ИНФОРМАТИКА

Е. В. Санников

«СОЛОН»

Курс практического программирования в Delphi

Объектно-ориентированное программирование

- курс-практикум для начинающих программистов
- от простого к сложному
- 15 практических работ
- формирование навыков при разработке приложений
- книга для тех, кто хочет научиться объектно-ориентированному программированию!

ISBN 5-91359-122-7
9 785913 591227

ЭЛЕКТИВНЫЙ КУРС ✸ ПРОФИЛЬНОЕ ОБУЧЕНИЕ

Е. В. Санников

КУРС ПРАКТИЧЕСКОГО ПРОГРАММИРОВАНИЯ В DELPHI. ОБЪЕКТНО – ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ПРАКТИКУМ

Москва
СОЛОН-ПРЕСС
2013

УДК 681.3
ББК 32.973-018
С 185

Санников Е. В.
Курс практического программирования в Delphi. Объектно – ориентированное программирование. — М.: СОЛОН-ПРЕСС, 2013. — 188 с.: ил.

Данный курс-практикум рассчитан преимущественно на студентов информационных и технических специальностей, а также на начинающих программистов, желающих научиться разработке приложений в среде Delphi и понять суть объектного программирования. Также практикум может использоваться в качестве элективного курса у школьников старших классов.

Весь процесс создания приложений расписан очень подробно и иллюстрирован рисунками, а приведенный программный код в большинстве случаев снабжен комментариями.

Особенное внимание уделено именно практической части и формированию умений при разработке приложения.

Практикум состоит из 15 практических работ, основанных на принципе «от простого к сложному».

Сайт журнала «Ремонт & Сервис»: www.remserv.ru
Сайт издательства «СОЛОН-ПРЕСС»: www.solon-press.ru

КНИГА — ПОЧТОЙ

Книги издательства «СОЛОН-ПРЕСС» можно заказать наложенным платежом (оплата при получении) по фиксированной цене. Заказ оформляется одним из трех способов:

1. Послать открытку или письмо по адресу: 123001, Москва, а/я 82.
2. Оформить заказ можно на сайте **www.solon-press.ru** в разделе «Книга — почтой».
3. Заказать по тел.: (499) 254-44-10, (499) 795-73-26.

Каталог издательства высылается по почте бесплатно.

При оформлении заказа следует правильно и полностью указать адрес, по которому должны быть высланы книги, а также фамилию, имя и отчество получателя. Желательно указать дополнительно свой телефон и адрес электронной почты.

Через Интернет вы можете в любое время получить свежий каталог издательства «СОЛОН-ПРЕСС», считав его с адреса **www.solon-press.ru/kat.doc**.

Интернет-магазин размещен на сайте **www.solon-press.ru**.

По вопросам приобретения обращаться:
ООО «АЛЬЯНС-БУКС»
Тел: (499) 725-54-09, 725-50-27,
www.alians-kniga.ru

ISBN 978-5-91359-122-7

© Обложка, «СОЛОН-ПРЕСС», 2013
© Санников Е. В., 2013

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ **5**

ПРАКТИЧЕСКАЯ РАБОТА №1 ..... **6**

1.1. Основы работы в среде объектно-ориентированного программирования Delphi7 6

ПРАКТИЧЕСКАЯ РАБОТА №2 ..... **20**

2.1. Работа с буфером обмена, рисование мышью, контекстное меню, компонент Memo1 и Image1 20
2.2. Рисование мышкой на форме. Контекстное меню 22
2.3. Компонент Memo 25

ПРАКТИЧЕСКАЯ РАБОТА № 3 ..... **28**

3.1. Перемещение мышью панели Panel1 на форме 28
3.2. Работа с таймером на примере программы «Бильярд» 29
3.3. Изменим надпись на кнопке «Пуск» 30

ПРАКТИЧЕСКАЯ РАБОТА №4 ..... **32**

4.1. Построение графиков в среде Delphi 32
4.2. Моделирование бегущих волн 35

ПРАКТИЧЕСКАЯ РАБОТА №5 ..... **41**

5.1. Работа с файлами (на примере текстового). Создание текстового файла. Запись информации в текстовый файл 41
5.2. Чтение из файла. Открытие файла в режиме чтения 44

ПРАКТИЧЕСКАЯ РАБОТА № 6 ..... **48**

6.1. Краткий обзор некоторых стандартных компонентов Delphi 48
6.2. Исследуем компоненты PageControl1, ListBox1, ComboBox1, а также еще ряд полезных компонентов 51

ПРАКТИЧЕСКАЯ РАБОТА № 7 ..... **57**

7.1. Процедуры и функции 57
7.2. Перегрузка функций 65

ПРАКТИЧЕСКАЯ РАБОТА № 8 ..... **69**

8.1. Создание приложения «МикроПроводник». Создание заставки 69
8.2. Запуск Программ 73
8.3. Окно заставки 74

ПРАКТИЧЕСКАЯ РАБОТА № 9 ..... **76**

9.1. Виртуальные и ASC II коды 76

ПРАКТИЧЕСКАЯ РАБОТА № 10 ..... **82**

10.0. Сообщения и их обработка 82
10.1. Исследуем стандартный виртуальный метод WndProc 84
10.2. Класс TApplication 88
10.3. Собственные обработчики событий. Функции SendMessage и PostMessage 91
10.4. Создание своего собственного сообщения 94

ПРАКТИЧЕСКАЯ РАБОТА № 11 ..... **96**

11.0. Базы Данных 96
11.1. Работа с мастером Баз Данных 96
11.2. Создание Базы Данных с использованием стандартного приложения Database Desktop 101
11.3. Работа с Базой Данных непосредственно в среде Delphi 104
11.4. Самостоятельные задания по работе с базами данных в Delphi 114

ПРАКТИЧЕСКАЯ РАБОТА № 12 ..... **115**

12.0. Библиотеки 115
12.1. Создание библиотеки с использованием Dll Wizard 115
12.2. Библиотека хранящая объекты 119
12.3. Динамически подключаемые библиотеки 122

ПРАКТИЧЕСКАЯ РАБОТА №13 ..... **125**

13.1. Создание собственного видеопроигрывателя 125
13.2. Вывод видео на стандартные компоненты Delphi. Описание свойства DeviceType компонента MediaPlayer1 128

ПРАКТИЧЕСКАЯ РАБОТА №14 ..... **132**

14.1. Классы и объекты 132
14.2. Наследование классов, класс TObject, назначение конструктора Create. Статические и виртуальные методы. 140

ПРАКТИЧЕСКАЯ РАБОТА №15 ..... **157**

15.1. Замена виртуальных методов. Цепочки Виртуальных методов. Директива reintroduce. Таблица виртуальных методов VMT 157
15.2. Перегрузка статических и виртуальных методов в разных взаимонаследующих классах 165
15.3. Инкапсуляция. Спецификаторы доступа public, private, protected 178
15.4. Свойства класса 181

БИБЛИОГРАФИЧЕСКИЙ СПИСОК **187**

ПРЕДИСЛОВИЕ

Уже много лет среда объектно-ориентированного программирования (ООП) Delphi корпорации Borland пользуется невероятной популярностью среди разработчиков во всем мире. Возможности этой среды программирования, действительно, впечатляют. Благодаря своей ориентации на визуально – компонентное проектирование, среда Delphi позволяет создавать полноценные windows-приложения за достаточно небольшой срок. При этом среда Delphi является очень дружественной к пользователю и интуитивно ему понятна.

На сегодняшний момент выпущена очередная версия Delphi с возможностью разрабатывать как стандартные windows-приложения, так и приложения с использованием относительно новой технологии .NET. Многие начинающие и профессиональные программисты выбирают для себя основной именно среду программирования Delphi. При этом наиболее удобной версией для желающих «с нуля» освоить Delphi является версия Delphi7. При работе с данной книгой желательно использовать версию Delphi7, хотя подойдет почти любая версия Delphi.

Данный курс-практикум «Курс Delphi и объектно-ориентированное программирование» рассчитан преимущественно на студентов информационных и технических специальностей, а также на начинающих программистов, желающих научиться разработке приложений в среде Delphi и понять суть объектного программирования. Также парактикум может выступать в качестве элективного курса у школьников старших классов.

Данный курс можно с успехом применять как лабораторный практикум в университете для обучения студентов программированию в среде Delphi и формированию у них фундаментальных понятий ООП.

На достаточно простых примерах подробно, в деталях изложен весь процесс разработки приложения от начала до конца. При этом особенное внимание в практикуме уделено именно практической части и формированию умений при разработке приложения. Поэтому студент очень быстро осваивает среду Delphi и начинает четко представлять себе концепцию разработки windows-приложений после первой же практической работы. Конечно, если читатель имеет навыки работы в языке Pascal (а в Delphi программирование идет на Object Pascal) или каком-то другом языке программирования, освоение Delphi пойдет еще быстрее.

Практикум состоит из 15 практических работ, основанных на принципе «от простого к сложному».

Каждая практическая работа имеет определенную тематику и состоит из пронумерованных заданий. Каждое задание имеет четкую цель, которая указывается в заголовке задания жирным шрифтом, а затем идет подробное, часто иллюстрированное рисунками и схемами, описание, как выполнять данное задание, а приведенный программный код в большинстве случаев снабжен комментариями.

В ряде практических работ имеется небольшая теоретическая часть, которая непосредственно связана с выполняемыми заданиями. Например, необходимая теория присутствует в практических работах, где рассматриваются достаточно важные вопросы программирования: создание библиотек, обработка сообщений windows, создание и работа с базами данных.

Достаточно тщательно разработаны две заключительные практические работы, связанные с основными принципами объектно-ориентированного программирования. Значительное внимание в них уделено понятиям *класс, объект, статические методы класса*. Подробно, на многочисленных примерах и схемах описана работа с виртуальными методами.

Описаны примеры применения ключевых слов virtual и override и директивы reintroduce в различных вариациях во взаимонаследующих классах, а также рассмотрена перегрузка статических и виртуальных методов.

Описывается доступ к таблице виртуальных методов и вызов метода по его адресу. На простых примерах подробно описывается работа с интерфейсом класса, то есть с его свойствами property. Уделено внимание вопросам работы со спецификаторами доступа к классам.

Две заключительные практические работы занимают значительную часть практикума и направленны на формирование фундаментальных принципов и понятий объектно-ориентированного программирования.

ПРАКТИЧЕСКАЯ РАБОТА №1

1.1. Основы работы в среде объектно-ориентированного программирования Delphi7

Для обучения и освоения объектно-ориенитрованной среды программирования Delphi наиболее оптимальными версиями являются Delphi 6 или Delphi7. Установите какую-либо из этих версий на свой компьютер и можно приступать к освоению этой популярной, обладающей значительными возможностями, среде программирования. Если вы имеете некоторые навыки работы с языком программирования Pascal, то обучение пойдет гораздо быстрее, поскольку в среде Delphi программы разрабатываются на языке Object Pascal. Запускается среда объектно-ориентированного программирования Delphi7 также, как и обычное приложение. Из общего меню Windows последовательно выберите: «Программы» → «Borland Delphi7», а затем просто щелкните на надписи «Delphi7» из раскрывшегося списка (рис. 1.1).

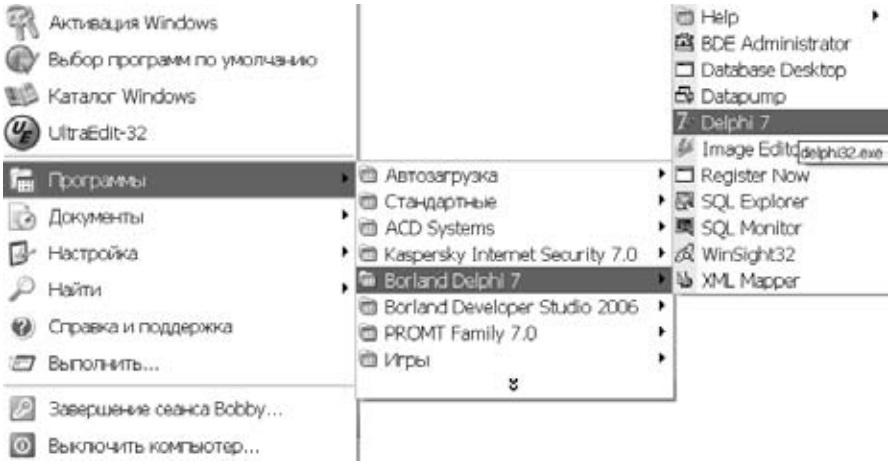


Рис. 1.1 Запуск Delphi7

Как только среда Delphi запустится, на экране сразу же появятся пять следующих окон (рис. 1.2):
главное окно – Delphi7;
окно стартовой формы – Form1;
окно редактора свойств и событий объектов – Object Inspector;
окно просмотра списка объектов – Object TreeView;
окно редактора кода – Unit1.pas.

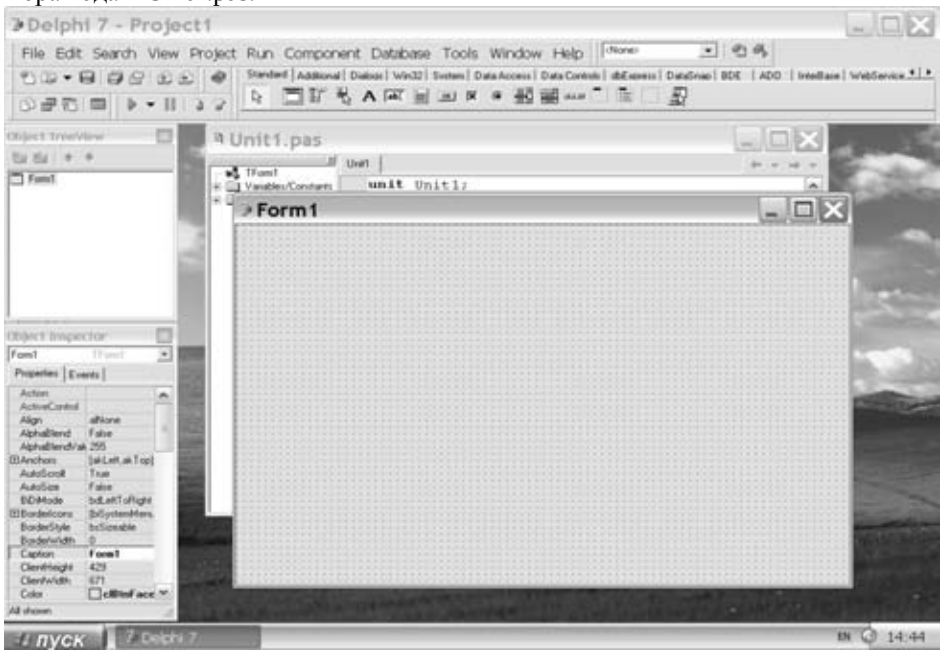


Рис. 1.2 Вид экрана сразу же после запуска Delphi

В главном окне (рис. 1.3) находится меню команд, панели инструментов и палитра компонентов (вкладки → «Standard», «Additional», «Win32» и т.д.). Палитра компонентов содержит компоненты, необходимые для разработки приложения. Часто компонент называют еще и объектом.

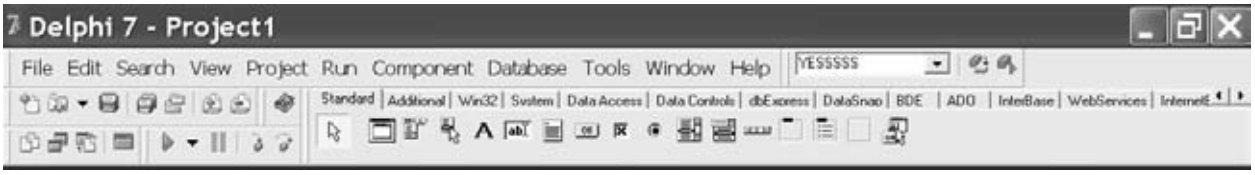


Рис. 1.3. Главное окно

Задание 1.1.0 Закроем проект и создадим его заново в среде Delphi.

В данном случае, после запуска Delphi, появившееся у вас на экране окно с заголовком Form1 и окно редактора кода с заголовком Unit1.pas представляют собой проект по умолчанию.

Закройте проект, последовательно выбрав «File» → «Close All», а затем снова создайте новый проект (приложение). Для этого последовательно выберите «File» → «New» → «Application».

Перед вами снова появится окно стартовой формы Form1, которое представляет собой заготовку главного окна разрабатываемого приложения (проекта) и окно с заголовком Unit1.pas.

Каждый объект в объектно-ориентированном программировании (например, объект с именем Form1) описывается определенным классом (например форма Form1 описывающая окно приложения, описывается классом TForm1). Понятие «класс» является логической абстракцией, описывающей некоторый объект.

Именно объект, созданный на основе некоторого класса, физически занимает определенную область в оперативной памяти компьютера и производит определенные действия с помощью своих методов над своими данными.

Фактически класс – это тип данных, определенный пользователем, а объект – его экземпляр. Часто говорят, что объект является экземпляром класса. В некотором смысле объект – это «воплощение класса в реальность».

Для сравнения можно сказать, что класс – это форма для печенья, а объект – само печенье [1]. Часто объектом в Delphi выступает обыкновенное окно операционной системы windows, именно такое окно и описывает класс TForm, то есть внешний вид окна, цвет заливки, надпись заголовка в окне и т.д.

У класса имеются свои поля и методы. В сущности, поля класса – это данные любого определенного типа, принадлежащие классу, а методы класса – это процедуры и функции класса. Методы непосредственно исполняют некоторые действия, в том числе могут работать с полями.

Например некоторые поля класса TForm содержат информацию о внешнем виде окна (ширина, высота, цвет и т. д.). Поля могут содержать текстовую информацию, некоторый адрес в памяти, просто целое или вещественное число, то есть могут содержать значение почти любого типа определенного в Delphi.

Чтобы объявить в программе два объекта (две переменные объектного типа) – Form1 и Form2 класса TForm1, требуется записать следующее объявление: var Form1, Form2: TForm1.

В данном случае объекты Form1 и Form2 полностью описываются классом TForm1. На самом деле Form1 и Form2 не являются объектами, они являются лишь ссылками на объекты, то есть содержат адрес объектов в оперативной памяти, но для упрощения рассуждений их часто можно считать самими объектами.

Во многих случаях понятия «тип» и «класс» можно считать синонимами.

Классы могут наследоваться. Это значит, что, если мы хотим создать новый класс TForm1, являющийся наследником TForm, то следует написать инструкцию: TForm1 = class (TForm).

Таким образом, новый класс TForm1 вбирает в себя все свойства, возможности и функциональность класса TForm, а также может, и скорее всего будет (специально для этого он и создан), дополнять их своими новыми.

В данном случае говорят, что класс TForm1 является производным классом от класса TForm, а класс TForm – базовым по отношению к классу TForm1.

Можно сказать, что класс TForm является родительским для класса TForm1, который, в свою очередь, является его дочерним классом.

Все компоненты в Delphi являются объектами, следовательно, описываются определенными классами.

В среде Delphi имеется очень важное окно **Object Inspector** (рис. 1.4**)** – окно редактора свойств и событий объектов, предназначенное для редактирования значений свойств объектов [2, 3] (**вызывается при нажатии на «F11»**). В терминологии компонентно-визуального проектирования объектами являются диалоговые окна и элементы управления (поля ввода и вывода, командные кнопки, переключатели и др.).

Например, сама форма Form1 является объектом. И все компоненты, что находятся в палитре компонентов, также являются объектами. Именно их свойства и события может отображать окно Object Inspector. На протяжении всего времени разработки приложений в Delphi вам придется очень интенсивно им пользоваться.

Properties (Свойства объекта) – это характеристики, определяющие вид, положение и поведение объекта. Находятся в Object Inspector слева.

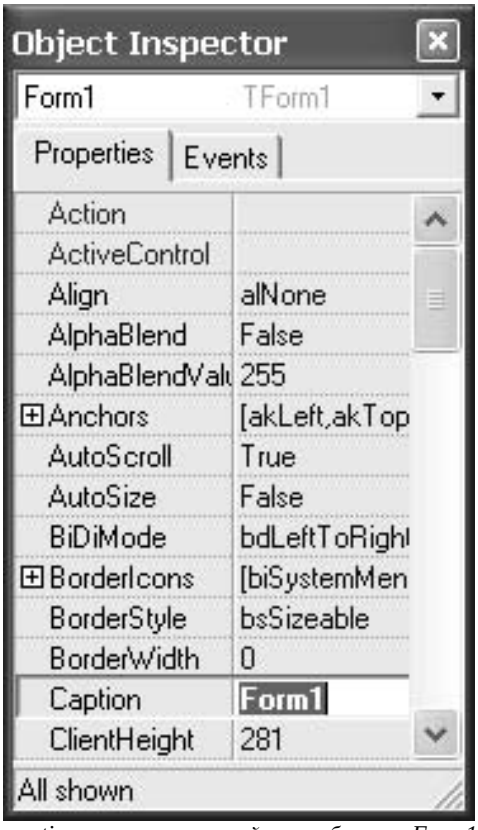


Рис. 1.4 На вкладке Properties перечислены свойства объекта-Form1 и указаны их значения

Задание 1.1.1 Изменим некоторые свойства формы – Form1

Активируйте окно Object Inspector, нажав «**F11»,** и выберете там из выпадающего списка форму Form1 или просто один раз щелкните по форме (в Object Inspector установится значение Form1, как на рис. 1.4). Поменяйте в «Object Inspector» → «Properties» значения свойств Width и Height (ширину и высоту) формы, свойства Top и Left – положение формы на экране.

Измените значение свойства Caption (заголовок) на “Работа в Delphi”.

Обратите внимание на то, что изменяя значение свойства Caption вы не меняйте имя самого объекта. Посмотрите на свойство Name – вы обнаружите там «Form1». Нужно отметить особое свойство Name, имеющееся у всех компонентов Delphi. Name – это имя компонента, его идентификатор.

