнув левой клавишей мыши по форме, мы получим обработчик события создания формы, где очистим текстовый редактор Memo1, затем запишем в него номер лабораторной работы, фамилию и номер группы студента, выполняющего работу. Здесь же очистим поле ввода значения x – Edit1.

Дважды щелкнув левой клавишей мыши по кнопке Button1, мы получим оформление обработчика нажатия этой кнопки. В нем определим переменные вещественного типа x и y, произведем вычисление **y** и выведем результаты вычислений в текстовый редактор Memo1.

В результате мы получим следующий текст модуля Unit1.pas с необходимыми пояснениями:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)      // Класс формы
    Label1: TLabel;
    Edit1: TEdit;
    Memo1: TMemo;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
// Обработчик создания формы
Procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Clear;                  //Очистка текстового редактора
  Memo1.Lines.Add('Лабораторная работа 1');     //Вывод пояснений
```

```
  Memo1.Lines.Add('Выполнил студент Иванов А.А., гр.320601');
  Edit1.Clear;                    // Очистка поля ввода
end;
// Обработчик нажатия кнопки
Procedure TForm1.Button1Click(Sender: TObject);
var x,y:extended;                 // Определение вещественных переменных x и y
begin
  x:=strtofloat(edit1.Text);  // Перевод строки в вещественное значение
  y:=sin(sqrt(abs(x)));           // Вычисление y
  Memo1.Lines.Add('x='+edit1.Text+'  y='+   // Добавление новой строки
  floattostr(y));                 // Перевод вещественного числа в строку
end;
end.
```

В данном тексте программы жирным шрифтом выделено все, что нужно было набрать самому студенту для выполнения задания. Все остальное среда Delphi добавляет автоматически сама. Следует иметь в виду, что имена компонентов отделяются от свойств и методов точкой.

В результате выполнения программы мы получим следующий вид формы:

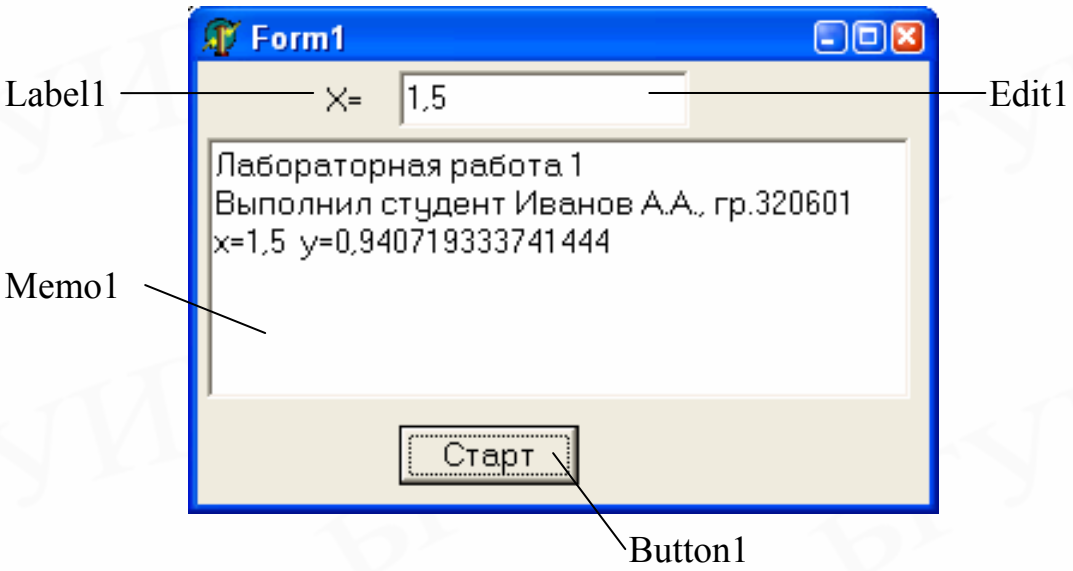


Рис.3.4. Форма простейшего примера

4. БАЗОВЫЕ ЭЛЕМЕНТЫ DELPHI

4.1. Алфавит среды Delphi

Для записи программы в Delphi можно использовать следующие символы:

1) буквы латинского алфавита A–Z, a–z (52 символа);
2) арабские цифры от 0 до 9;
3) специальные символы - . , : ; * / - + = ( ) [ ] { } ~ ! @ # $ % ^ & ‘ “ < > ? _;
4) буквы русского алфавита А–Я, а–я (64 символа). Их можно использовать только в текстовых константах и комментариях.

4.2. Константы

Константы могут быть разного типа:

1) целые – 1, -12, 1048;
2) вещественные – 1., -0.25, 0.25E-3 – эквивалентно $0{,}25*10^{-3}$;
3) логические – True – истина и False – ложь;
4) символьные – ’a’, ’Q’, #82 (символ # обозначает начало символьной константы, а 82 – номер символа в кодовой таблице);
5) строковые – ’Алгоритм’. Если в строку входит символ апострофа, то он записывается дважды, например, запись ’Дом ’’27’ эквивалентна строке – Дом ’27;
6) шестнадцатеричные – $A8, $FB3E, здесь символ $ обозначает начало шестнадцатеричной константы.

4.3. Переменные

Имя переменной должно начинаться с буквы и содержать не более 127 алфавитно-цифровых символов, но значащими будут только первые 63 символа. Например: A4, dw77eaaaabbbb, R_28i – все это имена переменных.

4.4. Основные типы переменных

Переменные целого типа

Целые числа в основном используются для нумерации и индексации различных данных. Они не могут иметь дробей и, например, при целочисленном делении 3 на 2 результатом будет целое число 1. Перечислим эти типы:

1) **Byte**. Целое число данного типа занимает в памяти ЭВМ 1 байт, и оно всегда положительно, т.е. его значения лежат в диапазоне от 0 до 255. Переменные этого типа можно объявить в разделе переменных следующим образом:

```
Var a, b1, Q32e:Byte;
```

Здесь объявлены три переменные типа Byte. Переменные отделяются друг от друга запятыми, а тип отделяется от списка переменных двоеточием;

2) **ShortInt,** или короткое целое. Число данного типа занимает в памяти тоже 1 байт, но старший двоичный разряд отведен для знака числа. 0 соответствует положительному числу, 1 – отрицательному числу. Диапазон значений от -128 до 127. Интересно отметить, что отрицательные целые числа хранятся в

виде дополнения к положительному числу. Например, положительное число $14_{(10)}=00001110_{(2)}=0E_{(16)}$, а отрицательное число $-14_{(1)}$ будет выглядеть следующим образом:

$$\begin{array}{r} 100000000 \\ \underline{-\ 00001110} \\ 11110010_{(2)}= -14_{(10)}= F2_{(16)}. \end{array}$$

Это позволяет ЭВМ не выполнять операцию вычитания целых чисел. В любом случае целые числа всегда складываются, а результат всегда будет правильным за счет особого способа хранения отрицательных чисел;

3) **Word**. Целое положительное число данного типа занимает в памяти ЭВМ 2 байта и может принимать значения в диапазоне от 0 до 65535;

4) **SmallInt**. Целое число данного типа занимает в памяти ЭВМ 2 байта и может принимать значения в диапазоне от -32768 до 32767;

5) **Integer** или **LongInt**. В первых версиях Delphi тип Integer занимал 2 байта, но начиная с Delphi–4 переменные этого типа занимают в памяти 4 байта, так же как и тип LongInt. Диапазон значений целых чисел лежит в интервале от -2 147 483 648 до 2 147 483 647;

6) **LongWord** или **Cardinal**. Целое положительное число данного типа занимает в памяти ЭВМ 4 байта и может принимать значения в диапазоне от 0 до 4 294 967 295;

7) **Int64**. Целое число данного типа занимает в памяти ЭВМ 8 байт и может принимать значения в диапазоне от -2^{63} до $2^{63}-1$.

Переменные вещественного типа

Вещественные числа – это числа, которые могут иметь дробную часть числа. Вешественные типы:

1) **Single**. Переменные этого типа занимают в памяти 4 байта. Диапазон представления чисел лежит в интервале от $1.5\cdot10^{-45}$ до $3.4\cdot10^{38}$. Точность представления таких чисел достигает 6 десятичных разрядов;

2) **Real48**. Ранее этот тип назывался просто Real и использовался на компьютерах, в которых отсутствовал математический сопроцессор. Переменные этого типа занимают в памяти 6 байт. Диапазон представления чисел лежит в интервале от $2{,}9\cdot10^{-39}$ до $1{,}7\cdot10^{38}$. Точность представления таких чисел достигает 10 десятичных разрядов. Переменные данного типа лучше не использовать в программе, так как математический сопроцессор не работает с таким представлением чисел и компилятор автоматически переводит их в тип Extended;

3) **Double** или **Real**. Переменные этого типа занимают в памяти 8 байт. Диапазон представления чисел лежит в интервале от $5{,}0\cdot10^{-324}$ до $1{,}7\cdot10^{308}$. Точность представления таких чисел достигает 16 десятичных разрядов;

4) **Extended**. Это основной вещественный тип, математический сопроцессор всегда работает с числами, представленными в этом типе. Переменные этого типа занимают в памяти 10 байт. Диапазон представления чисел лежит в

интервале от $3{,}6 \cdot 10^{-4951}$ до $1{,}1 \cdot 10^{4932}$. Точность представления таких чисел достигает 22 десятичных разрядов;

5) **Comp**. Это устаревший тип на 8 байт для работы с целыми числами, однако они воспринимаются как вещественные. Лучше вместо этого типа использовать тип Int64;

6) Тип **Currency**. Этот тип используется для работы с денежными значениями. Переменные этого типа в памяти занимают 8 байт. Это тип данных с фиксированной точкой, последние четыре десятичных разряда отведены для дробей или в денежном выражении – копеек и сотых долей копеек. Диапазон представления чисел лежит в интервале от – 92 233 720 3685 477,5808 до 922 337 203 685 477,5807.

Символьный тип

Язык Delphi поддерживает два символьных типа – **Ansichar** и **Widechar.** Тип **Ansichar** – это символы в кодировке ANSI, которым соответствуют числа в диапазоне от 0 до 255; тип **Widechar** – это символы в кодировке Unicode, им соответствуют числа от 0 до 65 535.

Delphi поддерживает и наиболее универсальный символьный тип – **Char**, который эквивалентен **Ansichar**.

Строковый тип

Строковые переменные могут быть следующих типов:

- **ShortString** – короткая строка с числом символов до 255,
- **AnsiString** – длинная строка с числом символов до 2^{31}-1,
- **WideString** – длинная строка с числом символов до 2^{30} и кодировкой UniCode, когда каждому коду символа выделяется по 2 байта.

Можно также использовать тип **String**. Он по умолчанию соответствует типу AnsiString, если явно не указана максимальная длина строки. В опциях транслятора ему в соответствие можно также поставить тип AnsiString.

Перечисляемый тип

Перечисляемый тип определяется как последовательность идентификаторов, которые не должны совпадать ни с какими другими именами переменных и зарезервированными словами Delphi. Он определяется в разделе типов. Например:

```
Type Tfio=(Иванов, Петров, Сидоров);
```

В программе в разделе переменных, допустим, определим переменную

```
Var Fio:TFio;
```

В разделе действий программы можно тогда записать

```
Fio:=Петров;
```

Тип диапазона значений

Тип диапазона значений представляет собой подмножество значений простых типов, при этом начальное и конечное значения разделяются двумя

точками. Например, Type Tlat=A..z; включает в себя подмножество всех латинских букв.

Логический тип

Существуют четыре логических типа: **Boolean, ByteBool, WordBool, and LongBool.** Переменные этих типов могут принимать только два логических значения: True – истина и False – ложь. Значению False соответствует нулевые значения полей этих переменных, а значению True – любые ненулевые значения.

4.5. Операции над переменными и константами

Арифметические операции:

1) + сложение,
2) - вычитание,
3) * умножение,
4) / вещественное деление,
5) **div** – целочисленное деление, результат всегда целый, а дробная часть отбрасывается,
6) **mod** – остаток от целочисленного деления, например 17 div 5 равно 2.

Операции отношения:

1) = равно,
2) <> не равно,
3) > больше,
4) < меньше,
5) >= больше или равно,
6) <= меньше или равно.

Результатом операции отношения всегда будет логическое значение, например, 7>9 равно значению – False.

Логические операции:

1. **OR** – логическое «или». Результат такой операции можно представить в виде следующей таблицы, где a и b операнды, а T (True) и F (False) – результаты этой операции:

a \ b	T	F
T	T	T
F	T	F

Например, если a = True, a b = False, то результат операции a OR b будет равен True.

2. **XOR** – исключающее «или» или сложение по модулю 2. Результат такой операции можно представить в виде следующей таблицы:

a \ b	T	F
T	F	T
F	T	F

3. **AND** – логическое «и». Результат такой операции можно представить в виде следующей таблицы:

a \ b	T	F
T	T	F
F	F	F

4. **NOT** – логическое отрицание. Результат такой операции можно представить в виде следующей таблицы:

a	Not a
T	F
F	T

Побитные логические операции

Перечисленные выше логические операции можно применять к переменным не только логического типа, но и к переменным целого типа. При этом эти операции производятся над соответствующими битами чисел. Например, 1 XOR 1 дает константу 0. Имеются еще две логоческие побитные операции:

- **shl** – побитный сдвиг влево, например 1 shl 1 равно 2,
- **shr** – побитный сдвиг вправо, например 16 shr 2 равно 4.

В выражениях без скобок операции выполняются в следующем порядке:

1. Not
2. / div mod and shl shr
3. + -or xor
4. = <> > < >= <= in

Например, выражение $y = \frac{a}{b \cdot c}$ можно запрограммировать как y:=a/b/c или со скобками y:=a/(b*c).

5. СТАНДАРТНЫЕ ФУНКЦИИ И ПОДПРОГРАММЫ

Для выполнения часто встречающихся вычислений и преобразований язык Delphi предоставляет программисту ряд стандартных функций.

Значение функции связано с ее именем. Поэтому функцию можно использовать в качестве операнда выражения, например в инструкции присваивания. Так, чтобы вычислить квадратный корень, достаточно записать k:=Sqrt(n), где Sqrt — функция вычисления квадратного корня, n — переменная, которая содержит число, квадратный корень которого надо вычислить.

Функция характеризуется типом значения и типом параметров. Тип переменной, которой присваивается значение функции, должен соответствовать типу функции. Точно так же тип фактического параметра функции, т.е. параметра, который указывается при обращении к функции, должен соответствовать типу формального параметра. Если это не так, компилятор выводит сообщение об ошибке.

5.1. Математические функции

Функция	*Значение*
Abs (x)	Абсолютное значение x
Sqrt (x)	Квадратный корень из x
Sqr (x)	Квадрат x
Sin (x)	Синус x
Cos (x)	Косинус x
Arctan (x)	Арктангенс x
Exp(x)	Экспонента x
Ln(x)	Натуральный логарифм x
Random(n)	Случайное целое число в диапазоне от 0 до n - 1
Random	Случайное вещественное число в диапазоне от 0 до 1
Exp(b*ln(a))	Возведение в степень a^b

Величина угла тригонометрических функций должна быть выражена в радианах.

5.2. Функции преобразования

Функции преобразования наиболее часто используются в инструкциях, обеспечивающих ввод и вывод информации. Например, для того чтобы вывести в поле вывода (компонент Label) диалогового окна значение переменной типа real, необходимо преобразовать число в строку символов, изображающую данное число. Это можно сделать при помощи функции FloatToStr, которая возвращает строковое представление значения выражения, указанного в качестве параметра функции.

Например, инструкция Labeil1.caption:=FioatTostr(x) выводит значение переменной x в поле Label1.

Функция	*Значение функции*
IntToStr(k)	Строка, являющаяся изображением целого k
Chr(n)	Символ, код которого равен n
Ord(c)	Код символа C
FloatToStr(x)	Строка, являющаяся изображением вещественного x
FloatToStrF(x,f,k,m)	Строка, являющаяся изображением вещественного x. При вызове функции указывают: f – формат (способ изображения, обычно – ffgeneral); k – точность (нужное общее количество цифр); m – количество цифр после десятичной точки
StrToInt(s)	Целое, полученное из строки s
StrToFloat(s)	Вещественное, полученное из строки s
Round(x)	Целое, полученное путем округления x по известным правилам

Trunc (x)	Целое, полученное путем отбрасывания дробной части x
Frac(x)	Вещественное, представляющее собой дробную часть вещественного x
Int (x)	Вещественное, представляющее собой целую часть вещественного x

5.3. Дополнительные системные подпрограммы и функции

Имя процедуры или функции	*Назначение*
Abort	Процедура окончания процесса без сообщения об ошибке
Halt(CodeError)	Процедура вызывает прекращение выполнения программы и устанавливает код ошибки CodeError
DateTimeToStr(Now)	Функция возвращает текущую дату и время в виде строки. Здесь функция Now дает текущую дату и время в виде 4-байтовой величины
Beep(F,D)	Функция из модуля Windows позволяет воспроизвести звук частотой F (герц) и длительностью D (миллисекунд)
Dec(X,N)	Процедура уменьшения значения числовой переменной X на единицу, если нет второго параметра, или на N
Inc(X,N)	Процедура увеличения значения числовой переменной X на единицу, если нет второго параметра, или на N
Pred(X)	Функция возвращает предыдущее порядковое значение числовой переменной X
Succ(X)	Функция возвращает следующее порядковое значение числовой переменной X
FillChar(X,Count,B)	Процедура заполнения поля переменной X Count значениями байта B
Hi(X)	Функция возвращает старший байт двухбайтовой переменной X
Lo(X)	Функция возвращает младший байт переменной X
LowerCase(S)	Функция возвращает строку S, в которой все латинские буквы будут маленькими
UpperCase(S)	Функция возвращает строку S, в которой все латинские буквы будут большими
SizeOf(X)	Функция возвращает длину переменной X в байтах
Val(S,X,Code)	Процедура перевода строки S в число X. Code – код ошибки перевода. Если Code не равен 0, эта переменная целого типа указывает на первый слева символ, который не может определять число
Odd(N)	Функция возвращает True, если аргумент N (целого типа) имеет нечетное значение

Move(Source,Dest, Count)	Процедура копирования Count байт из переменной Source в переменную Dest
Ramdomize	Процедура инициализации датчика случайных чисел, которая позволяет получать каждый раз новую последовательность случайных чисел с помощью функции Random

6. ОПЕРАТОРЫ DELPHI

6.1. Оператор присваивания

В результате выполнения оператора присваивания значение переменной меняется – ей присваивается новое значение. В общем виде оператор присваивания выглядит так: *Имя* **: =** *Выражение*, где *Имя* – переменная, значение которой изменяется в результате выполнения оператора присваивания; : = – символы инструкции присваивания; *Выражение* – выражение, значение которого присваивается переменной, имя которой указано слева от символа инструкции присваивания.

Пример:

Y:=a*b/10; S:=’Дом’; F1:= False;

Результат вычисления выражения всегда приводится к типу имени переменной и только затем записывается в поле переменной.

Каждый оператор Delphi заканчивается точкой с запятой (;), что позволяет в одной строке программы записывать несколько операторов.

6.2. Оператор безусловной передачи управления

Оператор безусловной передачи управления имеет вид

Goto метка;

Метка – это идентификатор, описанный в разделе меток – Label.

Например:

```
Label M1;
……
Goto M1;
……..
M1:y:=sin(x);
……
```

Оператор Goto передает управление оператору с указанной меткой, в нашем случае M1. Оператор, следующий за Goto, обязательно должен иметь метку, иначе он никогда не получит управление. По возможности следует избегать использования оператора Goto, так как это приводит к созданию неэффективных программ. Перескок внутри программы приводит к тому, что нужно заново обновлять очередь команд, готовых к выполнению в процессоре, и перенастраивать его управляющие регистры. Чем меньше в программе операторов Goto, тем выше квалификация программиста.

6.3. Условный оператор if

Этот оператор разветвления if в общем случае имеет вид

If условие **then** оператор1 **else** оператор2;

В этой записи зарезервированные слова выделены жирным шрифтом, а то, что подчеркнуто, может отсутствовать. Работает он следующим образом: если условие имеет значение «истина», то выполняется оператор1, иначе – оператор2.

Например:

If a>b then y:=a-b else y:=b-a;

В этом примере «y» всегда будет принимать положительные значения, независимо от значений a и b.

6.4. Оператор разветвления Case

Этот оператор в общем случае может иметь вид

Case выражение **of**

$знач_{11}..знач_{12}$:$оператор_1$;

$знач_{21}..знач_{22}$:$оператор_2$;

……..

$знач_{K1}..знач_{K2}$:$оператор_K$

else $оператор_0$;

end;

В зависимости от значения выражения выполняется тот или иной оператор. Если значение выражения не попадает ни в один из интервалов, то выполняется $оператор_0$.

Пример:

```
Var c:char; s:String;
…….
Case c of
  ’0’..’9’:s:=’Цифра’;
  ’A’..’z’:s:=’Латинская буква’;
..’A’..’я’:s:=’Русская буква’
  else s:=’Специальный символ’
  end;
```

6.5. Составной оператор

Этот оператор еще называют операторными скобками, он начинается ключевым словом Begin и заканчивается словом End. Между этими словами можно записывать любое число операторов, но чисто внешне он воспринимается как один оператор.

Пример:

```
If a>b then Begin
  Y:=a-b;
  Z:=a*b;
  End else Begin
```

```
    Y:=b-a;
    Z:=a/b;
    End;
```

В этом примере вместо одного оператора в инструкции if было использовано по два оператора присваивания.

Для лучшей читаемости программы рекомендуется после каждого оператора Begin все последующие строки программы сдвигать на две позиции вправо, а после оператора End – на две позиции влево. Это позволит легко отследить вложенность блоков Begin – End.

7. ОПЕРАТОРЫ ЦИКЛОВ

Алгоритм, в котором есть последовательность операций (группа инструкций), которая должна быть выполнена несколько раз, называется циклическим, а сама последовательность операций именуется циклом.

Хотя циклы можно легко организовывать с помощью оператора if, в Delphi есть три специальных оператора для организации циклов. Но вначале рассмотрим, как можно организовать цикл с помощью оператора if, например для задачи из разд. 3.1. Мы приведем только метод обработки нажатия клавиши «Старт»:

```
procedure TForm1.Button1Click(Sender: TObject);
// Обработчик нажатия кнопки
var x,y,h:extended;n,i,j:integer;// Определение внутренних переменных
Label M1, M2;
begin
  a:=strtofloat(edit1.Text); // Перевод строк в вещественное значение
  b:=strtofloat(edit2.Text);
  n:=strtoint(edit3.Text); // Перевод строки в целое значение
  j:=strtoint(edit4.Text);
  h:=(b-a)/10;  // Расчет шага по x
  x:=a;         // Подготовка цикла по x
  M1:y:=0;      // Подготовка цикла по i
  i:=1;
  M2:if i<>j then y:=y+sin(i*x)/(i-j);    // Расчет суммы
  i:=i+1;       // Наращивание переменной цикла i
  if i<=n then goto M2;      // Проверка окончания цикла по i
// Вывод результатов
  Memo1.lines.Add('x='+floattostr(x)+' y='+floattostr(y));
  x:=x+h;                    // Наращивание переменной цикла по x
  if x<=b then goto M1;      // Проверка окончания цикла по x
  end;
```

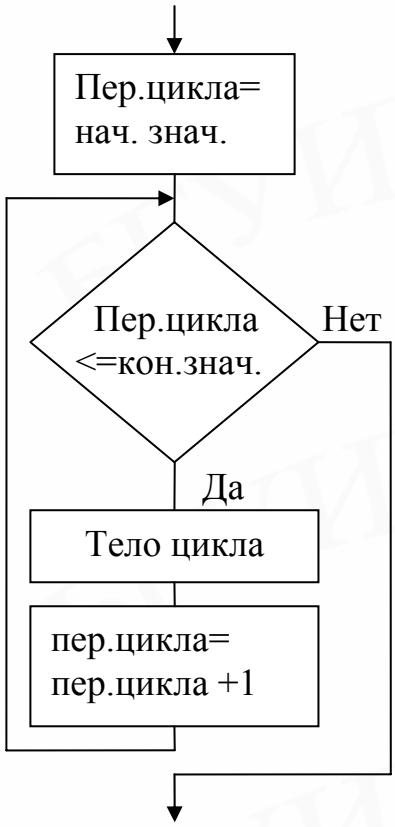
Как видно из этого примера, можно легко организовывать циклы с помощью оператора if, однако нам пришлось использовать две метки M1 и M2, что говорит о плохом стиле программирования.

7.1. Оператор цикла For

Этот оператор в общем случае имеет вид

For пер.цикла**:=** нач.знач. $\left\{ \begin{matrix} \textbf{To} \\ \textbf{Downto} \end{matrix} \right\}$ кон.знач. **Do** оператор$_1$;

Его следует понимать следующим образом. Для переменной цикла от начального значения до конечного значения выполнять оператор$_1$. В фигурных скобках показаны два варианта наращивания переменной цикла: To – соответствует шагу увеличения 1, а Downto – шагу -1. Переменная цикла, начальное и конечное значения обязательно должны быть целыми величинами. Переменная цикла должна быть внутренней переменной подпрограммы. Схему работы этого оператора можно изобразить следующим образом:



Правила использования оператора For:

1. Если начальное значение переменной цикла больше конечного, то цикл не выполняется ни разу.
2. Циклы можно вкладывать друг в друга, но не пересекать.
3. Можно передавать управление из цикла вовне его, но извне передать управление внутрь цикла нельзя.
4. Можно передать управление из цикла вовне его с последующим входом в этот же цикл.
5. Внутри цикла нельзя самим изменять значения переменной цикла.
6. После окончания цикла переменная цикла становится не определенной и ее значение уже нельзя использовать.
7. Если внутри цикла встречается оператор Break, то происходит принудительный выход из этого цикла.

8. Если внутри цикла встречается оператор Continue, то происходит переход к следующей итерации цикла.

Рассмотрим предыдущий пример программы, только внутренний цикл оформим с помощь оператора For. Эта часть программы теперь будет выглядеть следующим образом:

```
h:=(b-a)/10;   // Расчет шага по x
x:=a;          // Подготовка цикла по x
M1:y:=0;       // Подготовка цикла по i
For i:=1 to n Do if i<>j then y:=y+sin(i*x)/(i-j);    // Расчет суммы
// Вывод результатов
Memo1.lines.Add('x='+floattostr(x)+' y='+floattostr(y));
x:=x+h;                    // Наращивание переменной цикла по x
if x<=b then goto M1;      // Проверка окончания цикла по x
```

Как видим, число операторов программы уменьшилось на 3 и нет необходимости в метке M2.

7.2. Оператор цикла Repeat

Он имеет вид

Repeat тело цикла **Until** логическое выражение;

Выполнение цикла будет повторяться до тех пор, пока логическое выражение не примет значение «истина». Тело этого цикла в любом случае будет выполнено хотя бы один раз, так как проверка на окончание цикла здесь выполняется после выполнения операторов тела цикла. Внутри тела цикла обязательно должна изменяться какая-либо переменная, которая входит в логическое выражение и определяет условие выхода из цикла. В противном случае есть большая вероятность, что может получиться бесконечный цикл, т.е. программа зациклится. Давайте в предыдущем примере оформим внешний цикл по x с помощью оператора Repeat:

```
h:=(b-a)/10;   // Расчет шага по x
x:=a;          // Подготовка цикла по x
Repeat
 y:=0;         // Подготовка цикла по i
 For i:=1 to n Do if i<>j then y:=y+sin(i*x)/(i-j);  // Расчет суммы
// Вывод результатов
 Memo1.lines.Add('x='+floattostr(x)+' y='+floattostr(y));
 x:=x+h;       // Наращивание переменной цикла по x
 Until x>b;    // Проверка окончания цикла по x
```

Программа еще более упростилась и исчезла метка M1. В данном случае переменной цикла является x и она же входит в логическое выражение, определяющее условие выхода из цикла. В этом цикле переменная цикла может иметь любой тип в отличие от цикла For, где она должна иметь только целый тип.

7.3. Оператор цикла While

Он имеет вид

While логическое выражение **Do** оператор$_1$;

Пока логическое выражение принимает значение «истина», будет повторяться оператор$_1$. В этом цикле проверка на окончание цикла производится до выполнения оператора$_1$, и, если логическое выражение примет значение ложно, цикл заканчивает свою работу. В остальном – этот оператор похож на оператор **Repeat**. Давайте посмотрим, как предыдущий фрагмент программы будет выглядеть, если оператор **Repeat** заменить на оператор **While:**

```
h:=(b-a)/10;   // Расчет шага по x
x:=a;          // Подготовка цикла по x
While x<=b Do Begin      // Начало цикла по x и проверка окончания цикла
 y:=0;         // Подготовка цикла по i
 For i:=1 to n Do if i<>j then y:=y+sin(i*x)/(i-j);  // Расчет суммы
// Вывод результатов
 Memo1.lines.Add('x='+floattostr(x)+' y='+floattostr(y));
 x:=x+h;       // Наращивание переменной цикла по x
 end;
```

Как видим, этот фрагмент программы уменьшился еще на один оператор.

8. РАБОТА С МАССИВАМИ

Массив – это пронумерованный набор однотипных данных. Тип массива определяется в разделе типов и в общем случае имеет вид

Type имя массива=**Array[**список индексов и диапазоны их изменения**] of** тип элемента массива**;**

Пример объявления массивов:

```
Type Ta=array[1..10] of integer;
Tb=array[1..5,1..10] of extended;
Var a:Ta; b:Tb;
……….
a[1]:=25; b[5,10]:=-1.5;
```

Здесь объявлены: одномерный массив a целого типа, в котором элементы пронумерованы от 1 до 10, и двухмерный массив b вещественного типа, в котором первый индекс может изменяться в пределах от 1 до 5, а второй индекс – от 1 до 10. Индексы для элементов массивов могут быть любыми выражениями целого типа.

Рассмотрим пример написания обработчика нажатия какой-либо кнопки на форме, в котором произведем умножение матрицы A на вектор $\vec{X}$. В матричной форме это выглядит так:

$$\vec{Y} = A \cdot \vec{X} \text{ или } \begin{vmatrix} y_1 \\ .. \\ y_n \end{vmatrix} = \begin{vmatrix} a_{11} & .. & a_{1n} \\ .. & .. & .. \\ a_{n1} & .. & a_{nn} \end{vmatrix} \cdot \begin{vmatrix} x_1 \\ .. \\ x_n \end{vmatrix} \text{ или } y_i = \sum_{j=1}^{n} a_{ij} \cdot x_j ,$$

где n – размерность квадратной матрицы A и векторов Y и X.

Для визуализации массивов будем использовать табличные компоненты StringGrid1 для матрицы A, StringGrid2 для вектора X и StringGrid3 для вектора Y. Тогда обработчик события нажатия какой-либо кнопки будет выглядеть следующим образом:

```
// Обработчик нажатия кнопки
Procedure TForm1.Button1Click(Sender: TObject);
Const Nmax=10;      // Максимальная размерность массивов
Type Ta=array[1..Nmax,1..Nmax] of extended;
     Txy=array[1..Nmax] of extended;
Var A:Ta; x,y:Txy;n,I,j:integer;
Begin
  n:=StrtoInt(Edit1.Text);  // Определение текущей размерности массивов
  Randomize;                // Инициализация датчика случайных чисел
  with StringGrid1 do Begin       // Подготовка визуальных компонентов
    ColCount:=n+1;
    RowCount:=n+1;
    FixedCols:=1;
    FixedRows:=1;
    Sells[0,0]:='A';
    End;
  with StringGrid2 do Begin
    ColCount:=2;
    RowCount:=n+1;
    FixedCols:=1;
    FixedRows:=0;
    Sells[0,0,]:='X';
    End;
  with StringGrid3 do Begin
    ColCount:=2;
    RowCount:=n+1;
    FixedCols:=1;
    FixedRows:=0;
    Sells[0,0]:='Y';
    End;
  For i:=1 to n do Begin      // Заполнение массивов случайными числами
    StringGrid1[0,i]:=Inttostr(i);
    StringGrid1[i,0]:=Inttostr(i);
    StringGrid2[0,i]:=Inttostr(i);
    StringGrid3[0,i]:=Inttostr(i);
    For j:=1 to n do
    A[i,j]:=random;
    StringGrid1.Cells[j,i]:=Floattostr(a[i,j]);
    X[i]:=ramdom;
    StringGrid2.Cells[1,i]:=Floattostr(x[i]);
    End;
  For i:=1 to n do Begin      // Вычисление произведения матрицы на вектор
    Y[i]:=0;
    For j:=1 to n do
    Y[i]:=y[i]+a[i,j]*x[j];
```

```
  End;
 For i:=1 to n do              // Заполнение значениями компонента StringGrid3
 StringGrid3.Cells[1,i]:=FloattoStr(y[i]);
 End;
```

Рассмотрим теперь задачу сортировки элементов одномерного целого массива в порядке возрастания их значений методом «пузырька», когда можно менять местами только два соседних элемента массива. Размерность массива будет у нас задаваться в компоненте Edit1, начальный массив – в компоненте StringGrid1, а результат сортировки – в компоненте StringGrid2. Обработчик нажатия какой-либо кнопки будут иметь следующий вид:

```
// Обработчик нажатия кнопки
Procedure TForm1.Button1Click(Sender: TObject);
Const Nmax:=20;                  // Максимальная размерность массива
Type Ta=array[1..nmax] of integer;     // Тип Массива
Var a:Ta; i,j,k,n:integer;           // Внутренние переменные
Begin
 n:=StrtoInt(Edit1.Text);          // Определение размерности массива
 StringGrid1.ColCount:=1;          // Подготовка визуальных компонентов
 StringGrid1.RowCount:=n;
 StringGrid2.ColCount:=1;
 StringGrid2.RowCount:=n;
 Randomize;              // Инициализация датчика случайных чисел
// Заполнение начального массива случайными числами
 For i:=1 to n do Begin
  a[i]:= Random(2*n);
  StringGrid1.Cells[0,i-1]:=InttoStr(a[i]);
  End;
 Repeat            // Начало сортировки массива a
  J:=0;            // Зануление индикатора наличия перестановок
  For i:=1 to n-1 do          // Проход по всем элементам массива
  If a[i]>a[i+1] then Begin
    k:=a[i];               // Перестановка двух соседних элементов массива
    a[i]:=a[i+1];
    a[i+1]:=k;
    j:=j+1;
    end;
  until j<=0;   // Проверка наличия перестановок
 For i:=1 to n do StringGrid2.Cells[0,i-1]:=inttostr(a[i]);// Вывод результата
 End;
```

Решим теперь эту же задачу, но методом последовательного поиска минимального элемента. Сначала определим среди всех элементов массива минимальный и поменяем его местами с первым. Затем найдем минимальный, начиная со второго элемента, и поменяем найденный минимальный элемент уже со вторым и т.д. Запишем только алгоритм этой сортировки:

```
 For i:=1 to n-1 do Begin   // Начало сортировки
```

```
    j:=a[i]; jmax:=i;   // В качестве начального минимального берем I-й элемент
    For k:=i+1 to n do// Проходим по всем остальным элементам массива
    If j>a[k] then Begin        // Сравниваем текущий с запомненным
     j:=a[k]; jmax:=k;          // Запоминаем новый минимальный элемент
     end;
    if jmax<>i then Begin
  // Если нашли минимальный, то меняем его местами с I-м
     a[jmax]:=a[i];
     a[i]:=j;
     end;
  end;
```

Можно описывать и так называемые динамические массивы без указания диапазона значений индексов, например:

```
Var   A:array of integer;
      B:array of array of Byte;
```

Фактически это означает, что переменные A и B являются указателями на пока еще не выделенную область памяти. Выделение памяти под эти массивы должно происходить на этапе выполнения программой SetLength. В нашем случае это можно сделать, например, так:

```
SetLength(A, 10);
SetLength(B, 2);
SetLength(B[0],5);
SetLength(B[1],20);
```

Здесь массиву A выделяется память на 10 элементов массива, причем значение индекса начинается с нуля. Массиву B выделяется память под две строки, в первой строке может быть пять элементов, а во второй строке – 20 элементов.

Освобождение памяти производится процедурой **Finalize** или присвоением константы **Nil**, например:

```
Finalize(A);
B:=Nil;
```

9. РАБОТА СО СТРОКАМИ

Строки могут быть представлены следующими типами: Shortstring, AnsiString и Widestring. Различаются эти типы предельно допустимой длиной строки, способом выделения памяти для переменных и методом кодировки символов.

Переменной типа Sortstring память выделяется статически, т.е. до начала выполнения программы, и количество символов такой строки не может превышать 255. Переменным типа AnsiString и Widestring память выделяется динамически – во время работы программы, поэтому длина таких строк практически не ограничена.

Помимо перечисленных выше типов можно применять универсальный строковый тип String, который будет эквивалентен типу Shortstring или AnsiString в зависимости от настроек транслятора Delphi. Настройки можно

посмотреть, пройдя в основном меню Delphi путь: *Проект – Опции – Компилятор – Опции синтаксиса – Большие строки.*

Инструкция объявления строки в общем виде выглядит так:

Имя: String;

или

Имя: String [длина];

где

- имя – имя переменной;
- string – ключевое слово обозначения строкового типа;
- длина – константа целого типа, которая задает максимально допустимую длину строки.

Пример объявления переменных строкового типа:

name: string[20];

buff: string;

Если в объявлении строковой переменной длина строки не указана, то ее длина задается равной 255 символам, т.е. объявления

stroka: string [255]; и **stroka: string;**

эквивалентны.

В тексте программы последовательность символов, являющаяся строкой (строковой константой), заключается в одинарные кавычки. Например, чтобы присвоить строковой переменной S значение, нужно записать:

S:= 'Дует ветер';

Отдельные символы строки можно воспринимать как элементы одномерного массива. Для предыдущего примера s[2] определяет символ «у».

9.1. Процедуры работы со строками

Процедура **Delete**

Обращение к такой процедуре в общем виде выглядит так:

Delete(S,poz,n);

Эта процедура удаляет из строки S, начиная с позиции poz, n символов.

Например:

S:=’комплекс’;

Delete(S,3,4);

В результате строка S будет уже содержать слово «кокс».

Процедура **Insert**

Обращение к такой процедуре будет выглядеть так:

Insert(S1,S2,poz);

Эта процедура вставляет строку S1 в S2, начиная с позиции poz.

Пример:

S1:=’пре’;

S2:=’образование’;

Insert(s1,s2,0);

В результате строка S2 примет значение «преобразование».

Процедура **Str**

Обращение к такой процедуре будет выглядеть так:

Str(a,S);

Эта процедура переводит число a в строку S.

Пример.

a:=-25;

Str(a:5,S);

В результате строка S примет значение «__-25». Длина этой строки составит пять символов, но первые два будут пробелами.

Процедура **Val**

Обращение к такой процедуре будет выглядеть так:

Val(S,a,cod);

Эта процедура переводит значение строки S в число a. Cod – определяет ошибку перевода строки в число. Если Cod=0, то строка перевелась в число, а если нет, то Cod определяет первый слева символ в строке S, который не соответствует числовым данным.

Пример 1.

Val('-1.e2',a,cod);

В этом случае Cod=0 и a принимает значение -100.0. В

Пример 2.

Val('-1.ez3',a,cod);

А в этом случае Cod будет равен 5, так как пятым символом в строке стоит символ z, а он никак не может определять число.

9.2. Функции работы со строками

Функция **Copy**

Функция Copy позволяет выделить из строки подстроку, например:

S;=copy('строка',3,3);

Первая цифра 3 определяет, с какого символа начать копировать подстроку, а вторая цифра 3 определяет число копируемых символов. В результате переменной S будет присвоена строка «рок».

Функция ***Concat***

Функция Concat позволяет объединять произвольное число строк, записанных в качестве параметров этой функции, например:

S:=Concat('Грозовые',' ','облака');

Здесь переменная S примет строковое значение «Грозовые облака». Эту функцию можно просто заменить операцией сложения, например:

S:='Грозовые'+' '+'облака';

Функция ***Length***

Эта функция определяет текущую длину строки, например:

i:=Length('Студент');

Переменная i примет значение – 7.

*Функция **Pos***

Обращение к этой функции следующее:

Перем.:=Pos(S1,S2);

Эта функция ищет в строке S2 образ S1 и возвращает номер первого символа в строке S2, с которого начинается вхождение строки S1 в строку S2, например:

i:=Pos('ос','способность');

Здесь i примет значение 3, хотя образ «ос» встречается в строке 2 раза, начиная с позиции 3 и 8.

*Функция **UpCase***

Эта функция переводит строчные латинские буквы в заглавные, например:

c:=UpCase('q');

Здесь переменная символьного типа «c» примет значение «Q». Для перевода целой строки можно воспользоваться циклом For, например:

```
S:='Windows 2000/xp';
For i:=1 to Length(S) do
S[i]:=UpCase(S[i]);
```

В итоге строка S примет значение «WINDOWS 2000/XP».

Рассмотрим несколько примеров работы со строками.

Пример 1. Удаление лишних пробелов из строки. Лишними будем называть пробелы, которые стоят в начале и в конце строки, а также между словами, если их больше одного. Допустим, исходная строка будет задаваться в компоненте Edit1, а результат удаления пробелов – в компоненте Label1. Тогда обработчик кнопки удаления будет выглядеть так:

```
// Обработчик нажатия кнопки
Procedure TForm1.Button1Click(Sender: TObject);
Var S:String;
Begin
  S:=Edit1.Text;
// Удаляем начальные пробелы
  While ((Length(s)>0) and (Copy(s,1,1)=' ')) Do Delete (S,1,1);
// Удаляем конечные пробелы
  While ((Length(s)>0) and (Copy(s,length(s),1)=' '))
    Do Delete(S,Lengrh(s),1);
// Удаляем сдвоенные пробелы
  While ((Length(s)>0) and (Pos('  ',s)>0)) Do Delete(S,Pos('  ',s),1);
  Label1.Caption:=s;
  End;
```

Пример 2. Подсчет числа слов в строке. Слова могут разделяться одним или несколькими пробелами. Исходная строка пусть будет определяться в компоненте Edit1, а число слов – в компоненте Label1. Обработчик нажатия

клавиши «Вычислить» для компонента Button1 может выглядеть следующим образом:

```
// Обработчик нажатия кнопки
Procedure TForm1.Button1Click(Sender: TObject);
var s:String;
    i,ind,ns,l:Integer;
// ind – индикатор начала слова
// ns – счетчик числа слов
// l – длина строки
Begin
 S:=Edit1.Text;       // Читаем строку
 l:=length(s);  // Определяем ее длину
 ns:=0; Ind:=0;       // Зануляем число слов и индикатор начала слова
 For i:=1 to l Do Begin      // Открываем цикл по длине строки
  If ((ind=0) and (s[i]<>' ')) then Begin       // Определяем начало слова
    Inc(ns);            // Наращиваем число слов
    Ind:=1;             // Индикатор начала
    End else Begin
    If ((ind=1) and (s[i]=' ')) then ind:=0;
    End;
  End;
 Label1.Caption:='Число слов='+InttoStr(ns);
 End;
```

10. РАБОТА С ЗАПИСЯМИ

Запись – это набор данных, который может содержать поля разных типов и определяется ключевым словом – Record. Имена полей отделяются от имени записи точкой. Например, запись текущей даты можно определить следующим образом:

```
Type TDate=Record
            D:1..31;
            M:1..12;
            Y:1900..2200;
            End;
```

Здесь D, M и Y определяются диапазонами значений и соответствуют дню, месяцу и году. Теперь определим, например, запись типа анкеты сотрудника:

```
Type TAnketa=Record
      Number:integer;      // Порядковый номер
      FIO:String[80];      // Фамилия, имя и отчество
      DR:TDate;            // День рождения
      MR:String[50];       // Место рождения
      Dolg:String[40];     // Должность
      Oklad:Longint;       // Оклад
      End;
```

Чтобы не писать каждый раз имя записи и отделять его от поля точкой, можно использовать ключевое слово With с указанием имени записи. Затем в блоке Begin можно работать прямо с полями этой записи. Рассмотрим пример ввода полей записи для массива анкет сотрудников. Будем считать, что приведенные выше типы записей уже определены как глобальные типы. Определим глобальный массив таких записей и текущее число записей. На форме поля записей будут вводиться через элементы TEdit, здесь же будут присутствовать две кнопки – для ввода новой записи и для упорядочения записей по фамилиям сотрудников. Введенные записи будут отображаться в компоненте Memo1, а результат упорядочения – в Memo2:

```
// Определение глобальных констант и переменных
Const nmax=20;      // Максимальное число записей
Var n:integer;      // Текущее число записей
Ar:array[1..nmax] of TAnketa;  // Массив записей
// Определение обработчиков событий
// Обработчик создания формы
Procedure TForm1.FormCreate(Sender: TObject);
begin
  n:=0;             // Зануляем число записей
  Memo1.Clear;      // Очищаем текстовые редакторы
  Memo2.Clear;
end;
// Обработчик нажатия кнопки «Добавить запись»
procedure TForm1.Button1Click(Sender: TObject);
begin
  Inc(n);           // Наращиваем число записей
  With Ar[n] do Begin
    Number:=StrtoInt(Edit1.Text);
    Fio:=Edit2.Text;  //  Определение текущей записи анкеты
    Dr.D:=StrtoInt(Edit3.Text);
    Dr.M:=StrtoInt(Edit4.Text);
    Dr.Y:=StrtoInt(Edit5.Text);
    Mr:=Edit6.Text;
    Dolg:=Edit7.Text;
    Oklad:=StrtoInt(Edit8.Text);
    End;
  Memo1.Clear;      // Очищаем поле Memo1
  For i:=1 to n do With Ar[i] do Begin // Заново выводим все записи в Memo1.
    Memo1.Add('n='+InttoStr(Number)+' Fio='+FIO+' D='+InttoStr(Dr.D)+
    ' M='+InttoStr(Dr.M)+' Y='+InttoStr(Dr.Y)+' Mr='+Mr+' Dolg='+Dolg+
    ' Oklad='+InttoStr(Oklad));
    end;
  end;
// Обработчик нажатия кнопки «Сортировать»
procedure TForm1.Button2Click(Sender: TObject);
```

```
Var i,j,imin:Integer;          // Внутренние переменные
    Rt,Rmin:Tanketa;           // Внутренние записи
Begin
// Сортировка записей по фамилии методом последовательного
// поиска минимального значения
 For i:=1 to n-1 do do Begin
  Rmin:=Ar[i];
  Imin:=i;
  For j:=i+1 to n do Begin
   If Ar[j].FIO<Rmin.FIO then Begin
     Rmin:=Ar[j];
     Imin:=j;
     End;
   If imin<>I then Begin
     Rt:=Ar[i];          // Перестановка imin и i–й записей
     Ar[i]:=Rmin;
     Ar[imin]:=Rt;
     End;
   End;
  End;
 Memo2.Clear;          // Вывод массива упорядоченных записей в Memo2
 For i:=1 to n do With Ar[i] do Begin
  Memo1.Add('n='+InttoStr(Number)+' Fio='+FIO+' D='+InttoStr(Dr.D)+
  ' M='+InttoStr(Dr.M)+' Y='+InttoStr(Dr.Y)+' Mr='+Mr+' Dolg='+Dolg+
  ' Oklad='+InttoStr(Oklad));
  end;
 end;
end.
```

Записи могут иметь вариантную часть, которая похожа на оператор Case. В общем виде такая запись определяется так:

```
Type тип записи = record
  Поле–1: Тип–1;
  ...
  Поле–n: Тип–n;
 case переменная условия выбора: перечисляемый тип of
  Значение–1: (вариант полей 1);
  ...
  Значение–K: (вариант полей K);
 end;
```

Например, одна и та же память, выделяемая целой переменной длиной 4 байта, может быть разбита на отдельные четыре переменные, каждая из которых занимает в памяти 1 байт:

```
Type Tvar=(FourByte, OneByte);
     Tr=Record
          Case Tvar of
```

```
                Four Byte: FB:Integer;
                One Byte: B1,B2,B3,B4:Byte;
                End;
        End;
    Var R:Tr;
```

Теперь в программе можно использовать поле R.FB как переменную длиной 4 байта, а можно, например, использовать только младший байт из этой записи – R.B1.

11. ПРОЦЕДУРЫ И ФУНКЦИИ

Если необходимо многократно проводить какие-то вычисления для разных значений аргументов, то такие вычисления оформляются в виде подпрограммы «функции», если результатом вычислений является одна величина, в других случаях в виде подпрограммы – «процедуры».

В общем виде подпрограмма «процедура» определяется следующим образом:

Procedure Имя процедуры**(**Список формальных параметров**);** директивы**;**
Локальные описания
begin
Операторы процедуры
end;

Вызов процедуры осуществляется указанием имени процедуры со списком фактических параметров. Формальные параметры в списке разделяются точкой с запятой.

Для работы с подпрограммами используется специальная область памяти стек, который работает по принципу: первым вошел – последним вышел. Стек используется при вызове подпрограмм. В него записываются адрес возврата в вызывающую процедуру после окончания работы подпрограммы и передаваемые в подпрограмму параметры. В стеке отводится также место для всех внутренних переменных процедуры. При вызове подпрограммы происходит заполнение стека, а при выходе из процедуры из него исключается область памяти, которая была выделена процедуре при ее вызове. Одна процедура может вызвать другую, та – следующую и т.д., при этом происходит заполнение стека, а при выходе из каждой процедуры – его освобождение.

Существует пять видов формальных параметров:

1. Параметры–значения. Перед такими параметрами не ставится никаких ключевых слов. Значения таких параметров передаются через стек и в вызывающую программу они не возвращаются. Такими параметрами не могут быть файловые переменные и структуры, их содержащие.
2. Параметры–переменные. Перед такими параметрами записывается ключевое слово Var. В этом случае через стек в подпрограмму передается адрес фактического параметра, и поэтому изменение этого параметра в подпрограмме приводит к его изменению и для вызывающей программы.
3. Параметры–константы. Перед такими параметрами записывается ключевое слово Const. Эти параметры похожи на внутренние константы или

параметры, доступные только для чтения. Параметры константы подобны параметрам переменным, только внутри подпрограммы им нельзя присваивать никаких значений и их нельзя передавать в другие подпрограммы, как параметры переменные.

4. Выходные параметры. Перед такими параметрами записывается ключевое слово Out. Эти параметры имеют такие же свойства, как и параметры переменные, однако им не присваиваются какие–либо значения при вызове подпрограммы.

5. Нетипированные параметры. Для таких параметров не указывается тип параметра. При этом через стек передается адрес фактического параметра, а внутри подпрограммы по этому адресу можно расположить переменную любого типа.

При использовании подпрограмм нужно придерживаться следующих правил:

- фактическими параметрами при вызове подпрограмм могут быть переменные, константы и целые выражения;
- формальными параметрами при описании подпрограмм могут быть только имена переменных;
- фактические и формальные параметры должны быть согласованы по типу, порядку следования и количеству.

Для принудительного выхода из подпрограммы используется оператор **Return.**

Подпрограмма «функция» определяется следующим образом:

Function Имя функции**(**Список формальных параметров**):** тип результата**;**
директивы**;**
Локальные описания
begin
Операторы функции
end;

Вызов подпрограммы «функции» осуществляется указанием в правой части оператора присваивания имени функции со списком фактических параметров. Внутри функции обязательно нужно присвоить какое-то значение имени функции. Кроме имени функции можно применять ключевое слово Result, которое можно использовать в любом выражении, в то время как использование имени функции в выражении приводит к повторному вызову этой же функции, т.е. рекурсии.

Подпрограммы могут иметь директивы, которые записывают после объявления подпрограммы. Эти директивы могут определять правила вызова подпрограмм и передачи в них параметров. Директивой по умолчанию является директива **Register,** которая так же, как и **Pascal,** определяет передачу параметров в стек слева направо, т.е. так, как они записаны в определении подпрограммы.

Директивы **Cdecl, Stdcall** и **Safecall** определяют передачу параметров наоборот, т.е. справа налево, как это принято в языке Си.

Для всех директив, кроме **Cdecl,** освобождение стека происходит до выхода из подпрограммы, а для **Cdecl** – после передачи управления вызывающей программе.

Директива **Register** передает первые три параметра через регистры процессора, в то время как остальные передают все параметры через стек.

Директива **Safecall** используется при работе с OLE Automation интерфейсами.

Директивы **Near, Far** и **Export** использовались только в 16-битных приложениях и в новых версиях Delphi оставлены только для совместимости со старыми программами.

Директива **Forward** означает, что дано только описание параметров вызова процедуры, а само описание тела процедуры будет дано ниже. Такая ситуация возникает, когда, допустим, процедура A вызывает процедуру B, в то время как процедура B сама вызывает процедуру A. В этом случае можно сначала описать параметры процедуры B с директивой **Forward**, затем поместить процедуру A, а затем текст процедуры B. Это связано с тем, что Delphi не разрешает использовать не определенные заранее процедуры. Директива **Forward** может быть использована только в секции Implementation модуля Unit.

Директива **External** используется при описании правил вызова процедуры, когда ее тело извлекается из динамической библиотеки (DLL) или из объектного модуля (OBJ) – результата трансляции с языка Си.

Директива **OverLoad** используется тогда, когда две процедуры имеют одно имя, но разный список параметров. При вызове таких процедур предварительно проверяется список фактических параметров и вызывается та процедура, список формальных параметров которой совпадает по типу со списком фактических параметров.

В локальных описаниях процедур и функций могут быть объявлены внутренние типы, константы, переменные и другие процедуры и функции. Во внутренних процедурах и функциях доступны все локальные описания подпрограммы, в которой они сами описаны. Описания, сделанные в файле проекта с расширением *.dpr или в интерфейсной части модулей Unit, являются глобальными и доступными для всех подпрограмм проекта или модулей Unit. Эти переменные и константы располагаются в специальном сегменте данных, а не в стеке.

Для принудительного выхода из текущей процедуры или функции можно использовать процедуру Exit.

Рассмотрим пример расчета таблицы значений следующей функции:

$$y = \sin^2\left(\sum_{i=1}^{n} C_{2n}^{i} Cos(i \cdot x) \Big/ \sum_{j=1}^{n+1} C_{2(n+1)}^{j} Cos(2 \cdot j \cdot x) \right).$$

Переменная x будет изменяться в интервале $a \le x \le b$ с шагом h=(b - a)/10. Здесь $C_n^m = \dfrac{n!}{m! \cdot (n-m)!}$ – число сочетаний из n по m, $n! = \prod_{i:=1}^{n} i$ – число переста-

новок по *n*. Оформим расчет числа перестановок, сочетаний и сумм в виде подпрограмм функций. Исходные данные будем определять в компонентах TEdit, а результат вычислений поместим в многострочный редактор TMemo. Обработчик нажатия клавиши «Вычислить» будет иметь следующий вид:

```
Procedure TForm1.Button1Click(Sender: TObject);
  Var n:integer; a,b,h,x,y:Extended;
  Function F(n:Integer):Extended; // Функция расчета факториала
  Var i:Integer;
  Begin
    Result:=1;          // Вначале полагаем произведение равным единице
    For i:=1 to n do  // Организуем цикл умножений
    Result:=Result*i;
    End;
  Function C(n,m:Integer):Extended;          //  Расчет числа сочетаний
  Begin
    C:=F(n)/F(m)/F(n-m);
    End;
  Function Sum(n:Integer;x:Extended):Extended;        // Расчет суммы
  Var i:Integer;
  Begin
    Result:=0;          // Сумму полагаем равной нулю
    For i:=1 to n do  // Организуем цикл сложений
    Result:=Result+C(2*i,x)*Cos(i*x);
    End;
 Begin          // Тело обработчика нажатия клавиши «Выполнить»
  N:=strtoint(Edit1.Text); // Определяем начальные значения переменных
  a:=strtofloat(Edit2.Text);  b:=strtofloat(Edit3.Text);
  h:=(b-a)/10;                  // Расчет шага по X
  x:=a;                         // X полагаем равным начальному значению
  While x<b do Begin       // Организуем цикл по X
    Y:=Sqr(sin(Sum(n,x)/Sum(n+1,2*x)));    // Рассчитываем Y
    // Выводим результаты
    Memo1.Lines.Add('x='+Floattostr(x)+'  y='+Floattostr(y));
    x:=x+h;                     // Наращиваем X
    end;  end;
```

Рассмотрим еще один пример уже с использованием процедур. Необходимо написать программу по возведению квадратной матрицы в любую степень, т.е. $B = A^{\mathrm{m}}$, где A – заданная квадратная матрица размерности n:

$$A = \begin{vmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{vmatrix}.$$ Для решения этой задачи нужно написать подпрограмму

умножения двух матриц, т.е. $C = A \cdot B$ или $c_{ij} = \sum_{k:=1}^{n} a_{ik} \cdot b_{kj}$.

Размерность матрицы n и степень m будем задавать в компонентах TEdit. Начальную матрицу A будем определять случайным образом и помещать в компонент StringGrid1, а результат возведения в степень разместим в компоненте StringGrid2. Обработчик нажатия клавиши «Вычислить» будет иметь следующий вид:

```
Procedure TForm1.Button1Click(Sender: TObject);
Const nmax=20;       //Объявление констант, типов и переменных
Type Ta=Array[1..nmax,1..nmax] of extended;
Var i,j:integer;
    A,B,C:Ta;
// Процедура перемножения матриц
    Procedure UM(n:integer; var A,B,C:Ta);
    Var i,j,k:Integer; s:Extended;  // Внутренние переменные процедуры
    Begin
     For i:=1 to n do
     For j:=1 to n do Begin
      S:=0;
      For k:=1 to n do
      s:=s+A[i,k]*B[k,j];
      C[i,j]:=s;
      End;
     End;
Begin
 n:=StrtoInt(Edit1.Text);          // Определение размерности матриц
 m=StrtoInt(Edit2.Text);           // Определение степени возведения матрицы
 With StringGrid1 do Begin         // Подготовка визуальных компонентов
  RowCount:=n;
  ColCount:=n;
  FixedRows:=0;
  FixedCols:=0;
  End;
 With StringGrid2 do Begin
  RowCount:=n;
  ColCount:=n;
  FixedRows:=0;
  FixedCols:=0;
  End;
 Randomize;                        // Инициализация датчика случайных чисел
 For i:=1 to n do                  // Заполнение матрицы A случайными числами
 For j:=1 to n do Begin
  A[i,j]:=random;
  B[i,j]:=A[i,j];
```

```
  StringGrid1.Cells[j-1,i-1]:=FloattoStrF(A[i,j], ffGeneral, 10,5);
  End;
 For k:=1 to m-1 do Begin // Организация цикла возведения в степень A
  UM(n,A,B,C);
  For i:=1 to n do     // Запоминание текущего результата умножения матриц
  For j:=1 to n do
  B[i,j]:=C[i,j];
  End;
 For i:=1 to n do              // Вывод результатов вычислений
 For j:=1 to n do
 StringGrid2.Cells[j-1,i-1]:=FloattoStrF(C[i,j], ffGeneral, 10,5);
 End;
```

Процедуры и функции, которые прямо или косвенно вызывают сами себя, называют рекурсивными.

Приведем пример такой функции для расчета факториала:

```
Function F(n:Integer):Extended;
Begin
  If n>1 then F:=F(n-1)*n else F:=1;
  End;
```

Расчет большинства полиномов осуществляется по рекуррентным формулам, когда каждый последующий полином рассчитывается через предыдущие полиномы. Например, $B_0=1$, $B_1=X$, $B_k=2\cdot X\cdot B_{k-1}-B_{k-2}$. Для расчета таких полиномов можно тоже использовать рекурсивную функцию вида

```
Function B(n:Integer; x:Extended):Extended;
Begin
  If n>1 then B:=2*X*B(n-1,x)-B(n-2,x) else
  If n=1 then B:=x else B:=1;
  End;
```

Как видно из этих примеров, рекурсивные функции получаются очень компактными, но они требуют большой аккуратности при их написании – небольшая ошибка в программе может приводить к переполнению стека из–за бесконечного вызова функции самой себя.

12. МОДУЛЬ UNIT

Модуль Unit является отдельной программной единицей – он описывается в отдельном текстовом файле с расширением *.pas и транслируется отдельно. Результатом трансляции является машинный код, который записывается в файл с расширением *.dcu. Структура модуля Unit может иметь следующий вид:

```
Unit Имя модуля;
Interface        // Интерфейсная часть модуля
Uses             // Имена подключаемых модулей
// Объявления глобальных типов, констант, переменных,
// заголовков процедур и функций, которые будут доступны в других
// модулях, подключивших данный модуль
```

Implementation // Секция реализации модуля
Uses // Имена подключаемых модулей
// Здесь могут определяться внутренние константы, типы, переменные,
// процедуры и функции, которые будут доступны только внутри
// данного модуля. Здесь же дается реализация всех процедур и функций,
// объявленных в интерфейсной секции модуля
……...
Initialization // Секция инициализации модуля
// В этой секции записываются операторы, которые будут выполнены
// сразу после загрузки программы в память ЭВМ. Секции инициализации
// будут выполняться в том порядке, в каком модули Unit описаны
// в основной программе в операторе Uses
// Секция инициализации может отсутствовать в модуле Unit
Finalization // Секция завершения
// Эта секция может присутствовать в модуле Unit, только если в нем есть
// секция инициализации
// Выполнение операторов этой секции происходит после окончания работы
// программы перед выгрузкой ее из оперативной памяти ЭВМ
// Эти секции выполняются в обратном порядке по сравнению с порядком
// выполнения секций инициализации
End. // Конец модуля Unit

Все объявления, сделанные в интерфейсной секции, являются глобальными для программ, использующих данный модуль. Подключение модулей к другим модулям осуществляется оператором Uses со списком подключаемых модулей. Если в интерфейсных секциях модулей есть определения с одинаковым именем, то воспринимается определение того модуля, который находится в конце списке модулей в операторе Uses.

Определения, данные в секции Implementation, имеют силу только внутри данного модуля Unit.

Рассмотрим пример написания модуля Unit для работы с комплексными числами. Любое комплексное число имеет реальную и мнимую части, например: $\dot{a} = a_{re} + j \cdot a_{im}$, где a_{re} – реальная часть комплексного числа, a_{im} – мнимая часть комплексного числа, а $j = \sqrt{-1}$ – мнимая единица.

Сложение двух комплексных чисел осуществляется по формуле

$$\dot{c} = \dot{a} + \dot{b} = (a_{re} + b_{re}) + j_{im}(a_{im} + b_{im}) = c_{re} + jc_{im}.$$

Умножение комплексных чисел определяется формулой

$$\dot{c} = \dot{a} \cdot \dot{b} = (a_{re} \cdot b_{re} - a_{im} \cdot b_{im}) + j(a_{re} \cdot b_{im} + a_{im} \cdot b_{re}) = c_{re} + jc_{im}.$$

В модуль Unit включим две подпрограммы функции для расчета сложения и умножения комплексных чисел:

Unit Complex; // Начало модуля Complex
Interface // Интерфейсная секция модуля
Type TComp=Record // Объявление типа комплексного числа
Re,Im:Extended;

```
    End;
Const JC:TComp=(re:0.0;im:-1.0);      // Мнимая единица
// Функция сложения комплексных чисел
Function AddC(a,b:TComp):TComp;
// Функция умножения комплексных чисел
Function MulC(a,b,:TComp):TComp;
Implementation                  // Секция реализации модуля
Function AddC(a,b:TComp):TComp;
Begin
  AddC.re:=a.re+b.re;
  AddC.im:=a.im+b.im;
  End;
Function MulC(a,b:TComp):TCOmp;
Begin
  MulC.re:=a.re*b.re-a.im*b.im;
  MulC.im:=a.re*b.im+a.im*b.re;
  End;
End.                  // Конец модуля Unit
```

В модуле формы мы должны в конце списка модулей в операторе Uses добавить имя модуля Complex. Пусть начальные значения для комплексных чисел $\dot{a}$ *и* $\dot{b}$ будут определяться в компонентах TEdit, а результат вычисления выражения $\dot{c} = (\dot{a} + \dot{b}) \cdot \dot{a}$ поместим в компонент Memo1. Тогда начало модуля формы Form1 и обработчик события «Вычислить» будут иметь вид

```
Unit Unit1;
Uses ……., Complex;
……….……
Procedure Button1Click(Sender:Tobject);
Var a,b,c:TComp;
Begin
  a.re:=StrtoFloat(Edit1.text);
  a.im:=StrtoFloat(Edit2.text);
  b.re:=StrtoFloat(Edit3.text);
  b.im:=StrtoFloat(Edit4.text);
// Вычисление заданного комплексного выражения
  c:=MulC(AddC(a,b),a);
// Вывод результата
  Memo1.Lines.Add('c.re='+FloattoStr(c.re)+'  c.im='+FloattoStr(c.im));
  End;
```

13. РАБОТА СО МНОЖЕСТВАМИ

Множество представляет собой набор однотипных элементов, при этом ни один элемент множества не может входить в него дважды и порядок расположения элементов в множестве не имеет никакого значения. В Delphi

размер множества не может превышать 256 значений, т.е. множество занимает в памяти компьютера всего 256/8=32 байта.

Множество определяется ключевым словом Set, за которым определяется тип элементов множества. Например, в разделе типов можно записать:

```
Type  TLatChar=Set of 'A'..'z';      // Множество латинских букв
      TSimv=Set of Char;             // Множество любых символов
      TColor=Set of (Red,Blue,Green,White,Black);
      // Множество цветов перечисляемого типа
```

Теперь можно определить сами множества и проделать некоторые операции над ними:

```
Var   L1,L2:TLatChar;
      S1,S2:TSimv;
      C1,C2:TColor;
Begin
  L1:=['A','B','C'];      // Множество из трех латинских букв
  L2:=['a'..'z'];         // Множество строчных латинских букв
  S1:=['z','ф'];          // Множество из двух символов
  S2:=[];                 // Пустое множество
  C1:=[Red];              // Множество из одного значения Red
  C2:=[Green];            // Множество из одного значения Green
  C2:=C1+C2;              // Множество уже из двух значений
```

Над множествами можно выполнять операции сложения «+» (объединения множеств), вычитания «-» (исключения из множества), умножения множеств «*» (пересечение множеств) и вхождения во множество «in». Кроме этого, можно использовать процедуры:

Include(S,i) – добавление элемента i во множество S,

Exclude(S,i) – исключение элемента i из множества S.

Над множествами можно выполнять операции сравнения, например:

S1=S2 будет равно True, если множества одинаковы;

S1<>S2 будет равно True, если множества различаются хоть немного;

S1>=S2 будет равно True, если S2 является подмножеством S1;

'z' in S1 будет равно True, если элемент 'z' принадлежит S1.

Рассмотрим пример работы со множествами. Допустим, что на форме находится компонент Memo1 и нам нужно, чтобы при нажатии на клавиатуре клавиши с латинскими буквами звучал сигнал одной частоты, с русскими буквами – другой частоты и с цифрами – третьей частоты. Для этого объявим глобальный тип множества символов и переменные этого типа в интерфейсной секции модуля Unit так:

```
Type Tc=Set of Char;
Var CL,CR,C0:Tc;
```

В обработчике события создания формы определим множества латинских, русских и цифровых символов следующим образом:

```
Procedure Tform1.FormCreate(Sender:TObject);
Begin
  CL:=['A'..'z'];
```

```
CR:=['А'..'я'];
C0:=['0' '9'];
End;
```

Обработчик события нажатия любой клавиши для компонента Memo1 будет выглядеть так:

```
Procedure TForm1.Memo1KeyPress(Sender: TObject; var Key: Char);
Begin
 if key in cl then windows.Beep(1000,100);
 if key in cr then windows.beep(2000,100);
 if key in c0 then windows.beep(4000,100);
 end;
```

Здесь через параметр Key передается в процедуру код нажатой клавиши. Если это клавиша с латинским символом, то подается звуковой сигнал частотой 1 000 Гц и длительностью 100 миллисекунд. Для русских букв этот сигнал будет с частотой 2 000 Гц, а для цифр – 4 000 Гц.

14. РАБОТА С ФАЙЛАМИ

Файл – это набор данных на внешнем устройстве хранения информации. В Delphi файлы могут быть следующих типов:

1) текстовые;
2) типированные;
3) нетипированные.

14.1. Текстовые файлы

Текстовые файлы отличаются от всех других тем, что каждая строка файла заканчивается двумя специальными символами:

$0D – конец строки,

$0A – перевод каретки или переход в начало следующей строки.

В таких файлах каждая строка может иметь свою длину. Такие файлы называются файлами с последовательным доступом. Читать их можно только последовательно – строка за строкой, а записывать новые строки можно только в конец файла. Если произвести запись новой строки в середину файла, то все последующие строки будут удалены из файла.

В Delphi многие компоненты имеют методы прямого чтения и записи в текстовый файл. Например, компонент TMemo имеет свойство Lines, в котором есть два метода для чтения и записи строк прямо из файла или в файл. Для чтения текстового файла, например, программы P1.pas с диска «D» из каталога «My» и записи его строк в поле TMemo достаточно записать один оператор:

```
Memo1.Lines.LoadFromFile('D:\My\P1.pas');
```

После коррекции строк в Memo можно обратно записать эти строки в тот же файл одним оператором:

```
Memo1.Lines.SaveToFile('D:\My\P1.pas');
```

Но иногда, например при разработке собственного текстового редактора, возникает необходимость прямой работы с текстовым файлом и его отдельными строками, тогда следует использовать процедуры и функции из системного модуля System.

Каждый текстовый файл определяется файловой переменной, которая может быть только одна для одного файла. Файловую переменную текстового типа определяют с помощь ключевых слов Text или TextFile. Например, можно объявить файловую переменную текстового типа оператором

Var F:Text;

Есть специальные файловые переменные для текстовых файлов – это Input и Output. Обычно они соответствуют клавиатуре и дисплею. Поэтому, если в операторах ввода не указана файловая переменная, то по умолчанию она соответствует Input, а при выводе – Output.

Основные процедуры и функции для работы с текстовыми файлами:

Procedure AssignFile(var F, FileName:String); – процедура назначения файловой переменной F пути к файлу FileName. Эта процедура не может вызвать ошибку при выполнении программы, так как она не проверяет наличие указанного файла;

Procedure Append(var F:Text); – процедура открытия файла на запись в конец файла. При этом проверяется наличие указанного файла, и если его нет, то возникает ошибка ввода-вывода или исключительная ситуация с кодом EInOutError. Операционная система при этом выводит окно с пояснениями ошибки. После этого программа заканчивает свою работу. Если программист хочет, чтобы программа при этом не прерывала свою работу, то он должен взять на себя обработку таких ошибок или с помощью оператора Try, или путем отключения стандартной обработки ошибок ввода-вывода с помощью директивы транслятору {$I-}. После этого можно с помощью функции *Function IOError:Integer;* определить код ошибки и программу дальнейших действий. Включить стандартную обработку ошибок ввода-вывода можно директивой {$I+}. Следует помнить, что функцию IOError можно использовать для анализа ошибки только один раз. Повторный вызов этой функции для одной и той же ошибки ввода-вывода уже не даст кода ошибки;

Procedure ReWrite(var f:Text); – открытие нового файла на запись. При этом, если уже существовал такой файл, то он станет пустым;

Procedure Reset(var f:Text); – открытие файла на чтение с его начала;

Procedure Read(var f:Text; список переменных); – чтение значений переменных из текстового файла. При вводе значений числовых переменных читается символьный образ числа до появления первого разделителя чисел – пробела или конца строки, а затем этот образ преобразуется в число, которое и присваивается текущей переменной из списка ввода. Например, если оператором Read(f, a,n); из файла f читается строка вида « -1.5e-3 27»,то переменной «a» будет присвоено значение $-1.5*10^{-3}$, а переменной «n» значение 27. Но если в сроке встретятся символы, не соответствующие образу числа, то возникнет ошибка ввода-вывода.

При вводе строковых переменных из файла читается столько символов, сколько определено в объявлении строки, или чтение идет до конца текущей строки в текстовом файле. Не следует в одном операторе ввода читать числовые переменные и строковые переменные вперемешку;

Procedure ReadLn(var f:Text; список переменных); – процедура чтения строки из текстового файла с обязательным переходом в начало следующей строки, даже в случае наличия в строке непрочитанных значений;

Procedure Write(var f:Text; список выражений); – вывод в файл значений списка выражений. Здесь выражениями могут быть константы, переменные, числовые выражения. Значения числовых переменных при этом переводятся в строковые значения, которые и записываются в файл;

Procedure WriteLn(var f:Text; список выражений); – эта процедура отличается от предыдущей только тем, что после вывода значений обязательно дописываются два символа – конца строки и перевода каретки;

Procedure CloseFile(var f:Text); – процедура закрытия файла. При этом происходит физическая запись из буфера обмена последних данных и освобождается файловая переменная (ее можно использовать уже для другого файла). В конце работы программы происходит автоматическое закрытие всех незакрытых файлов;

Function Eof(var f:Text):boolean; – функция проверки на окончание файла. Она возвращает истину, если достигнут конец файла;

Function Eoln(var f:Text):boolean; – функция возвращает истину, если достигнут конец строки;

Function SeekEoln(var f:Text):boolean; – функция проверки текущего указателя файла на наличие конца строки. При этом пропускаются пробелы и знаки табуляции (код $09);

Function SeekEof(var f:Text):boolean; – функция проверки текущего указателя файла на наличие конца файла. При этом пропускаются пробелы и знаки табуляции;

Procedure Flush(var f:Text); – процедура очистки буфера обмена с текстовым файлом и переноса его содержимого непосредственно на внешнее устройство. Это предотвращает потерю информации, например, при зависании компьютера или исчезновении напряжения в сети.

Рассмотрим пример программы, которая будет принудительно форматировать текст программы, написанный на Delphi. Вначале она должна убрать все лишние пробелы из строк программы, затем после каждого появления ключевых слов Class, Begin, Repeat и Case в сроке программы все остальные строки сдвигать на две позиции вправо, а после ключевых слов End и Until – на две позиции влево. Это позволит легко определять пределы циклов и вложенных блоков и программа станет легко читаемой. Пусть имя текстового файла задается в компоненте Edit1, начальный текст программы поместим в компонент Memo1, а результат форматирования в – Memo2 и обратно в файл с этим же именем, но расширение заменим на «ра~». Тогда обработчик нажатия клавиши «Выполнить» может иметь следующий вид:

```
Procedure Button1Click(Sender:Tobject);
Var f1,f2:Text;
    l,io:integer;
    s1,s2,s,sp:String;  // Sp – определяет начальные пробелы в строке
Function Udpr(s:String):String; // Функция удаления лишних пробелов
```

```
Begin
// Удаляем начальные пробелы
  While ((Length(s)>0) and (Copy(s,1,1)=’  ’)) Do Delete (S,1,1);
// Удаляем конечные пробелы
  While ((Length(s)>0) and (Copy(s,length(s),1)=’  ’)) Do De-
lete(S,Lengrh(s),1);
// Удаляем сдвоенные пробелы
  While ((Length(s)>0) and (Pos(’  ’,s)>0)) Do Delete(S,Pos(‘     ’,s),1);
  End;
Begin
  s1:=Edit1.Text;                // Путь к текстовому файлу
  s2:=s1; l:=length(s2); s2[l]:=’~’;          // Путь к результирующему файлу
  AssignFile(f1,s1);    // Назначение файловых переменных
  AssignFile(f2,s2);
  {$I-}     // Отключение стандартной обработки ошибок ввода-вывода
  Reset(f1);       // Открываем исходный файл на чтение
  Io:=IOResult;          // Запоминаем ошибку открытия файла
  {$I+}     // Восстанавливаем стандартную обработку ошибок ввода-вывода
  If io<>0 then Begin // Проверяем наличие ошибок открытия файла
// Если есть ошибки, то выводим предупреждающее сообщение
    MessageDlg('Ошибка при открытии файла '+s1, mtWarning, [mbOk],
0);
    Return
    End;
  Memo1.lines.Loadfromfile(s1); // Выводим в Memo1 начальный файл
  ReWrite(f2);  // Открываем на запись результирующий файл
  Sp:=’’;          // Зануляем начальные пробелы строки
  While not eof(f1) do Begin         // Открываем цикл просмотра исх. файла
    Readln(f1,s);          // Читаем текущую строку
    S:=Udpr(s); /          / Удаляем из нее лишние пробелы
    S:=sp+s;               // К новой строке добавляем начальные пробелы
    Writeln(f2,s);         // Записываем в файл f2 результирующую строку
    If pos(’Class’,s)>0 then sp:=sp+’     ’; // Наращивание начальных пробелов
    If pos(’Begin’,s)>0 then sp:=sp+’     ’;
    If pos(’Repeat’,s)>0 then sp:=sp+’     ’;
    If pos(’Case’,s)>0 then sp:=sp+’     ’;
     // Удаление начальных пробелов
    If ((length(sp)>0)and(Pos(’End’,s)>0)) then delete(sp,1,2);
    If ((length(sp)>0)and(Pos(’Until’,s)>0)) then delete(sp,1,2);
    End;
  CloseFile(f1);//  Закрытие файлов
  CloseFile(f2);
// Вывод в Memo2 результата форматирования
  Memo2.Lines.LoadFromFile(s2);
  End;
```

14.2. Типированные файлы

Типированный файл – это файл с прямым доступом, в котором все записи имеют одинаковую длину. Например, можно объявить типированный файл следующим образом:

```
Type Tz=Record
        Fio:String[40];
        Voz:Byte;
        End;
Var f:file of Tz;
```

В этом примере все записи фала будут одинаковой длины и равны 42 байтам. В таком файле можно читать и записывать записи в произвольном порядке в отличие от текстовых файлов. Рассмотренные в предыдущем параграфе процедуры и функции можно применять и для типированных файлов, кроме тех, которые заканчиваются символами «ln».

Рассмотрим дополнительные процедуры и функции для работы с типированными файлами:

Procedure Seek(var f; n:LongInt); – перемещает текущий указатель файла в начало n–й записи. Записи здесь нумеруются с нуля;

Function FileSize(var f):LongInt; – возвращает длину файла в записях;

Function FilePos(var f):LongInt; – возвращает номер текущей записи;

Procedure Truncate(var f); – усекает файл с текущей позиции, т.е. отсекаются все записи, начиная с текущей записи.

Рассмотрим пример замены в любом файле одного символа на другой. Будем использовать те же компоненты, что и в предыдущем примере, только результат замены будем записывать прямо в исходный файл. Заменим, например, во всем файле строчную букву «я» на прописную «Я». Обработчик нажатия кнопки «Выполнить» будет тогда иметь следующий вид:

```
Procedure Button1Click(Sender:Tobject);
Var     f1:File of Char;   //  Определяем типированный файл с длиной
        // одной записи в один символ
   C:Char;l:longint;
Begin
 s1:=Edit1.Text;      // Путь к текстовому файлу
 AssignFile(f1,s1);   // Назначение файловых переменных
 {$I-}    // Отключение стандартной обработки ошибок ввода-вывода
 Reset(f1);           // Открываем исходный файл на чтение
 Io:=IOResult;        // Запоминаем ошибку открытия файла
 {$I+}    // Восстанавливаем стандартную обработку ошибок ввода-вывода
 If io<>0 then Begin // Проверяем наличие ошибок открытия файла
// Если есть ошибки, то выводим предупреждающее сообщение
   MessageDlg('Ошибка при открытии файла '+s1, mtWarning, [mbOk],
0);
   Return
   End;
```

```
  Memo1.lines.Loadfromfile(s1); // Выводим в Memo1 начальный файл
// Открываем цикл посимвольного чтения исходного файла
  For l:=0 to Pred(FileSize(f1)) Do Begin
    Read(f1,c);
    If c='я' then Begin
      c:='Я';
      seek(f1,l);
      Write(f1,c);
      End;
    End;
  CloseFile(f1);
  Memo2.Lines.LoadFromFile(s1);
  End;
```

14.3. Нетипированные файлы

Файлы без типа, нетипированные файлы, определяются, например, оператором

```
Var F:file;
```

Эти файлы являются файлами с прямым доступом и фиксированной длиной одного блока. Длина одного блока по умолчанию равняется 128 байтам. Приведем дополнительные процедуры и функции для работы именно с такими файлами:

Procedure ReWrite(var f:file[;RecSize:word]); – открытие нового файла на запись, RecSize – длина одного блока в байтах;

Procedure Reset(var f:file[;RecSize:Word]); – открытие файла на чтение, RecSize – длина одного блока в байтах;

Procedure BlockRead(var f:file; var Buf; Count:Integer[;var Re sult:Integer]); – чтение блоков из файла f и запись их в буфер – Buf. Count – число блоков, которые хотим прочитать, Result – число блоков, реально прочитанных;

Procedure BlockWrite(var f:file; var Buf; Count:Integer [; var Result:Integer]); – запись блоков в файл f из буфера Buf. Count – число блоков, которые хотим записать, Result – число блоков, реально записанных в файл;

Function FileSize(var f:file):Integer; – возвращает число записанных блоков в файле f.

В качестве примера работы с нетипированными файлами приведем текст программы быстрого копирования файлов, которая может быть вызвана прямо из командной строки операционной системы:

```
Program CopyMy;
Const LBuf=65000;  // Длина буфера
Var       f1,f2:File;   // f1 – файл чтения, f2 – файл записи
    s1,s2:String;       // Строки для хранения путей к файлам
    Buf:Array[1..Lbuf] of byte;   // Буфер
    io,lr,lrt,lw,lwt:integer;  // Внутренние переменные
Begin
```

```
// Проверка наличия двух параметров в командной строке
 If paramcount<2 then Begin
  ShowMessage('Обращение к программе: CopyMy f1 f2');
  Halt;   // Аварийный выход из программы
  End;
 S1:=paramstr(1);   // Из командной строки извлекаем путь к файлу чтения
 S2:=paramstr(2);   // Из командной строки извлекаем путь к файлу записи
 Assignfile(f1,s1);
 {$I-}
 Reset(f1,1);   // Открываем файл f1 на чтение с длиной блока в 1 байт
 Io:=IOResult;// Запоминаем ошибку открытия файла f1
 {$I+}
 if io<>0 then Begin
  ShowMessage('Файла '+s1+'  нет!');
  Halt;
  End;
 Assignfile(f2,s2);
 ReWrite(f2,1);        // Открываем файл f2 на запись с длиной блока в 1 байт
 While not Eof(f1) do Begin // Открываем цикл переписи файла f1
  lr:=lbuf;
  BlockRead(f1,Buf,lr,lrt);         // Читаем из f1 lr блоков
  BlockWrite(f2,Buf,lrt,lwt);       // Записываем в f2 lrt блоков
  If lrt<>lwt then Begin    // Проверяем правильность записи
   ShowMessege('Ошибка при переписи файла');
   Halt;
   End;
  End;
 CloseFile(f1);// Закрываем файлы
 CloseFile(f2);
 End.
```

Обращение к такой программе из командной строки, например, может быть таким:

CopyMy A:\p1.pas   C:\my\p1.pas

15. РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ

Для работы с файловой системой используются следующие функции и процедуры:

Function DiskSize(Drive: Byte): Int64; – получение размера диска в байтах. Параметр Drive определяет номер дисковода: 0 – текущий дисковод, 1 – A, 2 – B и т.д.;

Function DiskFree(Drive: Byte): Int64; – получение размера свободной области на диске;

Procedure ChDir(const S: string);overload; – смена текущей директории, например ChDir('C:\My\');

Procedure RmDir(const S: string);overload; – удаление пустой директории;

Procedure MkDir(const S: string); overload; – создание новой поддиректории;

Procedure GetDir(D: Byte; var S: string); – получение текущей директории на устройстве D;

Procedure Rename(var F; Newname: string); – переименование файла, связанного с файловой переменной F, в файл с именем Newname;

Procedure Erase(var F); – удаление закрытого файла, связанного с переменной F;

Function DirectoryExists(const Directory: string): Boolean; – проверка существования директории. Если она есть, то функция возвращает – True;

Function FileExists(const FileName: string): Boolean; – проверка наличия на диске файла с именем FileName;

Function FileGetAttr(const FileName: string): Integer; – получение атрибутов файла FileName;

Function FileAge(const FileName: string): Integer; – получение в сжатом виде даты создания файла для файла FileName;

Function FileDateToDateTime(FileDate: Integer): TDateTime; – перевод сжатой информации о дате создания файла в стандартный формат даты-времени. Для перевода его в строку следует использовать функцию DateToStr;

Function FileSearch(const Name, DirList: string): string; – поиск файла Name в списке каталогов DirList;

Function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer; – функция первого поиска файла в заданной директории и с заданным атрибутом, результат поиска помещается в запись F, она возвращает 0, если файл найден, иначе – код ошибки;

Function FindNext(var F: TSearchRec): Integer; – функция последующего поиска файла, удовлетворяющая условиям поиска в функции FindFirst, она возвращает 0, если файл найден;

Procedure FindClose(var F: TSearchRec); – освобождение памяти, выделенной ранее функцией FindFirst.

Рассмотрим пример применения этих трех функций. На форме должны быть помещены следующие компоненты:

Edit1 – для задания образа искомого файла, например, ’*.pas’ – для поиска всех файлов с расширением pas;

CheckBox1 – для задания атрибутов искомых файлов;

StringGrid1 – для вывода имен и длин найденных файлов;

Button1 – для запуска процесса поиска файлов.

Тогда обработчик события нажатия кнопки будет выглядеть следующим образом:

```
Procedure TForm1.Button1Click(Sender: TObject);
Var sr: TSearchRec;// Запись для параметров найденного файла
FileAttrs: Integer;  // Переменная для задания атрибута искомых файлов
begin
 StringGrid1.RowCount := 1;  // Задаем одну строку для вывода имен файлов
```

```
  if CheckBox1.Checked then      // Определяем атрибут только для чтения файла
    FileAttrs := faReadOnly
  Else FileAttrs := 0;
  if CheckBox2.Checked then
    FileAttrs := FileAttrs + faHidden; // Определяем атрибут для скрытых файлов
  if CheckBox3.Checked then
// Определяем атрибут для системных файлов
    FileAttrs:=FileAttrs+faSysFile;
  if CheckBox4.Checked then
    FileAttrs := FileAttrs + faVolumeID;// Атрибут метки тома
  if CheckBox5.Checked then
    FileAttrs := FileAttrs + faDirectory;// Атрибут для каталога
  if CheckBox6.Checked then
    FileAttrs := FileAttrs + faArchive;   // Атрибут для архивных файлов
  if CheckBox7.Checked then
    FileAttrs := FileAttrs + faAnyFile;   // Атрибут для любых файлов
  with StringGrid1 do
  begin
    RowCount := 0; ColCount:=2;          // Ищем первый файл
    if FindFirst(Edit1.Text, FileAttrs, sr) = 0 then
    begin
      repeat        // Открываем цикл поиска всех остальных файлов
        if (sr.Attr and FileAttrs) = sr.Attr then
        begin
// Увеличиваем счетчик найденных файлов на 1
        RowCount := RowCount + 1;
        Cells[0,RowCount-1] := sr.Name; // В первую колонку пишем имя файла
        Cells[1,RowCount-1] := IntToStr(sr.Size); // Во вторую – размер файла
        end;
      until FindNext(sr) <> 0;         // Ищем следующий файл
      FindClose(sr);     // Освобождаем память
    end;
  end;
end;
```

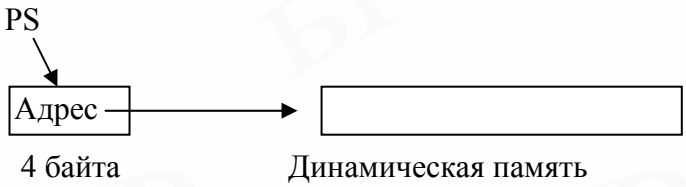
16. ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ ДАННЫХ

16.1. Динамические переменные

Динамические переменные представляют собой указатели на области памяти, где хранятся какие-то данные. Каждая из таких переменных занимает в памяти 4 байта. Например, в разделе объявления переменных можно определить указатель на строку следующим образом:

```
Var PS:^String;
```

Знак «^» – тильда, поставленный перед типом String, обозначает описание указателя на память, которая может быть выделена строке. Переменная PS будет занимать в памяти 4 байта, в них и будет храниться адрес начала выделенной памяти для строки. Здесь под адресом будем понимать не физический адрес памяти, а особым образом рассчитанный адрес внутри участка памяти, выделенного для динамических переменных. Этот участок памяти описывается отдельным дискриптором, как это принято в защищенном режиме работы процессора. Весь процесс выделения и освобождения динамической памяти контролируется системой Delphi. Графически это можно представить так:



Память под динамические переменные выделяется и освобождается во время выполнения программы по мере необходимости. Приведем основные процедуры и функции для работы с динамическими переменными:

Procedure New(var P:Pointer); – процедура выделения памяти под динамическую переменную P. Здесь тип Pointer определяет любой указатель на область памяти. Эта процедура выделяет блок памяти такой длины, как это было указано в типе динамической переменной P. В P записывается адрес начала блока выделенной памяти;

Procedure Dispose(var P:Pointer); – процедура освобождения динамической памяти, которая ранее была выделена процедурой New для переменной P;

Procedure GetMem(var P:Pointer; Size:Integer); – процедура выделения блока динамической памяти длиной Size байт и записи адреса начала этой памяти в переменную P. Эта процедура позволяет выделять блок памяти любой длины, независимо от типа указателя P;

Procedure FreeMem(var P:Pointer; Size:Integer); – процедура освобождения динамической памяти, которая ранее была выделена переменной P процедурой GetMem;

Function AllocMem(Size:Cardinal):Pointer; – функция возвращает указатель на блок выделенной динамической памяти длиной Size байт и обнуляет эту память, в отличие от процедуры GetMem. Освобождать эту память можно процедурой FreeMem;

Var AllocMemCount:Integer; – системная переменная, которая определяет число динамических блоков памяти для данного приложения. Эта переменная доступна только для чтения;

Var AllocMemSize:Integer; – системная переменная, которая определяет общий размер всех динамических блоков памяти для данного приложения. Такие переменные позволяют контролировать процесс выделения и освобождения динамической памяти;

Function SysGetMem(size:Integer):Pointer; – функция выделения системной динамической памяти. Эта память может быть доступна одновременно нескольким процессам;

Function SysFreeMem(var P:Pointer):Integer; – функция освобождения системной динамической памяти, которая ранее была выделена функцией SysGetMem;

Function Addr(X):Ponter; – функция получения адреса или указателя на любую переменную X. Эта функция эквивалентна оператору «@». Можно получить адрес переменной, записав:

P:=Addr(x); или

P:=@X;.

Процесс выделения и освобождения памяти довольно сложен и находится под контролем операционной системы. Память под динамические переменные выделяется в специальной области памяти Heap-области, или так называемой «куче». Для предотвращения сильной фрагментации этой области памяти необходимо придерживаться следующего правила: освобождать память следует в обратном порядке по сравнению с ее выделением. Другими словами, если мы выделили память под переменную P1, а затем под переменную P2, то освобождать следует сначала память, выделенную под переменную P2, и только потом освобождать память, выделенную под переменную P1.

Для доступа к содержимому динамической памяти следует после имени указателя ставить символ «^» – тильда.

Рассмотрим пример программы умножения матрицы на вектор, но при этом матрицу и вектор расположим в динамической области памяти. Размер матрицы будем задавать в компоненте Edit1. Матрицу A будем выводить в компоненте StringGrid1, вектор X – в компоненте StringGrid2, а результат умножения поместим в компонент StringGrid3. Тогда обработчик события нажатия клавиши «Вычислить» примет вид

```
Procedure Button1Click(Sender:Tobject);
Type Ta=array[1..1] of Extended;
    // Тип одномерного массива вещественного типа
    Tpa=^Ta;  // Тип указателя на одномерный массив вещественного типа
    Tapa=array[1..1] of Tpa;       // Тип массива указателей
    Tpapa=^Tapa;    // Тип указателя на массив указателей
Var      Px,Py:Tpa; // Указатели на массивы X и Y
    Pa:Tpapa;  // Указатель на массив указателей на одномерные массивы
    // вещественных чисел.
    i,j,n:Integer;       // Внутренние переменные
Procedure MulAX(n:integer, var Px,Py:Tpa; var Pa:Tpapa);
    // Процедура умножения матрицы A на вектор X, результат в векторе Y
Var i,j:Integer; s:extended;       // Внутренние переменные
Begin
  For i:=1 to n do Begin      // Цикл по строкам
    S:=0;
    For j:=1 to n do          // Цикл по столбцам
```

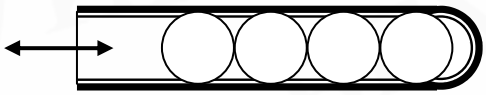
```
    s:=s+Pa^[i]^[j]*Px^[j];
    Py^[i]:=s;    // Запоминаем результат
    End;
  End;
Begin     // Начало раздела действий процедуры Button1Click
  n:=StrToInt(Edit1.Text); // Определяем размерность массивов
  with StringGrid1 do Begin // Подготовка визуальных компонентов
    ColCount:=n+1;
    RowCount:=n+1;
    FixedCols:=1;
    FixedRows:=1;
    End;
  with StringGrid2 do Begin
    ColCount:=2;
    RowCount:=n;
    FixedCols:=1;
    FixedRows:=0;
    End;
  with StringGrid3 do Begin
    ColCount:=2;
    RowCount:=n;
    FixedCols:=1;
    FixedRows:=0;
    End;
  GetMem(Px,n*Sizeof(extended));      // Выделение памяти для массива X
  GetMem(Py,n*Sizeof(extended));      // Выделение памяти для массива Y
  GetMem(Pa,n*Sizeof(Pointer));
     // Выделение памяти для указателей на строки массива A
  For i:=1 to n do Begin
    GetMem(Pa^[i],n*Sizeof(Extended));
     // Выделение памяти под строки массива A
    Px^[i]:=Random;  // Задание случайным образом вектора X
    StringGrid2.Sells[0,i-1]:=Floattostr(i);      // Оформление таблиц
    StringGrid2.Sells[1,i]:=Floattostr(Px^[i]);
    StringGrid1.Sells[0,i-1]:=inttostr(i);
    StringGrid1.Sells[i-1,0]:=inttostr(i);
    StringGrid3.Sells[i-1,0]:=inttostr(i);
    For j:=1 to n do Begin
      Pa^[i]^[j]:=Random;  // Задание случайным образом значений массива A
      StringGrid1.Sells[j,i]:=Floattostr(Pa^[i]^[j]);
      End;
    end;
  MulAX(n,Px,Py,Pa); // Вызов процедуры умножения матрицы на вектор
  For i:=1 to n do Begin
    StringGrid3.Sells[1,i-1]:=Floattostr(Py^[i]); // Отображение результатов
```

```
    FreeMem(Pa^[n-i+1],n*sizeof(Extended);
     // Освобождение в обратном порядке памяти для строк матрицы A
    End;
  FreeMem(Pa,n*sizeof(Pointer));
    // Освобождение массива указателей на строки массива A
  FreeMem(Py,n*sizeof(extended)); // Освобождение памяти для вектора Y
  FreeMem(Px,n*sizeof(Extended)); // Освобождение памяти для вектора X
  End;
```

В данном примере размерность массивов в принципе может быть любая. Динамической памяти под массивы выделяется ровно столько, сколько нужно для заданной размерности n.

16.2. Работа со стеком

Стек – это структура данных, организованная по принципу «первый вошел – последний вышел». Образно это можно представить как запаянную с одной стороны трубку, в которую закатываются шарики:



Первый шарик всегда будет доставаться из трубки последним. Элементы в стек можно добавлять или извлекать только через его вершину. Программно стек реализуется в виде однонаправленного списка с одной точкой входа (вершиной стека).

Для работы со стеком можно определить следующий тип:

```
Type Tps=^Ts;   // Указатель на запись
     Ts=Record;// Сама запись элемента стека
          inf:TInf;    // Информационная часть элемента стека,
                       // например Tinf=String
          Ps: Tps;     // Указатель на предыдущий элемент стека
          End;
```

Графически стек с тремя элементами можно представить следующим образом:

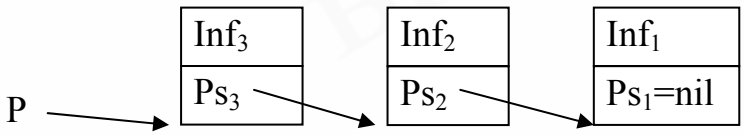


Рис.16.1. Структура стека

Здесь P – указатель на вершину стека. Самый первый элемент стека не имеет впереди себя других элементов, поэтому в указатель Ps записывается пустой указатель – Nil. Следует помнить, что процедуры освобождения памяти Dipose и FreeMem не полагают освобождаемый указатель константе Nil.

Рассмотрим пример программы, которая формирует стек из случайных целых чисел, затем его упорядочивает методом «пузырька», выводит на экран и

одновременно освобождает стек. Приведем полный текст модуля Unit, соответствующего форме Form1, на которой размещены два текстовых редактора Memo и две кнопки: первая – для получения стека из случайных величин, а вторая – для упорядочения элементов стека в порядке возрастания его значений:

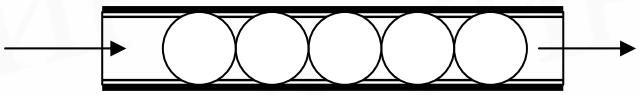
```
unit Ustack;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
Type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Memo2: TMemo;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
type Tps=^Ts; // Определяет тип элемента стека
     Ts=record
        inf:word; // Информационная часть элемента стека
        ps:Tps;    // Указатель на предыдущий элемент стека
        end;
var p:Tps;       // Указатель на вершину стека
const n=10;      // Размерность стека
implementation
{$R *.dfm}
// Процедура добавления нового элемента в стек
Procedure AddStack(var P:Tps; n:word);
var pt:Tps;
Begin
  new(pt);         // Выделяем память для нового элемента стека
  pt^.inf:=n;      // Записываем новое число в элемент стека
  pt^.ps:=p;       // Запоминаем указатель на предыдущий элемент стека
  p:=pt;   // Возвращаем этот новый указатель на вершину стека
  end;
// Процедура извлечения числа из стека с освобождением памяти
Procedure FromStack(var p:Tps;var n:word);
```

```
var pt:Tps;
Begin
  if p<>nil then Begin
    pt:=p;          // Запоминаем старое значение вершины стека
    n:=p^.inf;      // Извлекаем число из текущего элемента стека
    p:=p^.ps;       // Устанавливаем новый указатель на вершину стека
    dispose(pt);  // Освобождаем память старого элемента стека
    end else n:=0;
  end;
// Процедура однократного прохода по стеку и замены значений соседних
// элементов стека, если их значения в направлении от вершины ко дну стека
// не возрастают
Procedure exchange(var p:Tps; var n:word);
var nt:word;
Begin
  if p^.ps<>nil then Begin   // Проверка наличия следующего элемента стека
    if p^.inf>p^.ps^.inf then Begin        // Проверка на возрастание значений
стека
      nt:=p^.inf;          // Запоминаем значение числа в вершине стека
      p^.inf:=p^.ps^.inf;       // Переставляем числа соседних элементов стека
      p^.ps^.inf:=nt;
      inc(n);                        // Увеличиваем счетчик перестановок на единицу
      end;
    exchange(p^.ps,n);        // Рекурсивно вновь вызываем процедуру переста-
новок
    end;
  end;
//  Процедура упорядочения элементов стека методом «пузырька»
Procedure SortStack(var p:Tps);
var n:word;
Begin
  Repeat // Открытие цикла упорядочений
   n:=0;// Счетчик числа перестановок за один проход полагаем равным нулю
    exchange(p,n); // Вызываем процедуру однопроходного упорядочения
// Продолжаем цикл, пока не будет ни одной перестановки
    until n=0;
  end;
// Процедура формирования начального стека
procedure TForm1.Button1Click(Sender: TObject);
var i,j:integer;
begin
  randomize;    // Инициируем датчик случайных чисел
  memo1.Clear;// Очищаем текстовый редактор Memo1
  p:=nil;          // Указатель на вершину стека полагаем равным константе nil
  for i:=1 to n do Begin       // Открываем цикл записей в стек
```

```
    j:=random(20);     // Получаем случайное число
    addstack(p,j);     // Записываем его в стек
    memo1.lines.Add(inttostr(j)); // Выводим его значение в Memo1
    end;
  end;
// Процедура упорядочения элементов стека и вывод результата
procedure TForm1.Button2Click(Sender: TObject);
var k:word;
begin
  memo2.Clear;       // Очистка Memo2
  SortStack(p);      // Сортировка стека
  while p<>nil do Begin     // Проход по стеку
    Fromstack(p,k);  // Извлечение элемента из стека и освобождение памяти
    Memo2.Lines.Add(inttostr(k));       // Запись элемента стека в Memo2
    end;
  end;
end.
```

16.3. Работа со списками или очередями

Очередь – это структура данных, работающая по принципу: первый вошел, первый вышел. Образно такую структуру можно представить в виде открытой с двух сторон трубки – с одной стороны мы можем закатывать шарики, а из другого конца их извлекать:



Списки могут быть односвязными, когда в элементе списка есть указатель на предыдущий элемент списка, или двухсвязными, когда в одном элементе есть указатель на предыдущий элемент списка и указатель на последующий элемент списка.

Для односвязного списка можно определить тип записи следующим образом:

```
Type TpL=^TL;
TL=Record
      Inf:Tinf;
      PL:TpL;
      End;
```

Он в принципе ничем не отличается от записи стека.

Для двухсвязного списка тип записи уже будет следующим:

```
Type TpL=^TL;
TL=Record
      Inf:Tinf;
      PL1,PL2:TpL;
      End;
```

Здесь PL1 указывает на предшествующий элемент очереди, а PL2 – на стоящий за ним элемент очереди. Графически это можно представить следующим образом:

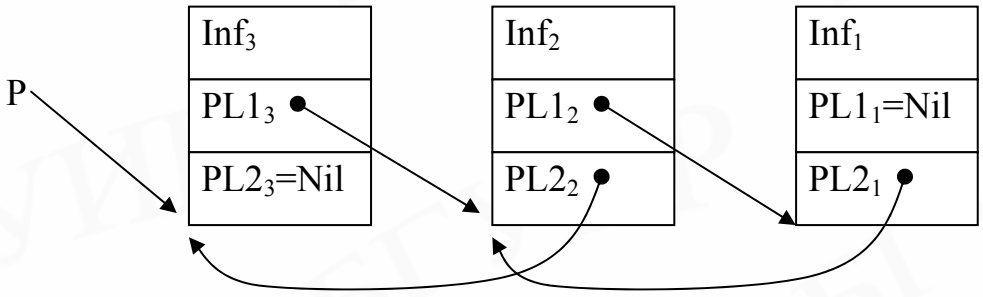


Рис.16.2. Структура очереди

Двухсвязный список очень легко позволяет продвигаться как вперед, так и назад по списку. Его легче редактировать. Рассмотрим несколько процедур работы с двухсвязными списками.

Процедура добавления нового элемента в виде строки в конец очереди:

```
Procedure Addl(var p:TPL; s:String);
Var Pt:TPL;
Begin
  New(Pt);       // Выделяем память под новый элемент
  Pt^.inf:=s;    // Записываем значение строки в новый элемент очереди
  Pt^.PL2:=nil;// За последним элементом нет других элементов очереди
  If p=nil then Pt^.PL1:=nil else Begin // Если это первый элемент очереди,
    Pt^.PL1:=p; // то запоминаем указатель на предыдущий элемент очереди
    P^.PL2:=Pt; // В предыдущем элементе записываем указатель на новый
    End;
  P:=Pt;         // Возвращаем указатель на новый конец очереди
  End;
```

Приведем теперь пример подпрограммы, которая удаляет из очереди элемент с заданным значением строки:

```
Procedure Dell(var p:TPL;var s:String);
// При выходе строка S должна быть пустой, если был удален элемент очереди
Var Pt:TPL;
Begin
  Pt:=p;  // Запоминаем конец очереди
  While p<> nil do Begin    // Открываем цикл прохода по очереди
   If s=p^.inf then Begin    // Нашли элемент для удаления
     If p^.PL2=nil then Begin // Удаляем первый элемент с конца очереди
       If p^.PL1<>nil then begin   // Впереди есть элемент
// Следующий элемент очереди становится последним
         p^.PL1^.PL2:=nil;
         p:=p^.PL1;     // Запоминаем указатель на новый последний элемент
         end else p:=nil; // Удаляем единственный элемент очереди
        dispose(pt);       // Освобождаем память
        s:='';             // Очищаем строку поиска
```

```
      End else If p^.PL1=nil then Begin// Удалять нужно последний элемент
      P^.PL2^.PL1:=nil;
// Второй элемент очереди теперь не должен иметь впереди стоящего
      Dispose(P);       // Освобождаем память
      P:=pt;            // Возвращаем указатель на конец очереди
      S:='';            // Строку поиска полагаем равной пустой строке
      End else Begin  // Удалять нужно средний элемент очереди
      p^.PL2^.PL1:=p^.PL1; // В сзади стоящем элементе очереди записыва–
    // ем указатель на впереди стоящий элемент
      p^.PL1^.PL2:=p^.PL2;      // Во впереди стоящем элементе записываем
    // указатель на новый, сзади стоящий элемент очереди
      dispose(p);       // Освобождаем память
      s:='';            // Очищаем строку поиска
      p:=pt;            // Возвращаем указатель на конец очереди
      end;
     End;               // Закончили блок удаления элемента очереди
// Выходим из процедуры, если был удален элемент очереди
   If s:='' then exit;
   P:=p^.PL1; // Переходим на просмотр впереди стоящего элемента очереди
   End;          // Конец цикла просмотра элементов очереди
 End;
```

16.4. Работа с деревьями

Дерево – это иерархическая информационная структура, состоящая из узлов, в каждом из которых может быть несколько указателей на дочерние узлы следующего по иерархии уровня.

Прародитель всех узлов называется корнем дерева. Узлы, не имеющие дочерних узлов, называются листьями. Промежуточные узлы называются ветвями. Степень дерева – максимальный порядок его узлов. Глубина дерева – это максимальная глубина его узлов.

Древовидное размещение списка данных (*abcdefgkl*) можно изобразить, например, следующим образом:

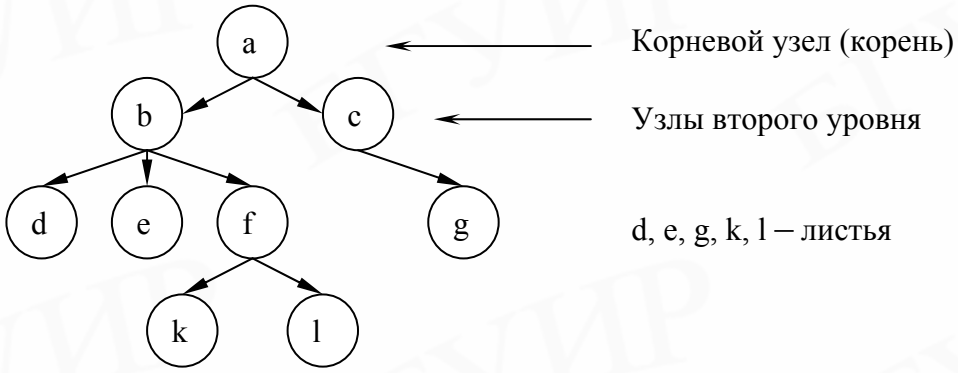


Рис.16.3. Структура дерева

Дерево, представленное выше, имеет степень 3 (троичное), глубину 4, корневой узел a.

Для работы, например, с двоичным деревом можно определить следующие типы:

```
Type Tpz=^Tz;        // Указатель на запись
   Tz=Record         // Запись узла дерева
        Inf:string;  // Информационная часть узла дерева
        Kl:integer;  // Уникальный ключ узла дерева
        P1,P2:Tpz;   // Указатели на дочерние узлы дерева
        End;
```

Для работы с деревьями в основном используют рекурсивные процедуры – процедуры, которые вызывают сами себя. Они позволяют проходить по всему дереву только по одному вызову таких процедур.

Прежде чем переходить к рассмотрению примера работы с деревьями, дадим описание некоторых свойств компонента TTreeView и класса TTreeNode, которые будут использованы в примере.

Компонент TTreeView предназначен для отображения ветвящихся иерархических структур в виде горизонтальных деревьев, например каталогов файловой системы дисков. Основным свойством этого компонента является Items, которое представляет собой массив элементов типа TTreeNode, каждый из которых описывает один узел дерева. Всего в Items – count узлов. Рассмотрим некоторые методы класса TTreeNodes:

Function AddFirst(Node:TTreeNode; const S:String):TTreeNode; – этот метод создает первый дочерний узел у узла Node. Если Node=Nil, то создается корневой узел. S – это строка содержания узла. Функция возвращает указатель на созданный узел;

Function AddChild(Node:TTreeNode; const S:String):TTreeNode; – эта функция добавляет очередной дочерний узел к узлу Node;

Procedure Clear; – этот метод очищает список узлов, описанных в свойстве Items;

Function DisplayRect(TextOnly: Boolean): TRect; – позволяет получить прямоугольник на канве компонента TTreeView, в котором описан текст узла TTreeNode, если аргумент TextOnly равен «True», иначе – весь прямоугольник узла.

В качестве примера рассмотрим проект, который создает дерево, отображает его в компоненте TTreeView, находит узел по ключу и может очистить дерево. На форме будут располагаться: Button1 – для создания дерева, Button2 – для удаления дерева, Button3 – для поиска узла по ключу и TTreeView – для графического отображения дерева:

```
unit U1; // Текст модуля
interface
// подключаемые модули
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids, ComCtrls;
Type      // Класс формы
  TForm1 = class(TForm)
    Button1: TButton;
```

```
    TreeView1: TTreeView;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
Type Tpz=^Tz;        // Тип указателя на запись
       Tz=Record     // Определение типа записи узла дерева
       Inf:string;   // Информационная часть узла
       kl:integer;   // Ключ узла дерева
       pl,pr:tpz;    // Указатели на исходящие ветви узла дерева
       end;
var  Form1: TForm1;
       Proot:Tpz;    // Указатель на корень дерева
       Level:integer; // Уровень узла дерева
       Kluch:integer;// Ключ узла дерева
       skluch:string; // Строка ключа узла дерева при поиске
       sitem:string;  // Строка информационной части узла при поиске
implementation
{$R *.dfm}

// Рекурсивная процедура создания и отображения дерева в TreeView
Procedure CreateTree(var p,ppar:Tpz);
// p – Указатель на текущий узел дерева
// ppar – Указатель на родительский узел дерева
var pt1:Tpz;   // Временный указатель на узел дерева
Begin with form1 do Begin
  inc(level);
// При входе в процедуру увеличиваем уровень узла
  if p=nil then Begin
// Если p=nil, то можно создать новый узел
// С помощью типового диалога решаем, создавать ли узел дерева
    case messagedlg('Создать новый элемент дерева уровня'
    +' '+inttostr(level)+' с ключом '+inttostr(Kluch+1),
      mtConfirmation,[mbok,mbno],0) of
      mrok:Begin      // Создаем новый узел
        new(pt1);     // Выделяем узлу память
// С помощью типового диалога вводим информацию в узел
        pt1^.Inf:=inputbox('Ввод инф. части дерева',
```

```
        'Введите текст элемента дерева','');
        inc(kluch);      // Увеличиваем номер узла на единицу
        pt1^.kl:=kluch;        // Запоминаем этот номер как ключ
        pt1^.pl:=nil;
// Записываем nil для дочернего левого узла
        pt1^.pr:=nil;
// Записываем nil для дочернего правого узла
// Если kluch=1, т.е. это – корень дерева, то используем метод
// AddFirst для отображения этого узла, иначе – метод AddChild
        if kluch=1 then Treeview1.Items.AddFirst(nil,Pt1^.inf+
        ' '+inttostr(kluch))
        else Treeview1.Items.Addchild(treeview1.Items[ppar^.kl-1],
        Pt1^.inf+' '+inttostr(kluch));
        Treeview1.FullExpand;
// Раскрываем все дерево
        createtree(pt1.pl,pt1);        // Создаем левую ветвь дерева
        createtree(pt1.pr,pt1);        // Создаем правую ветвь дерева
        p:=pt1;
        // Программа будет возвращать указатель на новый узел дерева
        end;
    mrno:Begin // В случае ответа «нет», ничего не делаем
       end;
     end;
   end;
  dec(level);
// При выходе из подпрограммы уменьшаем текущий уровень узла дерева
  end;
end;

// Обработчик нажатия кнопки «Создать дерево»
procedure TForm1.Button1Click(Sender: TObject);
begin
    createtree(proot,proot);
    end;

// Отработчик события создания формы
procedure TForm1.FormCreate(Sender: TObject);
begin
    Proot:=nil; // Указатель на корень дерева полагаем равным константе nil
    Level:=0;    // Текущий уровень узла дерева зануляем
    Kluch:=0;   // Текущий номер узла дерева зануляем
    end;

// Процедура освобождения памяти для дерева
Procedure DeleteTree(var p:Tpz);
```

```
// Здесь P – указатель на удаляемую ветвь дерева
Begin
    If p=nil then exit;
    if ((p^.pl=nil)and(p^.pr=nil))then Begin
// Если у узла нет дочерних узлов, то его можно удалить
        Dispose(p);
        p:=nil;
        exit;
        end;
// Если у узла есть дочерние узлы, то следует сначала их удалить
// с помощью программы DeleteTree
    if p^.pl<>nil then deleteTree(p^.pl);
    if p^.pr<>nil then deleteTree(p^.pr);
    end;

// Обработчик нажания кнопки «Удалить дерево»
procedure TForm1.Button2Click(Sender: TObject);
begin
// Удаляем дерево с корневого узла
    DeleteTree(proot);
    proot:=nil;
    level:=0;
    kluch:=0;
// Очищаем компонент TreeView
    TreeView1.Items.Clear;
    end;

// Процедура поиска в дереве узла с заданным ключом
Procedure SeekTree(var p:Tpz;s:string);
// Здесь P – указатель на узел дерева
// S – искомый ключ узла дерева
var i:integer;
Begin
 if p=nil then exit;
 if s=inttostr(p^.kl) then Begin
   skluch:=s;
   sitem:=p^.inf;
   end;
 if p^.pr<>nil then seektree(p^.pr,s);
 if p^.pl<>nil then seektree(p^.pl,s);
 end;

// Обработчик нажатия кнопки «Поиск узла»
procedure TForm1.Button3Click(Sender: TObject);
var s:String;
```

```
  rect1:Trect;          // Прямоугольник для узла дерева
  Color1:Tcolor;        // Цвет шрифта узла дерева
begin
    skluch:='';
    sitem:='';
    s:=Inputbox('Поиск узла',
    'Введите ключ искомого узла','');
    if length(s)>0 then Begin
      SeekTree(Proot,s);  // Ищем узел с ключом s
      if length(skluch)>0 then with treeview1 do Begin
    // Отметим красным цветом найденный узел
        for i:=0 to items.count-1 do Begin
          if pos(s,items[i].text)>0 then Begin
            rect1:=items[i].DisplayRect(true);
            color1:=canvas.font.color;
            canvas.font.Color:=clred;
            canvas.FillRect(rect1);
            canvas.
            TextOut(rect1.Left,rect1.Top,items[i].text);
            canvas.Font.Color:=color1;
            break;
            end;
          end
        Showmessage('Найден узел с ключом '
        +skluch+' и полем inf='+sitem)
        else Showmessage('Узла с ключом '+skluch+
        ' нет в дереве!');
        end;
    end;
end.
```

17. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

17.1. Объекты и классы

Класс – это тип объекта. Он характеризует объект вместе с его свойствами и правилами поведения. Объект есть экземпляр класса. В Delphi все объекты динамические. Для выделения памяти под объект используется специальный метод данного класса Constructor Create – это функция, которая возвращает адрес на вновь созданный объект. Освобождение памяти, выделенной ранее объекту, осуществляется методом Dectructor Destroy. Для объектов и классов сделано одно исключение из общих правил работы на Паскале с динамическими переменными – для них не используется символ ^ (тильда) для указателей и содержимого указываемой памяти. Переменные, описанные в классе, называют полями. Любая процедура или функция, описанная в классе, является уже методом. При вызове любого метода ему неявным образом передается первым параметром Self, который является указателем на объект, который вызвал данный метод. Если процедура описана вне класса, но за ее описанием следуют слова of object, то это тоже будет метод. Классы могут быть описаны или в интерфейсной части модуля Unit, или в самом начале секции реализации Implementation. Не допускается их описание внутри процедур и функций. Если перед описанием метода стоит ключевое слово class, то это классовый метод и его можно вызывать даже в том случае, если объекту еще не была выделена память. Типичный пример класса:

```
Type TmyObject=class(Tobject)
  x,y:integer;
  Constructor Create;
  Destructor Destroy;virtual;
  Procedure Show;
  End;
```

В скобках после ключевого слова class указывается наследуемый класс. Delphi поддерживает только единичное наследование классов. Объект Tobject является прародителем всех классов и не наследует никаких других классов.

17.2. Области видимости класса

При описании класса можно задать разные по возможностям области видимости полей и методов класса. Существуют четыре области видимости класса:

1. Private – личная, внутренняя область класса. Поля и методы, описанные в этой области, доступны только внутри модуля Unit, где описан данный класс. Для всех других модулей, которые подсоединяют данный модуль и наследуют этот класс, они недоступны.

2. Protected – защищенная область. Поля и методы этой области доступны только внутри классов, наследующих данный класс.

3. Public – общедоступная область. Поля и методы этой области не имеют ограничений на видимость.

4. Published – область публикаций. Поля и методы этой области имеют такую же видимость, как для области Public, но они еще видны инспектору объектов на этапе разработки программы. В дочерних классах можно переносить методы и свойства из области Protected в область Published и обратно.

17.3. Свойства (Property) и инкапсуляция

Объектно-ориентированное программирование (ООП) основано на трех принципах – инкапсуляция, наследование и полиморфизм. Классическое ООП утверждает, что чтение и обновление полей должно производиться только специальными методами и не допускается прямое обращение к полям класса. Это правило и называется инкапсуляцией, а такие поля – свойствами. Свойство определяется полем и двумя методами, которые осуществляют чтение и запись заданных значений в поле. Пример определения свойства:

```
Type TmyObject=class(Tobject)
    Privete
        FmyField:String;
    Protected
        Procedure SetMyField(Value:String);
    Published
        Property MyProp:String Read FmyField
            Write SetMyField
            Default 'Начальное значение';
    End;
```

Здесь в классе TmyObject определено свойство MyProp строкового типа. В качестве метода чтения выступает само значение строки, а запись осуществляется методом SetMyField. Само поле FmyField определено в области Privete, и поэтому к нему нет прямого доступа из других модулей. Метод чтения этого поля находится в защищенной области, а свойство MyProp – в области публикаций, и оно доступно инспектору объектов во время проектирования программы.

17.4. Методы, наследование и полиморфизм

Наследование означает, что при создании нового класса он наследует все поля, свойства и методы, определенные в родительском классе. В новом классе только добавляются новые поля, методы и свойства. Унаследованные от предка поля и методы доступны в дочернем классе, но с учетом областей видимости. Если имеет место совпадение имен, то говорят, что они перекрываются. В Delphi допускается только последовательное единичное наследование классов.

Методы подразделяются на четыре группы:

статические (Static),
виртуальные (Virtual),
динамические (Dynamic),
абстрактные (Abstract).

Адрес вызова статического метода определяется на этапе трансляции проекта, и вызов этих методов осуществляется быстрее всех остальных методов. Такие методы можно без ограничений перекрывать и даже менять список передаваемых параметров. По умолчанию методы объектов являются статическими.

Адреса виртуальных и динамических методов определяются во время выполнения программы и находятся в специальных таблицах: таблице виртуальных методов (VMT) и таблице динамических методов (DMT). В таблицу VMT включаются адреса всех определенных в данном классе виртуальных методов и всех наследуемых методов. В таблицу DMT включаются адреса динамических методов, определенных только в данном классе. Поэтому виртуальные методы вызываются быстрее динамических, но размеры таблиц VMT существенно больше таблиц DMT. Динамические методы позволяют экономить память, но их вызов осуществляется медленнее всех остальных методов, так как приходится для поиска адреса метода проходить по всем таблицам DMT родительских классов, пока не будет найден нужный динамический метод.

Для перекрытия виртуальных и динамических методов используется ключевое слово Override. Список параметров перекрываемых виртуальных и динамических методов не должен отличаться от списка параметров этих методов в родительском классе.

Абстрактные методы определяют только интерфейсную часть метода, такие методы нельзя использовать без перекрытия в дочерних классах, где должна находиться и реализация такого метода.

Рассмотрим следующий пример:

```
Type Tpoint=Class(Tobject)
                Constructor Create;
                Destructor Destroy;virtual;
                X,y:Integer;
                C:Tcolor;
                Procedure Show;virtual;
                End;
        Tcircle=Class(Tpoint)
                Constructor Create;
                Destructor Destroy;override;
                R:Integer;
                Procedure Show;override;
                End;
……….
Var   Point1:Tpoint;
      Circle1:Tcircle;
```

```
Begin
        Point1:=Tpoint.Create;
        Circle1:=Tcircle.Create;
        With Point1 do Begin
                X:=100;y:=50; C:=clRed;
                Show;
                End;
        With Circle1 do Begin
                X:=200;Y:=100:C:=clBlue;
                R:=50;
                Show;
                End;
……….
```

В данном примере определены два класса: Tpoint и Tcircle. Класс Tpoint наследует класс Tobject. В классе Tpoint определены поля X, Y, которые задают точку на экране дисплея, и C – цвет этой точки. В нем также описаны методы по выделению и освобождению памяти под объект и виртуальный метод Show– рисования точки на экране. Класс Tcircle наследует класс Tpoint и все его поля и методы. В нем дополнительно описаны поле R (радиус) и метод Show, который перекрывает аналогичный метод класса Tpoint. Метод Show класса Tcircle рисует теперь окружность. В результате два метода Show рисуют разные картинки в зависимости от того, какому классу они принадлежат. Это и называется полиморфизмом объектов.

17.5. События (Events)

События в Delphi – это свойства процедурного типа, предназначенные для создания пользовательской реакции на те или иные входные воздействия. Пример объявления события:

```
Property OnMyEvent:TmyEvent  Read  FOnMyEvent
                             Write  FonMyEvent;
```

Присвоить такому свойству значение – это значит указать адрес метода, который будет вызываться в момент наступления события. Такие методы называются обработчиками событий. События имеют разные типы, но общим для всех является параметр Sender – указатель на объект – источник события. Самый простой тип события это тип

```
TnotifyEvent=procedure(Sender:Tobject) of Object;
```

Здесь «of Object» означает, что данный тип определяет именно метод, а не обычную процедуру. Приставка On в имени свойства означает, что данное свойство является событием, хотя не каждое событие может иметь такую приставку. Для определения события необходимо в разделе Private объявить поле указателя на метод. В разделе Protected нужно объявить методы чтения и записи адреса обработчика события в это поле. Затем в разделе Published объявить само событие как свойство процедурного типа. В инспекторе объектов есть две страницы: страница свойств (Properties) и страница событий (Events). Двойной щелчок левой клавишей мыши по событию приводит к

появлению обрамления обработчика события в тексте программного модуля Unit.

Общими для всех компонентов являются события (наследники класса TControl):

OnClick – нажатие левой клавиши мыши,

OnDblClick – двойной щелчок левой клавиши мыши,

OnMouseDown – нажатие любой клавиши мыши,

OnMouseMove – перемещение курсора мыши по компоненту,

OnMouseUp – отжатие кнопки мыши.

Общими для оконных элементов управления являются события (наследники класса TWinControl):

OnEnter – перемещение фокуса ввода на компонент, который становится активным,

OnExit – потеря активности компонентом,

OnKeyDown – нажатие клавиши или комбинации клавиш,

OnKeyPress – нажатие каждой одиночной клавиши,

On KeyUp – отпускание клавиши.

18. ВЫДЕЛЕНИЕ ПАМЯТИ ПОД ОБЪЕКТ И ПРАРОДИТЕЛЬ ВСЕХ КЛАССОВ – TOBJECT

18.1. Выделение памяти под объект

Рассмотрим более детально, что собой представляет объект. Допустим, что мы определили класс следующим образом:

```
Type TMyClass=Class
        F1:Integer;
        F2:String[20];
        Procedure Sm1;
        Procedure Vm1;virtual;
        Procedure Vm2;virtual;
        Procedure Dm1;dynamic;
        Procedure Dm2;dynamic;
        End;
```

В обработчике какого-то события в описании переменных определим указатель на объект класса TMyObject:

Var Obj1:TMyClass;

В разделе действий мы создадим экземпляр этого объекта как

Obj1:=TMyClass.Create;

На рис.18.1 показано, как будет выглядеть внутренняя структура этого объекта.

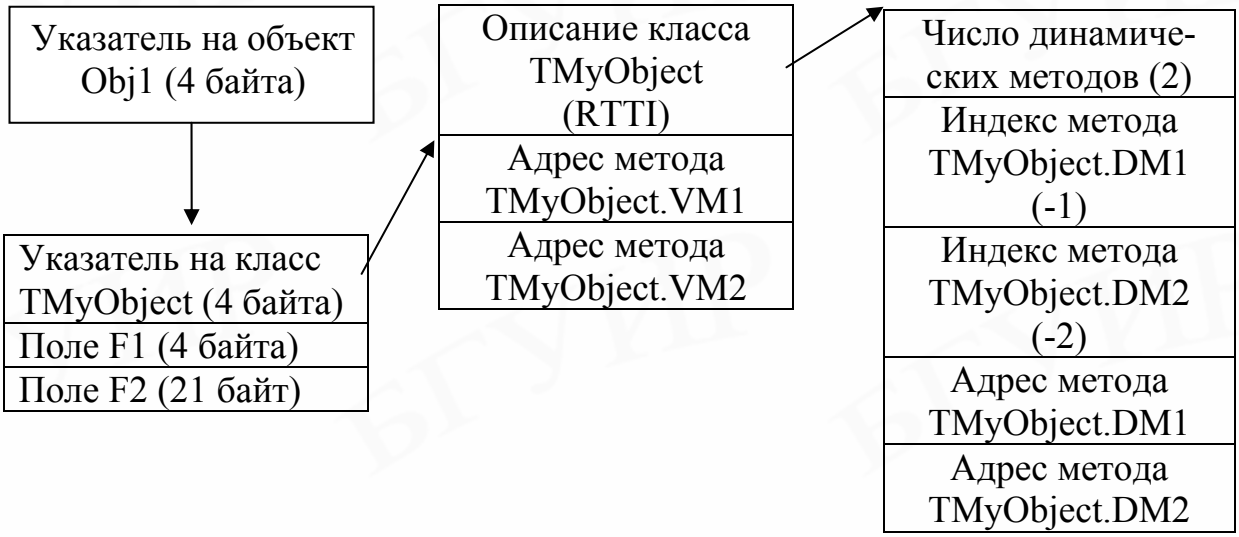


Рис.18.1. Структура связи объекта с описанием класса

Первое поле каждого экземпляра объекта содержит указатель на его класс. Затем идут поля объекта в том порядке, как они описаны в классе, и всех наследуемых классах. Класс как структура состоит из двух частей. Указатель на класс определяет начало таблицы указателей на виртуальные методы класса. В эту таблицу включаются все наследуемые и вновь описанные в данном классе виртуальные методы. Перед таблицей виртуальных методов с отрицательным смещением располагается информация о классе, так называемая информация о типе времени выполнения (RunTime Type Information – RTTI). Описание этой информации можно найти в модуле System.pas (см. константы с приставкой vmt*). От версии к версии Delphi объем этой информации все возрастает и, например, для Delphi 6 составляет 76 байт. В этой области находятся 19 указателей на различные таблицы и параметры класса. Так, например, там есть указатели на классовые методы, которые можно вызывать, даже не выделяя объекту память, есть указатели на таблицы динамических методов, таблицу реализуемых интерфейсов (для совместимости с объектами COM) и т.д. Но к этой таблице напрямую нужно обращаться только в крайнем случае и имея большой опыт программирования на Delphi. Все основные параметры класса можно получать, используя методы прародителя всех классов класса TObject.

18.2. Описание класса TObject

Все описание класса TObject полностью наследуются остальными классами. Рассмотрим более подробно этот класс:

```
TObject = class
  constructor Create;                  // Функция создания объекта
  destructor Destroy; virtual;         // Процедура разрушения объекта
  procedure Free;        // Процедура освобождения памяти с предварительной
      // проверкой – а была ли ранее выделена память под объект?
  class function InitInstance(Instance: Pointer): TObject; // Классовая функция,
      // которая создает новый экземпляр объекта с занулением его полей и
```

```
    // инициализации таблицы виртуальных методов. Эту функцию явно
    // не вызывают, вместо нее для создания нового объекта вызывается
    // метод NewInstance в конструкторе Create
class function NewInstance: TObject; virtual; // Классовая функция для
    // создания нового объекта
procedure CleanupInstance;       // Процедура освобождения памяти из-под
    // длинных строк, переменных типа Variant и интерфейсов. Прямо ее
    // не вызывают, она автоматически вызывается при вызове деструктора
procedure FreeInstance; virtual; // Процедура освобождения памяти,
    // выделенной ранее объекту методом NewInstance, она вызывается
    // автоматически при вызове деструктора
function ClassType: TClass;      // Функция, возвращающая ссылку на класс
    // Используется обычно в операторах is или as
class function ClassName: ShortString;        // Классовая функция,
    // возвращающая имя класса
class function ClassNameIs(const Name: string): Boolean;      // Классовая
    // функция, проверяющая принадлежность объекта к классу с именем
    // Name
class function ClassParent: TClass;     // Классовая функция, возвращающая
    // указатель на родительский класс
class function ClassInfo: Pointer;       // Классовая функция, которая
    // возвращает указатель на описание класса – область RTTI
class function InstanceSize: Longint;   // Классовая функция, которая
    // возвращает длину объекта в байтах
class function InheritsFrom(AClass: TClass): Boolean;   // Классовая
    // функция для проверки условия – является ли данный класс
    // наследником класса AClass
class function MethodAddress(const Name: ShortString): Pointer;
    // Классовая функция, которая возвращает указатель на метод с именем
    // Name
class function MethodName(Address: Pointer): ShortString;    // Классовая
    // функция, которая возвращает имя метода по его адресу
function FieldAddress(const Name: ShortString): Pointer;      // Классовая
    // функция, которая возвращает указатель на поле с именем Name
function GetInterface(const IID: TGUID; out Obj): Boolean;   // Функция
    // получения указателя на интерфейс с глобальным идентификатором IID.
    // Если такой интерфейс присутствует в системе, то эта функция
    // возвращает истину
class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
    // Классовая функция получения указателя на интерфейс с глобальным
    // идентификатором IID
class function GetInterfaceTable: PInterfaceTable;       // Классовая
    // функция получения указателя на начало таблицы интерфейсов
function SafeCallException(ExceptObject: TObject;
     ExceptAddr: Pointer): HResult; virtual; // Функция получения указателя
```

```
    // на процедуру обработки исключительной ситуации, используя SafeCall
    // соглашение при вызове подпрограмм
  procedure AfterConstruction; virtual;  // Метод, который будет вызываться
    // после создания объекта
  procedure BeforeDestruction; virtual;  // Метод, который будет вызываться
    // перед разрушениям объекта
  procedure Dispatch(var Message); virtual;    // Процедура диспетчеризации
    // вызовов динамических методов с помощью Windows сообщений
  procedure DefaultHandler(var Message); virtual;  // Обработчик Windows
    // сообщений по умолчанию
end;
```

Как видно из этого описания, большинство методов этого класса – классовые методы, и их можно вызывать, даже не создав ни одного объекта, так как информация о классе создается еще на этапе проектирования проекта.

18.3. Операторы приведения типов классов

В обработчики событий обычно первым параметром передается источник события Sender с типом TObject. На самом деле Sender может соответствовать любому объекту, например форме или другому компоненту. Поэтому, чтобы использовать их свойства и методы, следует использовать оператор As, например:

(Sender as TEdit).Text:=’Начальное значение’;

Оператор As используется для приведения объектных типов, причем производится проверка на совместимость типов во время выполнения программы. Попытка приведения несовместных типов приводит к исключительной ситуации – EInvalidCast. После применения оператора As сам объект остается неизменным, но можно вызывать его методы, которые соответствуют присваиваемому классу.

Оператор Is используется для проверки совместимости по присваиванию экземпляра объекта с заданным классом, т.е. является ли данный экземпляр объектом этого класса или одного из классов, порожденных от заданного, например: If Form1 is TForm then ……. Else ……..;

Эта проверка осуществляется еще на этапе компиляции и, если формально объект и класс несовместимы, выдается сообщение об ошибке в таком операторе.

19. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

19.1. Два вида оператора Try

Исключительные ситуации возникают при невозможности выполнения запланированных действий программой, например: при делении на ноль, попытке открытия несуществующего файла, попытке извлечения корня от отрицательного числа или обращения к области памяти, выходящей за пределы, отведенные данной программе, и т.д. Стандартный обработчик таких событий выводит окно с сообщением об ошибке, и выполнение программы на этом заканчивается. Но иногда программист хочет сам обработать такие исключи-

тельные ситуации и принять решение о дальнейшем направлении выполнения программы. В Delphi это можно сделать несколькими способами. Например, как это было показано в лекции по работе с файлами, можно отключить стандартную обработку ошибок ввода-вывода вставкой в текст программы оператора {$I-}. Затем выполняются любые действия с файлами и проверяется код ошибки с помощью функции IOResult. Если он отличен от нуля, то можно проанализировать эту ошибку и принять какое-то решение. Затем можно опять включить стандартную отработку ошибок ввода-вывода оператором {$I+}. Можно также анализировать ситуацию до появления возможной ошибки с помощью операторов If и принимать какие-то решения до возникновения исключительной ситуации. Но все эти способы в настоящее время используются очень редко. В Delphi есть специальный оператор для защиты кода от исключительных ситуаций – это оператор

```
Try
    {Защищаемый код }
    Finally
    {Операторы завершения защищаемого кода}
    End;
```

и вторая его модификация:

```
Try
    {Защищаемый код}
    Except
    On Exception1 Do Оператор1:
    On Exception1 Do Оператор2;
    ……
    else
    {Операторы для всех остальных исключительных ситуаций}
    end;
```

Для первого варианта оператора Try в любом случае, произошла ли исключительная ситуация или нет, все равно выполняются операторы из секции Finally. Во втором варианте оператора Try при возникновении исключительной ситуации можно предусмотреть отдельную обработку любой из таких ситуаций. Для всех остальных исключительных ситуаций, для которых не предусмотрена отдельная обработка, будут выполняться операторы, следующие за ключевым словом Else. Чем же отличаются исключительные ситуации? Поскольку это объекты, то они отличаются классом или объектным типом. Объектный тип Exception описан в модуле SysUtils.pas. Этот класс является родительским для многочисленных дочерних классов, которые соответствуют различным исключительным ситуациям. Все имена потомков этого класса начинаются с буквы E, например, класс EZeroDevide соответствует исключительной ситуации деления на ноль, а класс EInOutError – ошибке ввода-вывода. В последнем классе есть поле ErrorCode, которое определяет код ошибки ввода-вывода. Используя его, можно детализировать исключительную ситуацию, например:

```
Try
```

```
……
except
on E:EInOutError do
Case E.ErrorCode of
        2:ShowMessage('Файл не найден!');
        3:ShowMessage('Путь не найден!');
        5:ShowMessage('Доступ запрещен!');
        32:ShowMessage('Файл занят!');
        101:ShowMessage('Диск переполнен!);
        103:ShowMessage('Файл не открыт!');
        ……..
        end;
end;
```

Исключительные ситуации EInOutError возникают только тогда, когда установлена опция транслятора {$I+}, иначе надо самому проверять код ошибки с помощью функции IOResult. Следует помнить, что в директиве On указываются имена классов и проверка на совпадение идет в порядке следования имен исключительных ситуаций после слова Except, поэтому сначала следует располагать дочерние классы и только потом их родительские классы, иначе далее родительского класса проверка не пойдет. Например, класс EDivByZero является дочерним от класса EIntError, и, если мы хотим обрабатывать ситуацию EDivByZero, проверку на эту исключительную ситуацию нужно проводить до проверки ситуации EIntError, иначе мы никогда не дойдем до проверки ситуации EDivByZero.

19.2. Программное создание исключительной ситуации

В программе можно самим создать любую исключительную ситуацию с помощью оператора

Raise <конструктор исключительной ситуации> [at <адрес>];

То, что указано в квадратных скобках, может отсутствовать. Если в программе записать только слово Raise, то возбудится исключительная ситуация самого общего класса Exception. Если мы хотим возбудить исключительную ситуацию определенного типа, то необходимо вызвать конструктор соответствующей исключительной ситуации, например:

Raise Exception.Create('Ошибочный параметр') at @MyFunction;

Здесь создается обычная исключительная ситуация при вызове подпрограммы MyFunction. Можно и самим создавать любые исключительные ситуации путем создания собственных дочерних классов от класса Exception.

19.3. Основные исключительные ситуации

Приведем список основных исключительных ситуаций:

- Exception – базовый класс-предок всех обработчиков исключительных ситуаций;
- EAbort – «скрытое» исключение. Его используют тогда, когда нужно прервать процесс с условием, что пользователь программы не должен ви-

деть сообщения об ошибке. Для удобства использования в модуле SysUtils предусмотрена процедура Abort;

- EComponentError – вызывается в двух ситуациях:
  1) при попытке регистрации компонента за пределами процедуры Register;
  2) когда имя компонента не уникально или не допустимо;
- EConvertError – происходит в случае возникновения ошибки при выполнении функций StrToInt и StrToFloat, когда конвертация строки в соответствующий числовой тип невозможна;
- EInOutError – происходит при ошибках ввода-вывода при включенной директиве {$I+};
- EIntError – предок исключений, случающихся при выполнении целочисленных операций:
  - EDivByZero – вызывается в случае деления на ноль как результат RunTime Error 200;
  - EIntOverflow – вызывается при попытке выполнения операций, приводящих к переполнению целых переменных, как результат RunTime Error 215 при включенной директиве {$Q+};
  - ERangeError – вызывается при попытке обращения к элементам массива по индексу, выходящему за пределы массива, как результат RunTime Error 201 при включенной директиве {$R+};
- EInvalidCast – происходит при попытке приведения переменных одного класса к другому классу, не совместимому с первым (например, приведение переменной типа TListBox к TMemo);
- EInvalidGraphic – вызывается при попытке передачи в LoadFromFile файла, несовместимого графического формата.
- EInvalidGraphicOperation – вызывается при попытке выполнения операций, не применимых для данного графического формата (например, Resize для TIcon);
- EInvalidObject – реально нигде не используется, объявлен в Controls.pas;
- EInvalidOperation – вызывается при попытке отображения или обращения по Windows-обработчику (handle) контрольного элемента, не имеющего владельца (например, сразу после вызова MyControl:=TListBox.Create(...) происходит обращение к методу Refresh);
- EInvalidPointer – происходит при попытке освобождения уже освобожденного или еще не инициализированного указателя при вызове Dispose(), FreeMem() или деструктора класса;
- EListError – вызывается при обращении к элементу наследника TList по индексу, выходящему за пределы допустимых значений (например, объект TStringList содержит только 10 строк, а происходит обращение к одиннадцатой строке);
- EMathError – предок исключений, случающихся при выполнении операций с плавающей точкой:
  - EInvalidOp – происходит, когда математическому сопроцессору передается ошибочная инструкция. Такое исключение не будет до

конца обработано, пока Вы контролируете сопроцессор напрямую из ассемблерного кода;
    - EOverflow – происходит как результат переполнения операций с плавающей точкой при слишком больших величинах. Соответствует RunTime Error 205;
    - EUnderflow – происходит как результат переполнения операций с плавающей точкой при слишком малых величинах. Соответствует RunTime Error 206;
    - EZeroDivide – вызывается в результате деления на ноль;
- EMenuError – вызывается в случае любых ошибок при работе с пунктами меню для компонентов TMenu, TMenuItem, TPopupMenu и их наследников;
- EOutlineError – вызывается в случае любых ошибок при работе с TOutLine и любыми его наследниками;
- EOutOfMemory – происходит в случае вызовов New(), GetMem() или конструкторов классов при невозможности распределения памяти. Соответствует RunTime Error 203;
- EOutOfResources – происходит в том случае, когда невозможно выполнение запроса на выделение или заполнение тех или иных Windows ресурсов (например таких, как обработчики – handles);
- EParserError – вызывается, когда Delphi не может произвести разбор и перевод текста описания формы в двоичный вид (часто происходит в случае исправления текста описания формы вручную в IDE Delphi);
- EPrinter – вызывается в случае любых ошибок при работе с принтером.
- EProcessorException – предок исключений, вызываемых в случае прерывания процессора – hardware breakpoint. Никогда не включается в DLL, может обрабатываться только в «цельном» приложении.
    - EBreakpoint вызывается в случае останова на точке прерывания при отладке в IDE Delphi. Среда Delphi обрабатывает это исключение самостоятельно;
    - EFault – предок исключений, вызываемых в случае невозможности обработки процессором тех или иных операций;
- EGPFault – вызывается, когда происходит «общее нарушение защиты» – General Protection Fault. Соответствует RunTime Error 216;
- EInvalidOpCode – вызывается, когда процессор пытается выполнить недопустимые инструкции;
- EPageFault – обычно происходит как результат ошибки менеджера памяти Windows вследствие некоторых ошибок в коде вашего приложения. После такого исключения рекомендуется перезапустить Windows;
- EStackFault – происходит при ошибках работы со стеком часто вследствие некорректных попыток доступа к стеку из фрагментов кода на ассемблере. Компиляция ваших программ со включенной проверкой работы со стеком {$S+} помогает отследить такого рода ошибки;

    - ESingleStep – аналогично EBreakpoint, это исключение происходит при пошаговом выполнении приложения в IDE Delphi, которое само его и обрабатывает;
- EPropertyError – вызывается в случае ошибок в редакторах свойств, встраиваемых в IDE Delphi. Имеет большое значение для написания надежных property editors. Определен в модуле DsgnIntf.pas;
- EResNotFound – происходит в случае тех или иных проблем, имеющих место при попытке загрузки ресурсов форм – файлов .DFM в режиме дизайнера. Часто причиной таких исключений бывает нарушение соответствия между определением класса формы и ее описанием на уровне ресурса (например, вследствие изменения порядка следования полей-ссылок на компоненты, вставленные в форму в режиме дизайнера);
- EStreamError – предок исключений, вызываемых при работе с потоками;
    - EFCreateError – происходит в случае ошибок создания потока (например, при некорректном задании файла потока).
    - EFilerError – вызывается при попытке вторичной регистрации уже зарегистрированного класса (компонента). Является также предком специализированных обработчиков исключений, возникающих при работе с классами компонентов;
- EClassNotFound – обычно происходит, когда в описании класса формы удалено поле-ссылка на компонент, вставленный в форму в режиме дизайнера. Вызывается, в отличие от EResNotFound, в RunTime;
- EInvalidImage – вызывается при попытке чтения файла, не являющегося ресурсом, или разрушенного файла ресурса специализированными функциями чтения ресурсов (например, функцией ReadComponent);
- EMethodNotFound – аналогично EClassNotFound, только при несоответствии методов, связанных с теми или иными обработчиками событий;
- EReadError – происходит в том случае, когда невозможно прочитать значение свойства или другого набора байт из потока (в том числе ресурса);
    - EFOpenError – вызывается, когда тот или иной специфированный поток не может быть открыт (например, когда поток не существует);
- EStringListError – происходит при ошибках работы с объектом TStringList (кроме ошибок, обрабатываемых TListError).

Среда Delphi по умолчанию перехватывает все возникающие исключительные ситуации. Но в случае проверки правильности обработки в вашей программе исключительных ситуаций следует в меню Tools/Debugging Options… отключить флаг Stop On Delphi Exceptions, и тогда среда Delphi перестанет реагировать на исключительные ситуации, возникающие в отлаживаемой программе.

Для целей отладки программы в Delphi имеется процедура

Procedure Assert(Key:Boolean [; const msg:String]);

Ей соответствует исключительная ситуация EAsserttionFailed. При вызове этой функции проверяется значение параметра Key: если оно равно True, то ничего

не происходит, а если – False, то возникает исключительная ситуация EAssertionFailed. Стандартный обработчик этой ситуации показывает в диалоговом окне имя файла с исходным текстом и номер строки, где произошла исключительная ситуация. Включение и отключение этой исключительной ситуации можно осуществлять директивой компилятора {$C+} или {$C-}.

20. ОСНОВНЫЕ КЛАССЫ И ОБЩИЕ СВОЙСТВА КОМПОНЕНТОВ

Рассмотрим основные наиболее часто используемые классы, которые определяют свойства многих компонентов.

20.1. Класс TList

Он представляет собой массив нетипированных указателей и может использоваться для хранения любых данных. Этот класс является прямым потомком класса TObject. Основными его свойствами являются:

Property Capacity:Integer; – характеризует начальный размер массива указателей. Если число элементов списка превысит это значение, то в динамической памяти будет выделено место под новый массив с увеличенным на Capacity числом указателей, туда будут переписаны все старые указатели, и старый массив будет уничтожен. Поэтому нужно заранее приблизительно знать размер будущего списка для сокращения времени вычислений;

Property Count:Integer; – текущее количество элементов списка;

Property Items [Index: Integer]: Pointer; – это как раз и есть массив указателей. Элементы массива нумеруются с нуля;

Property List: PPointerList; – указатель на начало массива указателей.

Рассмотрим некоторые методы данного класса:

Function Add(Item: Pointer): Integer; – функция добавления нового элемента списка Item в конец списка. Она возвращает номер этого элемента в списке;

Procedure Clear; – очищает список, но не освобождает память, которую занимали отдельные элементов списка;

Procedure Delete(Index: Integer); – удаляет из списка указатель на элемент с номером Index;

Procedure Insert(Index: Integer; Item: Pointer); – вставка нового элемента Items перед элементом с номером Index.

Другие методы этого класса рассматривать не будем.

20.2. Класс TStrings

Это базовый, абстрактный класс для объектов, которые представляют список строк. Многие компоненты имеют свойства этого класса, и в них перекрываются абстрактные методы этого класса. Поэтому для создания собственного списка следует использовать класс TStringList, который является наследником класса TStrings. Кроме уже рассмотренных в предыдущем параграфе свойств Capacity и Count, класс TStrings содержит следующие свойства:

Property Strings[Index: Integer]: string; – массив строк списка;

Property Objects[Index: Integer]: TObject; – массив соответствующих строкам объектов. Объекты могут быть любого типа, но в основном здесь хранят картинки. Такая двойственность этого класса позволяет сохранять объекты с текстовыми примечаниями, сортировать их и создавать многомерные наборы строк, так как объектами могут быть потомки опять же класса TStringsж;

Property Sorted: Boolean; – свойство, определяющее необходимость автоматической сортировки строк в алфавитном порядке;

Property Duplicates: TDuplicates; – свойство, управляющее возможностью размещения в списке одинаковых строк. Здесь возможны следующие варианты значения этого свойства:

duIgnore – нельзя добавить новую строку, если она уже есть в списке,

duError – возбуждается исключительная ситуация EListError, если появляется дубль строки,

duAccept – разрешается иметь в списке дубли строк;

Property CaseSensitive: Boolean; – свойство, определяющее возможности сортировки и поиска дублей с различием в строках прописных и строчных букв или без этого различия.

Рассмотрим некоторые специфические методы этого класса:

Function Add(const S: string): Integer; – добавление в конец списка новой строки. Функция возвращает номер добавленной строки в списке. Нумеруются строки с нуля;

Function AddObject(const S: string; AObject: TObject): Integer; – добавление пары строка-объект в конец списка;

Procedure Insert(Index: Integer; const S: string); – вставка новой строки S перед строкой с номером Index;

Procedure LoadFromFile(const FileName: string);– загрузка списка строк из текстового файла FileName;

Procedure SaveToFile(const FileName: string); – сохранение текущего списка строк в текстовом файле FileName.

Последние два метода позволяют очень просто загружать и сохранять списки строк в текстовом файле. Это можно сделать одним оператором, при этом не нужно ни назначать файловой переменной путь к файлу, ни открывать файл, ни переписывать его построчно, ни закрывать. Достаточно, например для компонента Memo1, записать в программе

Memo1.Lines.LoadFromFile('c:\MyDir\Prog1.pas');

и текст программы Prog1.pas будет отображен в текстовом редакторе Memo1. Свойство Lines в компоненте Memo1 как раз имеет тип TStrings, что и позволяет использовать метод LoadFromFile.

20.3. Общие свойства компонентов

Все компоненты Delphi являются потомками класса TComponent, который в свою очередь произошел от класса Tpersistent. Класс Tpersistent передает своим потомкам важный виртуальный метод

Procedure Assing(Source:TPersistent);
Этот метод позволяет копировать поля и свойства объекта Source в объект, вызвавший метод Assign.

Класс TComponent является предком для класса TControl, от которого в свою очередь произошли классы TWinControl и TGraphicControl. Компоненты, которые наследуют класс TWinControl, имеют оконный ресурс, т.е. они способны получать и обрабатывать сообщения Windows.

Рассмотрим некоторые общие для всех свойства компонентов:

Property Name:TComponentName; – имя компонента, обычно оно дается самой системой Delphi, например Button1, и на начальном этапе обучения программированию лучше его не изменять;

Property Tag:Integer; – определяет 4 байта в любом компоненте для личного использования;

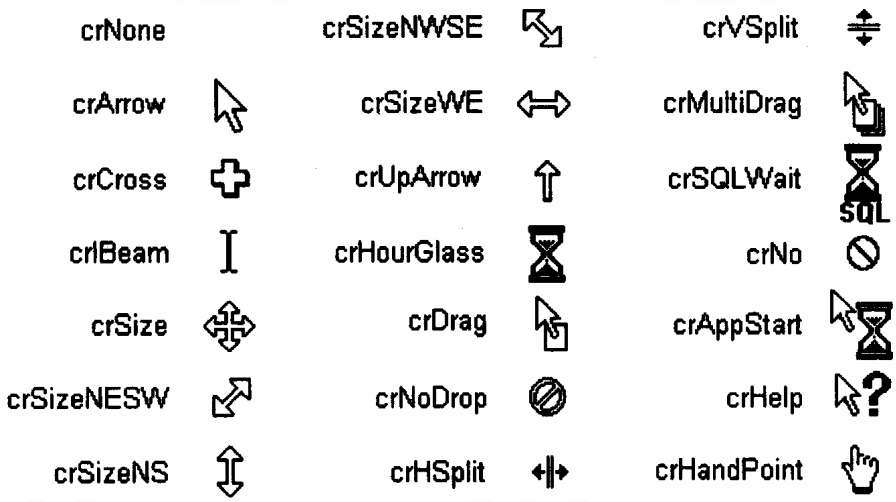
Property Owner:TComponent; – указатель на владельца данного компонента;

Property Parent:TWinControl; – определяет родительское окно для данного компонента. Следует иметь в виду, что владелец создает компонент, а родитель им управляет как дочерним окном;

Property Caption:TCaption; – заголовок компонента, именно он, например для кнопки, определяет видимую надпись на компоненте;

Property Text:TCaption; – текст на компоненте, он не совместим со свойством Caption. У компонента может быть свойство или Caption, или Text;

Property Cursor:TCursor; – определяет вид курсора при его нахождении над данным компонентом. Все курсоры принадлежат глобальному объекту Screen. Ниже приведены стандартные виды курсора:



Стандартные курсоры системы Windows имеют номера от -17 до 0. Можно создать свой вид курсора в каком-нибудь графическом редакторе, сохранить его в ресурсном файле с расширением *.res, внести его в список курсоров объекта Screen, а затем назначить его любому компоненту. Пример:

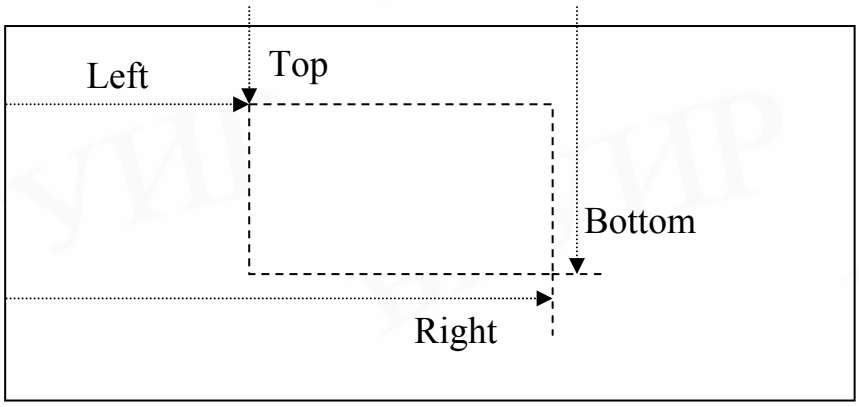
```
{$R Cursor.res}
Screen.Cursor[1]:=LoadCursor(HInstance,'Cur_1');
Button1.Cursor:=1 ;
```

Здесь сначала читается ресурсный файл Cursor.res, затем в объект Screen загружается курсор с именем Cur_1, регистрируется под номером 1, и этот вид курсора назначается компоненту Button1.

Property BaundsRect:TRect; – определяет прямоугольное окно, в котором находится компонент вместе с его граничным окаймлением. Здесь