Аналогично настроив Object Inspector на любой другой компонент, можно менять значения его свойств.

Задание 1.1.2 Кратко рассмотрим редактор кода.

Помимо формы, имеется еще и редактор кода (рис. 1.5), там, где непосредственно находится текст программы. **Редактор кода предназначен для редактирования модуля, который содержится в файле с расширением *.pas. По умолчанию имя файла модуля «Unit1.pas».**

Нажмите «**F12»** и вы будете переключаться между редактором кода и формой.

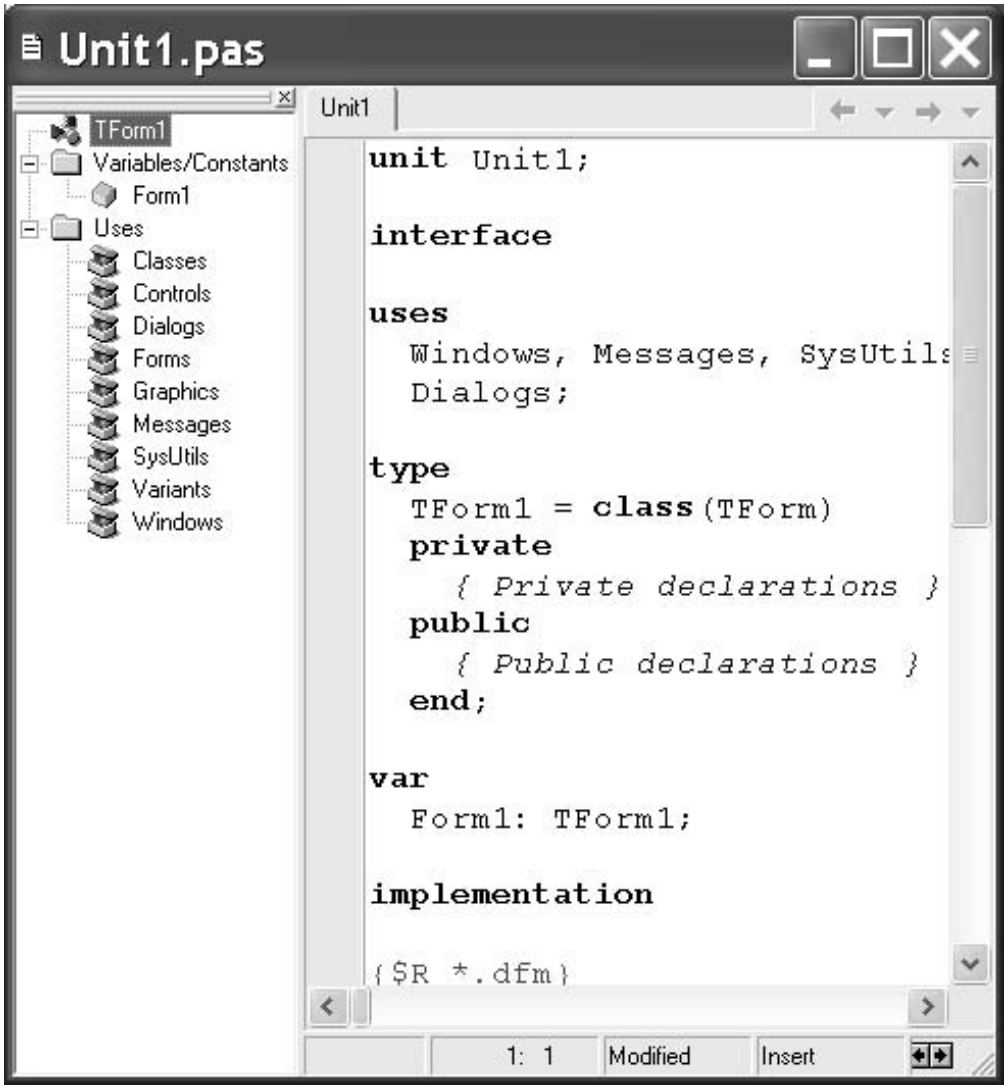


Рис. 1.5. Окно редактора кода

В окне редактора кода, которое можно увидеть, отодвинув в сторону окно формы, следует набирать текст программы. В начале работы над новым проектом окно редактора кода содержит сформированный средой Delphi шаблон программы.

В Delphi в основном создаются Windows-приложения, **часто одному приложению сопоставляется один проект, который хранится в одной папке.**

Задание 1.1.3 Сохраним проект.

Проект Delphi – это папка, в которой хранится несколько файлов.

Из них основные два файла – файл проекта (*.dpr), в котором хранится информация о проекте в целом и файл модуля (*.pas), в котором находится программный код.

Сохраняется проект путем последовательного сохранения этих двух файлов (файла проекта и файла модуля) в одну папку:

Последовательно выберите «File» → «Save All» или один раз щелкните мышкой на соответствующей пиктограмме на панели инструментов, обведенной кружком (рис. 1.6а)

а) Сохраняем проект

б) Открываем проект

Рис. 1.6. Пиктограммы сохранения и открытия проекта

Перед вами появится диалоговое окно, предлагающее сохранить файл модуля unit1.pas в месте, определенном по умолчанию (рис. 1.7).

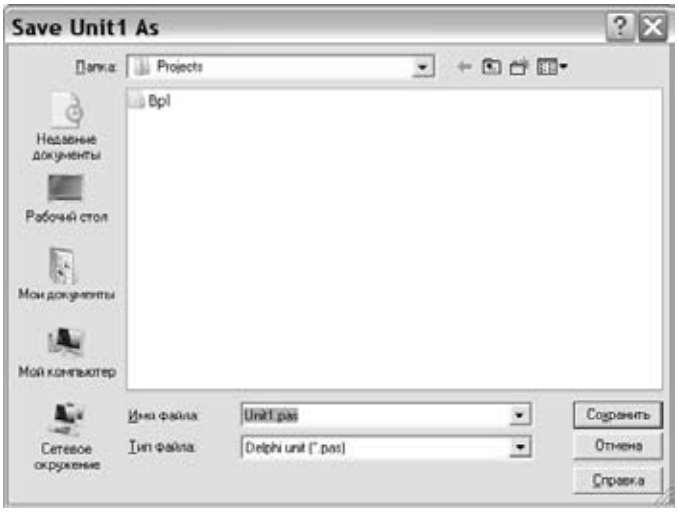


Рис. 1.7. Диалоговое окно, предлагающее сохранить вам файл модуля unit1.pas в месте, определенном по умолчанию. Не сохраняйтесь в этом месте, лучше выберите другое место

Однако вы, с помощью данного диалогового окна, выберете какое-нибудь другое место на диске, где должен располагаться ваш проект (например, «Рабочий стол» или «Диск С»), и создайте в этом месте – папку проекта, например, с именем «Proekt» *(щелкните на иконке «создание новой папки» или правой кнопкой мыши «создать папку»).* Затем зайдите в эту папку «Proekt» и, уже там, сохраните файл модуля unit1.pas, щелкнув один раз на кнопке «Сохранить» (рис. 1.8).

Рис. 1.8. Диалоговое окно сохранения файла модуля unit1.pas

После этого, сразу же, появится следующее диалоговое окно с предложением сохранить файл проекта project1.dpr в этой же папке.

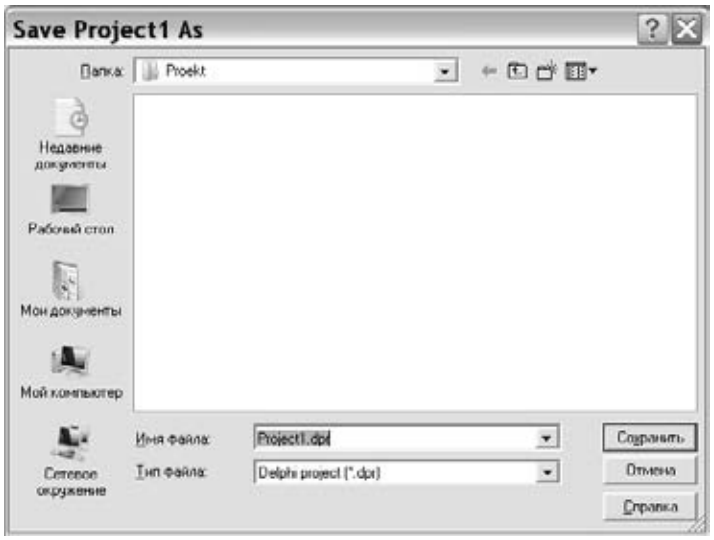


Рис. 1.9. Окно сохранения файла проекта Project1.dpr

Сохраните его в этой же папке.

Теперь, когда проект создан и основные файлы сохранены в одной папке, на форму можно добавлять остальные компоненты, менять программный код и для сохранения изменений уже просто нажимать на пиктограмме Save All.

Чтобы посмотреть содержимое файла проекта *.dpr и перемещаться между файлами модулей и файлом проекта, в окне редактора кода нажмите сочетание **«Ctrl+F12»** – появится следующее окно:

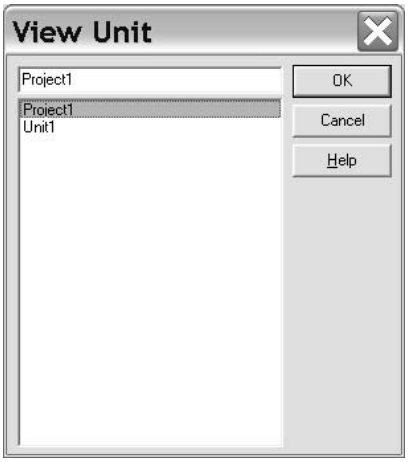


Рис. 1.10. Окно выбора файла модуля или файла проекта для его отображения в окне редактора кода

Выберите Project1 и нажмите OK. Перед вами в окне редактора кода появится содержимое файла проекта Project1.dpr.

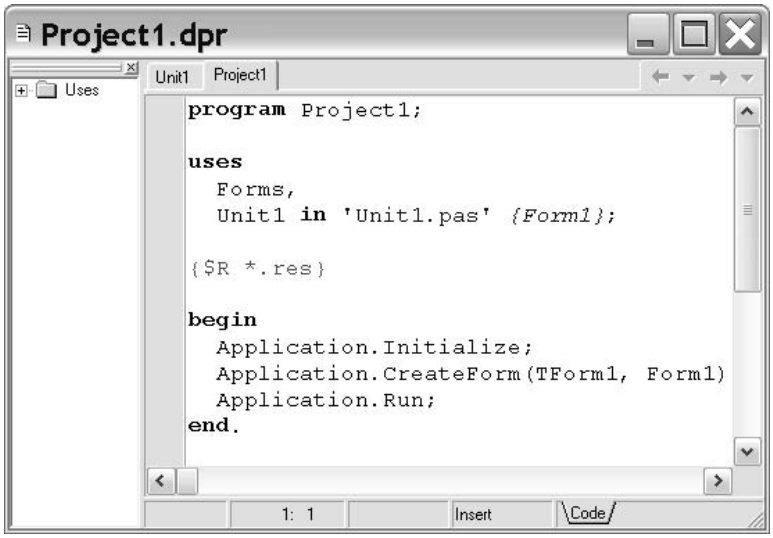


Рис. 1.11 Содержимое файла проекта

В этом файле, как и в unit1.pas, можно изменять программный код, но в основном при разработке несложных приложений его изменение не требуется. Например, при добавлении дополнительной формы в приложение среда Delphi сама внесет необходимый код в данный файл.

Задание 1.1.4 Переместим на форму три компонента Edit.

Наиболее часто используемые компоненты находятся на вкладке Standard (рис. 1.12).

Компонент Edit предназначен для ввода значений в его текстовое поле.

Рис. 1.12 Вкладка Standard содержит наиболее часто используемые компоненты

Для того чтобы добавить в форму компонент Edit1, необходимо в палитре компонентов Standard выбрать его, щелкнув один раз левой кнопкой мыши на его пиктограмме, далее подвести указатель мыши в ту точку

формы, в которой должен быть размещен левый верхний угол компонента, и еще раз щелкнуть левой кнопкой мыши по форме. В результате на форме появится компонент стандартного размера.

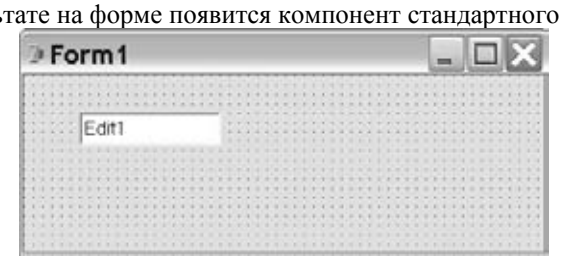


Рис. 1.13 Компонент Edit1 размещенный на форме

Каждый компонент в Delphi является объектом, и следовательно описывается классом. Подведите указатель мыши к компоненту Edit1, который уже на форме, и задержите его на пару секунд. Вы увидите надпись Edit1: TEdit. Следовательно, объект с именем Edit1 описывается классом TEdit.

Таким образом, имя переменной-объекта помещенного на форму состоит из 2-х частей:

«имя класса компонента без первой буквы Т» + «порядковый номер», то есть в случае с Edit1 будет «Edit» + «1». А если добавить второй компонент класса Tedit, будет «Edit» + «2», то есть Edit2.

Добавьте компонент Edit2 и Edit3 на форму.

Название класса по принятым соглашениям предваряется буквой «Т», так как оно происходит от слова «Type» – что означает тип или класс.

Программист путем изменения значения свойства Name может изменять имя компонента. В простых программах имена компонентов, как правило, не изменяют.

Задание 1.1.5 Переместим компонент Button1 (кнопка) на форму.

Выберете из палитры компонентов Standard, компонент Button1 и поместите его на форму. Этот компонент является кнопкой и описывается классом TButton. По умолчанию кнопка будет иметь имя Button1, это можно увидеть в свойстве name окна Object Inspector.

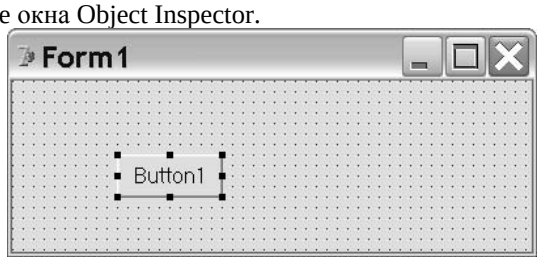
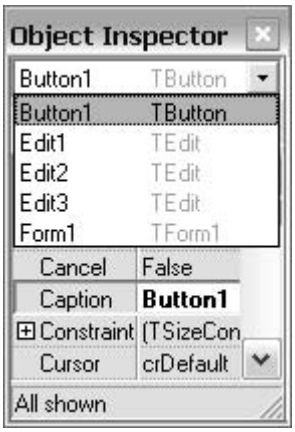


Рис. 1.14 Компонент Button1 размещенный на форме

Настройте Object Inspector (F11) на Button1. Для этого в верхней части Object Inspector из раскрывающегося списка просто выберете Button1.



а)

б)

Рис. 1.15.
а) Раскрывающийся список Object Inspector; б) Object Inspector настроен на Button1

Или выберите одинарным щелчком Button1 на форме, что также сразу настроит Object Inspector на Button1.

Растяните кнопку пошире в длину. Посмотрите, как при этом изменится значение свойства Width кнопки, а также в высоту (свойство Height) в Object Inspector. Перемещайте Button1 по форме. Посмотрите, что

изменяются значения свойств Left и Top – определяющие координаты Button1 относительно левого верхнего угла формы. Попробуйте поменять эти значения вручную.

Поменяйте значения свойства Caption с «Button1» на «PUSK». Тем не менее, имя кнопки останется Button1, вы меняете лишь надпись на ней.

Некоторые свойства компонента Button1

Свойство	Описание
Name	**Имя компонента. Используется в программе для доступа к компоненту и его свойствам**
Caption	**Текст на кнопке**
Enabled	**Признак доступности кнопки. Кнопка доступна, если значение свойства равно True, и недоступна, если значение свойства равно False**
Left	**Расстояние от левой границы кнопки до левой границы формы**
Top	**Расстояние от верхней границы кнопки до верхней границы формы**
Height	**Высота кнопки**
Width	**Ширина кнопки**

Задержите курсор мыши на кнопке (не щелкаете) – и вы увидите ее имя; или посмотрите значения свойства name кнопки в Object Inspector.

ВАЖНО! У всякого объекта (например, Button1), помимо Properties (свойств), имеются еще и Events (События). С объектом могут происходить события, и он должен на них как-то реагировать. Например, с кнопкой самое частое событие -это щелчок на ней.

Для каждого события можно определить реакцию объекта на него, поэтому существуют процедуры и функции, которые обрабатывают реакцию объекта на событие – они называются обработчиками события.

Задание 1.1.6. Создадим обработчик события «Щелчок на кнопке».

Настройте Object Inspector на Button1, затем выберете вкладку Events. Подведите указатель мышки к событию onClick и щелкните двойным щелчком чуть справа от надписи «onClick» на белом фоне, где мигает курсор (рис. 1.16).

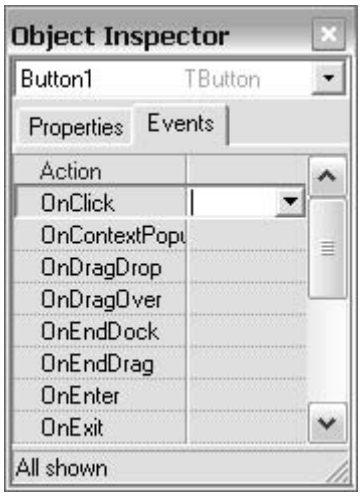


Рис. 1.16. Вкладка Events в Object Inspector

Вы переместитесь в окно редактора кода, и появится заготовка обработчика события onClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Вот и все, заготовка обработчика создана. Таким способом **можно создать обработчик события для любого компонента. Для этого нужно настроить Object Inspector на компонент (выбрать этот компонент одинарным щелчком мышкой или в Object Inspector из выпадающего списка), затем выбрать вкладку Events и щелкнуть двойным щелчком чуть справа от надписи (на белом фоне) нужного собы-**

тия, где мигает курсор. Появится заготовка обработчика, которую можно заполнять программным кодом.

Ту же самую заготовку обработчика события onClick можно сделать, если просто двойным щелчком быстро щелкнуть на Button1. Проверьте это. Но это формирует обработчик только для события – щелчок на кнопке. Если нужно создать обработчик только щелчка на кнопке, просто два раза быстро щелкните на Button1. Если же нужно создать обработчик для какого-либо другого события, то это нужно делать именно с помощью Object Inspector.

В данном случае обработчиком события «Щелчок на кнопке» является метод-процедура с именем Button1Click, принимающая параметр Sender типа TObject и принадлежащая к классу TForm1.

Между begin и end заготовки обработчика события onClick напишите следующее:

```
ShowMessage('Моя первая программа на Дельфи');
```

Оператор ShowMessage выдает текст, заключенный между кавычками, на экран в виде текстового сообщения. В этом случае обработчик щелчка на Button1 полностью должен выглядеть так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
ShowMessage('Моя первая программа на Дельфи');
end.
```

Сохраните в проекте все изменения, щелкнув на пиктограмме Save All. Теперь можно приступать к запуску программы на выполнение.

Откомпилируйте программу на предмет ошибок, нажав «CTRL+F9».

Если в тексте вашей программы есть ошибки, Delphi отметит их, выделив красной чертой.

Если ошибок нет, запустите программу на выполнение, нажав «F9», либо выберете последовательно «Run» → « Run», либо щелкните по зеленому треугольнику в главном окне.

Перед вами появится ваш проект, но уже в режиме выполнения, а не режиме разработки. **В режиме выполнения вы уже не можете изменять проект или программировать, а в режиме программирования можете.**

Щелкните по кнопке, и вы получите то, что запрограммировали. При нажатии на Button1 будет выдаваться текстовое сообщение «Моя первая программа на Дельфи».

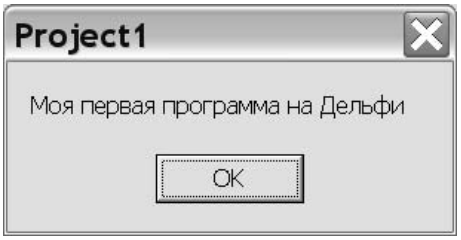


Рис. 1.17. Результат щелчка на Button1

Вот и все, ваше первое приложение в Delphi готово!

Затем, чтобы иметь возможность программировать и изменять проект, вернитесь в **режим программирования (режим разработки)**, для этого просто закройте окно вашего приложения одинарным щелчком мыши или нажмите «Alt»+«F4» как для обыкновенного окна.

После этого в режиме программирования просто закроем проект.

Выберете «File»→»Close All» и проект будет закрыт, затем закройте и саму среду Delphi.

Задание 1.1.7. Найдите исполняемый Project1.exe файл программы.

Найдите папку вашего проекта «Proekt». Помимо project1.dpr и unit1.pas, вы найдете полноценный исполняемый файл **Project1.exe,** готовый к выполнению, независимо, есть ли Delphi на данном компьютере или нет. Запустите его. Запустится ваше приложение, которое должно работать нормально. Таким образом, у вас готово стандартное оконное приложение (программа) Windows. После закройте приложение.

Задание 1.1.8. Откроем уже готовый файл проекта Project1.dpr в среде Delphi.

Здесь же в папке проекта «Proekt» щелкните двойным щелчком по файлу Project1.dpr и автоматически загрузится Delphi с вашим проектом. Вы также можете открыть проект из самого Delphi; сначала закройте имеющийся открытый проект, если он есть, последовательно выбрав «File» → «Close All», а затем последо-

вательно выберите «File» → «Open» или «File» → «Open Project» либо щелкните на эквивалентной пиктограмме (рис. 1.6б). В результате появится диалоговое окно с предложением открыть нужный файл проекта с расширением dpr (рис. 1.18).

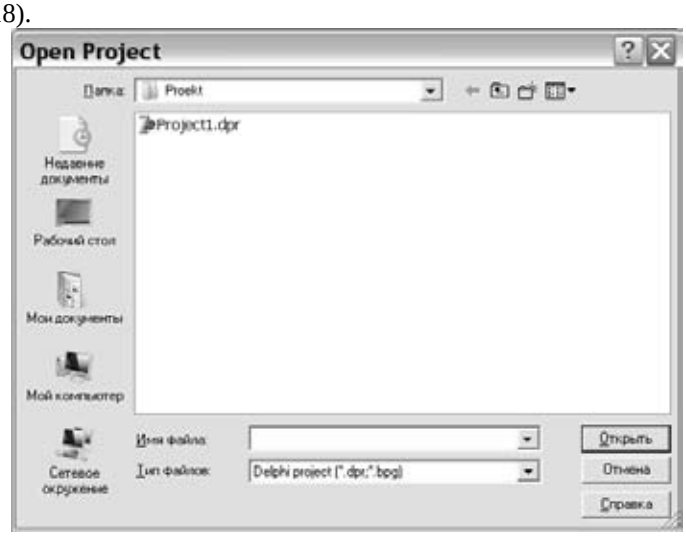


Рис. 1.18. Диалоговое окно предлагающее открыть проект

Выбирите с помощью данного окна, например, файл вашего проекта «Project1.dpr» в папке «Proekt» и щелкните на кнопке «Открыть». Выбранный проект нормально загрузится в среде Delphi в режиме программирования.

1.2. Создание проекта «Квадратное уравнение» с рассмотрением операторов типа IntToStr. Объявление глобальной переменной. Работа с таймером и компонентом Image.

Задание 1.2.1 Создадим программу «Квадратное уравнение», рассчитывающую корни квадратного уравнения $ax^2+bx+c=0$.

Создайте новый проект в папке KvadrUrovn, сразу же сохранив данный проект.

Установите значение свойства Caption формы в – Квадратное Уравнение

Перенесите по очереди 5 компонентов Label и 5 компонентов Edit, а также одну кнопку Button1, из палитры Standard, на форму (рис. 1.19).

Рис. 1.19. Окно программы «Квадратное уравнение»

Компонент Label используется как поле для вывода текста, а Edit в основном как поле для ввода текста (но туда также можно выводить текст).

Установите для компонента Label1 значение свойства Caption – “ a ”
Установите для компонента Label2 значение свойства Caption – “ b ”
Установите для компонента Label3 значение свойства Caption – “ c ”
Установите для компонента Label4 значение свойства Caption – “ x1 ”
Установите для компонента Label5 значение свойства Caption – “ x2 ”
Установите для компонента Button1 значение свойства Caption – “ Посчитать ”
Для каждого компонента Edit удалите текст из его свойства Text, чтобы там стало пусто (рис. 1.20).

Рис. 1.20. Окно программы «Квадратное уравнение»

Далее войдите в обработчик события onClick нажатия кнопки Button1 через вкладку Events или просто щелкните двойным щелчком по кнопке Button1.

Добавьте над словом begin ключевое слово var.

Учитывайте, что в тексте приводятся комментарии – это текст, который вы пишите для себя, он не воспринимается компилятором. Комментарии – это либо два слэша «//» – они действуют только на одну строку, либо фигурные скобки {} – они могут распространяться на несколько строк.

Напишите в обработчике следующий текст, в результате чего обработчик будет выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var a,b,D,x1,x2:real; c:integer;
begin
a:=StrToFloat(Edit1.Text); //в Edit1 можем вводить или вещественное или целое значение
b:=StrToFloat(Edit2.Text); // в Edit2 можем вводить или вещественное или целое значение
c:=StrToInt(Edit3.Text); // в Edit3 можем вводить только целое значение
D:=b*b-4*a*c;
if D<0 then
   begin
   ShowMessage('Корней        нет');
   halt;// немедленное прекращение программы
   end;

x1:=(-b+sqrt(D))/(2*a);//вычислили первый корень
x2:=(-b-sqrt(D))/(2*a);// вычислили второй корень
edit4.Text:=FloatToStr(x1);//выводим значение x1 на экран в Edit4
edit5.Text:=FloatToStr(x2);// выводим значение x2 на экран в Edit5
end;
```

Запустите программу, например, введите в поле Edit1 значения 9, в поле Edit2 значение 5, в поле Edit3 значения -4. Убедитесь, что она работает правильно и выдает корни x1 и x2.

Проанализируйте текст в комментариях кода. **Вернитесь в режим программирования.**

Проанализируем выражение a:=StrToFloat(Edit1.Text). Здесь компонент Edit1 является полем для ввода текста в его свойство Edit1.text строкового типа, поэтому введенное туда значение будет интерпретироваться программой как текстовый тип, строка, набор символов. Поэтому мы не можем просто написать a:= Edit1.Text, так как это ошибка. В противном случае компилятор начнет «возмущаться», поскольку нельзя присвоить переменной «a» вещественного типа real, текстовое или строковое значение типа string, хотя бы и в виде цифр. Мы должны указать компилятору, как интерпретировать введенное значение, и применяем оператор StrToFloat(‘тут текстовое представление числа’), его можно трактовать как «Строка_В_Вещественное число». Оператор StrToFloat преобразует текстовое представление числа в вещественный тип.

При этом, если текстовый тип преобразуем в вещественный тип, пишем StrToFloat, а если текстовый тип преобразуем в целый тип, пишем StrToInt – «Строка_В_Целое число».

Сравните, например, a:=StrToFloat(Edit1.Text) и c:=StrToInt(Edit3.Text).

Далее мы выводим значение переменной x1 вещественного типа real на экран в Edit4.text и поэтому пишем edit4.Text:=FloatToStr(x1). Здесь уже, наоборот, с помощью оператора FloatToStr преобразуем вещественный тип real в строковый тип: «Вещественное число_В_Строку», ситуация обратная предыдущей.

Задание 1.2.2 Объявим глобальную переменную X типа integer.

Обратите внимание, что все переменные - a, b, D, x1, x2 и c в обработчике нажатия кнопки procedure TForm1.Button1Click(Sender: TObject), идущие после слова var, – являются локальными по отношению к обработчику и не будут видны из других мест модуля Unit1.pas кроме этой процедуры, в отличие от глобальных переменных.

Чтобы создать глобальную переменную X, в окне редактора кода найдите ключевое слово var, оно расположено над ключевым словом implemantation, рядом с ним найдете запись Form1: TForm1; объявите там имя X и тип глобальной переменной integer (рис. 1.21).

var // Это место условимся называть «главный var модуля»
Form1: TForm1; X:integer; // глобальная переменная X видна во всех
// процедурах и функциях модуля Unit1.pas

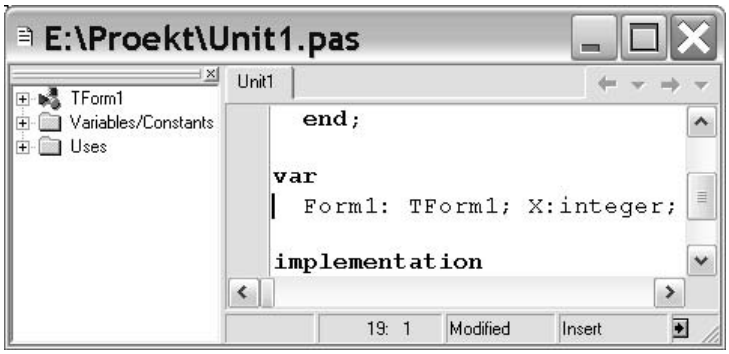


Рис. 1.21. Окно редактора кода. Объявляем глобальные переменные Form1 и X.

Таким образом, объявление переменной X в этом месте модуля делает ее глобальной по отношению ко всему модулю, то есть переменная X будет доступна в любом месте модуля приложения.

Задание 1.2.3 Используя компонент Timer1, запрограммируем так, чтобы через 1 секунду значение переменной целого типа увеличивалось и отображалась на форме.

Создайте новый проект в папке с названием «Timer».

Переместите на форму Label1, checkbox1, а также таймер timer1 из вкладки System палитры компонентов (рис. 1.22).

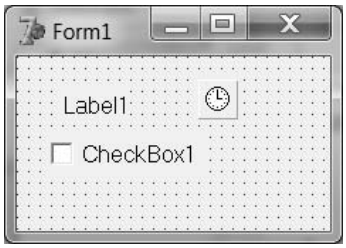


Рис. 1.22. Окно проекта с названием «Timer»

Установите следующие значения свойств компонентов приведенные в таблице:

Компонент	Свойство (properties)	Значение свойства
Label1	Caption	0
checkbox1	Checked	False
	Caption	Запуск
Timer1	Interval	1000
	Enabled	False

Здесь свойство Checked компонента checkbox1 является – булевским (логическим) значением, отвечающим на вопрос, поставлена (True) галочка в checkbox1 или нет (False).

Свойство Interval компонента timer1 определяет промежуток времени (в миллисекундах, 1000 миллисекунд это 1 секунда), через который будет генерироваться событие onTimer и, следовательно, будет вызываться обработчик данного события. У таймера всего одно событие -onTimer.

Создайте теперь глобальную переменную X типа integer (как ее создавать, сказано выше).

Затем настройте Object Inspector на форму Form1, выберите вкладку Events и создайте обработчик события onCreate.

Рядом с ключевым словом var имеется запись Form1:TForm1, поэтому в памяти компьютера может быть создан объект (переменная-указатель на объект Form1) класса TForm1. При этом каждый объект имеет определенный специфичный метод – constructor, который вызывается единственный раз и формирует объект в оперативной памяти. Можно сказать, что конструктор – это главный метод, который создает объект. При этом часто конструктор используется для инициализации (присвоения начальных значений) полей объекта. В Delphi в большинстве случаев имя конструктора называется «Create», что в переводе означает – «создать».

Поэтому в начале, когда создается объект Form1 – генерируется событие onCreate и в обработчике данного события onCreate мы присвоим начальное нулевое значение переменной X.

Используя объект инспектор перейдите в данный обработчик и напишите в этом обработчике всего одну строчку:

```
X:=0;//Инициализируем X начальным значением 0
```

Полностью данный обработчик будет выглядеть так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
X:=0; //Инициализируем X начальным значением 0
end;
```

Далее создайте обработчик события onClik (щелчка) на компоненте checkbox1 и напишите там следующее:

```
if CheckBox1.Checked=true then timer1.Enabled:=true
else timer1.Enabled:=false.
```

Если у компонента CheckBox1 поставлена галочка (CheckBox1.Checked=true), то нужно запустить таймер, другими словами, активировать таймер – timer1.Enabled:=true, а иначе – сделать таймер недоступным – timer1.Enabled=false, не выполняющим никаких действий.

Теперь пропишем обработчик таймера. Используя «Object Inspector» → «Events», создадим обработчик события onTimer для таймера Timer1. Полностью обработчик должен выглядеть так:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
X:=X+1; //увеличиваем значение X на 1 при каждом сообщении от таймера
Label1.Caption:=IntToStr(X); // выводим это значение в Label1
end.
```

Запустите программу, ставьте и снимайте галочку у Checkbox1. **Вернитесь в режим программирования.**

Задание 1.2.4. Загрузим в графический компонент Image1 картинку из графического файла.

В этом же проекте скиньте на форму графический компонент Image1 из вкладки Additional. Растяните его пошире, сбросьте рядом кнопку SpeedButton1, а также компонент OpenDialog1 из вкладки Dialogs, который представляет собой диалоговое окно выбора фалов.

Для Speed Button1 измените значение свойства Caption на «Загрузить картинку».

В обработчике щелчка на SpeedButton1 напишите:

```
if OpenDialog1.Execute=true then
Image1.Picture.LoadFromFile(OpenDialog1.FileName);
```

Запустите программу, нажмите кнопку, в появившемся диалоговом окне выберите какой-нибудь *.bmp файл и нажмите OK. Картинка должна нормально загрузиться в компонент Image1.

Рис. 1.23. Картинка помещенная в Image1

В данном случае, если диалоговое окно нормально загрузилось в память (if OpenDialog1.Execute=true), то в свойство Picture графического контейнера Image1 загружается рисунок из файла методом LoadFromFile. Имя файла содержится в свойстве FileName объекта OpenDialog1, мы этот файл выбираем мышкой. Пока вы не можете открыть *.jpg файл, как это сделать, будет рассказано ниже. **Вернитесь в режим программирования.**

Чтобы перемещаться между файлами модуля и главным файлом проекта, следует пользоваться сочетанием клавиш «Ctrl+F12».

ПРАКТИЧЕСКАЯ РАБОТА №2

2.1. Работа с буфером обмена, рисование мышью, контекстное меню, компонент Memo1 и Image1

Часто в приложениях нужно открыть какой – нибудь графический файл, изменить его содержимое, затем поместить это содержимое в буфер обмена, чтобы оно могло быть доступно другим приложениям и т.д. Также полезно иметь возможность рисования мышкой в окне созданного приложения. Часто при различных расчетах нужно в оперативном порядке вывести куда-нибудь рассчитанные программой результаты. Очень удобно и просто сделать это, используя стандартный компонент Memo1, очень похожий на стандартное приложение «Блокнот» операционной системы Windows. Описанные вещи кратко рассматриваются в данной практической работе.

Задание 2.1.1 Научимся работать с буфером обмена помещая и извлекая оттуда графические изображения.

Поместите на форму следующие компоненты: Image1 и Image2 (палитра Additional), OpenDialog1 (палитра Dialogs), а также 3 кнопки класса TButton, с заголовками на них так, как указано на рис. 2.1.

Рис. 2.1 Окно приложения «Буфер обмена»

Назначение кнопок данного приложения будет следующим:

Обработчик щелчка Button1 – Откроет картинку в Image1.
Обработчик щелчка Button2 – Скопирует картинку в буфер обмена – объект Clipboard.
Обработчик щелчка Button3 – Загрузит картинку в Image2.

Далее напишите имя подключаемого модуля, который описывает класс объекта Clipboard, т.е. буфера. Для этого в редакторе кода в самом верху после описания стандартных подключаемых модулей, идущих после ключевого слова **uses,** подпишите через запятую ClipBrd [2] (для буфера) и Jpeg для возможности открытия файлов *.jpg (рис. 2.2).

Также для обоих компонентов Image с помощью Object Inspector установите их свойство Stretch в значение True, чтобы рисунок подстраивался под их размер.

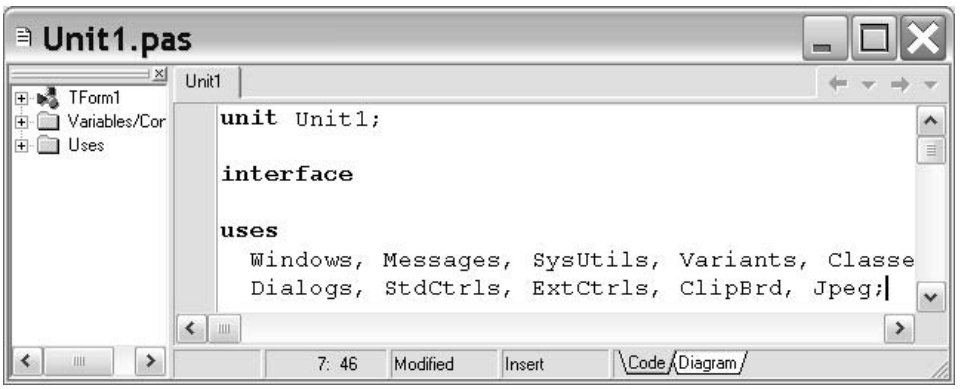


*Рис. 2.2 Подключаем модуль Jpeg для просмотра *.jpg файлов*

В обработчике щелчка Button1 напишите следующее:

```
opendialog1.execute; //запустить диалог выбора файлов

image1.Picture.LoadFromFile(opendialog1.FileName);
{Загрузить выбранный в диалоговом окне файл с именем opendialog1.FileName в свойство Picture компонента Image1}
```

В обработчике щелчка Button2 напишите следующее:

```
ClipBoard.Assign(Image1.Picture);
{Метод Assign копирует в буфер обмена графические объекты классов TGraphic, TBitmap, TPicture и т.д.}
```

И наконец, в обработчике щелчка Button3 напишите следующее:

```
If Clipboard.HasFormat(cf_picture)=true
then Image2.Picture.Assign(ClipBoard);

{Метод HasFormat определяет тип содержимого буфера обмена, то есть содержит ли буфер графический объект типа cf_picture}
```

Проанализируйте комментарии, а затем запустите программу. Сначала щелкните на Button1, в появившемся диалоговом окне выберите какой – нибудь графический файл с расширением *.bmp или *.jpg. После этого картинка из него отобразится на вашей форме в Image1. Далее щелкните на Button2 – скопировав картинку из Image1 в буфер. И как завершающий этап – сделайте щелчок на Button3, поместив изображение из буфера в Image2. Программа должна работать правильно. **Вернитесь в режим программирования.**

Задание 2.1.2 Нарисуем российский флаг в компоненте Image1

Покажем на данном примере, как работать с отдельным пикселом компонента Image1.

Создайте новый проект. Сбросьте на форму компонент Image1 и две кнопки Button. У компонента Image1 установите значения его обоих свойств Height и Width равными 400. То есть компонент Image1 будет иметь число пикселей, равное 400x400. Заголовок Caption кнопки Button1 установите в значение «Рисовать», а заголовок Button2 – в «Добавить оси». При щелчке на Button1 будет нарисован флаг России, а при щелчке на Button2 к нему добавятся координатные оси в центре. Это сделано для того, чтобы показать, что компонент Image1, в принципе, годится и для построения обыкновенных графиков или диаграмм.

Создайте обработчик щелчка на Button1, который должен выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var i,j,k,x,y:integer;
begin

{Рисуем флаг России 400x400}
for y:=1 to 400 do begin {цикл по всем y}
case y of
                {цикл по всем x, первые 133 строчки закрашиваем белым цветом}
1..133:begin for x:=1 to 400 do Image1.Canvas.Pixels[x,y]:=clWhite;  end;
      {цикл по всем x, вторые 133 строчки закрашиваем синим цветом}
134..266:begin for x:=1 to 400 do Image1.Canvas.Pixels[x,y]:=clBlue;  end;
      {цикл по всем x, третьи 133 строчки закрашиваем красным цветом}
267..400:begin for x:=1 to 400 do Image1.Canvas.Pixels[x,y]:=clRed;  end;
end;
                    end;
```

Нужно сказать, что очень многие графические операции выполняются в контексте специального свойства Canvas, которое содержится во многих компонентах. Например, у объекта Form1 также имеется свойство Canvas.

Canvas является полноценным объектом, имеющим свои методы (то есть функции и процедуры), ориентированные на работу с графикой, рисованием на форме и всем, что связано с графическим интерфейсом Windows [2]. В данном случае мы используем свойство Pixels[X, Y: Integer] объекта Canvas для доступа к цвету отдельного пикселя по его координатам относительно компонента Image1.

А теперь создайте обработчик щелчка на Button2, который должен выглядеть так:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
Image1.Canvas.MoveTo(1,200); Image1.Canvas.LineTo(400,200); {Рисуем Ось Y}
Image1.Canvas.TextOut(200+10,1,'Y');  {Подписываем Y}
Image1.Canvas.MoveTo(200,1); Image1.Canvas.LineTo(200,400); {Рисуем Ось X}
Image1.Canvas.TextOut(400-10,200-20,'X'); {Подписываем X}
        with Image1 do        begin
Canvas.TextOut(width div 2+2,Height div 2,IntToStr(0 ));
Canvas.TextOut(width div 2-100,Height div 2+5,IntToStr(-1));
Canvas.TextOut(width div 2-200,Height div 2+5,IntToStr(-2));
Canvas.TextOut(width div 2+100,Height div 2+5,IntToStr(1));
Canvas.TextOut(width div 2+194,Height div 2+5,IntToStr(2));
Canvas.TextOut(width div 2+5,Height div 2-100,IntToStr(1));
Canvas.TextOut(width div 2+5,Height div 2-200,IntToStr(2));
Canvas.TextOut(width div 2+5,Height div 2+100,IntToStr(-1));
Canvas.TextOut(width div 2+5,Height div 2+188,IntToStr(-2));
                              end;
end;
```

Здесь методом TextOut(X, Y: Integer; const Text: string) мы выводим нужную текстовую информацию, содержащуюся в переменной Text, в координаты X и Y, компонента Image1. Метод MoveTo объекта Canvas описан чуть ниже. Запустите программу. Щелкните сначала на кнопке Button1, в результате чего в компоненте Image1 нарисуется Российский флаг, а затем щелкните на кнопке Button2 и добавятся координатные оси в центре компонента.

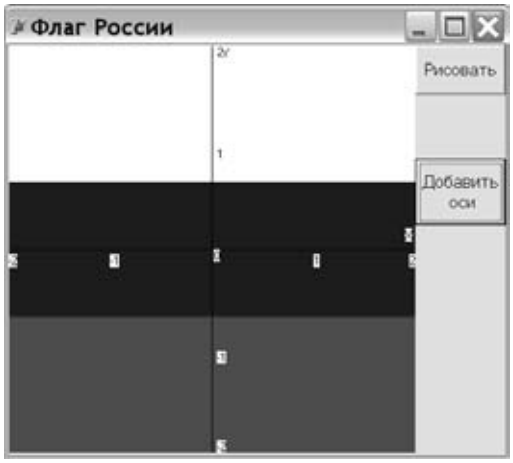


Рис. 2.3. Окно программы «Флаг России»

Данную заготовку в принципе можно использовать не только для рисунков, но и для построения некоторых диаграмм или графиков.

Вернитесь в режим программирования.

2.2. Рисование мышкой на форме. Контекстное меню

Задание 2.2.1 Запрограммируем возможность рисования мышкой на форме.