```
TRect = Record
 case Integer of
   0: (Left, Top, Right, Bottom: Integer);
   1: (TopLeft, BottomRight: TPoint);
   End;
 End;
TPoint=Record
 x,y:Integer;
 end;
```

Графически окно определяется следующим образом относительно координат владельца компонента:



Property ClientRect:TRect; – определяет клиентскую часть окна компонента, т.е. ту часть, которая доступна для отображения дочерних компонентов;

Property Align:TAlign; – определяет вид выравнивания компонента относительно границ родителя. Это свойство может принимать следующие значения:

alNone – выравнивание не производится;

alTop – компонент прижимается к верхней границе родителя и его ширина становится равной родительской ширине клиентской области;

alBottom – компонент прижимается к нижней границе родителя и его ширина становится равной родительской ширине клиентской области;

alLeft – компонент прижимается к левой границе родителя и его высота становится равной родительской высоте клиентской области;

alRight – компонент прижимается к правой границе родителя и его высота становится равной родительской высоте клиентской области;

alClient – компонент расширяется на всю клиентскую область родителя. Свойство Align часто используется при проектировании приложений рассчитанных для работы с экраном различного разрешения, например, 640 на 480 точек или 1024 на 768 точек. Оно позволяет автоматически настраивать вид формы и расположение компонентов на ней для любого разрешения экрана;

Property Visible:Boolean; – определяет видимость компонента. Можно прямо управлять значением этого свойства, а можно вызывать специальные методы: *Procedure Hide;* – скрыть;

Procedure Show; – показать;

Property Enable:Boolean; – определяет активность компонента. Если это свойство ложно, то этот компонент не может быть активным и отображается обычно серым цветом;

Property Color:Tcolor; – определяет цвет фона компонента. Цвет обычно задается или символьной константой, начинающейся приставкой «cl», или шестнадцатеричной восьмиразрядной константой, в которой старшие два разряда обычно полагают равными нулю, а следующие сдвоенные разряды определяют интенсивности синей, зеленой и красной составляющих цвета. Например, константа $000000FF соответствует ярко-красному цвету;

Property Hint:String; – текст оперативной подсказки, которая может всплывать рядом с компонентом, на который указывает курсор. Текст подсказки может состоять из двух частей, разделенных вертикальной линией. Первая часть текста подсказки будет появляться рядом с компонентом, а вторая часть – будет передаваться объекту Application в свойство Hint и может отображаться в любом визуальном компоненте, обычно в StatusBar, при написании обработчика события OnHint для объекта Application. Например, для кнопки можно определить подсказку вида

Button1.Hint:=’Открыть файл | Работа с файлами’;

С оперативной подсказкой непосредственно связаны следующие свойства:

Property ShowHint:Boolean; – разрешает вывод подсказки. Нужно помнить, что и у владельца данного компонента это свойство должно быть истинным, иначе у всех дочерних компонентов подсказка работать не будет;

Property HintColor:Tcolor; – определяет цвет фона подсказки;

Property HintPause:Integer; – определяет длительность паузы перед показом подсказки после появления курсора мыши над компонентом в миллисекундах;

Property HintHidePause:Integer; – определяет длительность показа подсказки в миллисекундах;

Property HelpContext:THelpContext; – определяет номер странички справочной системы, которая может вызываться для активного компонента при нажатии клавиши F1;

Property Canvas:TCanvas; – определяет холст фона компонента, вначале это обычно светло-серый фон, но на нем можно рисовать все, что угодно.

Мы рассмотрели только основные общие свойства компонентов. Более подробно о свойствах конкретных компонентов можно узнать, используя службу помощи системы Delphi или книги – справочники.

21. ГРАФИЧЕСКИЕ ВОЗМОЖНОСТИ DELPHI

21.1. Класс Tcanvas

Основу графики в Delphi представляет класс Tcanvas – это холст (контекст GDI в Windows) с набором инструментов для рисования. Основные свойства холста:

- *Property Pen:Tpen;* – карандаш;
- *Property Brush:Tbrush;* – кисть;
- *Property Font:Tfont;* – шрифт;
- *Property PenPos:Tpoint;* – текущая позиция карандаша в пикселях относительно левого верхнего угла канвы;
- *Property Pixels[x,y:Integer]:Tcolor.* – массив цветов холста;
- *Property CopyMode:TcopyMode;* – свойство, которое определяет, как графический рисунок копируется в канву. Оно используется при вызове метода CopyRect и при копировании объектов TbitMap. Возможные значения этого свойства:
  - cmBlackness – заполнение области рисования черным цветом;
  - cmDest – заполнение области рисования цветом фона;
  - cmMergeCopy – объединение изображения на канве и копируемого изображения с помощью операции AND;
  - cmMergePaint – объединение изображения на канве и копируемого изображения с помощью операции OR;
  - cmNotSrcCopy – отображение на канве инверсного изображения источника;
  - cmNotSrcErase – объединение изображения на канве и копируемого изображения с помощью операции OR и инвертированием полученного результата;
  - cmPatCopy – копирование шаблона источника;
  - cmPatInvert – объединение шаблона источника с изображением на канве с помощью операции XOR;
  - cmPatPaint – объединение инверсного изображения источника с исходным шаблоном с помощью операции OR, а затем объединение результата этого действия с изображением на холсте, используя ту же логическую операцию;
  - cmSrcAnd – объединение изображения источника и канвы с помощью операции AND;
  - cmSrcCopy – перенос изображения источника на канву;
  - cmSrcErase – инвертирование изображения на канве и объединение его с изображением источника с помощью операции AND;

- cmSrcInvert – объединение изображения на канве и источнике с помощью операции XOR. Отметим, что повторное объединение восстанавливает первоначальное изображение на канве. Это значение свойства CopyMode используется при создании игр, когда происходит движение объекта по фону;
- cmSrcPaint – объединение изображения на канве и источнике с помощью операции OR;
- CmWhiteness – заполнение области рисования белым цветом.

Канва не является компонентом, но во многих компонентах является свойством. С помощью свойства Pixels все пиксели канвы представляются в виде двухмерного массива точек. Изменяя цвет пикселей, можно прорисовывать изображение по отдельным точкам.

Методы канвы:

Рrocedure Аrc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer); – чертит дугу эллипса в охватывающем его прямоугольнике (X1, Y1) – (X2, У2). Начало дуги лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3, Y3), а конец – на пересечении с лучом из центра в точку (X4, У4). Дуга чертится против часовой стрелки (рис.21.1, а);

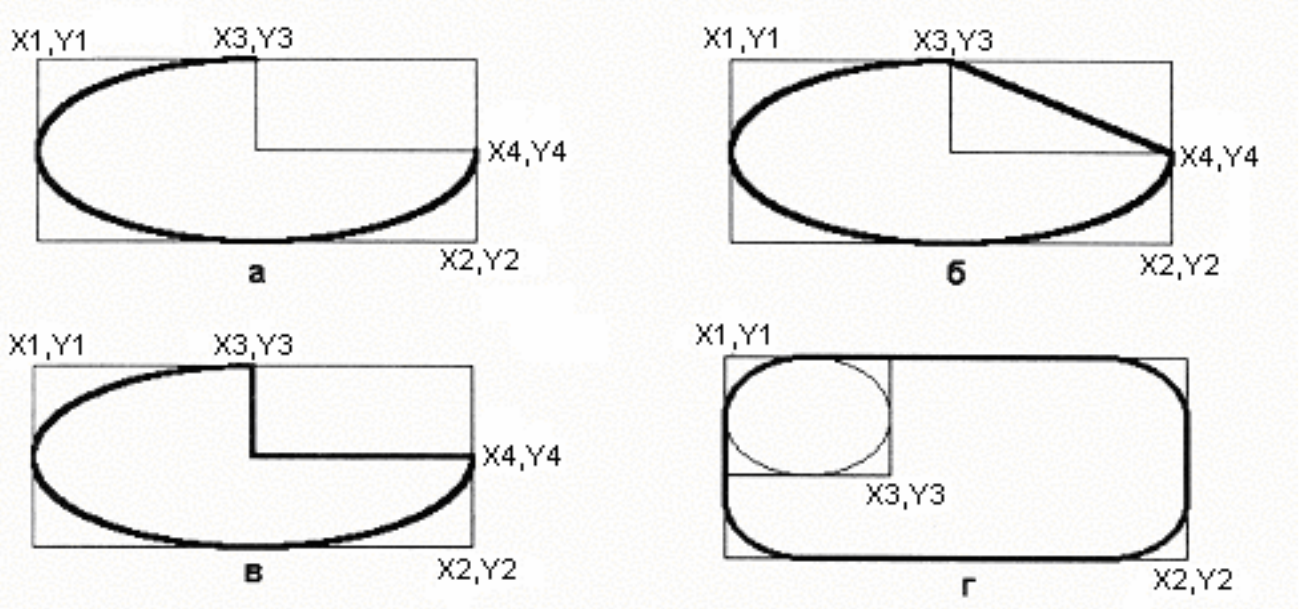


Рис. 21.1. Параметры обращения к методам:
а – Arc; б – Chord; в – Pie; г – RoundRect

Procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4:Integer) ; – чертит сегмент эллипса в охватывающем его прямоугольнике (XI, VI) – (X2, У2). Начало дуги сегмента лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3, Y3), а конец – на пересечении с лучом из центра в точку (X4, У4), дуга сегмента чертится против часовой стрелки, а начальная и конечная точки дуги соединяются прямой (рис. 21.1,б);

Procedure CopyRect(Dest: Trect; Canvas: Tcanvas; Source: Trect); – копирует изображение Source канвы Canvas в участок Dest текущей канвы. При этом свойство CopyMode определяет различные эффекты копирования;

Pгocedure Draw(X, Y: Integer; Graphic: TGraphic); – осуществляет вывод на канву графического объекта Graphic так, чтобы левый верхний угол объекта расположился в точке (X, Y);

Pгocedure E11ipse(X1, Y1, X2, Y2: Integer); – чертит эллипс в охватывающем его прямоугольнике (X1, Y1) – (X2, Y2), заполняет внутреннее пространство эллипса текущей кистью;

Pгocedure FillRect(const Rect: Trect); – заполняет текущей кистью прямоугольную область Rect, включая ее левую и верхнюю границы, но не затрагивая правую и нижнюю границы;

Pгocedure FloodFill(X, Y: Integer; Co1oг: TCo1oг; FillStyle: TFillStyle); – производит заливку канвы текущей кистью. Заливка начинается с точки (X, У) и распространяется во все стороны от нее. Если FillStyle= fsSurface, заливка распространяется на все соседние точки с цветом Co1oг. Если FillStyle=fsBorder, наоборот, заливка прекращается на точках с этим цветом;

Pгocedure FrameRect(const Rect:TRect) ; – очерчивает границы прямоугольника Rect текущей кистью толщиной в 1 пиксель без заполнения внутренней части прямоугольника;

Pгocedure LineTo (X, У: Integer); – чертит линию от текущего положения пера до точки (X, У);

Procedure MoveTo(X, У: Integer) ; – перемещает карандаш в положение (X, У) без вычерчивания линий;

Procedure Pie(X1, У1, X2, У2, X3, У3, X4, У4: Integer); – рисует сектор эллипса в охватывающем прямоугольнике (X1, У1) – (X2, Y2). Начало дуги лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3, У3), а конец – на пересечении с лучом из центра в точку (X4, У4). Дуга чертится против часовой стрелки. Начало и конец дуги соединяются прямыми с ее центром (см. рис. 21.1, в);

Pгocedure Po1ygon (Points: aггay of TPoint) ; – вычерчивает карандашом многоугольник по точкам, заданным в массиве Points. Конечная точка соединяется с начальной, и многоугольник заполняется кистью. Для вычерчивания без заполнения используется метод Po1yline;

Pгocedure Po1y1ine (Points: array of TPoint) ; – вычерчивает карандашом ломаную прямую по точкам, заданным в массиве Poimts;

Pгocedure Rectangle (X1, У1, X2, У2: Integer); – вычерчивает и заполняет прямоугольник (X1, У1) – (X2, У2). Для вычерчивания без заполнения используется методы FrameRect или Po1yLine;

Procedure ReFresh; – устанавливает в канве умалчиваемые шрифт, карандаш и кисть;

Pгocedure RoundRect (X1, У1, X2, У2, X3, У3: Integer) ; – вычерчивает и заполняет прямоугольник (XI, У1) – (X2, У2) со скругленными углами. Прямоугольник (X1, Y1) – (X3, У3) определяет дугу эллипса для округления углов (рис.22.1,г);

Pгocedure StretchDraw (const Rect: TRect; Graphic: TGraphic) ; – отображает и при необходимости масштабирует графический объект Graphic так, чтобы он полностью занял прямоугольник Rect;

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

Procedure TextRect (Rect: TRect; X, У: Integer; const Text: String) ; – выводит строку Text так, чтобы левый верхний угол прямоугольника, охватывающего текст, располагался в точке (X, У). Если при этом какая-либо часть надписи выходит из границ прямоугольника Rect, она отсекается и не будет видна.

21.2. Классы TGraphic и TPicture

Важное место в графическом инструментарии Delphi занимают классы TGraphic и Tpicture.

TGraphic – это абстрактный класс, инкапсулирующий общие свойства и методы трех своих потомков: значка (TIcon), метафайла (TmetaFile) и растрового изображения (TBitMap). Общей особенностью потомков TGraphic является то, что обычно они сохраняются в файлах определенного формата. Значки представляют собой небольшие растровые изображения, снабженные специальными средствами, регулирующими их прозрачность. Для файлов значков обычно используется расширение ICO. Метафайл – это изображение, построенное на графическом устройстве с помощью специальных команд, которые сохраняются в файле с расширением WMF или EMF. Растровые изображения – это произвольные графические изображения в файлах со стандартным расширением BMP.

Свойства класса TGraphic:

- *Pгoperty Empty: Boo1ean;* – содержит *Tгue,* если с объектом не связано графическое изображение;.
- *Pгoperty Height: Tnteger;* – содержит высоту изображения в пикселях;
- *Pгopeгty Modefied: Boo1ean;* – содержит *Tгue,* если графический объект изменялся;
- *Property Palette: HPALETTE;* – содержит цветовую палитру графического объекта.
- *Property PaletteModified: Boolean;* – содержит *True,* если менялась цветовая палитра графического объекта.
- *Property Transparent: Boolean;* – содержит *True,* если объект прозрачен для фона, на котором он изображен;
- *Property Width: Integer;* – содержит ширину изображения в пикселях.

Методы класса TGraphic:

Procedure LoadFromClipboardFormat (AFormat: Word; AData: THandle; Apalette: HPALETTE) ; – ищет в буфере межпрограммного обмена *Clipboard* зарегистрированный формат *AFormat* и, если формат найден, загружает, из буфера изображение *AData* и его палитру *APalette;*

Procedure LoadFromFile(const FileName: String) ; – загружает изображение из файла *FileName;*

Procedure LoadFromStream (Stream: TStream) ; – загружает изображение из потока данных *Stream;*

Procedure SaveToClipboardFormat (var AFormat: Word; var AData: THandle; var APalette: HPALETTE); – помещает графическое изображение *AData* и его цветовую палитру *APalette* в буфер межпрограммного обмена в формате *Aformat;*

Procedure SaveToFile(const FileName:String) ; – сохраняет изображение в файле *FileName;*

Procedure SaveToStream(Stream: TStream); – сохраняет изображение в потоке *Stream.*

Полнофункциональный класс *TPicture* инкапсулирует в себе все необходимое для работы с готовыми графическими изображениями – значком, растром или метафайлом. Его свойство *Graphic* может содержать объект любого из этих типов, обеспечивая нужный полиморфизм методов класса.

Свойства класса TPicture:

- *Property Bitmap: TBitmap;* – интерпретирует графический объект как растровое изображение;
- *Property Graphic: TGraphic;* – содержит графический объект;
- *Property Height: Integer;* – содержит высоту изображения в пикселях;
- *Property Icon: TIcon;* – интерпретирует графический объект как значок;
- *Property Metafile: TMetafile;* – интерпретирует графический объект как метафайл;
- *Property Width: Integer;* – содержит ширину изображения в пикселях.

Методы класса *TPicture:*

Procedure Assign(Source: TPersistent) ; – копирует содержимое объекта *Source* в объект *Graphic;*

Procedure LoadFromClipboardFormat (AFormat: Word; AData: THandle; APalette: HPALETTE); – ищет в буфере межпрограммного обмена *Clipboard* зарегистрированный формат *AFormat* и, если формат найден, загружает из буфера изображение *AData* и его палитру *Apalette;*

Procedure LoadFromFile(const FileName: String) ; – загружает изображение из файла *FileName;*

class Procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass: TGraphicClass) ; – используется для регистрации в *Clipboard* нового формата изображения;

class Procedure RegisterFileFormat (const AExtension, ADescription: String; AGraphicClass: TGraphicClass); – используется для регистрации нового файлового формата;

class Procedure RegisterFileFormatRes(const AExtension: String; ADescription-ResID: Integer; AGraphicClass: TGraphicClass); – используется для регистрации нового формата ресурсного файла;

Procedure SaveToClipboardFormat (var AFormat: Word; var AData: THandle; var APalette: HPALETTE) ; – помещает графическое изображение *AData* и его цветовую палитру *APalette* в буфер межпрограммного обмена в формате *AFormat;*

Procedure SaveToFile(const FileName: String) ; – сохраняет изображение в файле *FileName;*

class Function SupportsClipboardFormat(AFormat: Word): Boolean; – возвращает *True,* если формат *AFormat* зарегистрирован в буфере межпрограммного обмена *Clipbord;*

class Procedure UnregisterGraphicClass(AClass: TGraphicClass); – делает недоступными любые графические объекты класса *AClass.*

21.3. Классы TFont, TPen и TBrush

Класс TFont определяет объект «шрифт» для любого графического устройства (экран, принтер и т.д.).

Свойства класса TFont:

- *Property Charset: TFontCharSet;* – набор символов. Для русскоязычных программ это свойство обычно имеет значение *DEFAULTCHARSET* или *RUSSIAN CHARSET*. Используйте значение *OEMCHARSET* для отображения текста *MS-DOS* (альтернативная кодировка);
- *Property Color: TColor;* – цвет шрифта;
- *Property FontAdapter: IChangeNotifier;* – указатель на интерфейс для передачи информации о шрифте в компоненты *ActveX;*
- *Property Handle: hFont;* – дескриптор шрифта. Используется при непосредственном обращении к API-функциям *Windows;*
- *Property Height: Integer;* – высота шрифта в пикселях;
- *Property Name: TFontName;* – имя шрифта. По умолчанию имеет значение *MS Sans Serif;*
- *Property Pitch: TFontPitch;* – определяет способ расположения букв в тексте: значение *fpFixed* задает моноширинный текст, при котором каждая буква имеет одинаковую ширину; значение *fpVariabel* определяет пропорциональный текст, при котором ширина буквы зависит от ее начертания; *fpDefault* определяет ширину, принятую для текущего шрифта;

- *Property PixelPerInch: Integer;* – определяет количество пикселей экрана на один дюйм реальной длины. Это свойство не следует изменять, так как оно используется системой для обеспечения соответствия экранного шрифта шрифту принтера;
- *Property Size: Integer;* – высота шрифта в пунктах (1/72 дюйма). Изменение этого свойства автоматически изменяет свойство *Height* и наоборот;
- *Property Style: TFontStyles;* – стиль шрифта. Может принимать значение как комбинацию следующих *признаков: fsBold* (жирный), *fsItalic* (курсив), *fsUnderline* (подчеркнутый), *fsStrikeOut* (перечеркнутый).

Класс TPen определяет объект «перо» для рисования линий.

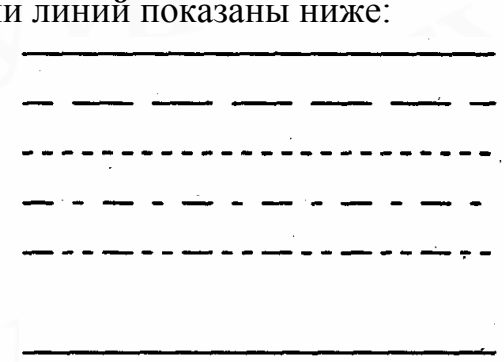
Его основные свойства:

- *Property Color:TColor;* – цвет вычерчиваемых пером линий;
- *Property Handle: Integer;* – дескриптор пера. Используется при непосредственном обращении к API-функциям *Windows;*
- *Property Mode:TPenMode ;* – определяет способ взаимодействия линий с фоном (см. ниже);
- *Property Style:TPenStyle;* – определяет стиль линий. Этот стиль имеет смысл только для толщины линий в 1 пиксель. Для толстых линий стиль всегда *psSolid* (сплошная);
- *Property Width: Integer;* – толщина линий в пикселях.

Значения свойства *Mode:*

- pmBlack – линии всегда черные. Свойства *Color* и *Style* игнорируются;
- pmWhite – линии всегда белые. Свойства *Color* и *Style* игнорируются;
- pmNop – цвет фона не меняется (линии не видны);
- pmNot – инверсия цвета фона. Свойства *Color* и *Style* игнорируются;
- pmCopy – цвет линий определяется свойством *Color* пера;
- pmMotCopy – инверсия цвета пера. Свойство *Style* игнорируется;
- pmMergePenNot – комбинация цвета пера и инверсионного цвета фона;
- pmMaskPenNot – комбинация общих цветов для пера и инверсионного цвета фона. Свойство *Style* игнорируется;
- pmMergeNotPen – комбинация инверсионного цвета пера и фона;
- pmMaskNotPen – комбинация общих цветов для инверсионного цвета пера и фона. Свойство *Style* игнорируется;
- pmMerge – комбинация цветов пера и фона;
- pmNotMerge – инверсия цветов пера и фона. Свойство *Style* игнорируется;
- pmMask – общие цвета пера и фона;
- pmNotMask – инверсия общих цветов пера и фона;
- pmXor – объединение цветов пера и фона операцией *XOR;*
- pmNotXor – инверсия объединения цветов пера и фона операцией *XOR.*

Стили линий показаны ниже:



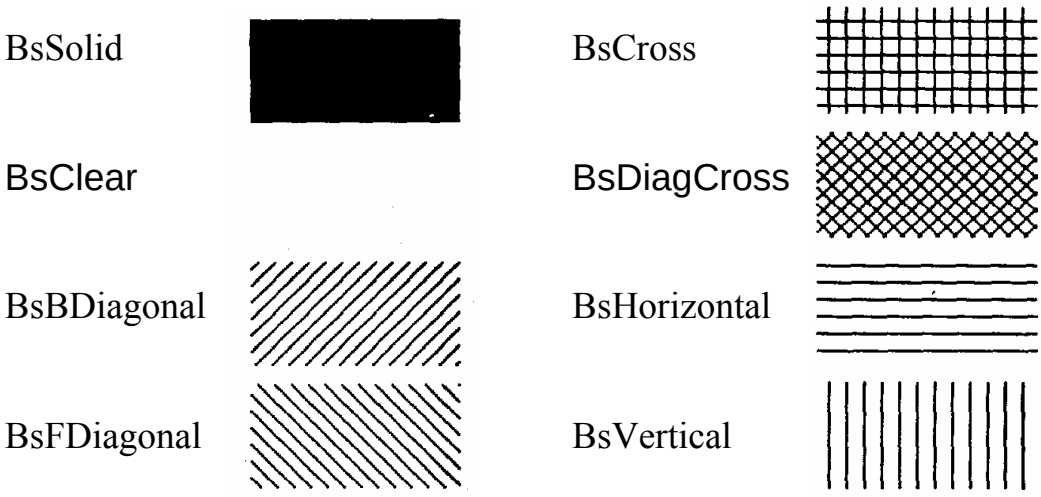
Следует заметить, что в системе Windows есть одна существенная недоработка, а именно: все линии, кроме psSolid, могут быть толщиной только в 1 пиксель.

Класс TBrush служит для описания параметров кисти для заполнения внутреннего пространства замкнутых фигур.

Свойства класса TBrush:

- *Property Bitmap: TBitmap;* – содержит растровое изображение, которое будет использоваться кистью для заполнения. Если это свойство определено, свойства *Color* и *Style* игнорируются;
- *Property Color: TColor;* – цвет кисти;
- *Property Handle: Integer;* – дескриптор кисти. Он используется при непосредственном обращении к API-функциям *Windows;*
- *Property Style:TBrushStyle;* – стиль кисти.

Стили кисти:



21.4. Работа с изображениями

Класс TCanvas, который является основой графической системы Delphi, не содержит методов сохранения отрисованных изображений. Методы сохранения картинок имеют классы: TPicture, TGraphic и TBitMap. Поэтому сохранение изображений из класса TCanvas можно осуществлять, используя

один из указанных выше классов. Приведем пример сохранения картинки из класса TCanvas объекта PaintBox1 в файл с помощью методов класса TBitMap:

```
Var Bm:TBitMap; // Определяем указатель на растровую картинку
    R:TRec;     // Выделяем память под прямоугольник
Begin
  Bm:=TBitMap.Create;       // Выделяем память под растровую картинку
  R:=PaintBox1. ClientRect; // Запоминаем размеры картинки
  Bm.Width:=R.Right-R.Left+1; // Передаем эти размеры объекту Bm
  Bm.Height:=R.Bottom-R.Top+1;
  Bm.Canvas.CopyRect(R,PaintBox1.Canvas,R); // Копируем картинку в Bm
  Bm.SaveToFile('c:\my\pic1.bmp'); // Сохраняем картинку в файле
  Bm.Free; // Освобождаем память выделенную объекту Bm
```

Операционная система Windows поддерживает только растровые картинки и файлы с расширением *.bmp. В этих файлах первые 14 байт определяют заголовок файла, следующие 40 байт – заголовок картинки (ширина, высота, разрешение по координатам для печати картинки, число цветовых плоскостей, число цветов в палитре, метод сжатия и т.д.). Затем идет цветовая палитра, где каждый цвет имеет свой номер и занимает в палитре 4 байта. Далее идет само растровое изображение – построчное перечисление номеров цветов для каждого пикселя (точки на экране), причем строки растра идут снизу вверх и длина каждой строки кратна 4 байтам.

Система Delphi позволяет работать и с картинками в формате JPEG (расширения файлов *.jpg). Для этого в Delphi используется специальный модуль Jpeg. В этом формате вся картинка сначала разбивается на квадратики 8*8 или 16*16 пикселей. Исходная растровая картинка каждого квадратика преобразуется, как в телевидении, в яркостную и две цветоразностные плоскости. Каждая из плоскостей подвергается дискретному, двухмерному, быстрому преобразованию Фурье. В результате этого преобразования получаются матрицы коэффициентов Фурье. В этих матрицах отбрасываются высшие составляющие коэффициентов Фурье. Оставшиеся коэффициенты матриц затем сжимаются по методу Хаффмана Число этих коэффициентов определяется параметром качества изображения. Этот формат позволяет сжимать растровые изображения в 5 – 20 раз в зависимости от структуры картинки, и он используется в основном для хранения фотографий людей, природы и т.д. Он также стал основой для цифрового телевидения и является основой формата MJPEG. Следует помнить, что этот формат хранит картинки с частичной потерей информации, которую человеческий глаз почти не замечает. В нем не следует хранить, например, графики функций, изображения с резким переходом яркости или цветности, так как вокруг линий будет появляться вуаль в виде отдельных точек, что становится уже заметным нашему глазу.

Приведем пример вывода в компонент Image1 картинки из файла с расширением *.jpg. В обработчике любого события можно записать:

```
Var AJpeg:TJpegImage;       // Определяем указатель на картинку
Begin
```

```
AJpeg:=TJpegImage.Create; // Выделяем память под картинку
AJpeg.LoadFromFile('C:\my\pic1.jpg');  // Читаем файл
Image1.Picture:=TPicture(AJpeg); // Отображаем картинку
AJpeg.Free;          // Освобождаем память из-под картинки
End;
```

Рассмотрим теперь пример чтения картинки из базы данных. Пусть, например, в таблице ADOTable1 есть поле Pic, в котором хранятся картинки в формате JPEG. Тогда в обработчике какого–то события можно написать:

```
Var  AJpeg:TJpegImage;      // Определяем указатель на картинку
 ABlobStream:TBlobStream; // Определяем указатель на поток байт
Begin
 ABlobStream:=TBlobStream.Create(ADOTable1.FieldByName ('Pic')
 as TBlobField, bmWrite);     // Связываем поле таблицы с потоком
 AJpeg:=TJpegImage.Create;         // Выделяем память под картинку
 AJpeg.LoadFromStream(ABlobStream);         // Загружаем картинку
 Imag1.Picture:=TPicture(AJpeg); // Отображаем картинку
 AJpeg.Free.                  // Освобождаем память из-под картинки
 End;
```

Отметим так же, что в Delphi можно работать и с картинками в формате GIF, в котором картинки хранятся в сжатом виде с использованием стандартного алгоритма LZW (Лемпела-Зива-Вootча). Этот алгоритм в процессе сжатия информации создает словарь слов и запоминает не сами слова, а их коды, которые занимают памяти существенно меньше, чем слова из словаря. Все современные архиваторы данных (ARJ, ZIP, RAR и др.) используют именно этот алгоритм сжатия информации, причем процесс сжатия идет без потери информации, как это наблюдается в картинках JPEG . Этот алгоритм может сжимать изображения в 2–5 раз. Формат GIF позволяет хранить сразу несколько картинок и чередовать их появление на экране, создавая подобие мультфильма. Этот формат картинок широко используется на страничках Интернета, в нем лучше всего хранить графики функций и контрастные изображения. Он официально не поддерживается в Delphi, но есть бесплатно рассылаемая библиотека компонентов RxLib, в которой есть поддержка такого формата картинок.

22. ВИЗУАЛЬНЫЕ КОМПОНЕНТЫ DELPHI

Рассмотрим некоторые, интересные с точки зрения использования, визуальные компоненты Delphi. Вся визуальная библиотека компонентов (VCL) Delphi содержит около 450 компонентов. К ним может быть добавлено неограниченное число компонентов, созданных различными компаниями и отдельными программистами. Delphi – открытая система для подключения любых нестандартных компонентов. Следует иметь в виду, что основная масса дополнительных компонентов поставляется в двоичном формате в файлах с расширением *.dcu, формат которого изменяется от одной к другой версий Delphi. Поэтому, приобретая какой-либо дополнительный компонент для одной

версии Delphi, нет никакой уверенности, что он будет работать с более новой версией Delphi.

Рассмотрим возможности некоторых компонентов, которые будут использованы при выполнении лабораторных работ по программировагнию.

22.1. Компонент TBitBtn

Этот компонент определяет кнопку с рисунком. Основные его свойства:

- Property Caption:Tcaption; – надпись на кнопке;
- Property Kind:TBitBtnKind; – вид кнопки. При значении этого свойства BkCustom вид кнопки определяется пользователем. Вид кнопок для других значений этого параметра приведен на рис.22.1;

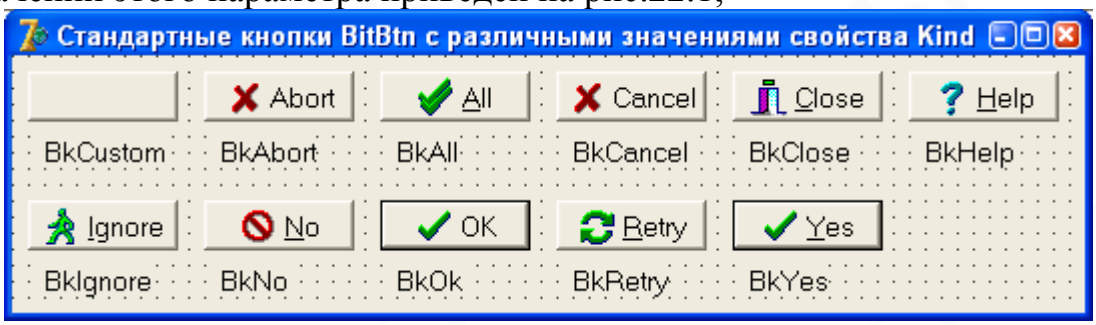


Рис.22.1. Виды стандартных кнопок

- *Property Glyph:TBitMap;* – растровое изображение картинок на кнопке;
- *Property NumGlyphs:TNumGlyphs;* – количество картинок в растре;
- *Property LayOut:TButtonLayOut;* – определяет край кнопки, к которому прижимается рисунок;
- *Property Margin:Integer;* – определяет расстояние от края кнопки до рисунка в пикселях;
- *Property Spacing:Integer;* – определяет расстояние от края рисунка до надписи на нем в пикселях.

Кнопки могут находиться в одном из следующих состояний:

1) обычное или нормальное,
2) кнопка не доступна,
3) кнопка нажата,
4) кнопка утоплена (это только для кнопок TSpeedButton).

Для каждого из этих состояний можно предусмотреть свою картинку на кнопке. Для этого в графическом редакторе нужно создать растровую картинку с несколькими изображениями состояний кнопки. Ниже приведен пример картинки для трех состояний кнопки «Старт». Сама картинка на кнопке будет размером 32 на 32 пикселя, но объединенная картинка для трех состояний будет иметь ширину 96 пикселей:

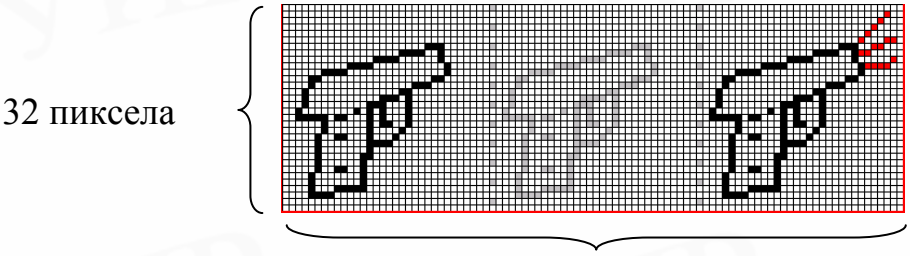


Рис.22.2. Пример картинки для кнопки TBitBtn

22.2. Компоненты TDrawGrid и TStringGrid

Компонент TDrawGrid представляет собой универсальную таблицу. Таблица делится на две области: фиксированную и подвижную. Фиксированная область предназначена для заголовков строк и столбцов.

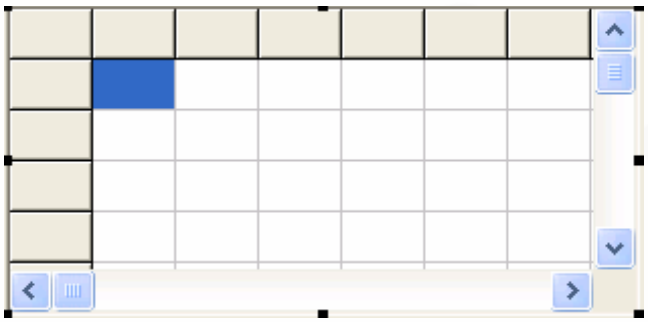


Рис.22.3. Вид компонента TDrawGrid

Основными свойствами этого компонента являются:

- *Property ColCount:Integer;* – количество столбцов;
- *Property RowCount:Integer;* – количество строк;
- *Property Col:Integer;* – текущая колонка;
- *Property Row:Integer;* – текущая строка;
- *Property FixedCols:Integer;* – число фиксированных столбцов;
- *Property FixedRows:Integer;* – число фиксированных строк;
- *Property DefoultDrawing:Boolean;* – если это свойство задать равным False, то программист должен предусмотреть отрисовку содержимого компонента в обработчике события:
- *Property OnDrawCell:TDrawCellEvent;* – в нем следует предусмотреть отрисовку каждой ячейки таблицы.

Рассмотрим пример отрисовки в компоненте DrawGrid1 шахматной доски:

```
Procedure TForm1.DrawGrid1DrawCell(Sender:TObject; Col,Row:Longint;
Rect:Trect; State:TGridDrawState);
    Begin
     With DrawGrid1.Canvas do Begin
      If not Odd(Col+Row) then Brush.Color:=clBlack else
      Brush.Color:=clWhite;       // Выбор цвета кисти
      FillRect(Rect); // Закраска текущей ячейки
```

```
    End;
End;
```

Здесь Col и Row определяют номер текущей ячейки таблицы, Rect – прямоугольник текущей ячейки, State – состояние текущей ячейки, которое может быть следующим:

gdSelected – текущая выбранная ячейка,
gdFocused – текущая ячейка имеет фокус,
gdFixed – текущая ячейка принадлежит фиксированной области.

Компонент TStringGrid наследует компонент TDrawGrid и все его свойства. Этот компонент используется для отображения таблицы строк. Его основным свойством является

- *Property Cells[ACol, ARow:Integer]:String;* – в этом свойстве хранят значения всех строк таблицы.

22.3. Компонент TPageControl

Компонент TPageControl со страницы Win32 используется для организации многостраничного блокнота, каждая страница которого представляется объектом типа TTabSheet (рис.22.4).

Рис.22.4. Вид компонента TPageControl

В приведенном выше примере компонент TPageControl имеет три страницы с именами A, B и C. Он позволяет на одной форме разместить достаточно большое количество разных компонентов. На каждой странице размещается свой набор компонентов. Переключение между страницами осуществляется щелчком мыши по соответствующей закладке. На этапе проектирования можно щелкнуть над компонентом правой кнопкой мыши, и появится всплывающее меню с описанием возможных действий. Основными свойствами этого компонента являются:

- *Property ActivePage:TTabSheet;* – текущая страница блокнота,
- *Property PageCount:Integer;* – число страниц в блокноте,
- *Property TabIndex:Integer;* – номер выбранной странички блокнота. Странички нумеруются с нуля.

22.4. Компонент TTimer

Данный компонент со странички System. Это не визуальный компонент, он используется для отсчета интервалов реального времени:

Его основные свойства:

- *Property Interval:word;* – задает интервал времени в миллисекундах, который проходит между событиями OnTimer. Величина этого интервала реально

всегда будет кратна 55 мс. Это определяется системными часами, которые вызывают прерывания по времени через каждые 55 мс;

- *Property Enable:Boolean;* – включает и выключает таймер.

В обработчике события OnTimer определяются те действия, которые должны быть выполнены через каждые Interval миллисекунд, например перемещение движка в компоненте TGauge.

22.5. Компонент TGauge

Он предназначен для отображения процесса изменения какой-либо величины, находится на страничке Semple и имеет несколько вариантов отображения, что определяется свойством (рис.22.5):

- *Property Kind:TGaugeKind;*

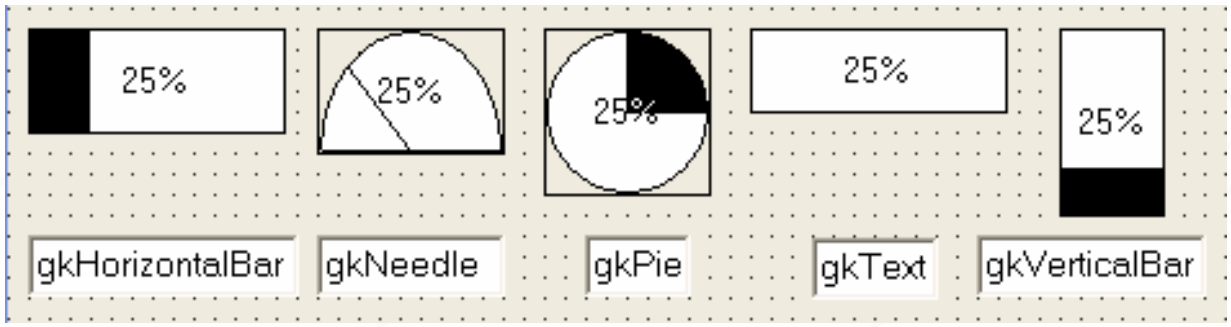


Рис.22.5. Варианты отображения TGauge

Остальные свойства имеют следующие значения:

- *Property BackColor: TColor;* – цвет незакрашенной части индикатора;
- *Property ForeColor: TColor;* – цвет закрашенной части индикатора;
- *Property MaxValue: Longint;* – максимальное значение диапазона изменения свойства Progress;
- *Property MinValue: Longint;* – минимальное значение диапазона изменения свойства Progress;
- *Property PercentDone: Longint;* – текущее значение Progress в процентах от его диапазона изменения;
- *Property Progress: Longint;* – текущее значение изменяющейся числовой величины;
- *Property ShowText: Boolean;* – если содержит True, в центре компонента выводится строковое представление значения PercentDone.

Метод *Procedure AddProgress(Value: Longint);* добавляет к текущему значению Progress величину Value. Аналогичные функции выполняют компоненты TProgressBar и TtrackBar.

22.6. Компонент TColorGrid

Этот компонент со страницы Samples предназначен для выбора цветов из 16-цветной палитры:

Основной цвет выбирается щелчком левой кнопки мыши и отображается символами FG, фоновый цвет выбирается правой кнопкой мыши и отображается символами BG, при совпадении цветов отображаются символы FB.

Выбранные цвета определяются свойствами:

- *Property BackGroundColor:TColor;* – цвет фона,
- *Property ForeGroundColor:Tcolor;* – основной цвет.

При смене цветов вызывается событие OnChange.

23. СТАНДАРТНЫЕ ДИАЛОГОВЫЕ ОКНА И ТИПОВЫЕ ДИАЛОГИ

23.1. Стандартные диалоговые окна

Стандартные диалоговые окна находятся на страничке Dialogs, на форме они отображаются ввиде значков и видны только на этапе проектирования программы. Все диалоговые окна вызываются в процессе работы программы с помощью метода Execute.

Для начала рассмотрим возможности компонента TOpenDialog  . Этот диалог предназначен для выбора файлов на диске. Основными его свойствами являются:

- *Property Filter:String;* – определяет фильтр для отбора файлов.

Он состоит из набора сдвоенных полей. В первом определяется имя фильтра, а во втором – его реализация. Все поля фильтра разделяются вертикальной чертой – |. Например, можно задать фильтр следующим образом:

OpenDialog1.Filter:=’Графические файлы | *.bmp, *.gif, *.jpg ’+
’| Текстовые файлы | *.txt, *.pas ’;

Здесь фильтр состоит из двух возможных фильтров: первый фильтр – для выбора графических файлов, а второй – для выбора текстовых файлов.

- *Property FileName:WideString;* – определяет имя выбранного файла.
- *Property Options:TOpenOptions;* – определяет множество возможностей выбора файлов.

Например, значение множества ***ofAllowMultiSelect*** позволяет выбрать не один, а произвольное множество файлов, имена которых размещаются в свойстве:

- *Property Files:TStrings;*

Для выбора нескольких файлов следует удерживать клавишу Ctrl и мышью отмечать выбранные файлы.

Рассмотрим пример обработчика события нажатия кнопки с надписью «Открыть файл», который производит поиск файла с расширением *.pas и затем воспроизводит содержимое этого файла в компоненте Memo1:

```
Procedure TForm1.Button1Click(Sender:TObject);
 Begin With OpenDialog1 do Begin
```

```
    Filter:='Паскалевские файлы | *.pas';
    If Execute then Memo1.Lines.LoadFromFile(FileName);
    End;
  End;
```

Приведем список остальных стандартных диалогов.

TSaveDialog  – сохранение файлов,

TOpenPictureDialog  – открытие картинки,

TSavePictureDialog  – сохранение картинки,

TFontDialog  – выбор шрифта,

TColorDialog  – выбор цвета,

TPrintDialog  – выбор текущего принтера из списка зарегистрированных в операционной системе,

TPrinterSetUpDialog  – настройка параметров печати,

TFindDialog  – поиск образа в тексте,

TReplaceDialog  – поиск образа и его замены в тексте,

TPageSetUpDialog  – настройка параметров печатаемой страницы.

23.2. Типовые диалоги

Типовые диалоговые окна вызываются как обычные процедуры или функциями по имени диалога. Рассмотрим основные типовые диалоги.

Procedure ShowMessage(const Msg:String); – вывод на экран окна сообщения с текстом Msg и одной кнопкой OK. Положение окна определяет сама операционная система.

Например:

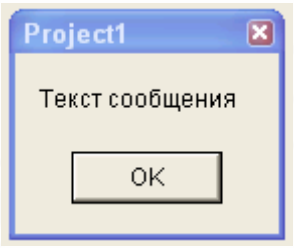


Рис.23.1. Вид окна диалога процедуры ShowMessage

Procedure ShowMessagePos(const Msg:String; x,y:Integer); – вывод окна диалога, где x и y определяют положение на экране верхнего левого угла окна.

Function (const Msg: string; DlgType: TMsgDlgType; Buttons: TMsgDlgButtons; HelpCtx: Longint): Word; – вывод окна диалога с возможным выбором ответа на сообщение Msg. Здесь второй параметр определяет тип диалога по отображаемому в окне значку. Возможны следующие варианты значков:

mtWarning – предупреждение  ,

mtError – ошибка  ,

mtInformation – информационное сообщение –  ,

mtConfirmation – знак вопроса  ,

mtCustom – отсутствие значка.

В окне диалога можно расположить любое множество предопределенных кнопок с текстом:

mbYes – 'Yes' – да,
mbNoA – 'No' – нет,
mbOK – 'OK' – хорошо,
mbCancel – 'Cancel' – закончить,
mbAbort – 'Abort' – прервать,
mbRetry – 'Retry' – повторить,
mbIgnore – 'Ignore' – игнорировать,
mbAll – 'All' – для всех,
mbNoToAll – 'No to All' – не для всех,
mbYesToAll – 'Yes to All' – да, для всех.

Последний формальный параметр определяет номер странички из файла помощи, которая будет появляться, если нажать клавишу F1.

Функция возвращает код нажатой кнопки. Коды определяются текстовыми константами, такими же, как и множество кнопок, только начинаются они с приставки «mr» вместо «mb».

Рассмотрим пример использования этого диалога:

```
Case MessageDlg('Продолжать выполнение программы?',
mtConfirmation, [mbYes,mbNo],0) of
  mrYes:ShowMessage('Нажата кнопка – Да');
  mrNo:ShowMessage('Нажата кнопка – Нет');
  end;
```

В результате вызова функции MessageDlg появится следующее окно с двумя кнопками:

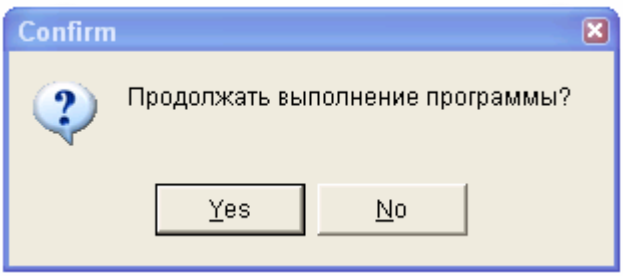


Рис.23.2. Вид окна диагола MessageDlg

Нажатие любой из кнопок будет сопровождаться появлением сообщения «Нажата кнопка – Да» или «Нажата кнопка – Нет».

Function InputBox(const ACaption, APrompt, ADefault: string): string; – вывод окна диалога для ввода строки. Здесь:

ACaption – заголовок окна диалога,

APrompt – пояснение для вводимого текста,

ADefault – начальное значение строки ввода.

Например, следующий оператор

S:=InputBox('Заголовок окна', 'Введите имя файла', 'c:\my\p1.pas');

вызовет появление следующего окна для ввода текста:

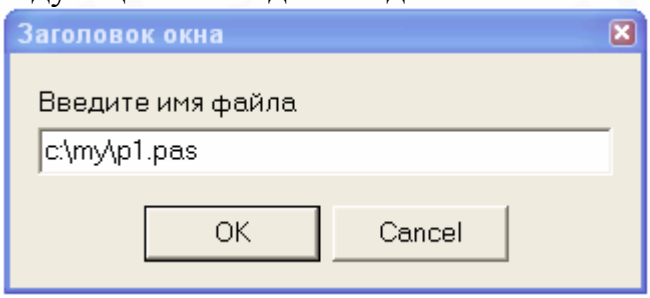


Рис.23.3. Вид окна диалога InputBox

24. ФОРМА, ПРИЛОЖЕНИЕ И ГЛОБАЛЬНЫЕ ОБЪЕКТЫ

24.1. Форма и ее свойства

Форма является основным строительным компонентом любой программы. Каждое приложение имеет хотя бы одну форму – главное окно программы. Программа может состоять из произвольного числа форм, которые появляются на экране по мере необходимости. Рассмотрим основные свойства формы:

- *Property FormStyle:TFormStyle;* – стиль формы. Стиль может принимать следующие значения:
  fsNormal – обычная форма;
  fsMDIForm – главная форма многодокументного интерфейса;
  fsMDIChild – дочерняя форма многодокументного интерфейса. Для многодокументного интерфейса (Multi Document Interface – MDI) характерно то, что дочерние формы могут создаваться только в границах основной MDI формы;
  fsStayOnTop – форма, перекрывающая все остальные формы. Такая форма всегда будет находиться поверх других форм и может быть только одна. Таким значением формы следует пользоваться очень осторожно, так как она может просто «мозолить» глаза;
- *Property BorderIcons:TBorderIcons;* – множество, определяющее наличие значков в заглавии окна. Возможны следующие значения элементов множества:
  biSystemMenu – значок вызова системного меню, обычно – «закрыть окно»,
  biMinimize – значок свертывания формы,
  biMaximize – значок развертывания формы,
  biHelp – значок вызова справочной службы;

- *Property BorderStyle:TFormBorderStyle;* – стиль границы формы. Возможны следующие варианты:
  bsNone – форма не имеет рамки и заголовка и не может перемещаться или изменять свои размеры,
  bsSingle – форма с рамкой в один пиксель, она не может менять свои размеры,
  bsSizeable – форма с обычной рамкой,
  bsDialog – форма не может менять свои размеры,
  bsToolWindow – подобно bsSingle, но с уменьшенным по высоте заголовком,
  bsSizeToolWin – подобно bsSizeable, но с уменьшенным по высоте заголовком;
- *Property Position:Tposition;* – определяет положение и размер формы в момент ее окрытия. Возможны следующие значения:
  poDesigned – все определяется на этапе проектирования формы,
  poDefault – положение и размеры окна определяет самой ОС Windows,
  poDefaultPosOnly – Windows определяет только положение окна на экране,
  poDefaultSizeOnly – Windows определяет только размеры окна на экране,
  poScreenCenter – форма располагается по центру экрана;
- *Property Icon:TIcon;* – значок окна для главной формы и значок приложения,
- *Property Menu:TMainMenu;* – основное меню формы,
- *Property Canvas:TCanvas;* – холст формы.

Рассмотрим возможность сделать в качестве фона формы какую-нибудь картинку. Для этого определим в качестве глобальной переменной указатель на картинку и два обработчика событий для формы, например:

```
Var gr:TBitMap;      // Определяем указатель на картинку
Procedure TForm1.FormCreate(Sender:TObject); // Создание формы
  Begin
    Gr:=TbitMap.Create; // Выделяем память под картинку
    Gr.LoadFromFile('c:\my\pic1.bmp'); // Загружаем картинку из файла
    End;
Procedure TForm1.FormPaint(Sender:TObject); // Перерисовка формы
  Begin
    Canvas.Draw(0,0,gr); // Отображаем картинку на форме
    End;
```

В этом примере загружают картинку в память из файла в обработчике события создания формы. Если бы здесь же отобразили картинку на форме, то при перемещении формы или изменении ее размеров эта картинка исчезла, ведь при каждом изменении размеров формы или ее перемещении по экрану форма заново перерисовывается. Обработчик события OnPaint вызывается при каждой перерисовке формы до момента отрисовки всех компонентов, размещенных на форме. В нем и вызывают метод отображения картинки на форме.

Все формы проекта обычно создаются в начале работы приложения, и конструкторы для их создания вызываются в файле проекта с расширением *.dpr.

Формы бывают модальными и немодальными. Модальная форма не позволяет переключиться на другую форму приложения, пока сама не будет закрыта. Немодальные формы не имеют такого ограничения. Главная форма проекта вызывается автоматически при запуске приложения. Для отображения всех остальных форм используются методы:

Procedure Show; – отображение немодальной формы;

Procedure ShowModal; – отображение модальной формы.

Для закрытия форм используются методы:

Procedure Close; – закрытие немодальной формы;

Procedure CloseModal; – закрытие модальной формы.

Например, из формы Form1 необходимо вызвать модальную форму Form2 в обработчике события нажатием какой-либо кнопки. Для этого можно написать

```
Procedure TForm1.Button1Click(Sender:TObject);
 Begin
  Form2.ShowModal;
  End;
```

24.2. Объект Application

Этот объект создается для каждой программы, написанной в Delphi, и является связующим звеном с операционной системой Windows. Объект Application доступен только на этапе выполнения программы. Основные его методы вызываются в файле проекта приложения. Например:

```
program Project1;
uses Forms,
  Unit1 in 'Unit1.pas' {Form1}; // Подключение текста программы
{$R *.res}   // Подключение файлов ресурсов
begin
  Application.Initialize; // Инициализация объекта Application
  Application.CreateForm(TForm1, Form1); // Создание формы
  Application.Run; // Запуск обработчика событий
  end.
```

Приведем некоторые свойства этого объекта:

- *Property ExeName:String;* – имя исполняемого файла программы,
- *Property HelpFile:String;* – имя файла справки программы,
- *Property Hint:String;* – вторая часть оперативной подсказки,
- *Property Title:String;* – текст на кнопке свернутой программы.

Особый интерес представляет метод *Procedure ProcessMessages;*

Он приостанавливает выполнение программы до тех пор, пока не будут обработаны все сообщения в очереди Windows. Вызывать метод ProcessMessages необходимо для обновления содержимого визуальных компонентов в случае, когда в программе выполняется большой объем вычислений. Любое изменение содержимого компонента попадает в очередь сообщений Windows и

остается в ней, пока не приостановится текущий вычислительный процесс. Поэтому в программах с большим объемом вычислений необходимо периодически (например, в конце итерации или на каждом шаге интегрирования) вызывать процедуру ProcessMessages, которая позволяет отслеживать этапы вычислительного процесса.

24.3. Глобальные объекты

Глобальные объекты доступны одновременно всем приложениям, выполняющимся на одном компьютере, и создаются автоматически при запуске приложения. Основными глобальными объектами являются: ClipBoard, Screen и Printer.

Объект ClipBoard

Объект *Clipboard* используется как буфер межпрограммного обмена в системе Windows. Для использования его возможностей достаточно в разделе **uses** подключить модуль *Clipbrd*.

Методы открытия и закрытия буфера обмена:

Procedure Open;

Procedure Close;

вызываются во всех остальных методах *TClipBoard***,** поэтому прямой их вызов практически не используется.

Очистка буфера обмена производится методом *Procedure Clear;*

С помощью свойств

- *Property FormatCount:Integer;* // число форматов
- *Property Formats[Index:Integer]:Word;* // номера форматов

можно узнать о форматах данных, записанных в буфер обмена.

Стандартно поддерживаются форматы, определяемые следующими текстовыми константами:

CF_TEXT – буфер содержит текст,

CF_BITMAP – буфер содержит картинку,

CF_METAFILEPICT – буфер содержит векторную картинку,

CF_PICTURE – буфер содержит объект типа *TPicture,*

CF_COMPONENT – буфер содержит компонент,

CF_PALETTE – буфер содержит палитру картинки.

Для работы с текстом предназначены следующие методы:

Function GetTextBuf(Buffer:PChar; BufSize:Integer):Integer;

Procedure SetTextBuf(Buffer:PChar);

Первый метод помещает текст из буфера обмена в переменную Buffer, длина которого ограничена значением BufSize. Функция возвращает истинную длину перемещенного текста. Второй метод помещает текст из переменной Buffer в буфер обмена в формате CF_TEXT.

Например, оператор

Clipboard.SetTextBuf(PChar(Memo1.SelText));

записывает в буфер обмена выделенный текст из компонента Memo1. Можно произвести и обратную операцию – поместить текст из буфера обмена, например в компонент Memo2:

Memo2.Lines.Add(ClipBoard.AsText);

Здесь используется свойство AsText для извлечения текста из буфера обмена. Используя метод

Procedure Assing(Source:TPersistent);

можно помещать в буфер обмена данные классов TBitMap (формат CF_BITMAP) или TMetaFile (формат CF_METAFILEPICT).

Следующий пример показывает возможность сохранения картинки из компонента Image1 в буфере обмена и последующей переписи этой картинки в компонент Image2:

```
ClipBoard.Asssign(Image1.Picture);
Image2.Picture.Assing(ClipBoard);
```

Для определения формата данных, находящихся в буфере обмена, можно использовать, например, следующий обработчик события нажатия кнопки Button1:

```
Procedure TForm1.Button1Click(Sender: TObject);
begin
 if Clipboard.HasFormat(CF_TEXT) then
 ShowMessage('Буфер содержит текст');
 if Clipboard.HasFormat(CF_BITMAP) then
 ShowMessage('Буфер содержит картинку');
 if Clipboard.HasFormat(CF_METAFILEPICT) then
 ShowMessage('Буфер содержит векторную картинку');
 if Clipboard.HasFormat(CF_PICTURE) then
 ShowMessage('Буфер содержит объект типа TPicture');
 if Clipboard.HasFormat(CF_COMPONENT) then
 ShowMessage('Буфер содержит компонент');
 end;
```

Можно также зарегистрировать свой собственный формат данных для буфера обмена с помощью процедуры **RegisterClipboardFormat**, но в этом случае нам самим придется написать процедуры записи информации в буфер обмена и чтения ее из буфера.

Объект Screen

Этот объект представляет свойства дисплея. Курсор приложения, общий для всех форм, доступен через свойство

- *Property Cursir:TCursor;*

которое уже было описано в 21-м разделе. Например, включить «песочные часы» на время выполнения длительной операции можно следующим образом:

```
Try
 Screen.Cursor:=crHourGlass;
 { выполнение операции }
 Finally
 ScreenCursor:=crDefault;
 End;
```

Имена всех зарегистрированных в системе шрифтов помещаются в список, определенный в свойстве

- *Property Fonts:TString;*

Размеры экрана в пикселях можно получить из свойств:

- *Property Height:Integer;*
- *Property Width:Integer;*

Эти свойства довольно часто используются для настройки размеров форм и других компонентов на текущее разрешение экрана дисплея.

Следующие два свойства указывают на активную в данный момент форму и ее активный элемент управления:

- *Property ActiveForm:TForm;*
- *Property ActiveControl:TWinControl;*

Объект Printer

Этот объект предназначается для печати из приложения. Выбрать текущий принтер и установить необходимые его параметры можно при помощи стандартных диалогов PrintDialog и PrintSetUpDialog, которые были рассмотрены в разд. 23.

Информацию о всех установленных с системе принтерах и текущем принтере можно получить из свойств:

- *Property Printers:TStrings;*
- *Property PrinterIndex:Integer;*

Свойство

- *Property Fonts:TStrings;* – содержит список шрифтов, поддерживаемых текущим принтером.

Расположение листа определяется свойством

- *Property Orientation:TPrinterOrientation;* Оно может принимать значения:

poPortrait – вертикальное расположение страницы,
poLandscape – горизонтальное расположение страницы.

Высоту и ширину листа бумаги содержат свойства:

- *Property PageHeihgt:Integer;*
- *Property PageWidth:Integer;*

Вывод информации на печать так же, как и на дисплей, осуществляется через свойство

- *Property Canvas:TCanvas;*

Начало и конец печати листа бумаги осуществляется с помощью методов:

Procedure BeginDoc;
Procedure EndDoc;

Следует заметить, что использование объекта Printer для печати документов достаточно сложно и неоправданно. Многие компоненты имеют свои методы печати, например: TcustomForm, TCustomRichEdit, TChart. Для печати же из Delphi документов можно использовать уже готовые заготовки, например COM-сервер приложения Word.Application.

25. МЕЖПРОГРАММНОЕ ВЗАИМОДЕЙСТВИЕ

В операционной системе Windows каждой программе выделяется определенный участок оперативной памяти, и никакая другая программа не

может ни прочитать, ни записать туда какую-либо информацию. Это защищает работу каждой программы от влияния извне, однако и приводит к определенным трудностям при организации межпрограммного взаимодействия. Один из способов передачи информации уже рассмотрен в предыдущем разделе – это использование буфера обмена ClipBoard. Такой способ обмена можно реализовать только на одной машине. Рассмотрим некоторые другие способы взаимодействия задач.

25.1. Обмен сообщениями

Посылка сообщений – это естественный способ связи между программами в Windows. В Delphi все участники обмена должны быть наследниками класса TWinControl, т.е. окнами, у которых есть свои номера или дискрипторы. Следующий пример показывает, как найти запущенное приложение с главным окном Server:

```
Var Svrh:THandle;
    Begin
     Svrh:=FindWindow(nil, 'Server');
     If Svrh=0 then ShowMassage('Сервер не найден');
```

Передать сообщение окну можно с помощью функции

Function SendMessage(Hwnd:HWND; Msg:Cardinal; WParam, LParam:Integer):LongBool;

Здесь Hwnd – номер окна, Msg – код команды, WParam и LParam – параметры, которые передаются окну. Эта функция ожидает окончания обработки переданного сообщения и возвращает его результат. Другая функция PostMessage имеет такие же параметры, но она не ждет окончания обработки посланного сообщения, а лишь сообщает о том, смогла ли она поставить сообщение в очередь.

Для создания обработчика Windows сообщений нужно в классе объявить метод с директивой **message** и указать номер обрабатываемой команды. Как видно из списка параметров функции посылки сообщения, каждому окну может быть передано не более 8 байт информации. Для передачи большего количества информации можно использовать команду WM_COPYDATA.

Рассмотрим пример создания сервера с именем Server, который каждому клиенту посылает информацию о текущем времени на сервере. В примере используются следующие типы:

```
TMsg = packed record    // Тип сообщения
   hwnd: HWND;  // Указатель на окно, которому посылается сообщение
   message: UINT;         // Код сообщения
   wParam: WPARAM;  // W параметр
   lParam: LPARAM;     // L параметр
   time: DWORD;         // Время посылки сообщения
   pt: TPoint; // Координаты мыши в момент отправки сообщения
  end;
TCopyDataStruct= record // Структура передаваемых данных
   dwData: Integer;         // Прямо передаваемые 4 байта
```

```
        cbData: Integer;          // Длина передаваемых данных
        lpData: Pointer;          // Указатель на начало передаваемых данных
      end;
```

Текст примера:

```
// Текст файла проекта сервера
Program server;
Uses Forms, windows,
  userver in 'userver.pas' {Form1}, // Включение в проект модуля Userver
  Udata1 in 'Udata1.pas' {DataModule1: TDataModule};
// Подключение модуля данных
Type TNewObject=class    // Объявляем новый класс
// Определяем классовый обработчик сообщений для сервера
    class procedure AppMsgHandler(var Msg:TMsg; var Handled:Boolean);
    end;
{$R *.res}
class procedure TNewObject.AppMsgHandler;
  var i:Integer;
  ClientWnd:Thandle;
  Begin
    ClientWnd:=THandle(Msg.WParam); // Получаем номер окна клиента
    if Msg.message=WM_ST then Begin
      for i:=0 to MaxClient-1 do Begin
        if TimeClientList[i]=0 then Begin
          TimeClientList[i]:=ClientWnd;
// Запоминаем номер окна в списке клиентов
          Handled:=True;
          exit;
          end;
        end;
      end;
    if Msg.Message=WM_UT then Begin
      for i:=0 to MaxClient-1 do Begin
        if TimeClientList[i]=ClientWnd then Begin
// Удаляем клиента из списка
          TimeClientList[i]:=0;
          Handled:=True;
          exit;
          end;
        end;
      end;
    end;
var i:Integer;
begin
// Очищаем список клиентов
  for i:=0 to MaxClient-1 do TimeClientList[i]:=0;
```

```
// Регистрируем в системе новые коды сообщений для окон
// Код соединения с сервером
  WM_ST:=RegisterWindowMessage('WM_ST');
// Код отсоединения
  WM_UT:=RegisterWindowMessage('WM_UT');
  Application.Initialize;
// Определяем обработчик Windows сообщения для сервера
  Application.OnMessage:=TNewObject.AppMsgHandler;
// Создаем формы
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TDataModule1, DataModule1);
// Запускаем обработчик сообщений сервера
  Application.Run;
  end.
// Текст модуля данных
Unit Udata1;
interface
uses SysUtils, Classes;
type
  TDataModule1 = class(TDataModule)
  end;
// Определяем глобальные переменные и константы сервера
var
  DataModule1: TDataModule1;
  WM_ST,WM_UT:THAndle;
  const maxclient=64; // Максимальное число клиентов
  var TimeClientList:Array[0..maxclient] of integer;
implementation
{$R *.dfm}
end.
// Текст модуля формы сервера
Unit userver;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, ExtCtrls,Udata1;
Type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
// Метод обработки тиков таймера сервера
```

```
Procedure TForm1.Timer1Timer(Sender: TObject);
var dt:TDateTime;
 i:Integer;
 cds:TCopyDataStruct;     // Структура передаваемых данных
begin
 dt:=Now;       // Определяем текущее время
 cds.cbData:=sizeof(dt);    // Запоминаем длину передаваемых данных
 cds.lpData:=@dt;           // Запоминаем указатель на передаваемые данные
 for i:=0 to MaxClient-1 do Begin
  if TimeClientList[i]<>0 then
// Посылаем данные всем зарегистрированным клиентам
   SendMessage(TimeClientList[i],WM_COPYDATA,Self.Handle,
   LongInt(@cds));
   end;
 end;
end.

// Текст модуля формы клиента
Unit Uclient1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, Buttons;
Type
 TForm1 = class(TForm)
   SpeedButton1: TSpeedButton;
   Label1: TLabel;
   Label2: TLabel;
   procedure SpeedButton1Click(Sender: TObject);
   procedure FormCreate(Sender: TObject);
 private
// Метод обработки сообщения WM_COPYDATA для клиента
   procedure WMCopyData(var msg:TMessage);message WM_COPYDATA;
 end;
var
 Form1: TForm1;
 WM_ST,WM_UT:THandle;
implementation
{$R *.dfm}
// Текст метода обработки сообщения WM_COPYDATA
Procedure TForm1.WMCopyData(var msg:TMessage);
var  dt:TDateTime;
Begin
 if PCopyDataStruct(msg.LParam)^.lpData <> nil then Begin
   dt:=TDateTime(PCopyDataStruct(msg.LParam)^.lpData^);
// Выводим на форму клиента врямя, переданное сервером
```

```
    Label2.Caption:=TimetoStr(dt);
    end;
  end;
Procedure TForm1.SpeedButton1Click(Sender: TObject);
var ServerHandle:THandle;
begin
// Получаем номер окна сервера
  ServerHandle:=FindWindow(nil,'server');
  if ServerHandle<>0 then Begin
    if SpeedButton1.Down then Begin
// Посылаем серверу сообщение для установления с ним соединения
      PostMessage(ServerHandle,WM_ST,Integer(Self.Handle),0);
      SpeedButton1.Caption:='Соединение есть!';
      end
    else Begin
// Посылаем серверу сообщение о прекращении с ним связи
      PostMessage(ServerHandle,WM_UT,Integer(Self.Handle),0);
      SpeedButton1.Caption:='Connect';
      Label2.Caption:='';
      end;
    end;
  end;
Procedure TForm1.FormCreate(Sender: TObject);
Begin
// Регистрируем в системе новые коды сообщений со стороны клиента
  WM_ST:=RegisterWindowMessage('WM_ST');
  WM_UT:=RegisterWindowMessage('WM_UT');
  end;
end.
```

Вид окна клиента может быть следующим:

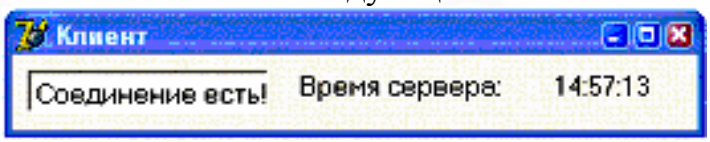


Рис.25.1. Вид формы клиента

25.2. Динамический обмен данными

Динамический обмен данными (Dynamic Data Exchange – DDE) является одним из первых протоколов обмена данными в системе Windows. На его основе были созданы протоколы OLE (Object Link and Embedding – связывания и внедрения объектов) сначала версии 1.0, а затем 2.0.

Приложения, использующие DDE, разделяются на две категории: клиенты, которые задают вопросы, и серверы, которые на них отвечают. Оба участника процесса осуществляют контакты (Conversations) по определенным темам (Topic), и в рамках темы производится обмен элементами данных (Items).

Для создания сервера на страничке компонентов System имеются два компонента: TDDEServerConv и TDDEServerItem. Для получения доступа к сервису DDE-сервера клиенту нужно знать несколько параметров:

- имя сервиса (Service Name) – это имя приложения (сервера) с путем к нему;
- имя темы (Topic Name);
- имя элемента данных (Item Name).

Объект TDDEServerItem связывается с TDDEServerConv и определяет, что будет пересылаться по DDE. Для этого у него есть свойства Lines и Text. При изменении значений этих свойств происходит автоматическая пересылка обновленных данных во все приложения-клиенты, которые установили связь с сервером по данной теме. Обработка сообщений от клиентов производится в обработчике события OnPokeData объекта TDDEServerItem.

Для создания клиента используются компоненты TDDEClientConv и TDDE-ClientItem. Установление связи с сервером во время выполнения программы осуществляется методом SetLink с передачей ему имен сервиса и темы. Передача данных серверу осуществляется методом PokeDataLines с передачей имени элемента данных и собственно самих данных. Получение данных от сервера происходит в методе обработки события OnChange объекта TDDECli-entItem.

25.3. Совместное использование общей памяти

Операционная система Windows выделяет каждому приложению свой участок оперативной памяти, который недоступен никакому другому приложению. Это позволяет существенно повысить надежность работы как самой операционной системы, так и всех приложений, так как они не могут влиять друг на друга. Однако это усложняет межпрограммное взаимодействие. Одним из вариантов организации такого взаимодействия является использование виртуальной памяти. Такой вид памяти используется операционной системой при ограниченной физической памяти компьютера. Система имеет в своем распоряжении так называемый файл подкачки (в системе Windows 98 это файл **win386.swp**, в системе Windows NT/XP – **pagefile.sys**). С его помощью система производит страничную перекачку память-файл и файл-память. Неактивные программы вытесняются на диск, а активные помещаются в физическую память компьютера. Когда памяти в компьютере не хватает, то можно наблюдать постоянную работу жесткого диска по перемещению данных из памяти в файл подкачки и обратно.

Такой файл можно создать и в Delphi-программе с помощью функций Windows API (Application Interface):

- *CreateFileMapping* – создание файла подкачки;
- *MapViewFile* – получение доступа к данным файла;
- *UnMapViewFile* – прекращение доступа к данным файла подкачки.

В разных приложениях для совместного использования памяти нужно задавать одинаковые параметры для этих функций. Не будем описывать правила использования этих функций. Однако отметим, что динамическая память практически не имеет ограничений на размер и ее следует использовать,

например, при обработке больших изображений в картографии. Кроме того, так как файл подкачки может почти весь располагаться в оперативной памяти ПК, то работа с ним будет осуществляться гораздо быстрее, чем с обычным файлом.

25.4. Каналы

Протокол DDE уже пережил период своего расцвета, на смену ему пришел сетевой протокол NDDE (NetWork DDE), однако и ему уже пришел на смену механизм каналов – **pipes**. Это основной протокол работы клиент-серверных приложений для Microsoft SQL Server. Канал можно представить себе как среду, через которую обмениваются два приложения. Будем рассматривать только именованные каналы, т.е. имя канала является одним для всех приложений, участвующих в обмене данными. Имя канала записывается в соответствии с соглашением UNC (Universal Naming Convention). Оно может выглядеть так: \\<имя сервера>\<каталог>\<имя канала>.

Для создания сервера используется специальная функция Windows API – CreateNamedPipe. Клиент подключается к уже созданному каналу с помощью функции ConnectNamedPipe. Дальнейшая работа с каналом напоминает работу с обычным файлом. Для этого используются функции ReadFile и WriteFile.
Отключение от канала осуществляется функцией DisConnectNamedPipe.

Каналы хорошо приспособлены для работы в локальной сети, однако в глобальных сетях используется другой механизм взаимодействия приложений – сокеты.

25.5. Сокеты

Сокеты отличаются от каналов тем, что поддерживают работу всех протоколов глобальной сети Интернет. Общепринятой и стандартизованной на международном уровне является семиуровневая модель структуры протоколов связи под названием *модель взаимодействия открытых систем* (Open System Interface – OSI). На каждом из уровней, от первого – физического до высших уровней, решается свой круг задач и используются свои инструменты. Роль каждого уровня – предоставлять определенный интерфейс протоколам следующего уровня. Перечислим эти уровни:

1) физический – на нем происходят преобразование бит в электромагнитный сигнал при передаче данных и обратное преобразование при приеме. Здесь все реализуется аппаратно, программные интерфейсы и протоколы отсутствуют;

2) канальный – обеспечивает связь между физическими устройствами в рамках одной сети;

3) сетевой – это уровень межсетевого общения;

4) транспортный – это уровень поддержки интернет-протоколов (IP). Основу этих протоколов составляют TCP (Transmission Control Protocol) и UDP (User Datagram Protocol);

5) сессионный – обеспечивает управление диалогом двух взаимодействующих сторон;

6) представлений – на нем решается задача приведения форматов данных к форматам, понятным приложениям высшего уровня;

7) приложений – на нем приложения обмениваются данными по глобальной сети.

Сокеты находятся над транспортным уровнем, а над ними – протоколы сеансового уровня, которые ориентированы на решение конкретных задач или сервисов Интернета. Например:

- FTP – сервис передачи файлов,
- HTTP – сервис передачи гипертекстовых страничек (Интернет-страниц),
- SMTP и POP3 – сервисы электронной почты и т.д.

Сокеты позволяют, не вдаваясь в детали передачи информации, обеспечивать решение большого круга задач, используя Интернет. Сокеты представляют собой модель одного конца сетевого соединения.

В Delphi имеется огромный выбор средств для работы на уровне сокетов. Во-первых, можно впрямую использовать Windows API. Все процедуры и функции для этого даны в модуле WinSock.pas. Исторически первыми компонентами для работы с сокетами были TClientSocket и TServerSocket. Их описание дано в модуле ScktComp.pas. Но в последних версиях Delphi появились более универсальные компоненты TTCPClient, TTCPServer и TUDPSocket (модуль Sockets.pas). Одновременно в палитре компонентов Delphi появились целые страницы компонентов Indy (Internet Direct).

Механизм соединения при помощи сокетов следующий. На клиентском конце создается сокет, для инициализации связи ему задается путь к серверному сокету, с которым необходимо установить соединение. В сетях TCP/IP путь задается двумя параметрами: адресом или равноценным ему именем хоста и номером порта. Хост – это система, в которой запущено приложение, содержащее сокет. Адрес хоста задается в виде четырех цифр в диапазоне от 0 до 255, разделенных точками. Вместо адреса можно задать имя хоста согласно правилам UNC (Universal Naming Convention), например http://bsuir.unibel.by. Соответствие адреса и имени хоста поддерживается специальным сервисом DNS (Domain Naming System). У компонента клиента адрес сервера задается двумя свойствами: Address и Port. На стороне сервера запускается свой сокет, который прослушивает (listening) состояние порта связи. После получения запроса от клиента устанавливается связь. Одновременно на сервере создается новый сокет для продолжения прослушивания.

В заключение можно сделать вывод, что сокеты – это основа создания Интернет-приложений или Web-служб.

26. ТЕХНОЛОГИЯ COM

Модель объектных компонентов – COM (Component Object Model) является базовой технологией операционной системы Windows. Это дальнейшее развитие технологии DDE, а затем и OLE. Технология COM представляет собой строго регламентированную спецификацию, определяющую требования к взаимодействующим программам. Она позволяет легко общаться программам, созданным на разных языках программирования и выполняющимся как на одном, так и на разных компьютерах.

Ключевым моментом, на котором основана модель COM, является понятие интерфейса.

26.1. Интерфейс

Интерфейс – это определение методов и свойств, которые могут быть реализованы классом. Все методы интерфейса рассматриваются как виртуальные и абстрактные по определению. Первоначально определенный интерфейс нельзя изменять в дальнейшем. Любое внесение изменений в интерфейс дает уже другой интерфейс. Каждый интерфейс определяется специальным, глобальным идентификатором – GUID (Global Unique IDentifier)– шестнадцатибайтовым числом. Для интерфейсов такой идентификатор носит название IID (Interface Identifier). В Delphi при нажатии клавиш Ctrl+Shift+G новый GUID формируется автоматически по специальным правилам, исключающим повторение значения GUID–а. Интерфейсы могут наследоваться. Прародителем всех интерфейсов является IUnKnown – неизвестный. В Windows он определен следующим образом:

```
Type IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}'] // GUID интерфейса
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    end;
```

Первая функция получает в качестве входного параметра идентификатор интерфейса. Если реализация такого интерфейса существует, то функция:

1) возвращает ссылку на него в параметре Obj;
2) вызывает метод _AddRef полученного интерфейса;
3) возвращает 0.

В противном случае функция возвращает код ошибки – ENOINTERFACE.

Функция _AddRef увеличивает счетчик ссылок на интерфейс на единицу и возвращает новое значение счетчика, а функция _Release уменьшает значение счетчика ссылок на интерфейс на единицу и возвращает новое значение счетчика. При достижении счетчиком нулевого значения она должна освободить память, занятую реализацией интерфейса.

В модуле System.pas объявлен класс TInterfaceObject, реализующий интерфейс IUnknown и его методы. Кроме того, поддержка интерфейсов реализована в базовом классе TObject.

Реализацией интерфейса в Delphi всегда выступает класс. Для этого в объявлении класса необходимо указать, какие интерфейсы он реализует. Например:

```
Type  TMyClass=Class(TComponent, IMy1Interface, IMy2Interface)
        // Реализация методов
        end;
```

В данном примере класс TMyClass является наследником класса TComponent и реализует два интерфейса: IMy1Interface и IMy2Interface. Таким образом, класс поддерживает множественное наследование интерфейсов и единичное

наследование классов. Когда класс реализует интерфейс, он ответственен за реализацию всех методов не только этого интерфейса, но и методов всех предков интерфейса.

Имя интерфейса принято начинать с буквы «I». Зарегистрированные в системе IID и имена интерфейсов хранятся в ветке реестра – HKEY_CLASSES_ROOT\Interface.

26.2. COM-сервер

COM-сервер – это специальным образом оформленное и зарегистрированное приложение, которое позволяет клиентам запрашивать создание реализованных в сервере объектов. Сервер может быть выполнен в виде либо динамической библиотеки (*.dll), либо исполняемого файла (*.exe).

Сервер в виде dll всегда выполняется в адресном пространстве активизировавшего его приложения. При этом производится быстрый вызов методов, но он менее надежен, так как не защищен от ошибок вызывающего его приложения.

Сервер в виде exe файла представляет собой обычный исполняемый файл, в котором реализована возможность создания COM-объектов по запросу клиента. Примером такого сервера может служить пакет Microsoft Office, приложения которого являются COM-серверами.

COM реализует механизм автоматического поиска серверов по запросу клиента. Каждый COM-объект имеет уникальный идентификатор Class Identifier (CLSID), построенный так же, как и GUID. Все COM-объекты регистрируются ОС Windows в ветке реестра HKEY_CLASSES_ROOT\CLSID. Для каждого сервера здесь прописывается информация, нужная для нахождения и загрузки его модуля, соответствующее «дружественное» имя или Programmatic Identifier (PROGID).

Серверы в виде исполняемых файлов автоматически регистрируются в системе при первом запуске программы на компьютере. Для регистрации DLL-серверов используется программа Regsvr32 из системы Windows или процедура RegisterCOMServer из поставки Delphi.

Windows – многозадачная и многопоточная среда с вытесняющейся многозадачностью. Для COM это означает, что клиент и сервер могут оказаться в разных процессах или потоках и нужно обеспечить обращение к серверу множества клиентов в непредсказуемое время. Технология COM решает эту проблему с помощью «комнат» (Apartments), в которых выполняются COM-клиенты и COM-серверы. Комнаты бывают однопоточные (Single Threaded Apartment – STA) и многопоточные (Multiple Threaded Apartment – MTA).

При STA COM создает невидимое окно, и при вызове любого метода COM-сервера Windows посылает сообщение этому окну с помощью функции PostMessage. Таким образом, образуется очередь сообщений для данного сервера. Это позволяет не заботиться о синхронизации методов и при этом нет опасности одновременного доступа нескольких методов к одному полю.

Для MTA в одной «комнате» может быть сколько угодно потоков и объектов, но при этом надо заботиться о синхронизации вызовов методов.

Разработка таких серверов сложна, но они обладают более высоким быстродействием.

Для EXE-серверов клиент и сервер находятся в разных адресных пространствах. Для их взаимодействия в «комнате» клиента создается представитель сервера – Proxy, который представляет собой объект, экспортирующий запрошенный интерфейс. Одновременно в комнате сервера создается объект-заглушка (Stub), принимающий вызовы от proxy и транслирующий их в вызовы сервера. Процесс обмена между клиентом и сервером называется Marshalling. При этом клиент и сервер могут находиться на разных компьютерах.

Процесс запуска COM-сервера по запросу клиента происходит следующим образом:

1) в реестре по запрошенному CLSID ведется поиск записи регистрации сервера;

2) из этой записи получаем имя исполняемого модуля сервера;

3) если это – исполняемый файл, то он запускается на выполнение. Любое приложение, реализующее COM-сервер, при старте регистрирует в системе интерфейс «фабрики объектов». После запуска и регистрации COM получает ссылку на «фабрику объектов»;

4) если это – DLL, то она загружается в адресное пространство вызвавшего процесса и вызывается ее функция DLLGetClassObject, возвращающая ссылку на реализованную в DLL «фабрику объектов»;

5) «фабрика объектов» – это COM-сервер, реализующий интерфейс IClassFactory. Ключевым методом этого интерфейса является CreateInstance, который и создает экземпляр требуемого объекта;

6) COM вызывает метод CreateInstance и передает полученный интерфейс клиенту.

Для создания COM-сервера в Delphi нужно в меню пройти путь

File → New → ActiveX

На страничке ActiveX находятся значки для управления процессом создания различных видов COM-объектов. Можно создать COM-сервер в виде EXE-файла путем добавления COM-объекта к текущему проекту. Для создания COM-сервера в виде DLL-библиотеки можно выбрать значок ActiveX Library. Обычно создание COM-сервера сопровождается созданием соответствующей библиотеки типов, предназначенной для документирования информации об объектах, интерфейсах, функциях и т.д. Для ее создания на страничке ActiveX есть значок Type Library. Традиционно для создания библиотек типов предназначен язык описания интерфейсов IDL (Interface Description Language). В Delphi в качестве основного варианта используются синтаксис и операторы языка Object Pascal. Для работы с библиотекой типов используется специальный редактор, который можно найти, пройдя путь Windows → *_TLB.pas Editor. Библиотека создается как обычный текстовый файл с расширением Pas и окончанием _TLB в названии. Редактор типов может экспортировать библиотеку из текстового формата в формат IDL (двоичный), что обеспечивает использование созданных объектов в любых других приложениях Windows. Библиотека типов дает возможность при создании клиента контролировать правильность

обращения к методам сервера еще на этапе проектирования. С другой стороны, можно импортировать в проект любую библиотеку типов, зарегистрированную в системе. Для этого следует пройти путь Project → Import Type Library.

27. ТЕХНОЛОГИЯ АВТОМАТИЗАЦИИ

27.1. Основы OLE Automation

Стандарт COM основан на едином для всех языков программирования формате таблицы, описывающей ссылки на методы объекта, реализующего интерфейс. Однако вызов методов из этой таблицы доступен только для тех языков программирования, для которых есть трансляторы, переводящие весь текст программы в язык машинных кодов. Такой доступ к методам называется ранним связыванием. Для использования возможностей COM в интерпретирующих (скриптовых) языках программирования, таких, как VBScript, Java, Perl и др., была разработана технология OLE Automation, которая позволяет вызывать методы объекта не по адресу, а по имени.

Технология автоматизации базируется на COM, однако накладывает на COM-серверы ряд дополнительных требований:

1. Интерфейс, реализуемый COM-сервером, должен наследоваться от IDispatch.
2. Должны использоваться типы данных из числа поддерживаемых OLE Automation (Byte, SmallInt, Integer, Currency, Single, Real48, Real, Double, ShortString, AnsiString, TDateTime, WordBool, Variant, OLEVariant).
3. Все методы должны быть процедурами или функциями, возвращающими значение типа HResult.
4. Все методы должны иметь соглашение о вызовах, соответствующее директиве safecall.

Центральным элементом технологии автоматизации является интерфейс IDispatch. В нем определяются методы GetIdsOfNames и Invoke, позволяющие клиенту узнать, поддерживает ли сервер метод с указанным именем, а затем, если метод поддерживается, вызвать его. Такой способ доступа к методам называется поздним связыванием. При позднем связывании не требуется наличие библиотеки типов, но при этом и не производится контроль правильности вызова метода на этапе проектирования программы, к тому же вызов методов осуществляется несколько медленнее, чем при раннем связывании.

Для контроля правильности вызова методов на этапе проектирования используется дополнительный диспинтерфейс (DispInterface). Он имеет тот же GUID, что и основной интерфейс, только его имя дополняется окончанием disp и каждому методу или свойству присваивается уникальный номер, который называется диспетчерским идентификатором (DISPID). Теперь методы сервера можно вызывать прямо через Invoke, передавая ему значение DISPID соответствующего метода. Следует отметить, что в описании диспинтерфейса могут присутствовать не все методы и свойства основного интерфейса.

Например, в библиотеке типов MS Office дано следующее определение интерфейса ICommandBarsEvents и его диспинтерфейса:

```
ICommandBarsEvents = interface(IDispatch)
  ['{55F88892-7708-11D1-ACEB-006008961DA5}']
  procedure OnUpdate; stdcall;
  end;
ICommandBarsEventsDisp = dispinterface
  ['{55F88892-7708-11D1-ACEB-006008961DA5}']
  procedure OnUpdate; dispid 1;
  end;
```

Такое двойное описание интерфейса позволяет клиентам обращаться к методам сервера или по стандартам COM раннего связывания через таблицу виртуальных методов (VTable) или использовать позднее связывание через методы IDispatch. Большинство OLE-серверов реализуют двойной интерфейс.

27.2. Примеры использования серверов автоматизации

Позднее связывание

В данном примере создается Ole объект Word приложения из MS Office, в нем создается новый документ, в который записывается текущая дата, и он сохраняется на диске.

```
Users ….ComObj;
procedure TForm1.Button1Click(Sender: TObject);
var v:olevariant;
begin
  v:=CreateOleObject('Word.Application');  // Создание Word сервера
  V.Application.documents.add(); // Создание нового документа
// Записываем текущую дату
  v.Application.ActiveDocument.Range.InsertAfter(Now);
// Сохраняем документ на диске
  v.Application.ActiveDocument.SaveAs('d:\my\d1.doc');
  v.Application.Quit(True,0); // Закрываем сервер
  end;
```

На этапе проектирования данной программы Delphi не дает никаких подсказок по написанию методов и свойств объектов и не проверяет правильность их использования. Ошибки могут возникнуть только на этапе выполнения программы.

Использование Dispatch-интерфейса

Здесь создается Word-сервер с помощью методов Dispatch-интерфейса и открывается существующий на диске файл. При этом не используются никакие компоненты доступа к серверам автоматизации:

```
Users …comobj, OleServer, WordXP;
procedure TForm1.Button2Click(Sender: TObject);
var v:OLEVariant;
   Word1:_ApplicationDisp; // Определяем указатель на DispInterface
begin
  v:='d:\my\d1.doc';  // Задаем имя файла
// Создаем Word-сервер с помощью метода диспетчерского интерфейса
```

```
  Word1:=CoWordApplication.Create as _ApplicationDisp;
// Открываем документ
  (Word1.Documents as DocumentsDisp).Open(v,
// Далее идут пустые параметры типа OleVariant
  Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
  Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
  Emptyparam,Emptyparam,Emptyparam,Emptyparam);
// Показываем имя файла документа и путь к нему
  ShowMessage((Word1.Application as _application).
  ActiveDocument.Path+'  '+
  (Word1.Application as _application).ActiveDocument.Name);
// Закрываем сервер
  Word1.Quit(Emptyparam,Emptyparam,Emptyparam);
  end;
```

В этом примере на этапе проектирования Delphi подсказывает, какие есть методы и свойства у объекта, и контролирует их использование.

Использование VTable интерфейса

В этом примере из странички компонентов Servers на форму помещается компонент WordApplication1. После нажатия кнопки Button3 открывается документ d1.doc и в нем производится замена набора символов «@1» на текущую дату и время. Измененный таким образом документ записывается в файл d2.doc:

```
procedure TForm1.Button3Click(Sender: TObject);
var v1,v2,oldstr,newstr,replaceend:OleVariant;
begin
  v1:='d:\my\d1.doc';// Исходный документ
  v2:='d:\my\d2.doc';// Выходной документ
  WordApplication1.Documents.Open(v1, // Открываем документ
  Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
  Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
  Emptyparam,Emptyparam,Emptyparam,Emptyparam);
  Replace:=1;   // Определяем параметр замены
  Oldstr:='@1'; // Заменяемый шаблон
  Newstr:=Datetostr(now); // Строка замены
  While   // Цикл замен
    WordApplication1.ActiveDocument.Range(Emptyparam,Emptyparam).
    Find.Execute(oldstr,
    Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
    Emptyparam,Emptyparam,Emptyparam,Newstr,replace,
    Emptyparam,Emptyparam,Emptyparam,Emptyparam)
    do;
  WordApplication1.ActiveDocument.SaveAs(v2,   // Сохраняем документ
  Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
  Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam,
```

```
    Emptyparam,Emptyparam,Emptyparam,Emptyparam,Emptyparam);
  // Закрываем сервер
    WordApplication1.ActiveDocument.Close(Emptyparam,Emptyparam,
    Emptyparam);
    end;
```

В данном примере методы сервера вызываются напрямую с использованием таблицы VTable, что значительно ускоряет работу программы по сравнению с применением позднего связывания.

27.3. Компоненты ActiveX

Задачу визуального представления COM-объектов выполняет технология ActiveX. С точки зрения программиста, ActiveX – это черный ящик, обладающий свойствами, методами и событиями. С точки зрения модели объектов COM, элемент управления ActiveX – это сервер, поддерживающий технологию автоматизации, реализованный в виде динамической библиотеки, исполняемый в адресном пространстве клиента и допускающий визуальное редактирование.

С Delphi поставляется ряд компонентов ActiveX, которые находятся на одноименной страничке палитры компонентов.

Можно включить в палитру компонентов Delphi любой компонент ActiveX, зарегистрированный в системном реестре. Для этого нужно пройти путь Component → Import ActiveX Control, выбрать нужный компонент и нажать клавишу Install. С другой стороны, можно любой компонент или любую форму Delphi сделать компонентом ActiveX. В первом случае, пройдя путь File → New, на страничке ActiveX выберем значок ActiveX Control. В появившемся окне зададим необходимые параметры для реализации нового ActiveX компонента. Во втором случае на страничке ActiveX выберем значок Active Form. Задав имя нового компонента ActiveX, имя модуля Unit для реализации компонента, имя проекта, тип потоковой модели элемента ActiveX и сохранив на диске проект, мы получим 6 файлов с расширениями *.dpr, *.pas, *_tlb.pas, *.tlb, *.dfm, *.res, которые описывают:

- *.dpr – DLL библиотеку, реализующую COM сервер,
- *.pas – реализацию ActiveX формы,
- файл с окончанием _tlb и расширением pas – библиотеку типов COM-объекта в формате Delphi,
- *.tlb – библиотеку типов COM объекта в двоичном формате,
- *.dfm – форму в формате Delphi,
- *.res – ресурсы формы.

Теперь в среде Delphi можно переносить на форму любые компоненты и настраивать их свойства и события, одновременно будет изменяться содержимое описанных выше файлов. Остается только зарегистрировать новый ActiveX компонент и включить его в состав одного из пакетов.

28. ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

Динамически подключаемые библиотеки (Dynamic Link Library – DLL) играют важную роль в функционировании ОС Windows и прикладных программ.

DLL представляют собой файлы с откомпилированным исполняемым кодом, который используется приложениями и другими DLL. Реализация многих функций ОС вынесена в динамические библиотеки, которые используются по мере необходимости, обеспечивая тем самым экономию адресного пространства. DLL загружается в физическую память один раз, когда к ней обращается какой-либо процесс. Для всех процессов, использующих DLL, в их виртуальной памяти отображаются только образы этих библиотек, при этом для каждого процесса создается впечатление, что DLL загружена именно в его адресное пространство. По существу, динамические библиотеки отличаются от исполняемых файлов только одним: они не могут быть запущены самостоятельно. Следует отметить, что DLL не имеет своего стека и очереди сообщений Windows. При вызове подпрограмм из DLL используется стек вызывающей программы. Для того чтобы динамическая библиотека начала работать, необходимо, чтобы она была вызвана уже запущенной программой или работающей DLL.

Обычно в динамические библиотеки выносятся группы функций, которые применяются для решения сходных задач. Динамическая библиотека может использоваться несколькими приложениями, при этом не обязательно, чтобы все они были созданы при помощи одного языка программирования.

Разновидностью динамических библиотек являются пакеты Delphi, предназначенные для хранения кода компонентов для среды разработки и приложений. Применение динамических библиотек позволяет добиться ряда преимуществ:

- уменьшаются размер исполняемого файла приложения и занимаемые им ресурсы;
- функции DLL могут использовать несколько процессов одновременно;
- управление динамическими библиотеками возлагается на операционную систему;
- внесение изменений в DLL не требует перекомпиляции всего проекта;
- одну DLL могут использовать программы, написанные на разных языках программирования.

28.1. Создание DLL

Одним из способов создания динамической библиотеки является прохождение пути File → New → Other → New и использование значка «DLL Wizard». Будет создан пустой модуль с первым ключевым словом Library и заданным именем библиотеки. Библиотеку следует сохранять с тем же именем и расширением pas. Структура библиотеки похожа на структуру модуля Unit. Внутри библиотеки можно описывать типы, константы и переменные, но они будут доступны только внутри модуля. Процедуры и функции, описанные в библиотеке, будут доступны извне модуля, если их перечислить в разделе

Exports. В конце модуля библиотеки между словами Begin и End можно расположить операторы исполняемой части модуля, но они будут выполнены только один раз, при первой загрузке модуля в память. В общем случае библиотечный модуль может иметь вид

```
Library MyLib;
Users SysUtils,Classes;
Function F1(….):тип1;
Begin
……..
 end;
Procedure P2(….);
Begin
……..
 end;
Procedure P3(…..);
Begin
……..
 end;
Exports
  F1,
  P2 index 2,
  P3 index 3 name ’Proc3’;
Begin
 End.
```

Здесь описана библиотека с именем MyLib, в которую входят три подпрограммы F1, P2 и P3. Для внешнего использования можно вызывать программы или по именам – F1, P2 и Proc3, или по индексам – 2 и 3. После трансляции этого модуля будут создана библиотека MyLib.DLL.

28.2. Использование DLL

При программировании можно использовать статическую или динамическую загрузку DLL. При статической загрузке следует в разделе описаний основной программы вставить для приведенного выше примера следующие операторы:

```
Function F1(….):тип1; external ’MyLib’;
Procedure P2(….); external ’MyLib’;
Procedure Proc3(…..); external ’MyLib’;
```

Они определяют названия внешних процедур и функций и библиотеку, из которой их следует импортировать при запуске программы.

При динамической загрузке DLL следует в разделе типов определить типы импортируемых процедур и функций, зарезервировать переменную типа THandle для указателя на динамическую библиотеку и переменные для указателей на импортируемые процедуры и функции, загрузить библиотеку с помощью функции LoadLibrary и определить указатели на процедуры и

функции с помощью функции GetProcAddress. Для нашего примера это может выглядеть следующим образом:

```
Procedure TForm1.Button1Click(Sender:TObject);
Type TF1=Function(….):тип1;
     TP2=Procedure(….);
     TP3=Procedure(….);
Var  LibH:THandle;
     Fun1:TF1;
     Proc2:TP2;
     Proc3:TP3;
Begin
     LibH:=LoadLibrary('MyLib');
     @Fun1:=GetProcAddress(LibH,'F1');
     @Proc1:=GetProcAddress(LibH,Pchar(LongInt(2)));
     @Proc3:=GetProcAddress(LibH,'Proc3');
     ……
     FreeLibrary('MyLib');
End;
```

Для статической загрузки DLL проще создать интерфейсный модуль, в котором описать все типы, используемые в DLL, и определить импортируемые процедуры и функции с указанием, из каких DLL они импортируются. В основной программе достаточно будет только указать имя интерфейсного модуля в операторе Users.

28.3. Пример написания DLL

Рассмотрим пример практического использования создаваемой DLL библиотеки для проверки регистрационного номера любой программы при ее инсталяции с помощью поставляемой вместе с Delphi программы Install Shield Express. В разделе Dialogs выберем страничку Customer Information и в ней зададим следующие параметры:

Show Serial Number	Yes
Serial Number Template	????-????
Serial Number Validation DLL	d:\my\MyLib.DLL
Validate Function	ValNum
Success Return Value	24
Retry Limit	3

В данном случае серийный номер устанавливаемой программы будет состоять из двух частей по четыре произвольных символа в каждой части. Проверка номера будет проводиться программой ValNum из библиотеки MyLib. При правильно введенном регистрационном номере эта функция должна возвращать значение 24. Допускается не больше трех попыток введения правильного номера.

Для решения этой задачи можно написать следующую DLL:

```
Library MyLib;
uses
```

```
  SysUtils, Classes,
{$F+}
// При вызове функции Val используется стандартное для Windows
// соглашение о передаче параметров – stdcall
Function VAL(h:word;ps1,ps2,ps3,ps4:Pchar):longint; stdcall;
const sn='ABCD1234'; // Правильное значение серийного номера
var s:string[20];
Begin
  s:=strpas(ps3);   // Переводим третий параметр в обычную строку
  if s:=sn then val:=24 else val:=-1;   // Проверяем серийный номер
  end;
Exports
  VAL name 'ValNum';// Внешнее имя функции – ValNum
begin
end.
```

29. РАБОТА С БАЗАМИ ДАННЫХ

Среда визуального программирования Delphi изначально предназначалась для работы с базами данных (БД) в отличие от ее предшественника – VisualBasic. Фирма Borland разработала свой процессор баз данных – BDE (Borland Database Engine), который позволял создавать программы, работающие практически со всеми типами баз данных. В настоящее время Delphi остается самым простым, надежным и перспективным средством создания программ для работы с базами данных. Delphi стала надстройкой над процессорами баз данных, созданными другими фирмами, в частности MS Jet 4.0 и MS ADO. Это существенно повышает привлекательность среды Delphi даже в сравнении с C–Builder или Visual–C++.

29.1. Основные определения

База данных – это набор взаимосвязанных таблиц, которые могут храниться в отдельных файлах (Dbase, Paradox и т.д.) или в одном файле (Access).

Таблица – набор полей (столбцов), количество которых почти постоянно, и записей (строк), количество которых меняется в процессе работы.

Рассмотрим пример базы данных, содержащей две таблицы. Одна таблица будет представлять список сотрудников – TSPIS, а вторая – список должностей – TDOL:

Таблица TSPIS

ID	FIO	CODEDOL	STAG	DATEB	OKLAD
01	Иванов	2	10	11.02.1970	300 000
02	Сидоренко	1	5	21.11.1981	200 000
…	…	…	…	…	

В этой таблице шесть полей: ID – номер сотрудника; FIO – фамилия, имя и отчество; CODEDOL – код должности; STAG – стаж работы; DATEB – дата рождения; OKLAD – оклад сотрудника.

Таблица TDOL

CODE	NAMEDOL
1	Инженер
2	Техник
….	…..

В таблице TDOL всего два поля: CODE – код должности, NAMEDOL – наименование должности. Эти две таблицы будут в дальнейшем связаны по полю CODEDOL таблицы TSPIS и полю CODE таблицы TDOL, это позволяет существенно экономить память, занимаемую таблицами, так как код должности занимает в основной таблице гораздо меньше места, чем наименование должности. Обычно эти связываемые таблицы называют справочниками.

Первые поля таблиц определяют **ключ записи,** или строки. Его значение должно быть уникально и не может повторяться. Строки в таблицах реляционных баз данных образуют множество, они не имеют последовательных номеров, а для их идентификации используется значение ключа.

Индексные поля определяют порядок сортировки строк таблицы. Индекс может залаваться одним полем, несколькими полями или выражением от значений полей. Индексы позволяют значительно ускорить поиск информации в таблице по значению индексного поля за счет предварительной сортировки записей и использования бинарного поиска. Например, если таблица содержит 1 000 записей, то для поиска нужной строки в среднем надо произвести 500 сравнений. Бинарный поиск уменьшает число сравнений до 10, так как $1\ 000 \sim 2^{10}$. Индексы приводят к появлению дополнительных таблиц в базе данных.

Домен – это диапазон или набор значений какого-либо поля.

Сессия – представляет собой канал связи с базой данных. Приложение может иметь несколько сессий. Сессии являются владельцами курсоров и блокировок.

Курсор – специальный объект, в котором хранятся текущий набор строк таблицы и информация о закладках (BookMark). Все операции с таблицами осуществляются только через курсор. Он позволяет ускорить работу с таблицами, т.к. его буфер хранит не одну, а 100 … 1 000 строк таблицы, что позволяет существенно реже обращаться к файлам таблиц. Курсор может находиться как на стороне сервера, так и на стороне клиента.

Клиент – приложение, которое задает вопросы.

Сервер – приложение, которое отвечает на вопросы.

Запрос – оператор на языке SQL (Structured Query Language).

Транзакция – система взаимосвязанных действий по изменению базы данных. Она должна быть выполнена до конца или отменена полностью, т.е. должна быть предусмотрена возможность отката назад. Транзакция требует наличия дополнительной памяти для хранения временных изменений в базе данных.

Хранимая процедура – набор SQL запросов, хранимых на сервере, который можно вызвать со стороны клиента по одному имени.

29.2. Взаимодействие приложения на Delphi с базами данных

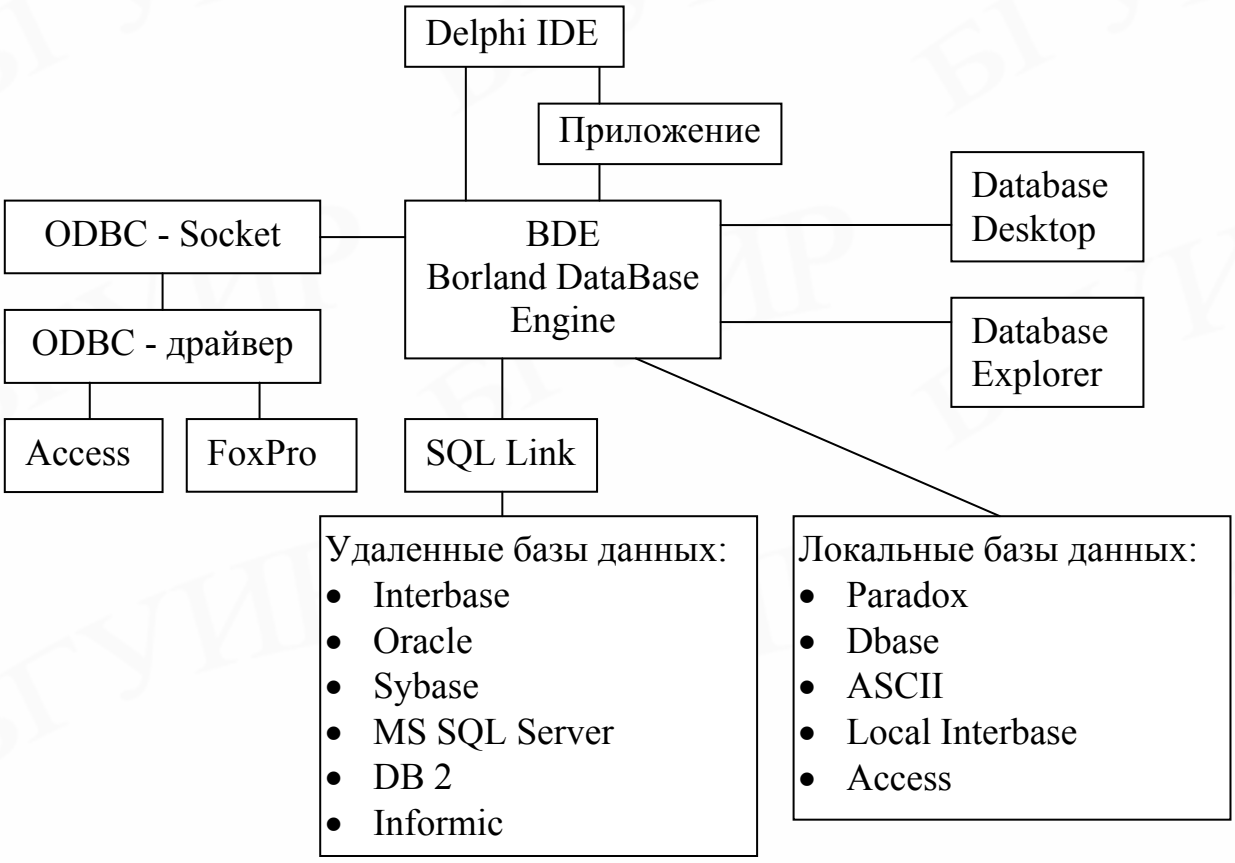


Рис.29.1 Схема взаимодействия приложения с базами данных через BDE

Здесь:

IDE – интегрированная среда разработки Delphi,

Database Desktop – программа прямой работы с базой данных (создание, просмотр, корректировка),

Database Explorer – программа настройки параметров базы данных и BDE,

ODBC – использование технологии Open Data Base Connectivity для доступа к некоторым локальным базам данных,

SQL Link – драйверы связи с удаленными клиент-серверными базами данных.

Как видно из рис.29.1, центральное место во взаимодействии приложения с базой данных занимает BDE. Это хорошо и одновременно плохо: при установке разработанной программы у заказчика придется одновременно установить и настроить на его компьютере BDE, а это около 15 Мбайт дополнительной памяти и возможные трудности с автоматической настройкой BDE.

Разработанный фирмой Borland процессор баз данных BDE в последнее время перестал обновляться. Небольшие фирмы не могут сравниться по возможностям с такими гигантами, как Microsoft. Поэтому в новых разработках небольшие фирмы все чаще стали использовать готовые решения от крупных фирм. Так, в Delphi появилась страничка компонентов доступа к базам данных ADO (ActiveX Data Object).

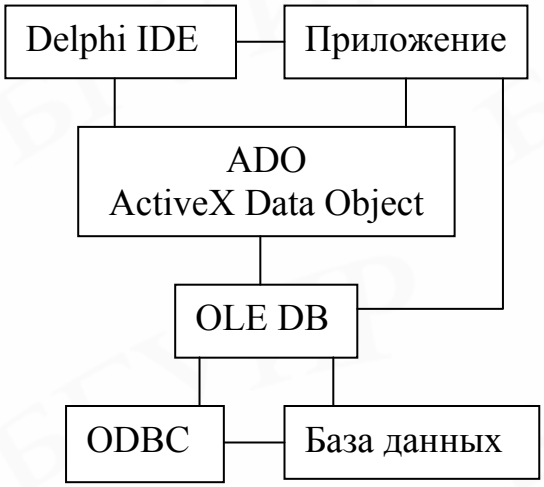


Рис.29.2. Схема взаимодействия с базами данных с использованием ADO

Здесь центральное место во взаимодействии с базами данных занимает OLE DB (Object Linking and Embedding Data Base). Однако это программное обеспечение входит как составная часть операционной системы Windows и не требует дополнительной установки. Вместе с Windows поставляются программы связи с различными типами баз данных. Эти программы называются провайдерами. Например, для базы данных Access поставляется программа-провайдер MS Jet 4.0 OLE DB. Использование компонентов ADO позволяет полностью отказаться от BDE.

29.3. Компоненты взаимодействия с базами данных

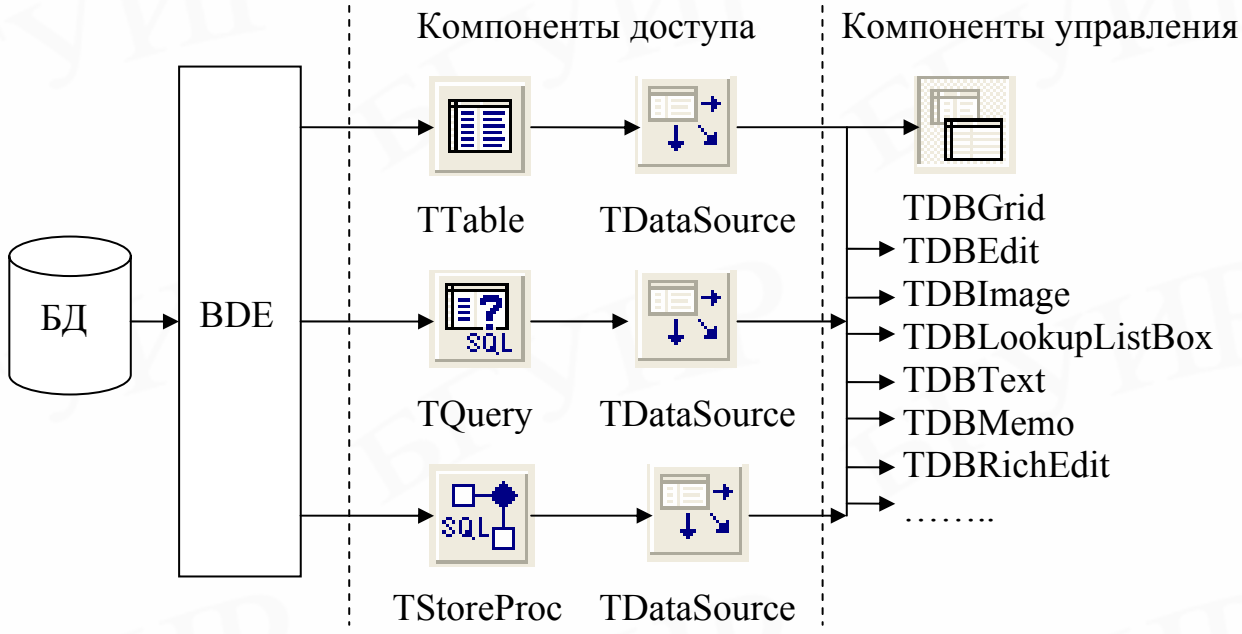


Рис.29.3. Схема взаимодействия компонентов с использованием BDE

Здесь приведены следующие компоненты:

TTable – доступа к таблице базы данных,

TQuery – запроса к базе данных,

TStoreProc – хранимой процедуры с запросом к базе данных,

TDataSource – связи компонентов доступа с компонентами управления и отображения данных,

TDBGrid – отображения данных в виде таблицы.

Компонент управления и отображения данных достаточно много, поэтому на схеме (рис 29.3) показаны только некоторые из них.

Во время проектирования невизуальные компоненты, в частности компоненты доступа к данным, помещенные на форму, представляются маленькими значками. Но из-за большого количества таких невизуальных компонентов утрачивается наглядность формы. Компоненты доступа к данным обычно располагаются в специальном модуле данных – TDataModule. Создать такой модуль можно, пройдя путь File → New → Data Module.

Как следует из рис.29.3, для отображения какой-нибудь таблицы базы данных на форму следует поместить как минимум три компонента, например: Table1, DataSource1 и DBGrid1. В свойстве TableName компонента Table1 нужно указать имя таблицы базы данных, в свойстве DataSet компонента DataSource1 – источник данных Table1, а в свойстве DataSource компонента DBGrid1 – источник отображаемых данных DataSource1.

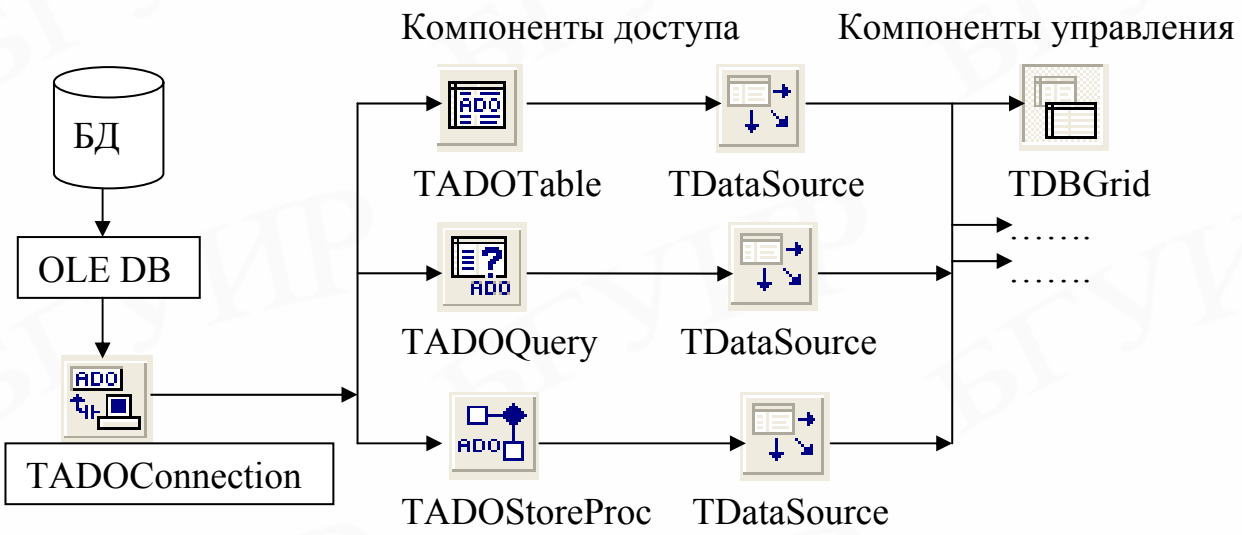


Рис.29.4. Схема взаимодействия компонентов с использованием ADO

На рис.29.4 приведены аналогичные компоненты по сравнению с предыдущим рисунком, только со странички ADO. В их тип входит сочетание букв – ADO. Следует отметить, что в свойствах компонента TADOConnection не следует явно указывать нахождение базы данных в свойстве ConnectionString и задавать программу-провайдер в свойстве Provider. Лучше создать специальный текстовый файл связи с базой данных с расширением *.URL. Затем с помощью программы MS Explorer, которая воспринимает файлы с расширением URL как файлы связи с базой данных, можно настроить этот файл на конкретную базу данных, выбрать программу-провайдер и задать ее параметры. Это позволяет легко переносить базы данных на любой диск без перетрансляции разработанного приложения.

Иерархия классов доступа к базам данных

Все классы доступа к базам данных являются наследниками класса TComponent:

```
TComponent                  // Компонент
  TSession                  // Сессия
  TDataSource               // Источник данных
  TCustomConnection         // Базовый класс для соединения с БД
    TDataBase               // База данных
    TADOConnection          // ADO соединение с БД
  TDataSet                  // Класс навигации по таблицам БД
    TBDEDataSet             // Набор данных, связанных с BDE
      TDBDataSet            // Редактирование и навигация по БД
        TTable              // Таблица
        TQuery              // Запрос
        TStoredProc         // Хранимая процедура
    TCustomADODataSet       // Базовый класс связи с БД через ADO
      TADODataSet           // Редактирование и навигация по ADO БД
      TADOTable             // ADO таблица
      TADOQuery             // ADO запрос
      TADOStoredProc        // ADO хранимая процедура
  TField                    // Базовый класс поля таблицы
    TStringField            // Строковое поле (до 256 символов)
    TNumericField           // Базовый класс числового поля
      TIntegerField         // Базовый класс целочисленного поля
        TTSmallIntField     // Малое целое поле (от –32 768 до 32 767)
        TWordField          // Целое число без знака (от 0 до 65 535)
        TautoIncField       // Автоинкрементное поле (4 байта)
      TLargeIntField        // Целое длинное поле (8 байт)
      TFloatField           // Поле вещественных чисел (8 байт)
        TCurrencyField      // Поле денежных величин (8 байт)
      TBCDField             // Двоично-десятичное поле (8 байт)
    TBooleanField           // Логическое поле
    TDateTimeField          // Базовый класс даты и времени
      TDateField            // Поле даты
      TTimeField            // Поле времени
    TBinaryField            // Базовый класс бинарных данных
      TBytesField           // Массив байт (16 байт)
        TVarBytesField      // Поле произвольного массива байт (<65 535)
    TBlobField              // Поле массива байт неопределенной длины
      TMemoField            // Поле текста неопределенной длины
      TGraphicField         // Поле картинки неопределенной длины
```

Следует отметить, что при всем многообразии типов полей следует по возможности использовать для столбцов строковой тип – TStringField. Это

облегчает программирование и упрощает работу с таблицами в среде MS Access.

Класс TField

Для каждого столбца набора данных автоматически создается потомок класса TField с типом поля столбца. С помощью редактора полей можно создать вычисляемые поля. Рассмотрим некоторые свойства полей:

- *Property AsBoolean:Boolean;* – содержимое поля воспринимается как логическое значение,
- *Property AsDateTime:TDateTime;* – содержимое поля воспринимается как поле даты и времени,
- *Property AsFloat:Double;* – содержимое поля воспринимается как вещественное значение,
- *Property AsInteger:LongInt;* – содержимое поля воспринимается как целое число,
- *Property AsString:String;* – содержимое поля воспринимается как строка,
- *Property EditMask:String;* – шаблон маски ввода. Шаблон состоит из трех секций, разделенных «;». Первая секция – тело маски, вторая секция – символ управления, если это –«0», то можно вводить только цифры, иначе – любые символы, третья секция – символ, определяющий пробел для маски. Например, маска для ввода номера телефона может иметь вид **Table1.FieldByName('TEL').EditMask:='000-00-00;0;_';**
- *Property ValidChars:Set of Char;* – множество допустимых символов для ввода. Например, для ввода только русских букв можно определить это свойство следующим образом:
  Table1.FieldByName('FIO').ValidChars:=['А'..'я','Ё','ё'];

Класс TADOTable

Этот класс определяет ADO таблицу. Рассмотрим некоторые из основных свойств и методов этого класса:

- *Property Connection: TADOConnection;* – определяет соединение с БД,
- *Property TableName: WideString;* – определяет имя таблицы,
- *Property Active: Boolean;* – определяет состояние таблицы: открыта она или нет,
- *Property Eof: Boolean;* – определяет конец таблицы,
- *Property MasterSource: TDataSource;* – определяет главную таблицу для связываемой таблицы,
- *Property MasterFields: String;* – определяет поля связи в главной таблице,
- *Procedure Append;* – переводит таблицу в режим редактирования и добавляет в конец таблицы пустую строку,
- *Procedure First;* – устанавливает указатель таблицы на первую запись,
- *Procedure Last;* – устанавливает указатель таблицы на последнюю запись,
- *Procedure Next;* – переход к следующей записи таблицы,
- *Procedure Prior;* – переход к предыдущей записи таблицы,

- *Procedure Edit;* – устанавливается режим редактирования,
- *Procedure Post; virtual;* – запись изменений в текущую строку,
- *Function Seek(const KeyValues: Variant; SeekOption: TSeekOption = soFirstEQ): Boolean;* – поиск записи по значению ключевого или индексного поля. Например, в таблице TSpis найдем «Иванова» по индексному полю FIO: **If ADOTable1.Seek('Иванов', soFirstEQ) then ……;** Перед этим оператором нужно будет определить текущим индексное поле,
- *Function Locate(const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean; override;* – поиск записи по заданным полям. Например, найдем «Иванова» с 10-летним стажем работы:
  If ADOTable1.Locate('FIO,STAG',varArrayOf(['Иванов','10']),[])Then …;

29.4. Работа с локальной базой данных

Приведенные выше свойства и методы таблиц и полей можно использовать только при работе с локальной базой данных, которая может находиться или на вашем компьютере, или в локальной сети.

Рассмотрим фрагмент программы, которая в таблице TSpis увеличивает зарплату сотрудников в 2 раза:

```
With ADOTable1 Do Begin
  DisableControls; // Отключаем связь визуальных компонентов с таблицей
  First; // Устанавливаем указатель в начало первой записи
  While not Eof do Begin // Открываем цикл просмотра таблицы
  // Переводим оклад сотрудника из строки в целое число
    I:=SysUtils.StrToInt(FieldByName('OKLAD').AsString);
    I:=I*2; // Увеличиваем оклад в 2 раза
    Edit; // Включаем режим редактирования таблицы
    // Записываем новый оклад в текущую запись
    FieldByName('OKLAD').AsString:=SysUtils.IntToStr(I);
    Post; // Сохраняем изменения в таблице
    Next; // Переходим к следующей записи
    End; // Конец цикла
  // Восстанавливаем связь визуальных компонентов с таблицей
  EnableControls;
  End;
```

В этом примере мы использовали методы DisableControls и EnableControls для отключения и восстановления связи таблицы с визуальными компонентами, чтобы избежать прокрутки значений таблицы в визуальных компонентах при изменении значений зарплаты сотрудников во всей таблице.

30. ОСНОВЫ ЯЗЫКА SQL

Язык структурированных запросов SQL (Structured Query Language) используется в основном для работы с удаленными базами данных. Однако многие программы взаимодействия с БД , например MS Accesss, работают и с локальными базами данных через SQL.

30.1. Составные части SQL

SQL состоит из трех частей:

- DML (Data Manipulation Language) – язык манипулирования данными,
- DDL (Data Definition Language) – язык определения данными,
- DCL (Data Control Language) – язык управления данными.

DML

DML состоит из четырех основных команд:

- Select – выбрать записи из таблиц,
- Insert – вставить записи в таблицу,
- UpDate – обновить записи в таблице,
- Delete – удалить записи из таблицы.

DDL

DDL состоит из следующих команд:

- Create DataBase – создать базу данных,
- Create Table – создать таблицу,
- Create View – создать временную таблицу,
- Create Trigger – создать триггер, т.е. хранимую процедуру, которая будет вызываться автоматически при наступлении какого-то события,
- Create Index – создать индекс,
- Create Procedure – создать хранимую процедуру,
- Alter DataBase – модифицировать базу данных,
- Alter Table – модифицировать таблицу,
- Alter View – модифицировать временную таблицу,
- Alter Trigger – модифицировать триггер,
- Alter Index – модифицировать индекс,
- Alter Procedure – модифицировать хранимую процедуру,
- Drop DataBase – удалить базу данных,
- Drop Table – удалить таблицу,
- Drop View – удалить временную таблицу,
- Drop Trigger – удалить триггер,
- Drop Index – удалить индекс,
- Drop Procedure – удалить хранимую процедуру.

DCL

DCL управляет доступом к базе данных:

- Grand – дать права доступа к базе данных,
- Revoke – отобрать права доступа к базе данных.

Остановимся на рассмотрении только наиболее важной команды языка манипулирования данными – Select.

30.2. Команда SELECT

Эта команда предназначена для выбора информации из базы данных. Результатом работы команды Select (выбор) всегда является таблица. Общая форма данной команды имеет вид

Select [Distinct] <список полей>
From <список таблиц>
[Where <условия отбора>**]**
[Group By <поля> **[Having** <условия отбора для группировки>**]**
[Union <другой Select>**]**
[Order By <поля или номера полей>**];**

Например, для выбора всех фамилий из таблицы TSpis можно записать команду Select следующим образом:

Select FIO From Tspis;

В результате мы получим следующую таблицу:

FIO
Иванов
Сидоренко
…….

Если вместо списка имен полей поставить символ «*» (звездочка), то в итоговую таблицу будут включены все поля таблицы.

При описании полей выбора перед каждым именем поля можно вставить текстовую константу (литерал), которая будет выступать в роли «псевдо-столбца». Например:

Select ”Фамилия”, FIO, ” получает”, OKLAD, ” рублей” From TSpis;

Теперь мы получим следующую таблицу:

	FIO		OKLAD	
Фамилия	Иванов	получает	300 000	рублей
Фамилия	Сидоренко	получает	200 000	рублей
…..	…..	…..	…..	…..

Квалификатор As

Квалификатор As позволяет при выводе итоговой таблицы изменять названия столбцов и таблиц, например:

Select FIO as Фамилия, OKLAD as Зарплата From TSpis;

Эта команда даст следующую таблицу:

Фамилия	Зарплата
Иванов	300 000
Сидоренко	200 000
…..	……

Агрегатные функции

Эти функции позволяют производить вычисления над значениями какого-то столбца или подсчитать число строк в таблице:

- Sum – суммирование значений столбца;
- Min – определение минимального значения в столбце;
- Max – определение максимального значения в столбце;
- Avg – определение среднего значения от значений столбца;

- First – первое значение в столбце;
- Last – последнее значение в столбце;
- StDev – оценка среднеарифметического отклонения;
- StDevp – несмещенная оценка среднеарифметического отклонения;
- Var – оценка дисперсии;
- Varp – оценка несмещенной дисперсии;
- Count – число строк в таблице.

Рассмотрим пример расчета средней зарплаты и полного числа сотрудников:

Select Avg(OKLAD) as [Средняя зарплата], Count(*) as ЧС From TSpis;

В результате выполнения этой команды получим следующую таблицу:

Средняя зарплата	ЧС
250 000	2

Условия отбора строк

Отбор строк осуществляется оператором Where <предикат>. В предикате могут быть использованы следующие операции:

- = – равно;
- <> или != – не равно;
- > – больше;
- < – меньше;
- >= – больше или равно;
- <= – меньше или равно;
- Between – между;
- In – вхождение в множество значений;
- Like – похоже на что-то;
- Is Null – значение не задано;
- Exit – значение определено;
- Any – любое значение;
- All – все значения.

В операторе Where можно также использовать следующие логические операции:

- And – логическое И;
- Or – логическое ИЛИ;
- Not – логическое отрицание.

Рассмотрим несколько примеров использования оператора Where:

1. Найдем все строки, содержащие фамилию «Иванов»:

Select * From TSpis Where FIO=”Иванов”;

Значение поля можно заключать в двойные кавычки, а если в значащей строке есть специальные символы, включая пробел, то это значение нужно заключать в квадратные скобки, например «Иванов», или [Иванов].

2. Найдем всех сотрудников, у кого оклад лежит в заданном интервале:

Select * From TSpis Where OKLAD Between ”100000” and ”300000”;

3. Найдем всех сотрудников, у кого фамилии начинаются с букв «Ив»:

Select * From TSpis Where FIO Like ”Ив%”;

Следует отметить, что символ % (проценты) означает любую последовательность символов для стандарта ANSI-92, а для стандарта ANSI-89 следует использовать символ * (звездочка). Аналогично символ _ (подчеркивания) означает любой символ в стандарте ANSI-92, а в стандарте ANSI-89 для этих же целей используется символ ? (вопросительный знак).

4. Найти всех сотрудников с фамилиями «Иванов» или «Петров»:

Select * From TSpis Where FIO in [”Иванов”, ”Петров”];

5. Найти всех сотрудников, кому еще не установлен оклад:

Select * From TSpis Where OKLAD is Null;

Упорядочение строк

Упорядочение строк осуществляется в команде Select с помощью ключевого слова Order. Пример упорядочения строк по фамилиям сотрудников:

Select * From TSpis Order by FIO;

Связывание таблиц

Связывание таблиц производится в операторе Where команды Select по совпадению значений каких-либо двух полей в разных таблицах. Пример связывания таблиц TSpis и TDol по коду должности:

```
Select a.ID as Номер,
       a.FIO as Фамилия,
       b.NAMEDOL as Должность,
       a.STAG as [Стаж работы],
       a.OKLAD as Зарплата
            From TSpis as a, TDol as b
            Where a.CODEDOL=b.CODE;
```

Модификатор Distinct

Модификатор Distinct позволяет устранить дублирование строк в результирующей таблице. Например, можно узнать, сколько различных уровней зарплаты имеют сотрудники, следующим образом:

```
Select Distinct OKLAD as Зарплата
       From TSpis Order by OKLAD;
```

Следующий пример показывает, как найти тех сотрудников, у кого совпадают дни рождения:

```
Select Distinct a.FIO as Фамилия, a.DATEB as [Дата рождения]
       From TSpis as a, TSpis as b
       Where a.DATEB=b.DATEB and a.ID ≤ b.ID
       Order by a.DATAB;
```

В этом примере мы одну и ту же таблицу TSpis обозначили как таблицу «a» и таблицу «b», что позволило найти тех сотрудников, у кого совпадают дни рождения, но разные идентификационные номера «ID». Для исключения дублирования вставлен модификатор Distinct и проведена сортировка результирующей таблицы по дате рождения.

Работа с датами

Найдем, например, всех сотрудников, родившихся с 1 апреля 1980 г. по 31 июля 1981 г.:

Select * From TSpis

Where DATEB Between ’01/04/1980’ and ’31/06/1981’;

Эта команда записана в стандарте ANSI–92. Для стандарта ANSI–89 дату нужно заключать не в апострофы, а в символы «#» (решетка).

Текущую дату можно получить, использую функцию Date() без аргументов. Для выделения части даты из поля даты, например DATEB из таблицы TSpis, можно использовать следующие функции:

Year(DATEB) – год;
Month(DATEB) – месяц;
DatePart(”q”,DATEB) – квартал;
DatePart(”y”,DATEB) – год;
DatePart(”m”,DATEB)– месяц;
DatePart(”d”,DATEB) – день.

Например, найдем всех сотрудников, родившихся в апреле:

Select FIO as Фамилия, DATEB as [Дата рождения]
From TSpis Where DatePart(”m”,DATEB)=”04”;

Команда Transform

Эта команда есть только в MS Access. Она является расширением стандарта SQL и используется для создания перекрестных таблиц. Общий вид ее следующий:

Transform <агрегатная функция**>**
Select <инструкция**>**
Pivot <заголовки столбцов**> [in [<**знач$_1$, знач$_2$, ….**>]];**

Названия строк образуют значения полей в операторе Select, а названия столбцов образуют значения полей, указанных в операторе Pivot. Например, чтобы построить таблицу, показывающую распределение числа сотрудников по должностям (по столбцам) и стажу работы (по стокам), можно написать следующую команду:

Transform Count(*)
Select a.STAG as Стаж
From TSpis as a, TDol as b
Where a.CODEDOL=b.CODE
Pivot b.NAMEDOL;

Таблица может принять, например, следующий вид:

Стаж	Инженер	Техник	…
10	4	7	…
5	5	6	…
…	…	…	…

30.3. Пример использования запросов в Delphi

Сформируем форму так, как это показано на рис.30.1, и напишем обработчики событий создания формы и нажатия кнопки «Выполнить» следующим образом:

```
Procedure TForm1.FormCreate(Sender:TObject);
  Begin      // Связь в БД определяется в файле DB1.udl
    ADOConnection1.ConnectionString:='File Name=DB1.udl';
    ADOConnection1.Open; // Открываем соединение с БД
     // Связываем компоненты доступа и управления БД
    ADOQuery1.Connection:=ADOConnection1;
    DataSource1.DataSet:=ADOQuery1;
    DBGrid1.DataSource:=DataSource1;
    End;
Procedure TForm1.Button1Click(Sender:TObject);
  Begin
     // Закрываем активный запрос
    If ADOQuery1.Active then ADOQuery1.Close;
    ADOQuery1.SQL.Clear; // Очищаем текст запроса
    ADOQuery1.SQL.Add('Select * From TSpis;');  // Записываем запрос
     // Выполняем запрос и выводим результат запроса в DBGrid1.
    ADOQuery1.Open;
    End;
```

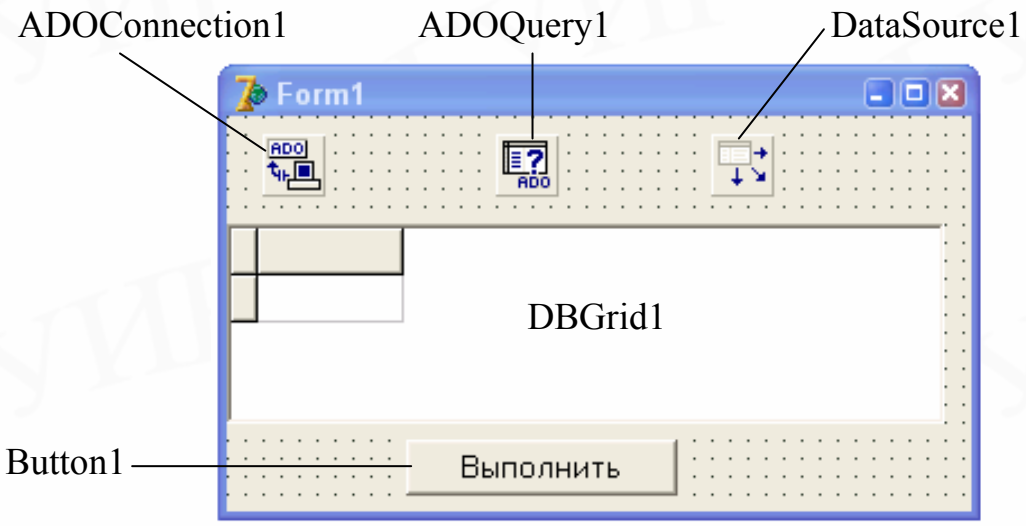


Рис.30.1. Форма с компонентами для работы с запросами

31. СОЗДАНИЕ СОБСТВЕННЫХ КОМПОНЕНТОВ

В состав Delphi входит более 450 стандартных компонентов. Но иногда возникает необходимость усовершенствовать некоторые компоненты или создать свои новые компоненты. Все компоненты должны являться наследниками класса TComponent.

В Delphi можно создать заготовку для нового компонента, пройдя путь

File → New → Component или
Component → NewComponent

После этого на экране появляется окно, в котором нужно определить:

- наследуемый класс (Ancestor Type);
- имя нового класса (Class Name);
- имя странички палитры компонентов (Paletter Page);
- место расположения файла с модулем нового компонента.

В результате получаем заготовку модуля Unit с процедурой регистрации компонента. Остается только дополнить этот модуль новыми возможностями и установить его, пройдя путь

Component → Install Component

В итоге в палитре компонентов Delphi появится новый компонент.

Для нового компонента можно создать свой значок, войдя в графический редактор **Tools → Image Editor**. В нем нужно создать компонентный ресурсный файл **File → New → Component Resource File**, в котором задать растровое изображение значка размером 24 x 24 пикселя с палитрой из 16 цветов: **Resource → New → BitMap**. Далее нужно назвать новый ресурс именем класса компонента, причем используя только прописные буквы, затем сохранить ресурсный файл с именем модуля, описывающего компонент, и расширением «.dcr». При регистрации нового компонента ему будет автоматически присвоен новый значок.

Рассмотрим пример создания нового компонента – наследника класса TListBox, который будет поддерживать табуляцию строк. Табуляция строк нужна для разбивки содержимого строк на колонки, начинающиеся с определенной позиции в строке. Например, мы хотим вывести в виде таблицы фамилии и должности сотрудников. Фамилии, допустим, будут начинаться с 5-й позиции в строке, а должности – с 40-й. Стандартный компонент TListBox не поддерживает табуляцию, и так как фамилии имеют разное количество символов, в таблице названия должностей будут начинаться с разных позиций, что затрудняет ее восприятие.

В новом компоненте предлагается в строки, помещаемые в новый LisBox, записывать между словами символы табуляции (код #09) и в класс компонента ввести новое свойство STabStop типа TStrings, в которое будем записывать в каждой ее строке номера позиций табуляции.

В результате получим компонент, который из таблицы

Иванов Инженер
Сидоренко Техник
Там Технолог

позволит получить более воспринимаемую таблицу:

Иванов	Инженер
Сидоренко	Техник
Там	Технолог

В этой таблице, в отличие от компонента TStringGrid, будет выбираться вся строка, а не отдельная ячейка таблицы.

Реализация такого компонента приведена ниже:

```
unit Ulistmy;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TListmy = class(TListBox) // Имя нового класса – TListMy
  private
    FfTabStop:TStrings; // Поле для хранения позиций табуляции
// Метод задания позиций табуляции
    Procedure SetTabStop(value:Tstrings);
  protected
  public
    Atabstop:Array[1..20] of integer; // Массив значений позиций табуляции
// Новый метод создания окна компонента
    Procedure CreateWnd; override;
// Новый метод определения параметров окна компонента
    Procedure CreateParams(var Params:TcreateParams);override;
// Новый конструктор компонента с выделением дополнительной памяти
    constructor create(Aowner:TComponent);override;
// Новый деструктор по освобождению памяти
    Destructor Destroy;override;
// Метод задания компоненту массива значений позиций табуляции
    Procedure tabset;
  Published
// Определяем новое свойство STabStop, доступное на этапе проектирования
    property STabStop:Tstrings read FfTabstop write SetTabStop;
  end;
procedure Register; // Метод регистрации компонента
implementation
procedure Register;
begin
// Регестрируем новый компонент ListMy на страничке Samples
  RegisterComponents('Samples', [TListmy]);
  end;
// Метод записи значений в свойство STabStop
Procedure Tlistmy.SetTabStop(value:TStrings);
  Begin
    ffTabStop.Assign(value); // Запись значений в поле FFTabStop
    tabset; // Устанавливаем в компоненте новую табуляцию
    Invalidate; // Перерисовываем компонент
    end;
```

```
Procedure TListmy.CreateParams(var Params:TCreateParams);
  Begin
// Вызываем наследуемый метод задания параметров компонента
    inherited CreateParams(Params);
    with Params do Begin
// Изменяем стиль окна нового компонента, разрешаем табуляцию
     Style:=Style or $80{Lbs_Usetabstops};
     end;
   end;
Procedure Tlistmy.CreateWnd;
  var i,j,err:integer;s:String;
  Begin
// Вызываем наследуемый метод создания окна
    Inherited CreateWnd;
    tabset; // Устанавливаем табуляцию для нашего окна
    end;
Procedure Tlistmy.Tabset;
  var i,j,err:integer;s:string;
  Begin
// Проверяем наличие строк, задающих позиции табуляции в FFTabStop
    if fFTabStop.count>0 then Begin
// Открываем цикл переписи позиций табуляции из FFTabStop в ATabStop
     for i:=1 to fFtabstop.count do Begin
      s:=Fftabstop.strings[i-1];
      val(s,j,err);       // Переводим строку s в число j
      if err<>0 then Atabstop[i]:=0 else
// Записываем позиции табуляции в ATabStop с учетом того, что ширина одного
// символа равна четырем базовым единицам Windows
       Atabstop[i]:=j*4;
      end;
     end else Begin
     Atabstop[1]:=0;
     end;
// Посылаем сообщение окну нового компонента, устанавливающее табуляцию
    SendMessage(Handle,LB_SetTabStops,
    Fftabstop.count,Longint(@Atabstop));
    end;
Constructor Tlistmy.Create(Aowner:Tcomponent);
  Begin
    inherited Create(Aowner); // Вызываем наследуемый конструктор Create
// Выделяем память под список позиций табуляции
    Fftabstop:=Tstringlist.create;
    end;
 Destructor Tlistmy.Destroy;
   Begin
```

```
    Fftabstop.Free; // Освобождаем память из-под списка табуляции
    Inherited Destroy; // Вызываем наследуемый деструктор Destroy
    end;
end.
```

Для подключения своего значка к вновь созданному компоненту можно создать компонентный ресурсный файл с именем UListMy.dcr, в котором необходимо определить растровое изображение значка размером 24 х 24 пикселя и названием TListMy.

Запись новый строки в компонент ListMy1 можно осуществить оператором:

```
ListMy1.Items.Add(#09+'Иванов'+#09+'Инженер');
```

При этом слова «Иванов» и «Инженер» будут располагаться с позиций, заданных в свойстве STabStop.

Если создается целый набор своих компонентов, то их можно поместить в один пакет. Для этого нужно пройти путь

Component → Install Packages → Add

Затем нужно создать свой пакет и поместить в него свои компоненты.

32. РАБОТА С РЕЕСТРОМ

Системный реестр операционной системы Windows представляет собой общедоступную древовидную базу данных для хранения настроечной информации как самой Windows, так и других программ, установленных на компьютере. В первых версиях Windows все настройки программ и самой операционной системы хранились в специальных текстовых файлах с расширением «ini». С развитием системы таких файлов становилось все больше и больше. Они были разбросаны по разным каталогам, и стало очень сложно управлять такой системой. Начиная с Windows 95, вся настроечная информация стала заноситься в одно место – реестр. Работа с реестром в Delphi осуществляется через модуль Registry, а работа с реестром вне Delphi – системной программой RegEdit.exe.

С помощью реестра можно настроить нужным образом любую установленную на компьютере программу. Однако возможности реестра представляют основной интерес при создании собственных программ широкого применения – это и запись в реестр серийного номера программы, и подсчет числа вызовов вашей программы, и защита программы от копирования, и периодическое информирование разработчика о масштабах распространения вашей программы и многое другое.

Весь реестр разделяется на пять ключевых областей:

- HKEY_CLASSES_ROOT – содержит информацию о расширениях файлов, классах и зарегистрированных в системе COM серверах. Содержимое этого ключа дублируется в разделе HKEY_ LOCAL_MACHINE\SOFTWARE\Classes;
- HKEY_CURRENT_USER – содержит множество настроек программного обеспечения (информацию о конфигурации рабочего стола, клавиатуры и информацию о параметрах меню Пуск) текущего пользователя, т.е. того

пользователя, который в настоящий момент зарегистрирован в системе. Реестр копирует содержимое одного из подразделов в разделе HKEY_USERS в данный раздел и по окончании работы его обновляет;

- HKEY_LOCAL_MACHINE – содержит информацию, относящуюся к компьютеру: местонахождение драйверов, параметры настроек, установленных на компьютере программ, и COM-серверов. Он самый крупный и самый важный из всех разделов реестра. В нем содержится в одном из подразделов копия ключа HKEY_CLASSES_ROOT и все возможные варианты в зависимости от текущей аппаратной конфигурации содержимого раздела HKEY_CURRENT_CONFIG;
- HKEY_USERS – содержит описания настроек компьютера для всех зарегистрированных в системе пользователей. Подраздел Default является разделом HKEY_CURRENT_USER для конфигурации пользователя Windows по умолчанию;
- HKEY_CURRENT_CONFIG – содержит описание текущей конфигурации оборудования компьютера и является отражением одной из ветвей раздела HKEY_LOCAL_MACHINE\Config, в котором описаны все созданные в системе конфигурации оборудования. Когда конфигурация меняется, меняется и содержимое раздела HKEY_CURRENT_CONFIG – он начинает «отражать» уже другую ветвь раздела HKEY_LOCAL_MACHINE\Config.

Физически содержимое реестра находится в нескольких файлах, и это зависит от версии Windows. Например, для Windows XP основные настройки реестра находятся в каталоге %SystemRoot%\System32\Config. Точный список файлов реестра можно посмотреть в разделе

HKEY_LOCAL_MACHINE\System \CotnrolSet\Control\HiveList\

Для работы с реестром в Delphi нужно:

- подключить модуль Registry, объявив его в операторе Uses,
- создать переменную типа TRegistry и вызвать ее конструктор,
- выполнить работы с реестром с помощью методов класса TRegistry,
- освободить память, занимаемую переменной типа TRegistry.

Остановимся на описании некоторых свойств и методов класса TRegistry:

- Property RootKey: HKEY; – определение текущего корневого ключа (один из возможных пяти, приведенных выше);
- Property CurrentKey: HKEY; – получение текущего открытого ключа реестра;
- Function OpenKey(const Key: String; CanCreate: Boolean): Boolean; – открытие ключа Key в текущем корневом каталоге, но если такого ключа нет, то он будет создан при CanCreate=True. Функция возвращает истину, если операция открытия ключа успешно выполнена;
- Function ReadString(const Name: String): String; – чтение значения параметра Name строкового типа;
- Procedure WriteString(const Name, Value: String); – запись значения Value в параметр Name строкового типа;
- Function CreateKey(const Key: String): Boolean; – создание нового ключа Key в текущем ключе;

- Procedure CloseKey; – закрытие открытого ключа и запись изменений в реестр;
- Function DeleteKey(const Key: String): Boolean; – удаление из реестра ключа Key. Корневые ключи не поддаются удалению;
- Function DeleteValue(const Name: String): Boolean; – очистка значения параметра Name;
- Function ReadInteger(const Name: String): Integer; – чтение значения параметра Name целого типа;
- Procedure WriteInteger(const Name: String; Value: Integer); – запись значения параметра Name целого типа.

Рассмотрим пример записи в реестр некоторой информации о своей программе.

```
Uses …., Registry; // Подключаем к программе модуль Registry
……..
Var Reg:TRegistry; // Определяем указатель на реестр
  Begin
    Reg:=TRegistry.Create; // Выделяем память под реестр
// Устанавливаем корневой ключ
    Reg.RootKey:=HKEY_LOCAL_MACHINE;
// Создаем в ключе Software ключ MyProgram
    Reg.OpenKey('Software\MyProgram',True);
// Создаем в текущем ключе параметр ID и записываем в него
// серийный номер нашей программы: ABCD-1234.
    Reg.WriteString('ID','ABCD–1234');
    Reg.CloseKey;  // Закрываем ключ
    Reg.Destroy;// Освобождаем память, выделенную ранее переменной Reg
    End;
```

Работать с реестром можно также с использованием программы RegEdit.exe путем создания текстового файла с нужными изменениями и последующего вызова из Delphi программы RegEdit с передачей ей текстового файла через параметры командной строки.

Операции, которые мы сделали в предыдущем примере, можно оформить и следующим образом:

```
// Первая строка текстового файла для формата NT4
Const RegID='REGEDIT4'+#13#10;
// Строка ключа
   RegPath:='[HKEY_LOCAL_MACHINE\SOFTWARE\MyProgram]'+
     #13#10;
// Задание имени параметра и его значения
      RegNew='"ID"="ABCD–1234"'+#13#10;
Var     RegFile:TFileStream; // Переменная файлового потока
   RegCmd:String; // Командная строка
   P:Pchar; // Указатель на строку с нулевым символом в конце строки
Begin
```

```
    // Создаем текстовый файл
        RegFile:=TFileStream.Create('MyFile.reg',fmCreate);
    // Записываем в него управляющую строку
        RegFile.Write(RegID[1], Length(RegID));
    // Записываем текущий ключ реестра
        RegFile.Write(RegPath[1], Length(RegPath));
    // Записываем определения параметров для текущего ключа
        RegFile.Wtite(RegNew[1], Length(RegNew));
    // Закрываем файл
        RegFile.Free;
    // Определяем командную строку
        RegCmd:='RegEdit /s '+'MyFile.reg';
    // Преобразуем обычную строку в указатель P
        StrPCopy(P, RegCmd);
    // Вызываем из Delphi программу EditReg и вносим в реестр изменения
        WinExec(P, SW_MINIMIZE);
    // Удаляем текстовый файл
        If FileExists('MyFile.reg') then DeleteFile('MyFile.reg);
        End;
```

Можно просканировать весь реестр в поисках заданных ключей и значений параметров с помощью следующей рекурсивной процедуры:

```
// Поиск в реестре значений параметров
Procedure Scanreg(skey,findpar:String;valpars:Tstringlist;
  var reg:Tregistry);
// SKey – ключ просмотра
// FindPar – имя искомого параметра
// ValPars – список найденных значений параметров
// Reg – указатель на реестр
var st:Tstringlist; // Внутренний список строк
   i:integer;s,s1:String;
   dt:TRegDataType;
  Begin
// Открытие текущего ключа
   if reg.OpenKeyReadOnly(skey) then Begin
     st:=Tstringlist.create;
     reg.GetValueNames(st);            // Получение списка параметров
     for i:=0 to st.count-1 do Begin   // Проход по списку
       if st.Strings[i]=findpar then Begin
         dt:=reg.GetDataType(findpar);      // Получение типа параметра
         if dt=rdstring then Begin
           s:=reg.ReadString(st.Strings[i]);    // Получение значения параметра
           valpars.add(s); // Добавление значения в список значений параметров
           end;
         end;
```

```
      end;
     st.Clear; // Очистка внутреннего списка параметров
     reg.GetKeyNames(st); // Получение списка ключей
     for i:=0 to st.count-1 do Begin // Проход по списку ключей
      s1:=st.Strings[i]; // Получение значения ключа
// Сканирование текущего ключа
      scanreg(skey+'\'+s1,findpar,valpars,reg);
      end;
     reg.CloseKey; // Закрытие текущего ключа
     st.Free; // Очистка списка ключей
     end;
   end;
```

С помощью этой процедуры можно, например, узнать адреса электронной почты всех пользователей данного компьютера. Для этого можно в обработчике какого-либо события написать следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
var s:string;
 ss:TStringList; // Указатель на список адресов электронной почты
begin
 reg1:=Tregistry.Create; // Подключение к реестру
// Установка корневого ключа
 reg1.RootKey:=HKEY_CURRENT_USER;
 ss:=Tstringlist.Create; // Выделение памяти под список адресов
 s:=''; // Вначале ключ полагаем равным пустой строке
 s1:='SMTP Email Address'; // Имя параметра электронной почты
 Scanreg(s,s1,ss,reg1);          // Сканируем реестр
 if ss.Count>0 then Begin
  Memo1.Lines:=ss;      // Записываем адреса электронной почты в Memo1
  end;
 ss.Free;              // Освобождаем память списка
 reg1.CloseKey;   // Закрываем реестр
 end;
```

33. ПЕРСПЕКТИВЫ ПРОГРАММИРОВАНИЯ В DELPHI

В лекциях не были затронуты некоторые вопросы программирования, такие, как работа с потоками и процессами, создание справочных систем, инсталляционных программ, Интернет-технологий, приложений для работы с распределенными базами данных. Но нельзя объять необъятное.

Ни одна отрасль науки не развивается такими быстрыми темпами, как информационные технологии. Каждый год выходят новые версии операционных систем, сред программирования и прикладных программ.

Какие же перемены ожидаются в ближайшем будущем в области информационных технологий? Компания Microsoft – фактический лидер в разработке операционных систем и прикладных пакетов программ – выпустила

в свет новую надстройку над операционной системой Windows – среду или платформу .NET Framework. Эта среда коренным образом изменяет подходы к разработке программного обеспечения. Основная задача новой среды состоит в существенном повышении надежности и безопасности разрабатываемого программного обеспечения.

Рассмотренные в предыдущих лекциях подходы к программ-мированию имеют много недостатков. Например:

- COM-технология требует регистрации всех компонентов в системном реестре, и новый вариант тех же компонентов сотрет информацию об их старой версии, что может привести к неработоспособности ранее разработанных программ, использующих предыдущие версии компонентов;
- COM-интерфейсы изначально предполагали неизменность своего описания. Однако даже сама фирма Microsoft при создании новых программных продуктов умудрялась изменять описания старых интерфейсов, что приводило к печальным последствиям;
- IUnKnown – интерфейс, который должен следить за числом ссылок на COM-объекты и удалять неиспользуемые объекты, в некоторых случаях (по вине программистов) давал сбои, что приводило к так называемой утечке памяти, когда при длительной работе программы размеры затребованной памяти превышают все разумные пределы;
- не было общих подходов к обработке ошибок.

Новая среда призвана оградить нас от этих и многих других неприятностей. NET Framework состоит из двух главных составляющих: библиотеки базовых классов и базового язык времени выполнения (Common Language Runtime – CLR).

Среда исполнения .NET-программ CLR – это краеугольный камень в фундаменте организации вычислительных процессов всей концепции .NET. Именно здесь решаются основные задачи повышения надежности и безопасности программ, а также платформной независимости. Теперь результатом работы любой системы программирования, в том числе и Delphi, должна быть программа, написанная на языке MSIL (Microsoft Intermediate Language) в полном соответствии со спецификацией CLS (Common Language Specification) и реализованная в виде двоичного байт-кода. CLR выполняет байт-код путем предварительной компиляции в машинный код отдельных фрагментов программы или приложения целиком. Программы на языке MSIL создают так называемый «управляемый код» (managed code). Это означает, что CLR не просто преобразует MSIL в машинные инструкции, а выполняет это действие с учетом внешних установок. Программа на MSIL является языково- и платформонезависимой. Это означает, что не важно, на каком языке высокого уровня создавалась программа – на C#, Visual Basic или Delphi, результатом работы трансляторов с этих языков для платформы .NET является программа на MSIL. Для каждой же физической платформы компьютера нужно создать свой компилятор с языка IL в язык машинных кодов – JIT компилятор (Just–InTime).

Структура CLR-модулей состоит из исполняемого кода и метаданных. Метаданные (например, различные декларации полей, методов, свойств и

событий) широко применяются и в COM-технологии, что и составляет ее основное отличие от обычных двоичных DLL. В случае же CLR состав метаданных значительно расширен, что позволяет эффективнее контролировать версии, проверять надежность источников поступления программ и пр.

Объединение отдельных .NET-компонентов в одно приложение непосредственно связано с новым понятием «сборка» (Assembly). Как известно, с контролем версий в COM дело обстояло, мягко говоря, не самым лучшим образом. Фактически поддержка совместимости версий была полностью возложена на разработчика COM-объектов. Технология .NET Assembly призвана решить все эти проблемы, известные под названием DLL Hell (ад DLL). В упрощенном виде идея заключается в переносе процедур регистрации объектов из системного реестра на уровень отдельных приложений. В сущности, сборка – это и есть .NET-приложение, она реализуется в виде расширенного варианта традиционного исполняемого модуля. Сборка может состоять из одного или нескольких файлов, причем они могут содержать не только исполняемый код, но также и графические изображения, исходные данные и прочие ресурсы. В архитектуре .NET сборки являются минимальным блоком, на уровне которого решаются вопросы внедрения, контроля версий, повторного использования и безопасности. Описание сборки содержится в секции метаданных (она называется манифестом) исполняемого модуля приложения.

На CLR теперь возложено все управление памятью приложения. Благодаря двухшаговой схеме компиляции (IL + JIT-компиляция), CLR имеет в своем распоряжении все списки глобальных переменных, знает местоположение локальных переменных в стеке, ссылок на объекты и т.д., что позволяет периодически контролировать правильность использования Heap памяти (кучи) и удалять из нее неиспользуемые объекты (производить сбор «мусора»).

Вторым основным составляющим .NET-технологии является введение единой библиотеки базовых функций и классов. Если раньше каждый язык программирования имел свою библиотеку функций и классов, то теперь такая библиотека только одна. Библиотека базовых классов .NET реализована в виде набора DLL (сейчас их 22), имена которых начинаются с идентификатора System.

Сегодня .NET Framework – это некая дополнительная операционная среда, устанавливаемая в Windows в качестве автономного программного компонента. Нет сомнений, что она станет неотъемлемой частью будущей версии Windows. Тем не менее еще несколько лет пользователи Windows будут иметь возможность работать как в режиме «Win API + COM», так и .NET. Но потом им придется забыть о «старом, добром Windows» и работать исключительно в режиме «управляемого кода» в среде CLR.

Последняя версия Delphi-8, сохраняя стиль программирования, заложенный в предыдущих версиях, ориентирована на разработку .NET Framework–приложений и не поддерживает Win32. По сути – это продукт, серьезно отличающийся от предыдущих версий Delphi, и более понятным было бы название, например Delphi .NET, так как это полностью новая версия Delphi.

Она позволит разработчикам создавать приложения для платформы .NET Framework и переделывать уже созданные ранее приложения для работы на данной платформе. Среда разработки Delphi 8 предполагает полную поддержку всех классов платформы .NET Framework. Здесь осуществляется поддержка технологий Microsoft ASP.NET Web Forms и Web–служб XML, что помогает ускорить разработку надежных решений для сети Интернет. В состав новой версии пакета входят элементы управления Windows Forms и библиотека элементов VCL (Visual Control Library). Для организации эффективной работы с базами данных разработчикам предлагаются технологии Microsoft ADO.NET и Borland Data Provider (BDP). Технология Borland ECO (Enterprise Core Objects) ускоряет процесс разработки приложений с опорой на дизайн (design–driven development). Новая среда разработки использует знакомый по предыдущим версиям Delphi синтаксис языка, что будет способствовать быстрому освоению новых инструментов и технологий. Визуальные элементы управления VCL обладают обратной совместимостью с уже существующим исходным кодом программ Delphi и способны взаимодействовать со средой .NET Framework. В пакете Delphi 8 реализована возможность обмена компонентами и исходным кодом с более чем 20 языками программирования, которые поддерживаются в среде NET Framework.

ЛИТЕРАТУРА

1. Фаронов В.В. Delphi. Программирование на языке высокого уровня. – СПб.: Питер, 2003.

2. Дарахвелидзе П.Г., Марков Е.П. Программирование в Delphi 7. – СПб.: Питер, 2003.

3. Дарахвелидзе П.Г., Марков Е.П. Разработка Web служб средствами Delphi. – СПб.: Питер, 2003.

4. Эбнер М. Delphi 5 руководство разработчика. – BHV, Киев, 2000.

5. Синицын А.К., Колосов С.В. и др. Программирование в среде Delphi: Лабораторный практикум по курсу «Программирование». Ч.1 и 2. – Мн.: БГУИР, 2002–2003.

6. Колосов С.В. Объектно-ориентированное программирование в среде Delphi: Лабораторный практикум для студентов всех специальностей. – Мн.: БГУИР, 2001.

Учебное издание

Колосов Станислав Васильевич

ПРОГРАММИРОВАНИЕ В СРЕДЕ DELPHI

Учебное пособие

Редактор *Н.А. Бебель*
Корректор *Н.В. Гриневич*
Компьютерная верстка *В.М. Ничипорович*

Подписано в печать 23.02.2005. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Печать ризографическая. Усл. печ. л. 9,77. Уч.-изд. л. 8,5. Тираж 400 экз. Заказ 523.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия на осуществление издательской деятельности №02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности №02330/0133108 от 30.04.2004.
220013, Минск, П. Бровки, 6