Достаточно часто представляет интерес обычное рисование мышью на форме, при этом с помощью рисунка иногда требуется выделить некоторую область на форме, рядом с кнопкой и т.д. Покажем, что рисование на форме в Delphi является достаточно простой задачей.

Создайте новый проект. Определим глобальную переменную булевского типа с именем Drug. Она будет являться идентификатором движения и принимать значение “Истина”, если движение есть, и “Ложь” в противном случае.

Для этого в окне редакторе кода объявите глобальной переменную Drug типа Boolean (рис. 2.4). Она будет устанавливаться в True, когда будет выполняться операция-**щелкни и перетащи.**

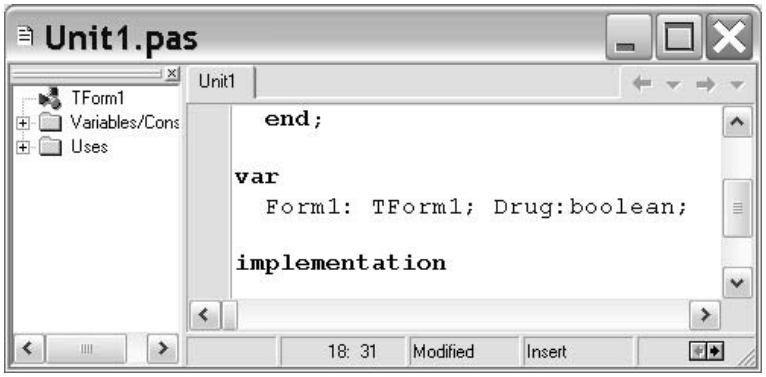


Рис. 2.4 Объявляем глобальную переменную Drug типа Boolean

Установите начальное значение Drug:=False (движения нет) в обработчике события onCreate для формы Form1.

Чтобы организовать операцию **«щелкни и перетащи»,** требуется 3 последовательных события и, соответственно, 3 обработчика этих событий.

Первое событие "Нажать кнопку мыши"- переходим в обработчик события onMouseDown нажатия кнопки мыши для формы и пишем там следующее:

Drug:=True; // устанавливаем, что движение должно быть
Canvas.MoveTo(X,Y); // устанавливаем координаты начала рисования

Обратите внимание, что в этом обработчике
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);

имеются значения X и Y типа Integer – это координаты текущего указателя мыши на форме. Именно они передаются в метод MoveTo(X,Y) объекта Canvas.

Таким образом, в метод MoveTo(X,Y) объекта Canvas передаются координаты текущего указателя мыши на форме, то есть данным методом устанавливаются координаты, с которых начнется рисование на форме Form1.

Второе событие "Движение мыши" – переходим в обработчик события onMouseMove движения мыши на форме Form1 и пишем там следующее:

if Drug=true then Canvas.LineTo(X,Y)

{ LineTo – рисует линию от предыдущей точки к текущей, здесь передаваемые X и Y будут уже другими, так как указатель мыши уже на другой точке формы после перемещения}

Третье событие "Отпускаем кнопку мыши" – переходим в обработчик события onMouseUp формы Form1 отпускания кнопки мыши и пишем там следующее:

Drug:=False; // устанавливаем что движения недолжно быть

Запустите программу. Попробуйте порисовать мышкой на форме. Обратите внимание, что, когда вы двигаете мышью без нажатой кнопки мыши, событие onMouseMove движения мыши на форме Form1, тем не менее, все равно генерируется, просто теперь Drug равен False и линия не рисуется. **Вернитесь в режим программирования.**

Задание 2.2.2 Будем изменять толщину и цвет рисуемых мышкой линий

Здесь же сбросьте на форму кнопку Button1 и следующие компоненты с соответствующими значениями их свойств, описанные в таблице:

Компонент	Свойство	Значение свойства
Label1	Caption	Цвет
Label2	Caption	Толщина
Edit1	Text	100000
Edit2	Text	10

Разместите их так, как показано на рисунке 2.5.

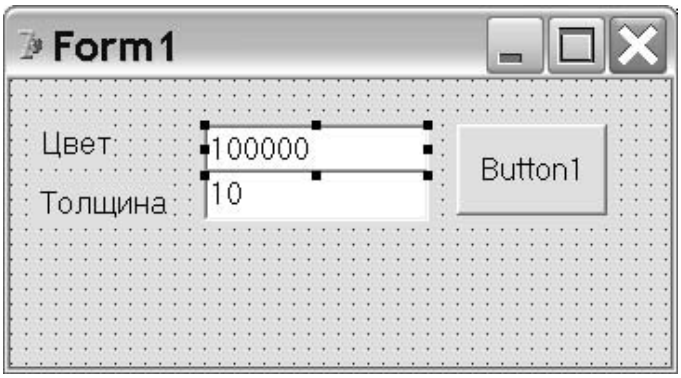


Рис. 2.5. Окно приложения «Толщина и цвет линий»

После щелчка на кнопке Button1, цвет и толщина рисуемых линий должны измениться.
Также ради интереса выведем текст “УРА” в область с координатами (0,0).
Для этого в обработчике щелчка на кнопке напишите:

```
Canvas.Pen.Color:=StrToInt(Edit1.Text); //Устанавливаем цвет
Canvas.Pen.Width:=StrToInt(Edit2.Text);//Устанавливаем толщину
Canvas.Font.Size:=18;//Устанавливаем размер шрифта для текста
Canvas.TextOut(0,0,'УРА');//Выводим на форму текст
```

Запустите программу. Щелкните на Button1 и попробуйте что-нибудь нарисовать на форме. Вы получите подобный рисунок:

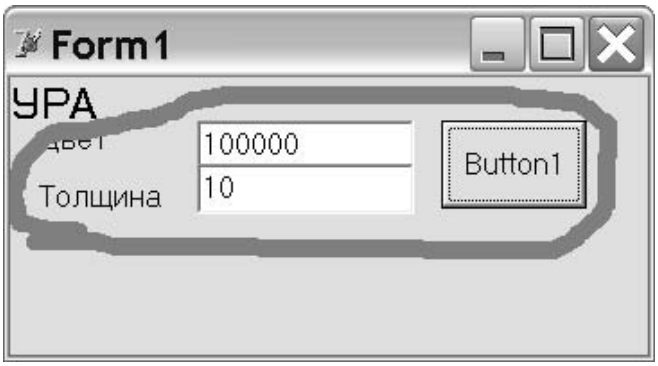


Рис. 2.6 Вид окна после запуска приложения

Вернитесь в режим программирования.

Задание 2.2.3 Научимся работать с контекстным меню MainMenu, в котором определим вкладки для очистки формы и закрытия приложения.

Добавьте на форму контекстное меню MainMenu1 (палитра Standard), щелкните по этому компоненту MainMenu1 двойным щелчком, и появится следующее окно (рис. 2.7)

Рис. 2.7 Окно приложения «Контекстное меню» пока еще пустое

Затем в свойстве Caption выделенной вкладки напишите “File”, а также добавьте еще вкладки “Очистить” и “Выход”, как показано на рисунке (рис. 2.8)

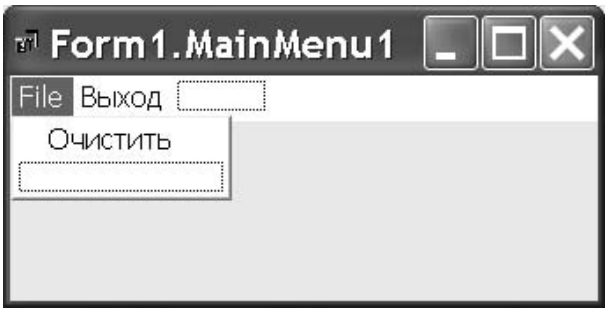


Рис. 2.8 Окно приложения «Контекстное меню» с пунктами меню

Щелкните двойным щелчком по вкладке “Очистить” и войдите в обработчик щелчка на вкладке – procedure TForm1.N1Click(Sender: TObject).

Здесь надпись “Очистить” – это значение свойства Caption объекта с именем N1, который принадлежит объекту MainMenu1, именно в его обработчик мы и зашли.

В этом обработчике напишите следующее:

```
Canvas.Brush:=Form1.Brush;
{Устанавливаем значение свойства Brush объекта Canvas в значение свойства Brush объекта формы.}
Canvas.FillRect(Form1.ClientRect);
{Методом FillRect объекта Canvas закрашиваем форму ее же цветом}
```

Аналогично щелкните двойным щелчком по вкладке “Выход” и напишите в обработчике:

```
Close; {завершить программу}
```

Запустите программу. Нарисуйте что-нибудь на форме. Затем, используя контекстное меню, очистите форму. Фактически очистить форму, в нашем случае, означает просто полностью закрасить форму ее же цветом. Поэтому мы передаем методу FillRect объекта Canvas всю клиентскую область формы Form1.ClientRect.

Вернитесь в режим программирования.

2.3. Компонент Memo

Задание 2.3.1 Изучим компонент Memo1

Часто в Windows-приложениях присутствует что-то наподобие стандартного текстового блокнота (Notepad). Компонент Memo как раз во многом подобен такому блокноту.

У компонента Memo, как и у Notepad, есть методы записи и загрузки текстовых файлов. Так как компонент Memo работает только с текстовым форматом, то при выводе в него некоторых числовых значений их требуется предварительно перевести в текстовый формат.

Создайте новый проект.

Переместите Memo1 на форму. Слегка его растяните и сбросьте кнопку Button1 на форму (рис. 2.9).

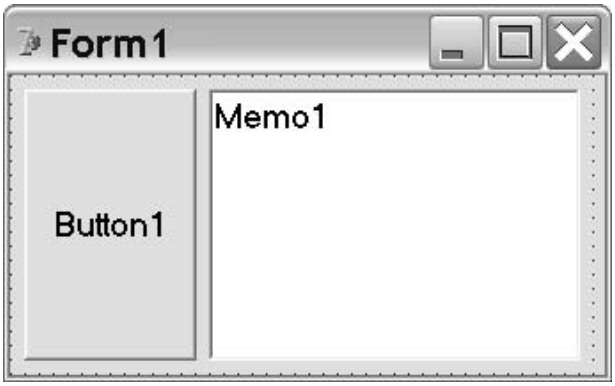


Рис. 2.9 Демонстрация работы компонента Memo1

С помощью Object Inspector зайдите в свойство Lines компонента Memo1. Для этого щелкните по трем точкам, что справа от надписи свойства Lines (рис. 2.10).

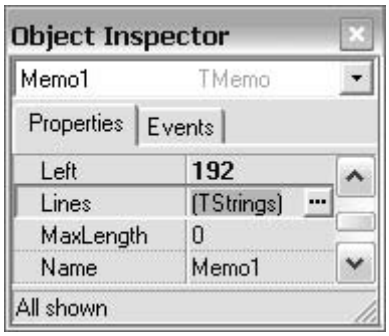


Рис. 2.10 Свойство Lines компонента Memo1 в Object Inspector

Перед вами откроется окно ввода строк String List Editor(рис. 2.11). Каждая строка компонента Memo1 является строкой типа String. Вы можете спокойно вводить сюда текст, но в данном задании этого делать не требуется, поэтому просто очистите строки компонента, стерев строку с надписью Memo1.

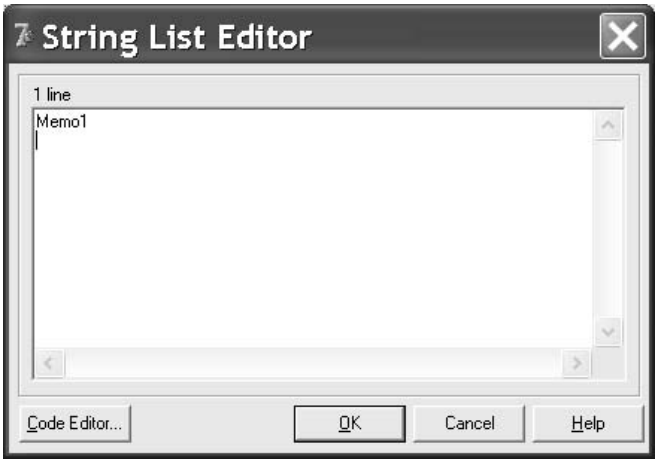


Рис. 2.11 Окно ввода строк String List компонента Memo1

Задание 2.3.2 Выведем 2 текстовых строки в компонент Memo1 при щелчке на Button1.

Зайдите в обработчик щелчка кнопки Button1. Напишите там следующее:

```
memo1.Lines.Add('Студент');
memo1.Lines.Add('Информатика');
```

Здесь метод Add() свойства Lines объекта Memo1 добавляет еще одну строку в Memo1. В скобках метода Add() должен быть либо текст между апострофами, например ‘Студент’, либо переменная строкового типа string.

Запустите программу. Щелкните на кнопке Button1. В Memo1 должны отобразиться запрограммированные вами слова. Обратите внимание, что метод Add при его вызове каждый раз добавляет именно новую строку.

Вернитесь в режим программирования.

Задание 2.3.3 Программно сформируем текстовую строку и выведем ее в компонент Memo1.

В данном случае программно сформируем простую текстовую строку«1 2 3 4 5», представляющую из себя разделенную пробелами последовательность цифр.

Для этого добавьте кнопку Button2 на форму. Обработчик щелчка на ней должен быть следующим:

```
var i:integer; s:string;
begin
s:=’’; {устанавливаем начальную строку пустой}
{формируем строку s накапливая в ней цифры и пробелы между ними}
for i:=1 to 5 do s:=s+’ ‘+IntToStr(i);
Memo1.Lines.Add(s); { выводим строку в Memo1}
end.
```

Запустите программу. Щелкните на кнопке Button2. В Memo1 должна отобразиться сформированная строка. **Вернитесь в режим программирования.**

Уточним, что к строкам компонента Memo1 можно обращаться как к массиву строк типа string через его свойство Lines. Например, чтобы обратиться к первой строке Memo1, требуется инструкция – Memo1.Lines[0], ко второй стоке – Memo1.Lines[1] и т.д.

Задание 2.3.4 Требуется вывести в Memo1 матрицу 3x4, заполненную случайными числами в диапазоне от нуля до 100.

Для этого в этом же проекте переместите еще одну кнопку Button3 на форму.
Полный код обработчика щелчка на ней должен быть следующий:

```
procedure TForm1.Button3Click(Sender: Tobject);
var
i, j:word;{Определяем переменные целого типа, индексирующие строку и столбец}
massiv:array[1..3,1..4] of word; {Определяем двумерный массив(матрицу) целых чисел }
s:string; {Определяем строку для заполнения ее цифрами и последующего вывода}

begin
Memo1.Clear;//Очищаем Memo1
randomize;// Инициализируем генератор случайных чисел
for i:=1 to 3 do //Делаем двойной цикл, чтобы заполнить массив massiv[i,j]
for j:=1 to 4 do
massiv[i,j]:=random(101);{ случайными числами от 0 до 100 функцией random}

for i:=1 to 3 do begin // Цикл по строкам с индексом i
s:=''; // очищаем переменную s, содержащую строку
        for j:=1 to 4 do // Цикл по столбцам с индексом j
        s:=s+' '+IntToStr(massiv[i,j]);//Формируем строку s из столбцов j
Memo1.Lines.Add(s);//Выводим строку в Memo1
                        end;

end;
```

Запустите программу. Щелкните на кнопке Button3. Вы получите примерно следующий результат:

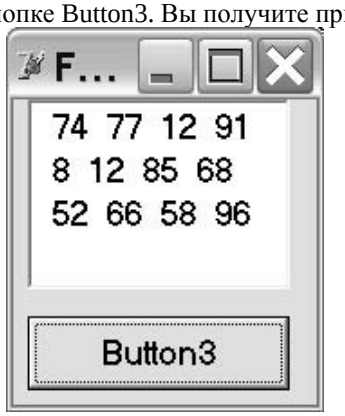


Рис. 2.12 Матрица 3x4, заполненная случайными числами в Memo1

Вернитесь в режим программирования.

Еще раз внимательно проанализируйте комментарии к программному коду. Попробуйте обменять элементы первого и последнего столбика матрицы.

Последний пример дополнительно демонстрирует, что компонент Memo1 также будет полезен при переносе программ написанных на языке Паскаль в Delphi. То есть теперь текстовый компонент Memo1 в каком – то смысле может заменить тот черный экран, куда выводились результаты работы программ, написанных на языке Паскаль.

ПРАКТИЧЕСКАЯ РАБОТА № 3

3.1. Перемещение мышью панели Panel1 на форме

Во многих оконных Windows – приложениях очень часто требуется перемещать мышкой панель с расположенными на ней кнопками, пиктограммами и т.д. При этом часто этих панелей бывает несколько. Поэтому разберемся, каким образом происходит такое перемещение. В данном случае программный код и основные действия будут аналогичны программированию рисования мышкой на форме из предыдущей работы.

Задание 3.1.1 Реализуем технологию «Drag and Drop» для перемещения мышкой панели.

Создайте новый проект. Поместите на форму компонент Panel1 из палитры Standard.

Координаты компонентов относительно формы, в том числе и компонента Panel1, определяются координатами точки левого верхнего угла компонента и определяются значениями свойств left и top. Так, значение Panel1.left соответствует обычной координате по оси X, направленной слева направо, а значение Panel1.top соответствует координате по оси Y, направленной сверху вниз (рис. 3.1). Эти координатные оси условно пересекаются в крайнем левом верхнем углу формы.

Объявите глобальными переменные Xset, Yset:integer; Drug:Boolean;

Присвойте начальное значение переменной Drug, записав инструкцию Drug:=False в обработчике события onCreate формы.

Задача разбивается на 3 этапа:
1) щелкаем мышью на панели,
2) не отпуская кнопки мыши двигаем панель,
3) отпускаем кнопку мыши.

Рассмотрим все эти этапы.

Этап первый – щелкаем мышью на панели (на рис. 3.1 место щелчка отмечено точкой).

Создайте обработчик события onMouseDown компонента Panel1, который должен выглядеть следующим образом:

```
procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin

Drug:=true; {определяем в переменную Drug что движение должно быть, так как произведен щелчок мыши по форме}
Xset:=X; {координаты указателя мыши относительно компонента Panel1}
Yset:=Y;
SetCapture(Form1.Handle); {устанавливаем перехват формой всех событий}

end.
```

В данном случае переменным Xset и Yset присваиваем координаты указателя мыши относительно компонента Panel1.

Обратите внимание, что в параметрах данного обработчика:

procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; **X, Y: Integer**);

уже имеются координаты указателя мыши относительно компонента Panel1, обозначенные как **X, Y: Integer** вот эти значения мы и присваиваем глобальным переменным Xset и Yset.

Оператор SetCapture(Form1.Handle) устанавливает перехват всех событий, сообщений и перенаправляет их от всех компонентов к форме Form1, чтобы только форма реагировала на эти события и обрабатывала их, поэтому уже движение мыши (хоть указатель мыши и находится на Panel1) будет относиться уже не Panel1, а к форме и все события будут обрабатываться именно формой, а не любым другим компонентом.

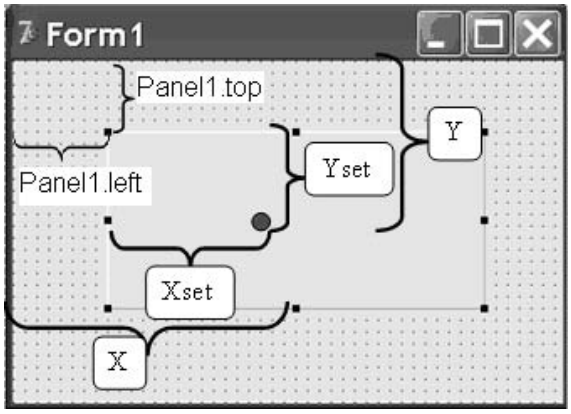


Рис. 3.1 Схема координат указателя мыши относительно Panel и Form1

Этап второй – не отпуская кнопки мыши, двигаем панель.

Но теперь мы должны знать координаты указателя мыши относительно формы, и панель также мы двигаем по форме, поэтому мы должны перейти в обработчик события onMouseMove движения мыши для формы Form1. Напишите в этом обработчике:

```
If Drug=true then    begin { проверка на нажатость кнопки мыши для движения }
                     Panel1.left:=X-Xset;
                     Panel1.top:=Y-Yset;
                              end.
```

В данном обработчике X и Y это координаты указателя мыши уже относительно Form1, так как форма перехватывает все события. Таким образом, мы легко вычисляем координаты Panel1 относительно формы Panel1.left и Panel1.top.

Этап третий – отпускаем кнопку мыши.

Зайдите в обработчик события onMouseUp формы Form1, напишите там следующее:

```
Drug:=False; {индикатор движения Drug устанавливаем ложным, так как отжали кнопку мыши}
ReleaseCapture; {нормальная обработка событий}
```

Здесь оператор ReleaseCapture [2] возвращает нормальный режим обработки событий.

Запустите программу. Попробуйте перемещать мышкой панель по форме. Она должна нормально двигаться. Убедитесь в правильности ее работы. **Перейдите назад в режим программирования.**

В этом же проекте скиньте на форму кнопку SpeedButton1 из палитры Additional.

Напишите в обработчике щелчка:

```
SpeedButton1.Parent:=Panel1;
SpeedButton1.Left:=0;
SpeedButton1.top:=0;
```

В данном случае выражение SpeedButton1.Parent:=Panel1 определяет родительский компонент Panel1, который содержит в себе данную кнопку SpeedButton1, а затем следующими двумя инструкциями устанавливаем координаты данной кнопки относительно Panel1.

Запустите программу. При щелчке на SpeedButton1 она должна переместиться на компонент Panel1. **Вернитесь в режим программирования.**

3.2. Работа с таймером на примере программы «Бильярд»

Задание 3.2.1 Бильярд. Запрограммируем движение компонента Shape1 по форме так, чтобы он отталкивался от ее краев как бильярдный шарик.

Создайте новый проект. Добавьте компоненты Shape1 и timer1 на форму. Здесь компонент Shape1 представляет собой одну из простых стандартных фигур.

Измените значение свойства Shape компонента Shape1 на stCircle (рис. 3.2)

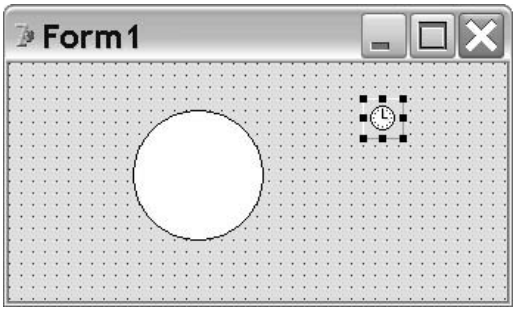


Рис. 3.2 Окно приложения «Бильярд»

Объявите глобальные переменные dX, dY:integer. Проинициализируйте их начальными значениями в обработчике события onCreate формы, написав там: dX:=1; dY:=1;

Измените значение свойства interval компонента timer1 на 100.
Зайдите в обработчик события onTimer таймера и напишите там код, определяющий движение по оси X:

```
Shape1.Left:=Shape1.Left+dx;
if (Form1.ClientWidth-Shape1.Left<=Shape1.Width)
or (Shape1.Left<=0) then dx:=-dx;
```

Выражение Shape1.Left:=Shape1.Left+dx определяет изменение координаты по оси X компонента Shape1.
Form1.ClientWidth – ширина всей формы
Shape1.Left – координата, соответствующая оси X компонента Shape1 относительно формы, отсчитывается от верхнего левого угла Shape1.
Shape1.Width – ширина Shape1.
Очевидно, что, когда одно из условий отталкивания Shape1 от правого края (Form1.ClientWidth-Shape1.Left<=Shape1.Width) или левого края (Shape1.Left<=0), формы будут равны True, то следует просто сменить знак приращения dx (то есть умножить на -1), чтобы значение Shape1.Left либо увеличивать, либо уменьшать.

Запустите программу. Компонент Shape1 в форме шарика должен летать по оси x, отталкиваясь от правого и левого краев формы. Убедитесь в правильности работы программы. **Вернитесь в режим программирования.**

Задание 3.2.2 Зайдите снова в обработчик таймера и самостоятельно добавьте движение по оси Y.

Учитываете следующее:
Form1.ClientHeight – высота всей формы.
Shape1.Top – координата, соответствующая оси Y компонента Shape1 относительно формы, отсчитывается от верхнего левого угла Shape1.
Shape1.Height – высота Shape1.

Запустите программу. Убедитесь в правильности ее работы. **Вернитесь в режим программирования.**

3.3. Изменим надпись на кнопке «Пуск»

Задание 3.3.1 Изменим надпись на кнопке «Пуск» основного меню Windows.
Данным маленьким примером просто покажем, что в Delphi можно менять надписи на некоторых кнопках стандартных приложений ОС Windows.
Создайте новый проект. Сбросьте на форму Edit1 и Button1. Установите значения соответствующих свойств Text и Caption (рис. 3.3).

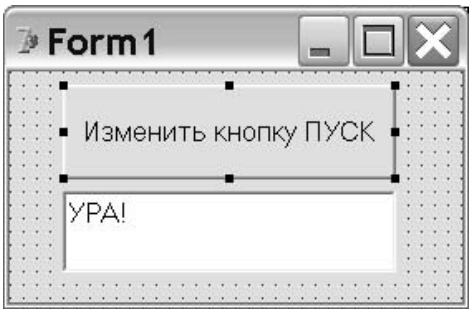


Рис. 3.3 Приложение, изменяющее надпись на стандартной кнопке «Пуск» меню Windows

Обработчик щелчка на Button1 должен быть следующим:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 h, h1: hwnd;  name:string;
begin
 name:=edit1.Text;
 h := findwindow('Shell_TrayWnd', nil);
 h1 := findwindowex(h, 0, 'Button', nil);
 setwindowtext(h1, PANSICHAR(name));
end;
```

Запустите программу. Напишите слово в поле Edit, нажмите вашу кнопку Button1, а затем подведите курсор мыши к кнопке ПУСК.

Не вдаваясь в подробности, можно сказать, что функции findwindow и findwindowex определяют нужные дескрипторы определенных окон Windows. Дескриптор окна это - уникальный индентификатор окна, который присваивается операционной системой во время создания окна.

Функция SetWindowText выводит текст в окно с дескриптором h1 типа HWND. **Вернитесь в режим программирования**.

ПРАКТИЧЕСКАЯ РАБОТА №4

4.1. Построение графиков в среде Delphi

Очень часто полученную в результате работы программы информацию приходится отображать в виде диаграмм или графиков. Среда объектно ориентированного программирования Delphi предоставляет удобный стандартный компонент Chart класса TChart.

Задание 4.1.1 Рассмотрим стандартный компонент диаграмм и графиков Chart1

Создайте новый проект. Перенесите на форму кнопки Button1, Button2, Button3, Button4, SpeedButton1, а также компонент Chart1 из палитры Additional, позволяющий строить различные графики и диаграммы.

Щелкните правой кнопкой мыши на Chart1. Появится контекстное меню. Затем выберите «Edit Chart» (либо просто дважды щелкните на объекте Chart1).

Появится следующее окно :

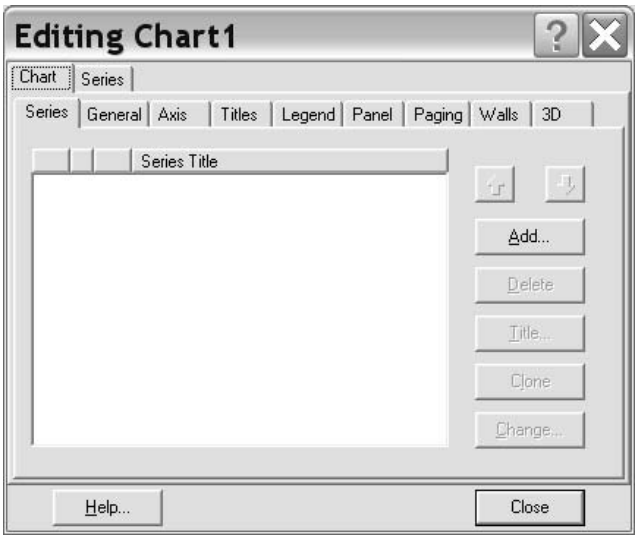


Рис. 4.1 Редактор компонента диаграмм и графиков Chart1

Чтобы добавить график или диаграмму, выберете вкладку Add. Появится диалоговое окно TeeChartGallery, затем выберите Line (он выбран по умолчанию) и снимите флажок трехмерного отображения (рис. 4.2)

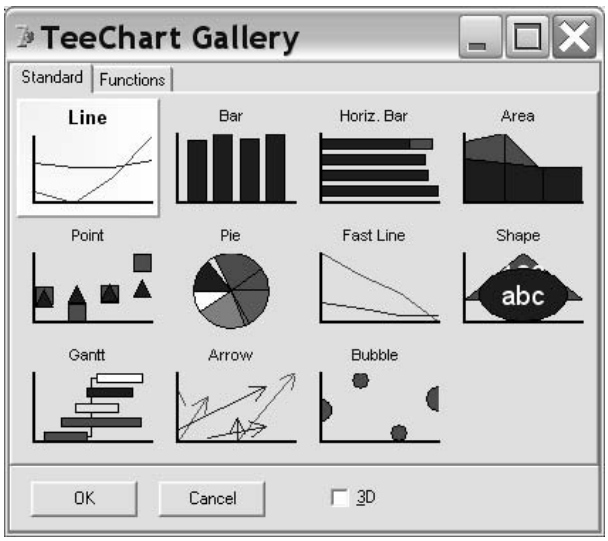


Рис. 4.2 Окно выбора отображения информации в Chart1

Нажмите OK. По умолчанию имя вашего графика Series1 (рис. 4.3) – это один из множества возможных наборов точек данных, принадлежащих объекту Chart1 [2]. Каждый конкретный набор точек данных – это также объект, принадлежащий диаграмме.

Каждый набор имеет множество опций для отображения его на диаграмме. В данном случае под диаграммой понимается сам объект Chart1. Необходимо отметить и то, что класс Tchart, которому принадлежит компонент Chart1, является наследником класса TPanel, поэтому он обладает родительским свойством parent и может содержать в себе некоторые другие объекты типа TButton, TSpeedButton и т.д.

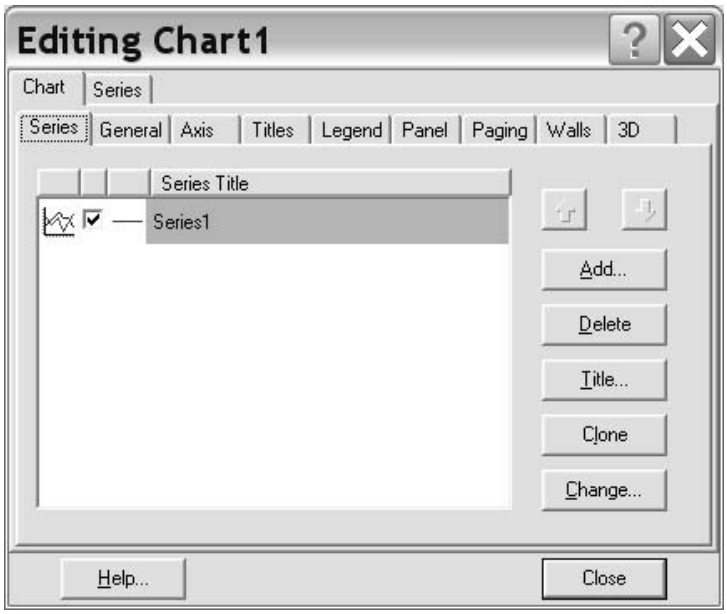


Рис. 4.3 С помощью кнопки «Add» можно добавлять графики и диаграммы с именами по умолчанию Series2, Series3 и т. д.

Теперь щелкните на вкладке Titles и поменяйте там заголовок по умолчанию на заголовок – “График функции”, должно получиться так (рис. 4.4):

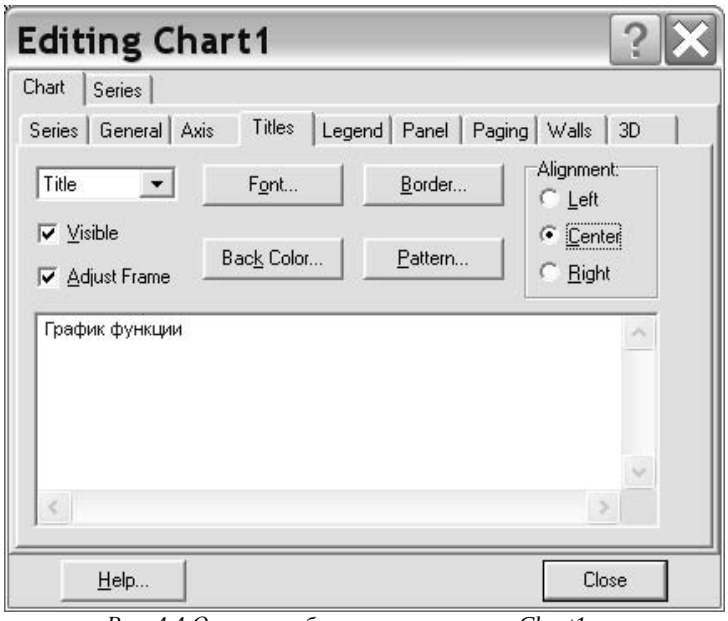


Рис. 4.4 Опции отображения заголовка Chart1

Нажмите Close. Далее мы непосредственно попробуем построить простейший график.

Задание 4.1.2 Запрограммируем щелчок кнопки так, чтобы построить график Series1 для точек с некоторыми координатами используя метод AddXY(x:double, y:double, [s:string])

Из самого названия ясно, что метод AddXY(x:double, y:double, *[s:string]*) устанавливает курсор в новую точку с координатами X и Y и при последующем вызове AddXY рисует линию между старой и новой точками. Параметр *[s:string]* является необязательным.

Создайте обработчик щелчка кнопки Button1
procedure TForm1.Button1Click(Sender: Tobject);

Напишите там следующее:

```
Series1.AddXY(10,10,'Hello');
Series1.AddXY(20,20,'World');
Series1.AddXY(30,30,'Russia');
Series1.AddXY(40,40,'Belarus');
Series1.AddXY(50,50,' Ukraine').
```

Запустите вашу программу и щелкните кнопку Button1. Попробуем изучить возможности созданного графика. Для изменения масштаба наведите указатель мыши на график и, нажав левую кнопку мыши, потяните чуть вправо и ниже, затем отпустите мышь. Масштаб увеличится.

Чтобы возвратить нормальный масштаб, аналогично потяните чуть влево вверх, а затем отпустите.

Чтобы двигать график в сторону, зажмите правую кнопку мыши и двигайте ею.

Чтобы очистить график, измените у кнопки SpeedButton1 значение свойства Caption на "Очистить", а в обработчике щелчка на SpeedButton1 просто напишите инструкцию Series1.clear – что означает очистку содержимого Series1.

Запустите программу. Программа должна работать верно, и при щелчке на SpeedButton1 график должен очищаться. **Вернитесь в режим программирования.**

Задание 4.1.3 Запрограммируем обработчик щелчка кнопки Button2 для построения графика функции $y=x^2$.

Создайте обработчик щелчка кнопки Button2, который должен выглядеть следующим образом:

```
procedure Tform1.Button2Click(Sender: Tobject);
var x:integer; y:double;
begin
Series1.clear; {очищаем график}
for x:=-10 to 10 do begin
y:=sqr(x); { рассчитываем значение y=x^2}
Series1.AddXY(x,y); {отображаем на диаграмме}
                                end;
end;
```

Запустите программу. Вы получите следующий результат:

Рис. 4.5 График функции y = x2. построенный в Chart1

Закройте программу. **Вернитесь в режим программирования.**

Задание 4.1.4 Самостоятельно. В этом же компоненте Chart1 создадим еще три набора Series2, Series3, Series4 для расчета конкретных функций.

Запрограммируйте обработчик щелчка кнопки Button3 так, чтобы построить график для Series2: функция $y=x^3$, x меняется от -100 до 100 с шагом 1.
Запрограммируйте обработчик щелчка кнопки Button4 так, чтобы построить график для Series3: функция y=2*cos(x), x меняется от -360 до 360 с шагом 1.

Примечание. В Delphi у тригонометрических функций типа cos(x), аргумент x – это значение данное в радианах. Чтобы перейти к градусам, нужно умножить x на·π/180. В Delphi значение числа π находится в переменной Pi.

Запрограммируйте обработчик щелчка кнопки Button5 так, чтобы построить график для Series4: функция y= Ln(x*0.1), x меняется от 1 до 100 с шагом 1.

Запустите программу, убедитесь, что она работает нормально и при нажатии на соответствующие кнопки график строятся. **Вернитесь в режим программирования.**

Задание 4.1.5 Сохраним построенный график в графический файл.

Создайте контекстное меню с вкладками «File» → «Save» и «Exit».
Запрограммируйте вкладку «Save» для сохранения вашего графика в файл, напишите в обработчике:

```
chart1.SaveToBitmapFile('graphic.bmp');
```

Запустите программу, отобразите любой график путем щелчка на любой кнопке Button и щелкните File → Save. Закройте программу. **Вернитесь в режим программирования.** Затем зайдите в папку вашего проекта и вы обнаружите там графический файл 'graphic.bmp' с рисунком вашего графика.

4.2. Моделирование бегущих волн

Бегущими волнами называются волны, которые переносят в пространстве энергию.
Уравнение бегущей волны следующее $\xi(x,t) = \mathrm{A}\cos(\omega \cdot t - \kappa \cdot x + \varphi_0)$, где:

A- амплитуда волны (в метрах).
ω – цикличиская частота волны (в радианах в секунду).
x – координата.
k – волновое число.
φ_0 – начальная фаза волны для простоты возьмем φ_0=0.

Задание 4.2.1. **Используя компонент Chart1 смоделируем бегущие волны.**

Создайте новый проект. Поместите компонент Chart1 на форму. Также добавьте на форму компоненты, указанные в таблице, со следующими значениями их свойств, указанными в этой же таблице:

Компонент	Значение свойства	Описание в контексте программы
	Caption	
Label1	W1	Частота первой волны
Label2	dT	Приращение времени
Label3	Ym1	Амплитуда первой волны
Label4	W2	Частота второй волны
Label5	Ym2	Амплитуда второй волны
Edit1	0,01	Значение частоты первой волны
Edit2	1	Значение приращения времени
Edit3	1	Значение амплитуды первой волны
Edit4	-0,05	Значение частоты второй волны
Edit5	3	Значение амплитуды второй волны

Компонент	Свойство	Значение свойства
checkbox1	**Checked**	**False**
	Caption	**Запуск волны**
checkbox2	**Checked**	**False**
	Caption	**3D**
Timer1	**Interval**	**10**
	Enabled	**False**
Button1	**Caption**	**Установить**

Разместите Label1 напротив Edit1, Label2 напротив Edit2, Label3 напротив Edit3, Label4 напротив Edit4, а также Label5 напротив Edit5. Разместите все эти компоненты так, как показано на рисунке 4.6. В компоненте Chart1 создайте графики Series1, Series2 и Series3.

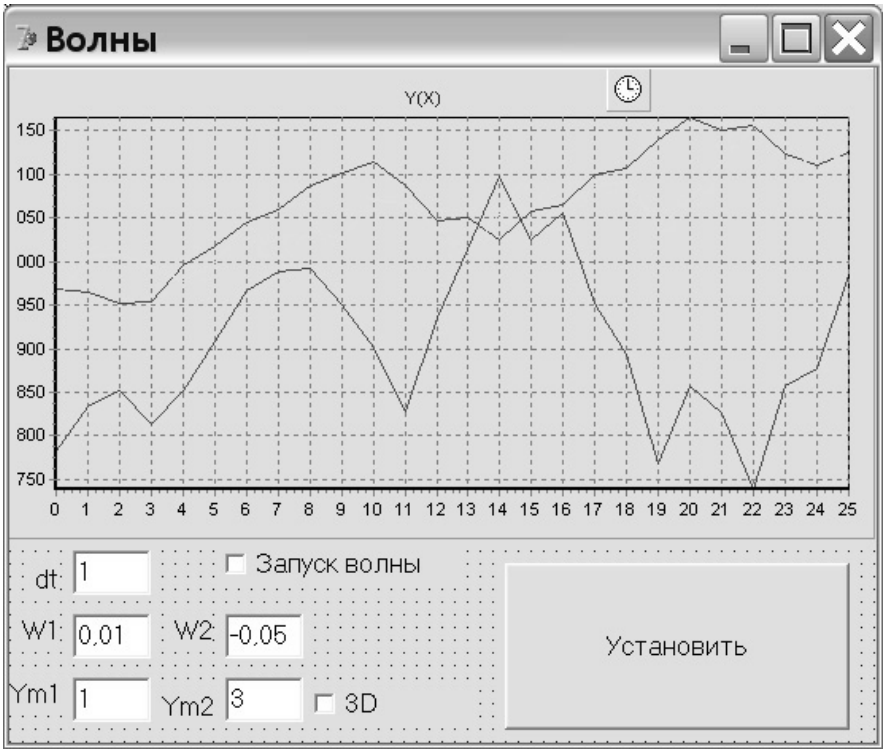


Рис. 4.6 Окно приложения «Бегущие Волны»

Задача создаваемого приложения состоит в следующем: сначала устанавливаем значения всех параметров (амплитуды, частоты), затем щелкаем на кнопке «Установить», затем ставим галочку на «Запуск волны» и моделируемые волны должны отображаться в компоненте Chart1.

Создайте глобальные переменные Ym1,W1,Ym2,W2:double; t, Dt:word.

Проинициализируете их начальными значениями в обработчике события onCreate формы, для этого напишите там следующее:

```
t:=0; {начальное значение времени}
Ym1:=StrToFloat(Edit3.Text);
W1:=StrToFloat(Edit1.Text);
Ym2:=StrToFloat(Edit5.Text);
W2:=StrToFloat(Edit4.Text);
Dt:=StrToInt(Edit2.Text).
```

То же самое напишите в обработчике нажатия кнопки Button1, **либо, (Важно!!!), чтобы не дублировать дважды один и тот же текст для Button1 на вкладке Events, для события onClick, щелкните справа на раскрывающемся списке и настройте это событие на обработчик FormCreate (путем простого выбора из списка). Аналогичные операции можно осуществлять с любыми событиями, настраивая события на любые другие обработчики, уже прописанные ранее, для других объектов.**

В обработчике события onClick щелчка на CheckBox1 напишите следующее:

```
if CheckBox1.Checked then timer1.Enabled:=true else
timer1.Enabled:=not true;
```

Таким образом, если галочка установлена, то активируем таймер, в котором уже прописано моделирование волны, иначе останавливаем таймер.

Теперь непосредственно выведем бегущую волну в объект Chart1, для этого запрограммируем обработчик таймера следующим образом:

```
var x:integer; Y1:double;
begin
Series1.Clear;
t:=t+dt;
for x:=-360 to 360 do begin          {*****Начало блока*****}
Y1:=Ym1*cos(x*pi/180+w1*t);
Series1.AddXY(X,Y1,'');
                end;                 {*****Конец блока*****}
end;
```

Запустите программу. Установите галочку напротив «Запуск волны». Вы получите смоделированную бегущую волну. Меняйте значения параметров, ставьте как положительные, так и отрицательные значения частоты, а затем гажимайте кнопку «Установить». Изучайте, как меняется при этом скорость и направление волны. Чтобы зафиксировать масштаб, щелкните по компоненту Chart1 правой кнопкой мыши. Чтобы остановить движение волны, снова снимите галочку напротив «Запуск волны».

Закройте программу и **вернитесь в режим программирования.**

Задание 4.2.2. Добавим вторую волну, в обработчик этого же таймера.

Указание. Для расчетов будем использовать уже готовый цикл for, имеющийся в обработчике, то есть для второй волны не нужно еще раз прописывать второй for.

Немного измените предыдущий обработчик таймера следующим образом:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var x:integer; Y1, Y2:double;
begin
Series1.Clear; Series2.Clear;
t:=t+dt;
for x:=-360 to 360 do begin          {*****Начало блока*****}
Y1:=Ym1*cos(x*pi/180+w1*t);  {первая волна}
Series1.AddXY(X,Y1,'');

Y2:=Ym2*cos(x*pi/180+w2*t);  {вторая волна}
Series2.AddXY(X,Y2,'');
                end;                 {*****Конец блока*****}
end;
```

Запустите программу и установите галочку на «Запуск волны». Вы получите две движущиеся бегущие волны.

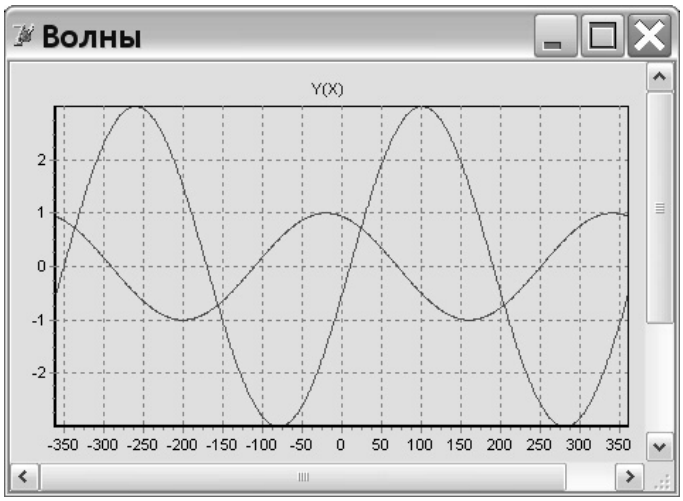


Рис. 4.7. Окно приложения «Волны» демонстрирует моделирование двух бегущих волн

Задание 4.2.3. Учитывая, что эти 2 волны складываются в пространстве (интерферируют), построим третью результирующую волну в этом же таймере.

У нас график Series3 будет соответствовать результирующей волне, то есть будет соответствовать интерференции двух волн, отображаемых графиками Series1 и Series2.

Смысл результирующей волны в следующем: к волнам применяется принцип суперпозиции волн – при распространении волн каждая из них распространяется так, как будто другие волны отсутствуют, а результирующее смещение частиц среды в любой момент времени равно геометрической сумме смещений, которые получают частицы, участвуя в каждом из слагающих волновых процессов.

Снова измените обработчик таймера следующим образом:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var x:integer; Y1, Y2, Y3:double;
begin
Series1.Clear; Series2.Clear; Series3.Clear;
t:=t+dt;
for x:=-360 to 360 do begin                {*****Начало блока*****}
Y1:=Ym1*cos(x*pi/180+w1*t);  {первая волна}
Series1.AddXY(X,Y1,'');

Y2:=Ym2*cos(x*pi/180+w2*t);  {вторая волна}
Series2.AddXY(X,Y2,'');

Y3:=Y1+Y2;
Series3.AddXY(X,Y3,'');    {результирующая волна}
                end;            {*****Конец блока*****}
end;
```

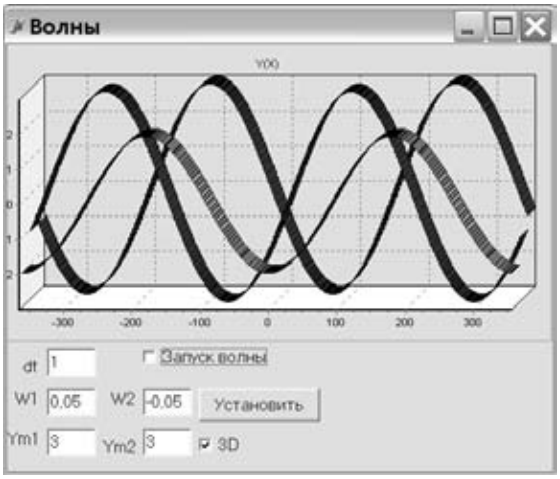
Запустите программу. Убедитесь, что теперь на грфике отображается результирующая волна №3, равная сумме двух других. Мы проанализируем поведение этой третьей, результирующей волны чуть ниже. Закройте программу и **вернитесь в режим программирования.**

Задание 4.2.4. Запрограммируем компонент Checkbox2 следующим образом. Если на нем выбрана галочка, то график должен быть объемным, а иначе обыкновенным.

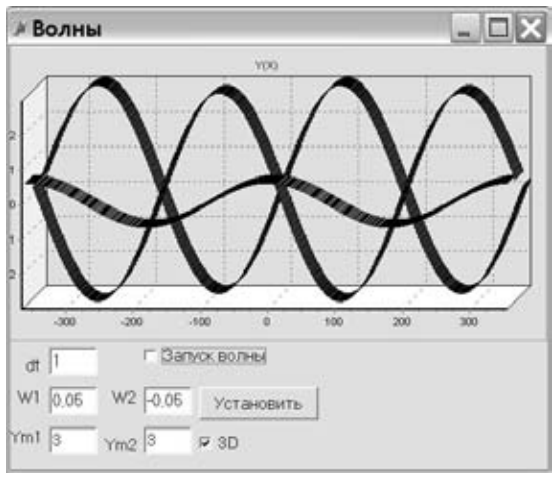
В обработчике события onClick щелчка на CheckBox2 напишите следующее:

```
if CheckBox2.Checked=true then Chart1.View3D:=true
else Chart1.View3D:=false.
```

Здесь мы используете булевское свойство View3D компонента chart1, отвечающее за объемность графика. Запустите программу. Установите обе амплитуды Ym1 и Ym2 двух складывающихся волн в одинаковое значение, например в 3, а их частоты W1 и W2 установите одинаковыми по модулю, но разными по знаку, например 0,05 и -0,05 соответственно. Установите галочку напротив «3D», чтобы график отображался в объемном виде и поставьте галочку напротив «Запуск волны». Вы получите примерно такой результат:



а) Волны 1 и 2 в некоторый момент времени



б) Волны 1 и 2 в противофазе

Рис. 4.8. Моделируемые волны в различные моменты времени.

Мы видим, что когда волны 1 и 2 в противофазе (рис. 4.8б), то есть в момент времени, когда у одной волны максимум совпадает с минимумом другой волны, значение результирующей волны обращается в ноль, то есть волна 1 и волна 2 гасят друг друга.

Самостоятельно зафиксируйте момент, когда волны находятся в резонансе, то есть максимально усиливают друг друга. **Вернитесь в режим программирования.**

Теперь зайдите у компонента Chart1 на вкладку «Chart» → «3D» (Рис. 4.9) и поставьте галочку напротив 3 Dimensios, также снимите галочку напротив Orthogonal и установите там следующие значения: Zoom=109, Rotation=345, а затем нажмите Close.

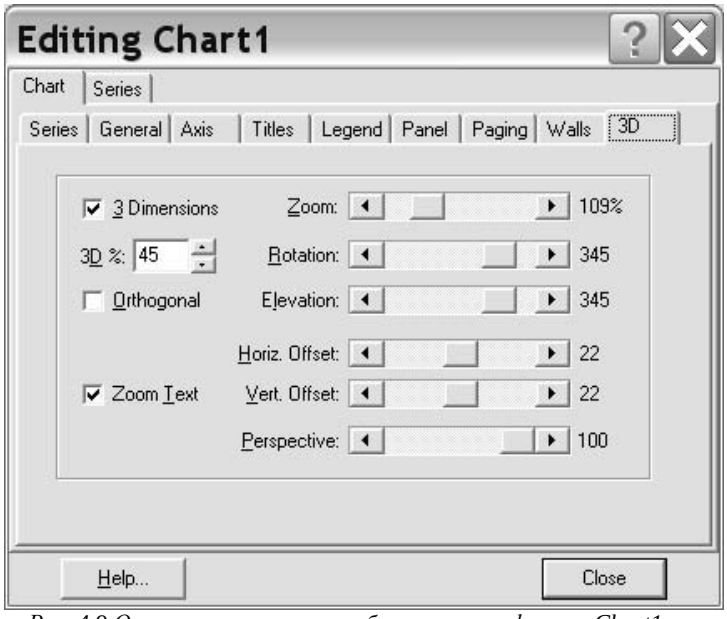


Рис. 4.9 Опции трехмерного отображения графиков в Chart1

Снова запустите программу, установив объемный вид графика. Посмотрите, что теперь график выглядит по-другому. Затем, снова используя вкладку «Chart» → «3D», попробуйте самостоятельно изменять значения Elevation, HorizOffset, Rotation, VertOffset и Perspective.

Посмотрите соответствующие изменения на графике в процессе изменения данных значений.

Задание 4.2.5. Изменим параметры отображения объемного графика, с помощью компонента TrackBar.

Настройте Object Inspector на Chart1. В его свойствах, найдите View3DOptions, щелкните на три точки справа, либо разверните, нажав на “+” справа. Измените значения Zoom, другие параметры, посмотрите, как это влияет на отображение графика. Далее добавьте 5 компонентов TrackBar. TrackBar – полезный компонент, расположенный на вкладке Win32 палитры компонентов, служит для установки конкретного целого значения, путем перемещения его указателя, при этом данное целое значение содержится в его свойстве Position.

Компонент	**Свойство**	**Значение свойства**
TrackBar1	**Position**	**1**
	min	**1**
	max	**500**

Запрограммируем так, чтобы при перемещении указателя TrackBar1 размер графика менялся от 1 до 500%. Для этого в обработчике изменения позиции указателя TrackBar1

procedure TForm1.TrackBar1Change(Sender: TObject) напишите следующую простую инструкцию:

```
chart1.View3DOptions.Zoom:=trackbar1.Position;
```

Запустите программу. Убедитесь в правильности ее работы. Закройте программу и **вернитесь в режим программирования.**

Обратите внимание, что для TrackBar1 значения свойств min=1 и max=500 выбраны исходя из минимального и максимального значения Zoom. Аналогично настройте остальные компоненты TrackBar используя View3DOptions компонента Chart1.

Компонент	**Меняет в Chart1 следующие значения:**
TrackBar2	**Elevation**
TrackBar3	**HorizOffset**
TrackBar4	**Rotation**
TrackBar5	**VertOffset**

Запустите программу. Убедитесь в правильности ее работы. Закройте программу и **вернитесь в режим программирования.**

ПРАКТИЧЕСКАЯ РАБОТА №5

5.1. Работа с файлами (на примере текстового). Создание текстового файла. Запись информации в текстовый файл

Среда Delphi предоставляет очень широкий спектр методов работы с файлами. Часть из них – это процедуры и функции самой среды Delphi, остальная часть – это некоторые API функции ОС Windows. Мы кратко рассмотрим работу только с текстовыми файлами [3], используя методы самой среды Delphi. Тип текстового файла в Delphi – TextFile.

Создайте новый проект. Поместите на форму следующие компоненты (рис. 5.1). с указанными в таблице значениями свойств.

Компонент	Свойство	Значение свойства
Label1	**Caption**	**Имя Файла**
Edit1	**Text**	**test.txt**
Button1	**Caption**	**Создать**
Button2	**Caption**	**Добавить**

Рис. 5.1 Приложение позволяющее создать текстовый файл

Задание 5.1.1 Требуется при щелчке на Button1 создать новый текстовый файл с именем ,указанным в текстовом поле Edit1, а затем добавить какие-нибудь данные в уже созданный файл.

Таким образом, создадим текстовый файл и откроем его в режиме добавления данных.
Обработчик щелчка кнопки с заголовком “Создать” должен выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var i,j:integer; f:TextFile; fname:string;
begin
fname:=edit1.Text; { задаем имя файла}
AssignFile(f,fname); { связываем файловую переменную f и имя файла fname}
rewrite(f); {Открытие файла в режиме создания нового файла или замены уже существующего}
write(f, 'Студент ',25,' Университет ',16,25); { записываем все в одну строчку }
for i:=1 to 4 do write(f, ' Строка ',i); { продолжаем записывать все в одну строчку}
for i:=1 to 3 do writeln(f, ' Строка ',i); {записываем несколько строчек подряд в столбик}
CloseFile(f); {закрываем файл}
if IoResult=0 then messageDlg('Файл удачно создан',mtInformation,[mbOK,mbCancel], 0) else
messageDlg('Произошла ошибка',mtError,[mbOK], 0);
end;
```

В данном случае при вызове функции rewrite(f) создается новый файл, ассоциированный с файловой переменной f и при этом в сам файл можно записывать информацию. Если файл с данным именем уже существовал, то его содержимое полностью удаляется.

Как только в новый созданный файл f была записана необходимая информация, его нужно обязательно закрыть инструкцией CloseFile(f). Логика записи информации в файл операторами write и writeln указана в комментариях к коду. Запустите программу, щелкните на кнопке “Создать”. Зайдите в папку вашего проекта найдите созданный вами файл «test.txt». Откройте его и изучите расположение данных. **Вернитесь в режим программирования.**

Также в программе присутствует оператор **AssignFile(f,fname),** связывающий файловую переменную f и имя файла fname. Здесь f является файловой переменной, а Fname – именем файла, поэтому запись Fname:='fail.txt' устанавливает связь между переменной f и файлом «fail.txt» в папке вашего проекта. Если мы не указываем место, где находится файл, как в последнем случае, то связь по умолчанию устанавливается на расположение файла в папке проекта, однако можно непосредственно указать место, где расположен или должен располагаться файл, например Fname:='C:\fail.txt'.

В общем виде инструкции ввода и вывода в текстовый файл выглядят так:
Write(ФайловаяПеременная, СписокВывода) и
Writeln(ФайловаяПеременная, СписокВывода).

Разница между ними состоит в том, что инструкция Writeln после записи всех значений в строку, указанных в списке вывода, затем записывает в файл символ – "новая строка", т.е. курсор «как бы» переходит на новую строку, в отличие от Write.

На всякий случай мы вызываем функцию IoResult (Input-Output Result- результат ввода-вывода), которая возвращает 0, если операция ввода-вывода завершилась успешно, в противном случае – не ноль.

Полезно запомнить удобную функцию MessageDlg(), она отображает на экране некоторое сообщение, при этом возвращает значение нажатой кнопки в виде mrOk или mrCancel. Например, допустима конструкция: if MessageDlg('Файл удачно создан',mtInformation,[mbOK,mbCancel], 0) = mrOk then…

После работы с файлом его нужно обязательно закрыть оператором CloseFile(f).

Еще раз отметим, что **процедура Rewrite(f) открывает файл в режиме создания нового файла, удаляя все данные из файла, если таковой был с данным именем и записывает новые данные**, убедитесь в этом, создав еще одну кнопку Button3 с надписью Создать2 (рис. 5.2).

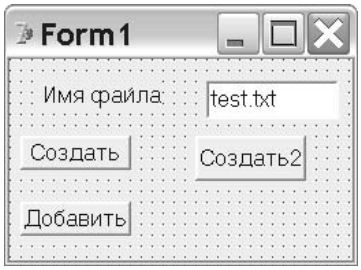


Рис. 5.2 Добавляем еще одну кнопку «Создать2»

В обработчике щелчка на Button3 напишите следующее:

```
var fname:string; f:TextFile;
begin
fname:=edit1.Text;
AssignFile(f,fname);
rewrite(f);
write(f,'Computer','Processor Core 2 Duo');
CloseFile(f);
end.
```

Запустите программу, имя файла должно быть тем же, щелкните на кнопке «Создать2».

Зайдите в папку вашего проекта найдите заново созданный вами файл «test.txt», откройте его и увидите, что он полностью переписан. Там будет лишь строчка "Computer','Processor Core 2 Duo". То есть те данные, которые были в файле, теперь полностью удалены, так как файл открылся в режиме создания нового файла с использованием функции rewrite(f), и затем переписаны некоторой новой информацией, в частности, строчкой "Computer','Processor Core 2 Duo".

Вернитесь в Режим Программирования.

Задание 5.1.2 Теперь откроем уже готовый, созданный файл в режиме добавления данных.

В обработчике кнопки с заголовком "Добавить" напишите:

```
var i:byte; fname:string; f:TextFile;
begin
fname:=edit1.Text;
```

```
AssignFile(f,fname);
append(f);{открываем файл f в режиме добавления}
writeln(f,'Dobavili',' USSR');
for i:=1 to 10 do
writeln(f,' Dobavili ', i);

CloseFile(f);
end.
```

Запустите программу, щелкните на кнопке «Добавить». Зайдите в папку вашего проекта найдите файл test.txt и откройте его. Вы обнаружите, что файл не переписан заново, а к старым данным вашего файла прибавились новые, добавленные данные. В данном случае мы воспользовались процедурой **Append(f) – открывающей файл в режиме добавления данных.**

Вернитесь в Режим программирования.

Задание 5.1.3 Создадим в виде текстового файла базу данных “Успеваемость”, содержащую информацию об успеваемости студентов.

Создайте новый проект. Создайте глобальную файловую переменную f. Сбросьте на форму компоненты Label, Edit1 и одну кнопку Button1 следующим образом (рис. 5.3)

Рис. 5.3 Окно приложения базы данных «Успеваемость»

В обработчике события FormActivate для формы напишите

```
AssignFile(f,'uspev.txt');
{$I-}
Append(f);
if IOResult=0 then Button1.Enabled:=True else
begin
Rewrite(f);
if IOResult=0 then Button1.Enabled:=True else
ShowMessage('Error Create File');

end;
```

Событие FormActivate формы Form1 генерируется, как только активизируется форма приложения, а активизируется она после создания формы или сразу после выделения формы мышью. Директива {$I-} указывает компилятору не обрабатывать автоматически операции ввода-вывода, мы будем сами их обрабатывать, используя переменную IOResult.

Таким образом, если произошла ошибка при выполнении процедуры Append(f), то программа будет далее работать по алгоритму и специфических сообщений об ошибках не поступит.

Поэтому если файл uspev.txt существует и вы попытались его открыть в режиме добавления Append(f) , и никакой ошибки не произошло, то делаем доступной кнопку Button1.Enabled:=True, а если файла не существует, то, естественно, произойдет ошибка открытия (в таком случае значение IOResult<>0), но мы об этом не узнаем, ибо мы отключили автоматическую обработку ошибок {$I-}. Таким образом, в случае несуществования файла мы просто его заново создаем Rewrite(f).

Теперь все прописано и осталось добавить в обработчик кнопки следующий текст

```
if (length(edit1.text)<>0) or (length(edit2.text)<>0) or
```

(length(edit3.text)<>0) or (length(edit4.text)<>0) { Если длина любого поля не равна нулю, то пишем в файл}
then writeln(f,edit1.text,' ',edit2.text,' ',edit3.text,' ',edit4.text) else
showMessage('Заполните все поля');

Функция length() определяет длину строки объекта edit1.text.
Теперь при активизации события закрытия формы запрограммируем закрытие файла.

В обработчике события onClose закрытия формы просто напишите
CloseFile(f).

В данном случае инструкцией length(edit1.text)<>0 мы проверяем факт отличия длины текстового поля text компонента edit1 от нуля. То есть, если в поле text находится
какой – нибудь текст, то это значит, что длина поля не равна нулю. Вообще, length возвращает число символов, из которых состоит строка. Таким образом, если во всех строках (во всех текстовых полях text) хоть что-то имеется (не пусто), то мы начинаем записывать эти значения в файл.
Запустите программу, введите в соответствующие текстовые поля компонентов TEdit любые текстовые данные, например «Иванов», «Иван», «Физика», «Отлично».
Затем щелкните на кнопке “Добавить в файл «uspev.txt». Введите еще несколько фамилий с данными. Закройте программу, **Вернитесь в Режим программирования.**

Зайдите в папку вашего проекта найдите файл «uspev.txt» и откройте его, все данные будут там. Изучите содержимое файла.
Заметьте, что данные будут в файле «uspev.txt» только после того, как вы закройте приложение, так как именно там мы прописали инструкцию CloseFile(f).

5.2. Чтение из файла. Открытие файла в режиме чтения

Задание 5.2.1 Откроем текстовый файл в режиме чтения и прочитаем оттуда имеющуюся там информацию.
Открытие текстового файла f в режиме чтения осуществляется вызовом процедуры Reset(f).
В общем виде инструкции чтения из файла выглядят так:

Read(ФайловаяПеременная, СписокПараметров) или
Readln(ФайловаяПеременная, СписокПараметров)

В текстовом файле находятся лишь символы. Функция Readln работает следующим образом. Из файла считывается последовательность символов до первого пробела или разделительного символа, если эта последовательность символов изображает число, то данная последовательность преобразуется в число и это число присваивается первому из перечисленных через запятую параметров «СпискаПараметров», если же последовательность состоит из символов, не являющихся цифрами, то она будет преобразовано в строку и также присвоено соответствующему параметру из «СпискаПараметров». Затем для следующей последовательности все снова повторяется и т.д.
Как только всем параметрам «СпискаПараметров» будут присвоены значения, происходит переход на следующую строку внутри текстового файла.
Функция Readln отличается от Read только тем, что после того, как параметры «СпискаПараметров» заполнены, происходит переход на новую строку.

Создайте новый проект, а также добавьте на форму следующие кнопки:
Button1 с заголовком «Читать из файла1», Button2 с заголовком «Читать из файла2» и Button3 с заголовком «Читать из файла3» и компонент Memo1. Также добавьте на форму кнопку BitBtn1 – она самая первая в палитре Additional, это функциональная кнопка. Измените значение ее свойства Kind на bkClose и теперь при нажатии на нее приложение будет закрываться.

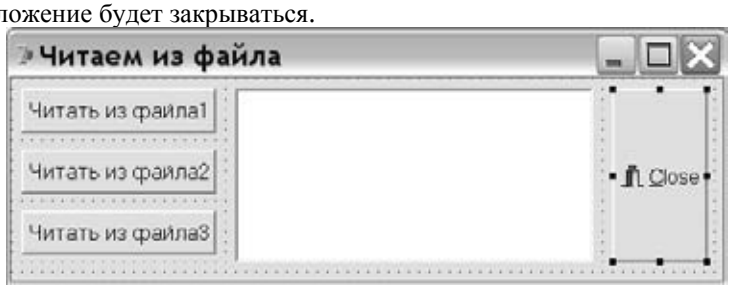


Рис. 5.4 Окно приложения «Читаем из файла»

В папке проекта вручную создайте файл «data.txt» пусть он содержит следующие строки:

12 3 45 Комдив
67 8 90 10
91 Чапаев 25

Обработчик щелчка на кнопке Button1 должен выглядеть так:

```
procedure TForm1.Button1Click(Sender: TObject);
var f:TextFile; a,b,c,d:integer;
begin
Memo1.Clear;
AssignFile(f,'data.txt');
reset(f); {открываем файл f для чтения}
read(f,a); {читаем значение в переменную a, остаемся в текущей строке}
readln(f,b,c); {читаем значения в переменные b и c, переходим на следующую строку}
read(f,d); {читаем значение в переменную d, остаемся в текущей строке}
memo1.Lines.Add('a='+IntToStr(a)+#9+'b='+IntToStr(b)+
#9+'c='+IntToStr(c)+#9+'d='+IntToStr(d)); {выводим значения a,b,c,d в Memo1}
closefile(f); {закрываем файл}
end;
```

Запустите программу. Щелкните на кнопке Button1. Вы получите следующий результат.

Рис. 5.5 Результат щелчка на кнопке «Читать из файла1»

Проанализируем, почему переменные a, b, c и d принимают такие значения.

После того как мы открыли файл для чтения процедурой reset, вначале мы применили инструкцию read(f,a). Поэтому в переменную a будет записано первое значение первой строки, то есть 12. Вернее сказать, сначала считывается последовательность символов 1 и 2, а так как между ними нет разрывов в виде пробелов, то эти символы представляют собой число 12, которое присваивается первому параметру из списка вывода процедуры read. В данном случае параметр в «СпискеПараметров» всего один – a.

При этом можно считать, что внутри файла находится курсор в определенной позиции, который может перемещается по всему файлу. Таким образом, после применения первой инструкции read в переменную a записывается число 12, а курсор после этого продолжает находиться все в той же строке файла. Далее мы применяем инструкцию readln(f,b,c). Поэтому в переменные b и c присвоятся значения 3 и 45, и курсор перейдет на начало следующей строки.

И затем, когда мы снова применяем read(f,d), то в d записывается значение текущей позиции курсора, то есть 67. **Вернитесь в режим программирования.**

Теперь полностью скопируйте код обработчика кнопки Button1 в обработчик щелчка кнопки Button2 и замените там все процедуры: read на readln, a readln на read.

Запустите программу. Вы получите следующий результат:

Рис. 5.5 Результат щелчка на кнопке «Читать из файла2»

Попробуйте сами объяснить, почему переменные a, b, c и d принимают такие значения.

Вернитесь в режим программирования.

Далее создайте обработчик щелчка кнопки Button3, который должен выглядеть следующим образом:

```
procedure TForm1.Button3Click(Sender: TObject);
var f:TextFile; a,b,c,d:integer; s:string;
begin
```

```
AssignFile(f,'data.txt');
Reset(f);
readln(f,a,b,c,s); {Правильно! В переменную строкового типа s можно записать строку «Комдив» }
//readln(f,a,b,c,d); { Неправильно! В переменную целого типа d нельзя записать строку «Комдив» }
memo1.Lines.Add('a='+IntToStr(a)+#9+'b='+IntToStr(b)+
#9+'c='+IntToStr(c)+#9+'s='+s);
closefile(f);
end;
```

Запустите программу. Вы получите следующий результат:

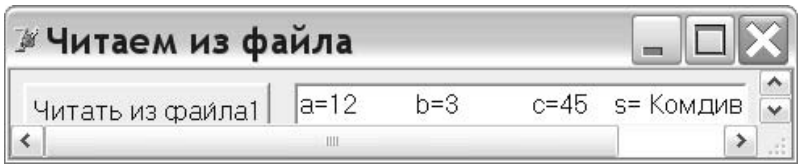


Рис. 5.6 *Результат щелчка на кнопке «Читать из файла3»*

В данном случае первая инструкция readln(f,a,b,c,s) работает с первой строкой файла «data.txt» следующим образом. Эта инструкция считывает из файла первые 3 целых числа, которые присваиваются соответствующим переменным целого типа: a, b и c, а затем встречает последовательность символов “Комдив”, в которой нет цифр, и, преобразовав ее в строку, присваивает эту строку значению параметра строкового типа s. И все работает хорошо. Но как только мы попробуем применить инструкцию readln(f,a,b,c,d), программа компилироваться не будет, так как в переменную целого типа d мы пытаемся записать значение строкового типа.

Проверьте это, закомментировав первую инструкцию readln программного кода и убрав комментарии со следующей инструкции readln.

Вернитесь в режим программирования.

Задание 5.2.2 Будем перемещаться по текстовому файлу и считывать от туда нужную информацию пока не обнаружим его конец.

Часто очень важно знать, когда же кончается файл, где находится его конец? Для того, чтобы это определить, существует специальная булевская функция EOF – расшифровывается как End Of File. Данная функция возвращает «True», если достигнут конец файла, и «False» в противном случае.

Создайте новый проект. Добавьте на форму компоненты Button1 и Memo1. В папке вашего проекта самостоятельно создайте файл rashod.txt, например пусть он содержит следующие 3 строки:

```
2500 Часы
900 Фонарик
100 Ручка
```

Попробуем открыть файл, подсчитать там общую сумму расходов 2500+900+100 и вывести значение в Memo1. Окно приложения должно выглядеть так (рис. 5.7).

Рис. 5.7 Окно приложения «Общая сумма расходов»

В обработчике кнопки Button1 напишите следующее:

```
var f:TextFile; i,sum,Obsh_sum:integer;
begin
Obsh_sum:=0; {общая сумма}
AssignFile(f,'rashod.txt');
Reset(f);

while NOT Eof(f) do { пока нет конца файла делать …}
        begin
```

```
readln(f,sum); {считываем значение конкретной суммы в переменную sum}
Obsh_sum:=Obsh_sum+sum; {наращиваем значение общей суммы}
        end;
memo1.Lines.Add('Общая сумма расходов ');
memo1.Lines.Add(IntToStr(Obsh_sum)); {выводим в Memo1 общую сумму}

closefile(f);
end;
```

Запустите программу. Вы получите следующий результат.

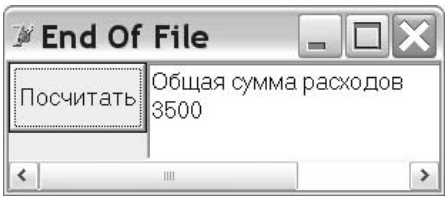


Рис. 5.8 Результат работы приложения «Общая сумма расходов» после щелчка на кнопке.

В данной программе мы применяем цикл – while NOT Eof(f) do, который будет продолжаться до тех пор, пока не встретится конец файла f. Пока нет конца файла, инструкцией readln(f,sum) мы просто считываем очередное целое значение в переменную sum, которую потом прибавляем к переменной Obsh_sum. То есть в цикле мы наращиваем значение в переменной Obsh_sum. Здесь, внутри цикла, инструкцией readln(f,sum) мы пробегаем только по всем первым значениям, записанным в начале очередной строки файла. Когда встречается конец файла, цикл заканчивается и в Memo1 выводится посчитанное значение общей суммы расходов.

Вернитесь в режим программирования.

Задание 5.2.3 Самостоятельное задание. Требуется прочитать данные из файла и добавить их на график.

Указание:

Кнопка Button1 должна создавать файл «funk.txt», записывать в него значения функции $y=x^3$ (интервал x от -10 до +10 с шагом 1) и закрывать файл (рис. 5.9).

Кнопка Button2 должна открывать файл для чтения, считывать из этого файла данные и отображать их на графике компонента Chart1.

Рис. 5.9 Окно приложения «Считывание данных из файла и добавление на график» в режиме разработки в Delphi7

ПРАКТИЧЕСКАЯ РАБОТА № 6

6.1. Краткий обзор некоторых стандартных компонентов Delphi

Задание 6.1.1. Исследуем компоненты GroupBox и RadioGroup

Рассмотрим компонент GroupBox – стандартный элемент управления Windows для объединения кнопок с зависимой фиксацией, флажков опций и других элементов управления, которые являются неотъемлемой частью большинства windows – приложений.

Создайте новый проект.

Перенесите компоненты GroupBox1, GroupBox2, Button1 на форму. Затем перенесите по 3 компонента RadioButton в GroupBox1 и в GroupBox2 (рис. 6.1)

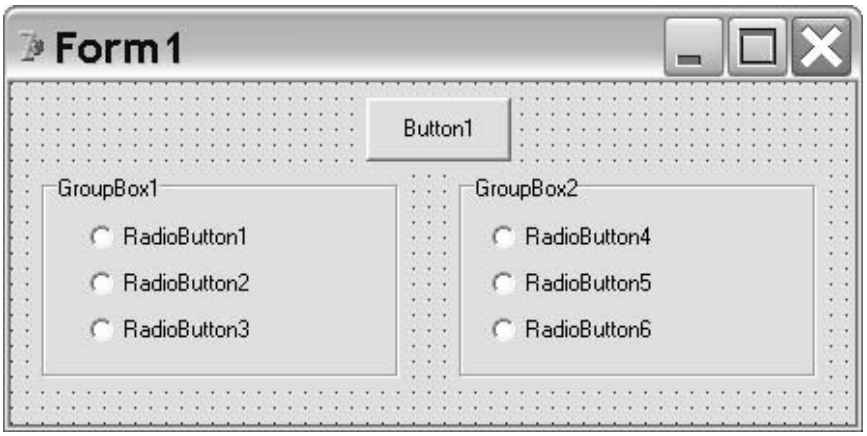


Рис. 6.1 Компоненты TGroupBox могут содержать в себе объекты TRadioButton

RadioButton – это кнопка с зависимой фиксацией [2,4], в любом наборе из таких кнопок для выбора доступна только 1 кнопка, каждый компонент GroupBox как раз объединяет один конкретный набор таких кнопок. Выбор конкретной кнопки указывается ее свойством Checked, точно так же, как и у компонента CheckBox.

Поэтому запрограммируем, обработчик щелчка на Button1 так, чтобы при щелчке выдавалось сообщение, какая кнопка RadioButton и в каком компоненте GroupBox она выбрана.

В обработчике щелчка на Button1 напишите следующее:

```
if RadioButton1.Checked
then MessageDlg('выбрана RadioButton1 в GroupBox1',mtInformation,[mbOk],0);
if RadioButton2.Checked
then MessageDlg('выбрана RadioButton2 в GroupBox1',mtInformation,[mbOk],0);
if RadioButton3.Checked
then MessageDlg(' выбрана RadioButton3 в GroupBox1',mtInformation,[mbOk],0);

if RadioButton4.Checked
then MessageDlg(' выбрана RadioButton4 в GroupBox2',mtInformation,[mbOk],0);
if RadioButton5.Checked
then MessageDlg(' выбрана RadioButton5 в GroupBox2',mtInformation,[mbOk],0);
if RadioButton6.Checked
then MessageDlg(' выбрана RadioButton6 в GroupBox2',mtInformation,[mbOk],0);
```

Далее добавьте на форму кнопку BitBtn1 из палитры Additional, это функциональная кнопка. Измените значение ее свойства Kind на bkClose и теперь при щелчке на ней приложение будет закрываться.

Запустите программу. Выберите произвольно некоторые кнопки в наборах и щелкните на Button1. Убедитесь в правильности работы программы. **Вернитесь в режим программирования.**

Задание 6.1.2. Исследуем компонент RadioGroup1

Есть еще более удобный способ объединения кнопок с зависимой фиксацией, который реализует компонент RadioGroup1 аналог компонента GroupBox. Сбросьте на форму компонент RadioGroup1. Выберите его свойство Items и чуть справа щелкните на на надписи TSrings (так же, как и для компонента Memo), наберите там вручную следующее:

Рис 6.2. Добавление имен кнопок TRadioButton в TRadioGroup

И нажмите OK. Получите уже сформировавшийся, готовый к работе набор кнопок RadioButton.

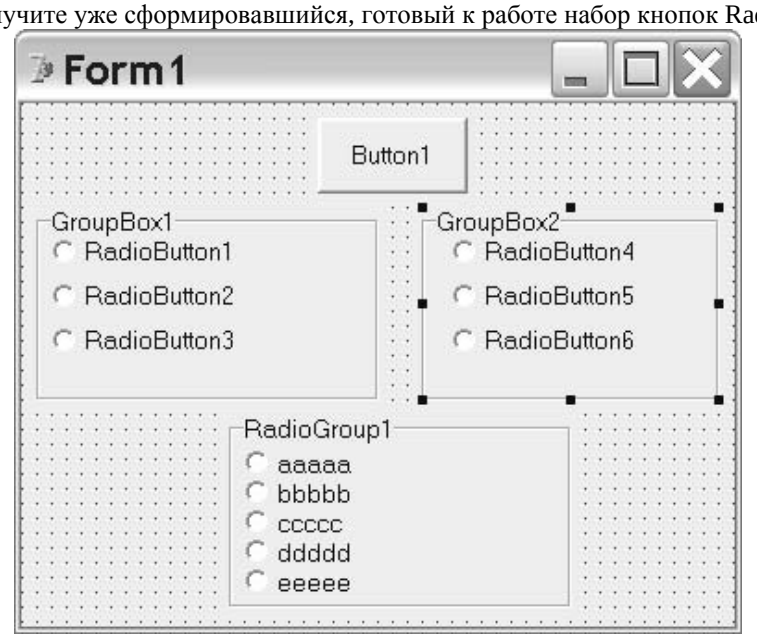


Рис. 6.3. Сформированные кнопки TRadioButton в TRadioGroup

Для RadioGroup1 измените значение свойства Columns – количество столбцов на 2 или 3. Чтобы добраться до конкретной кнопки RadioButton, нужно просматривать массив Items (свойство, представляющее из себя набор строк типа TStrings) компонента RadioGroup1.

Массив Items начинается с нуля, поэтому запись Items[0] определяет первую кнопку RadioButton внутри RadioGroup1, Items[1] определяет вторую кнопку RadioButton и т.д.

Значение свойства Count массива Items равно числу строк в свойстве Strings, и, следовательно, числу объектов-кнопок RadioButton, принадлежащих RadioGroup1.

Значение ItemIndex определяет индекс выбранной кнопки. Если ItemIndex=-1 не выбрана ни одна кнопка, если ItemIndex=0 выбрана первая кнопка, если ItemIndex= 1 вторая и т.д.

Сбросьте кнопку Button2 на форму и запрограммируете щелчок на ней так, чтобы при выборе кнопки с фиксацией выводилась информация: во-первых, присутствует ли в RadioGroup1 кнопка с надписью ‘aaaaa’, и, во- вторых, кнопка с каким индексом выбрана в данный момент.

Для этого в обработчике Button2 напишите следующее:

```
var i:integer;
begin
for I:=0 to RadioGroup1.Items.Count-1 do
if RadioGroup1.Items[i]='aaaaa' then
MessageDlg('Knopka aaaaa присутствует',mtinformation,[mbOk],0);
MessageDlg(RadioGroup1.Items[RadioGroup1.ItemIndex],mtinformation,[mbOk],0);
end;
```

Запустите программу. Выберите произвольно кнопку в RadioGroup1 и щелкните на Button2. Программа будет работать так, как и было запланировано. **Вернитесь в режим программирования.**

Задание 6.1.3 Изучим полезный оператор «with … do».

Далее в обработчике Button2 закомментируйте набранный ранее текст, чтобы его не воспринимал компилятор, или просто удалите его и измените обработчик следующим образом:

```
var i:integer;
begin
{ for i:=0 to RadioGroup1.Items.Count-1 do
if RadioGroup1.Items[i]='aaaaa' then
MessageDlg('Knopka aaaaa присутствует',mtinformation,[mbOk],0);
MessageDlg(RadioGroup1.Items[RadioGroup1.ItemIndex],mtinformation,[mbOk],0); }
with RadioGroup1 do
for i:=0 to Items.Count-1 do
if Items[i]='aaaaa' then MessageDlg('Knopka aaaaa присутствует',mtinformation,[mbOk],0);
with RadioGroup1 do
MessageDlg(Items[ItemIndex],mtinformation,[mbOk],0);
end;
```

Запустите программу. Выберите произвольно кнопку в RadioGroup1 и щелкните на Button2. Программа будет работать точно также как и в предыдущем случае. Мы просто немного изменили код. **Вернитесь в режим программирования.**

Данный оператор «with … do» используется для того, чтобы обращаться к полям, свойствам или методам объекта только один раз указав имя объекта, без использования операции доступа “.” что является довольно удобным.

В частности для компонента RadioGroup1 мы указали его имя единственный раз, а затем внутри инструкции with RadioGroup1 do мы уже просто записываем инструкцию Items.Count вместо RadioGroup1.Items.Count.

Или, для упрощения кода удобнее, например, написать вместо a:=RadioGroup1.Items.Count инструкцию with RadioGroup1, Items do a:=Count перечисляя через запятую члены класса. Подобные инструкции используются часто и делают код программы более читабельным.

Так как многие объекты (компоненты) в Delphi могут содержать другие объекты (компоненты), то для обработки этих других объектов имеется ряд полезных свойств благодаря которым можно до них добраться программным путем, 3 таких свойства приведем в следующей таблице:

Свойство	**Описание**
Components	**Массив содержит все внутренние объекты(компоненты), находящиеся в некотором объекте (компоненте). Нумеруется, начиная с нулевого номера, например, первый объект – это Components[0], второй – Components[1] и т.д.**
ComponentIndex	**Определяет номер внутреннего объекта (компонента), выбранного или активированного в данный момент времени в некотором объекте (компоненте). (0-первый, 1 – второй, 2 – третий и т.д.)**
ComponentCount	**Число всех внутренних объектов, содержащихся в некотором объекте.**

Приведенные свойства, в частности, имеются и у формы.

Задание 6.1.4 Используя данные свойства, запрограммируем щелчок на Button3 так, чтобы отменить выбор всех кнопок с зависимой фиксацией в RadioGroup1.

Здесь же сбросьте Button3 на форму.

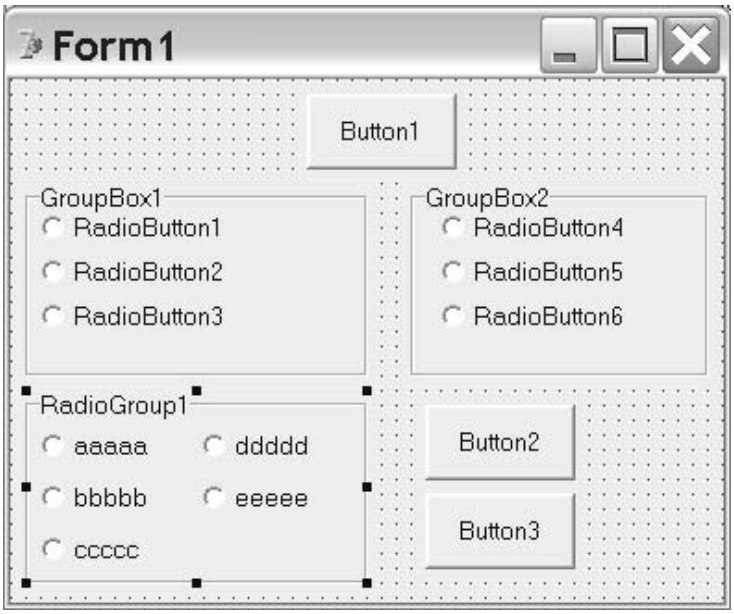


Рис. 6.4 Окно отмены выбора всех кнопок с зависимой фиксацией

А в обработчике Button3 напишите следующее:

```
var i:integer;
begin
with RadioGroup1 do
for i:=0 to ComponentCount-1 do
TRadiobutton(Components[i]).Checked:=False;
end;
```

Здесь мы пробегаем по массиву Components[i] компонента radioGroup1, начиная с нулевого индекса массива, а инструкция TRadiobutton(Components[i]).Checked:=False говорит о том, что любой объект компонента RadioGroup1, содержащийся в массиве, должен рассматриваться как объект класса TRadiobutton и к этому объекту мы применяем конкретное действие, в частности, изменяем значение его свойства Checked на False.

Запомните конструкцию: TRadiobutton(Components[i]).Checked:=False.

Подобные вещи в Delphi встречаются очень часто при программировании свойств объектов и др. В приведенном примере таится некоторая опасность, например, если объект из массива не принадлежит классу TRadiobutton? В этом случае появится сообщение об ошибке. Эта опасность исключается некоторыми приемами, мы их рассмотрим в следующем цикле практических работ.

Запустите программу. Программа должна работать так, как и было запланировано, и отменять выбор всех кнопок. **Вернитесь в режим программирования.**

6.2. Исследуем компоненты PageControl1, ListBox1, ComboBox1, а также еще ряд полезных компонентов

Задание 6.2.1. Кратко рассмотрим некоторые основные приемы работы с компонентами PageControl1, ListBox1 и ComboBox1

Если сказать кратко, то назначение компонента PageControl1 в следующем: это упрощение создания многостраничных диалоговых окон с отдельным набором элементов управления на каждой странице.

Создайте новый проект. Поместите PageControl1 из палитры win32 на форму.

Щелкните правой кнопкой мыши на PageControl1 и выберите New Page, появится вкладка TabSheet1, аналогично создайте еще 2 вкладки TabSheet2 и TabSheet3.

Теперь удалите TabSheet3 выбрав Delete Page.

Для того, чтобы PageControl1 заполнял всю клиентскую область формы, установите его соответствующее свойство, в значение, приведенное в таблице:

Компонент	Свойство	Значение свойства
PageControl1	**Align**	**alClient**

Поместите ListBox1 и ComboBox1 из палитры Standard на форму. Эти компоненты имеют такие же свойства Items, ItemIndex, что и RadioGroup1.Установите соответствующее свойство, в значение, приведенное в таблице:

Компонент	Свойство	Значение свойства
ComboBox1	**Text**	**Элементарные частицы**

У компонента ListBox1 выберите свойство Items, и чуть справа щелкните на надписи TSrings, затем наберите в открывшемся окне «String List Editor» следующее:

«Материя – философская категория для обозначения
объективной реальности, которая дана человеку в ощущениях его,
которая копируется, фотографируется, отображается
нашими ощущениями, существуя независимо от них
В. И. Л е н и н (том 18, стр. 131)»

Все должно выглядеть примерно так же как на рисунке:

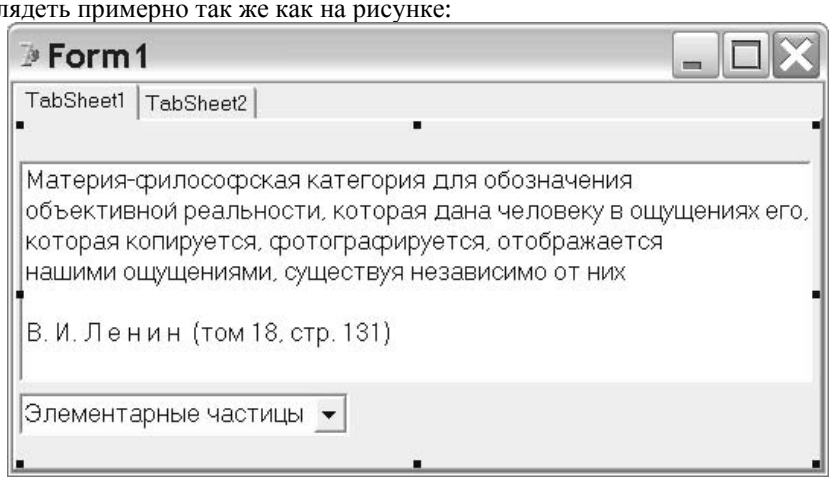


Рис. 6.5 Компонент TPageControl – это набор окон-компонентов TTabSheet, на которых можно размещать компоненты обычным образом

Аналогично у компонента ComboBox1 выберите свойство Items и, чуть справа щелкнув на надписи TSrings, наберите там следующее:

Молекула
Атом
Протон
Нейтрон
Электрон
Пи-мезон
Альфа-частица
Гиперон
мюон
тахион
позитрон
бозон

Запрограммируем так, чтобы при выборе строки в ListBox1 она отображалась в заголовке TabSheet1, а при выборе из выпадающего списка ComboBox1 она отображалась в заголовке TabSheet2.

Для этого в обработчике щелчка (click) на ListBox1 напишите следующее:
TabSheet1.Caption:=ListBox1.Items[ListBox1.ItemIndex];

А в обработчике щелчка(click) на ComboBox1 напишите следующее:

```
TabSheet2.Caption:=ComboBox1.Items[ComboBox1.ItemIndex];
//TabSheet2.Caption:=ComboBox1.Text; //альтернативный вариант
```

Запустите программу. Выбирайте произвольно значения в ListBox1 и ComboBox1. . Программа должна работать так, как и было запланировано. **Вернитесь в режим программирования.**

Задание 6.2.2 Изучим еще несколько полезных компонентов
Поместите на TabSheet2 следующие компоненты:

Компонент	**Палитра**
StringGrid1	**Additional**
CheckListBox1	**Additional**
ColorBox1	**Additional**
DateTimePicker1	**Win32**
MonthCalendar1	**Win32**
6 компонентов Animate	**Win32**
UpDown1	**Win32**
OleContainer1	**System**

А также еще кнопку BitBtn1, изменив значение ее свойства Kind на bkClose, и компонент CheckBox1 напротив DateTimePicker1.

Компонент	**Свойство**	**Значение свойства**
CheckBox1	**Caption**	**Изменять DateTimePicker**

Разместите все компоненты так, как показано на рисунке 6.6.

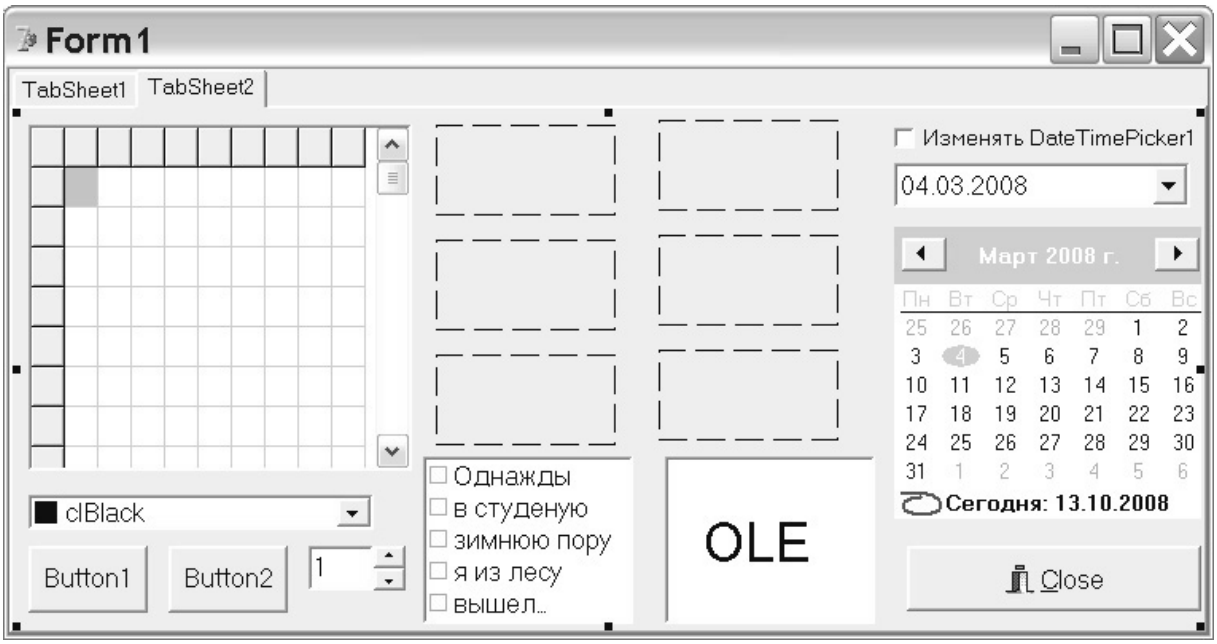


Рис. 6.6 Набор некоторых полезных компонентов

Задание 6.2.3 Изменим вид отображения компонента DateTimePicker1, при щелчке на CheckBox1.
В обработчике щелчка на CheckBox1 напишите:

```
if CheckBox1.Checked then DateTimePicker1.Kind:=dtkTime else
DateTimePicker1.Kind:=dtkDate;
```

Запустите программу. Щелкните на CheckBox1, в результате чего вид отображения компонента DateTimePicker1 должен измениться. **Вернитесь в режим программирования.**

Установите следующие значения:

Компонент	Свойство	Значение свойства	Описание
StringGrid1	DefaultColWidth	20	Ширина столбцов таблицы
	ColCount	10	Число столбцов таблицы
	RowCount	10	Число строк таблицы
	Cells[i,j]		Соответствующий таблице двумерный строковый массив
	FixedCols		Число зафиксированных слева столбцов таблицы
	FixedRows		Число зафиксированных сверху строк таблицы

Задание 6.2.4 Запрограммируем обработчик щелчка на Button1 так, чтобы он заполнял StringGrid1 как матрицу, заполненную случайными числами.

Для этого напишите в обработчике следующее:

```
var i,j:integer;
begin
for i:=1 to 7 do
for j:=1 to 7 do
StringGrid1.Cells[i,j]:=IntToStr(random(11));
end;
```

Запустите программу. Щелкните на Button1, и двумерная таблица, которая представляет собой компонент StringGrid1, заполнится псевдослучайными числами. **Вернитесь в режим программирования.**

Задание 6.2.5 Исследуем функциональность компонента UpDown1

Здесь же скиньте на форму CheckListBox со следующими значениями:

Компонент	*Свойство*	*Значение свойства*	Описание
CheckListBox1	**Items**	**Однажды в студеную зимнюю пору я из лесу вышел…**	**Массив содержит текстовые записи(строки), они нумеруется с нуля**

Щелкните на свойстве Associate компонента UpDown1 и из раскрывшегося списка выберите компонент Edit1. Вот и все. Теперь UpDown1 визуально прицепился к Edit1 и проассоциирован с ним, настроившись на него.

Компонент	Свойство	Значение свойства	Описание
UpDown1	position	1	позиция
	Min	0	минимум position
	Max	4	максимум position
	Increment	1	приращение величины

Запустите программу. Щелкайте на стрелках «вверх» или «вниз» компонента UpDown1, и значение в Edit1 будет соответственно меняться, но в пределах, указанных для UpDown1, т.е. Min=0 и Max=2 свойства position. **Вернитесь в режим программирования.**

Задание 6.2.6 Пусть при щелчке на UpDown1, в компоненте CheckListBox1 напротив текстовых значений (строк) ставится галочка, а во всех остальных ее не будет.

На самом деле при щелчке на UpDown1 значение будет изменяться в Edit1, поэтому фактически будем обрабатывать значение Edit1.text.

Напишите в обработчике события изменения значения в Edit1 (данное событие называется onChange) следующее:

```
var i:integer;
```

```
begin
With CheckListBox1 do
for i:=0 to Items.Count-1 do Checked[i]:=false; { устанавливаем все строки в
невыделенное состояние}
CheckListBox1.Checked[StrToInt(Edit1.Text)]:=true; {ставим галочку напротив}
CheckListBox1.Selected[StrToInt(Edit1.Text)]:=true; {выделяем строку}
end;
```

Запустите программу. Программа должна работать так, как и было запланировано. **Вернитесь в режим программирования.**

Задание 6.2.7 Пусть компонент ColorBox1 меняет цвет компонента-таблицы StringGrid1.
Напишите в обработчике ColorBox1Change следующее:

```
StringGrid1.Color:=ColorBox1.Selected;
```

Здесь свойство Selected определяет текущий выбранный цвет в ColorBox1.
Запустите программу. Выбирайте цвет в ColorBox1. Убедитесь в правильности работы программы. **Вернитесь в режим программирования.**

Задание 6.2.8 Рассмотрим компонент, генерирующий стандартную анимацию при копировании файлов, поиске файлов, очистки корзины и т.д.

Переместите на форму 6 компонентов Animate из палитры Win32. Поместите их рядом. Установите следующие значения:

Компонент	Свойство	Значение свойства
Animate1	**CommonAvi**	**aviCopyFile**
Animate2	**CommonAvi**	**aviDeleteFile**
Animate3	**CommonAvi**	**aviEmptyRecycle**
Animate4	**CommonAvi**	**aviFindComputer**
Animate5	**CommonAvi**	**aviFindFile**
Animate6	**CommonAvi**	**aviFindFolder**

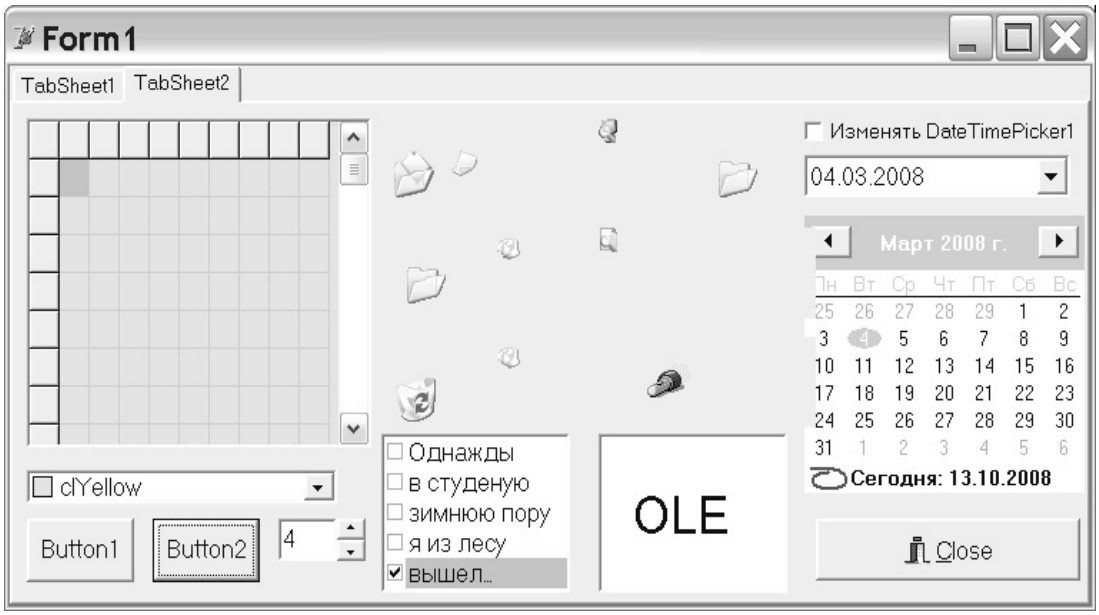


Рис. 6.7 Выше описанное приложение в режиме работы

А в обработчике щелчка кнопки Button2 напишите следующее:

```
Animate1.Play(1,1000,10); Animate2.Play(1,1000,10); Animate3.Play(1,1000,10);
Animate4.Play(1,1000,10); Animate5.Play(1,1000,10); Animate6.Play(1,1000,10);
```

Запустите программу. Щелкните на Button2. Должны выполняться стандартные анимационные эффекты. Убедитесь в правильности работы программы. **Вернитесь в режим программирования.**

Задание 6.2.9 Кратко рассмотрим компонент OleContainer1

Данный компонент использует OLE технологию [2,4], позволяющую связывать и внедрять объекты из одного приложения в другое.

Щелкните правой кнопкой на компоненте OleContainer1. Выберите вкладку «Insert Object», появится стандартное диалоговое окно выбора для вставки объекта, выберите, например, точечный рисунок.

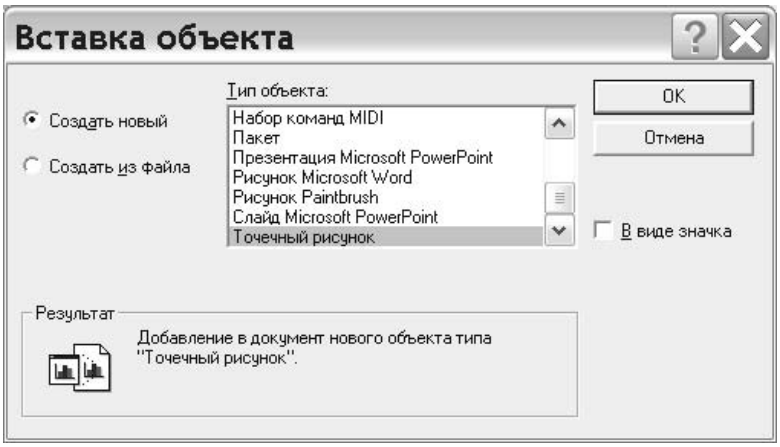


Рис. 6.8 Применение технологии OLE с использованием компонента OleContainer

Затем запустите программу. Еще раз правой кнопкой щелкните на компоненте OleContainer1 и выберите вкладку «Открыть». Откроется стандартное приложение Microsoft Paint, нарисуйте там что-нибудь и сохраните. **Вернитесь в режим программирования.**

ПРАКТИЧЕСКАЯ РАБОТА № 7

7.1. Процедуры и функции

Современную серьезную программу трудно представить без множества процедур и функций, которые преследуют принцип модульности, введенный с целью упрощения кода программы.

В Object Pascal существуют процедуры и функции, которые являются подпрограммами, позволяющими разбить на более простые блоки сложную программу. Функцию или процедуру можно вызывать из других частей программы многократно.

Процедуры или функции могут иметь или не иметь входных параметров, могут изменять или не изменять их.

Особенность **функции** в том, что она **обязательно возвращает конкретное значение определенного типа. Процедура** аналогична функции, но **у процедуры нет возвращаемого значения.**

При этом процедуры и функции, принадлежащие определенному классу, называются **методами класса.** От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут данные именно того объекта, который вызвал метод.

Процедура объявляется следующим образом:

procedure <имя процедуры> (список параметров)
<объявления локальных переменных, типов, констант>
Begin
<операторы тела функции>
End;

Функция объявляется следующим образом:

function <имя процедуры>(список параметров): <тип возвращаемого значения> ;
<объявления локальных переменных, типов, констант>
Begin
<операторы тела функции>
end.

Задание 7.1.1 Опишем процедуру вычисления корней квадратного уравнения.

Создайте новый проект. Перенесите на форму Button1. Будем создавать процедуру и последующие несколько процедур как методы класса TForm1.

В окне редактора кода в секции Public класса TForm1 объявите **прототип вашей процедуры** KvadrKoren(a,b,c:real; var x1,x2:real), который должен выглядеть так:

```
type
 TForm1 = class(TForm)
  Button1: TButton;
 private
  { Private declarations }
 public
procedure KvadrKoren(a,b,c:real; var x1,x2:real); // это прототип

 { Public declarations }
 end;
```

Теперь обязательно поставьте курсор на данную строчку описания процедуры в любое место записи прототипа [2,3,4]:

```
 procedure KvadrKoren(a,b,c:real; var x1,x2:real);
```

И нажмите Ctrl+Shift+C [2]. Перед вами появится сформированная, пока еще пустая заготовка описания вашей процедуры:

```
procedure TForm1.KvadrKoren(a, b, c: real; var x1, x2: real);
begin

end;
```

Заметьте, что прототип процедуры указывается в разделе interface, а заготовка процедуры в разделе реализации, который предваряет ключевое слово implementation.

В данном случае процедура KvadrKoren принадлежит классу TForm1 и следовательно является методом данного класса.

Опишем конкретно нашу процедуру. Измените заготовку процедуры следующим образом:

```
procedure TForm1.KvadrKoren(a, b, c: real; var x1, x2: real);
var D:real;
begin
D:=sqr(b)-4*a*c;
if D<0 then begin ShowMessage('Корней нет');exit; end;
x1:=(-b+sqrt(D))/(2*a); x2:=(-b-sqrt(D))/(2*a);
a:=10000; b:=10000; c:=10000; {попытка изменить значения переданных трех первых параметров Эта строка нужна в учебных целях}
end;
```

Все, процедура готова. Теперь просто будем вызывать ее из другой процедуры, в частности, из обработчика щелчка Button1 (метода класса TForm1).

Поэтому сбросьте на форму одну кнопку Button1. Обработчик щелчка на Button1 должен выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var a1,b1,c1, x1,x2:real;
begin
a1:=1; b1:=2; c1:=-3; //начальные значения a1, b1 и c1
KvadrKoren(a1, b1, c1, x1,x2); {Вызываем процедуру и передаем ей аргументы}
InputBox('Процедура квадр', 'корни', 'x1= '+FloatToStr(x1)+' x2= '+FloatToStr(x2));
// Выводим значения x1 и x2
Showmessage('a1= '+FloatToStr(a1)+' b1= '+FloatToStr(b1)+' c1= '+FloatToStr(c1));
        {Проверяем, изменились ли значения a1, b1 и c1 на 10000 как записано в процедуре}
end;
```

Обратите внимание на вызов процедуры KvadrKoren(a1, b1, c1, x1,x2) из обработчика щелчка на Button1

Здесь будет следующее соответствие аргументов и параметров, получаемых процедурой:

в качестве a будет получено значение a1;

в качестве b будет получено значение b1;

в качестве c будет получено значение c1.

При этом допускается, чтобы имена параметров, описанных в прототипе процедуры и переданных в нее, совпадали, поэтому можно не вводить переменные a1, b1 и c1, а просто в обработчике щелчка определить переменные a, b, c и передать их процедуре, что и было сделано только для x1 и x2.

Разберем описание прототипа процедуры KvadrKoren(a, b, c: real; var x1, x2: real). Здесь первые 3 параметра a, b, c имеют тип real; а вторые 2 параметра x1 и x2 также имеют тип real, но предваряются ключевым словом Var.

Дело здесь состоит в следующем: **если в параметрах процедуры (функции) отсутствует ключевое слово Var, то при вызове процедуры (функции) с параметрами ей передаются не сами параметры, а их копии, и она работает лишь с копиями этих параметров, не изменяя значений самих переданных параметров. Этот способ называется передачей параметров по значению.**

Мы намеренно написали в коде процедуры KvadrKoren инструкции

a:=10000; b:=10000; c:=10000; то есть мы хотим, чтобы она изменила переданные ей значения a1, b1 и c1 на 10000, так как считаем, что есть соответствие между a и a1, b и b1, c и c1.

А затем уже из обработчика щелчка на Button1, после вызова процедуры, проверяем, поменяла ли она значения a1, b1 и c1 на 10000 так, как в ней записано.

Результат таков: первые 3 параметра a1, b1, c1 не изменятся процедурой, как бы она “ни старалась”, а для проверки мы выводим их значения на экран, после передачи их в процедуру.

Запустите программу и щелкните по кнопке. Сначала вы получите значения корней x1 и x2 (рис. 7.1).

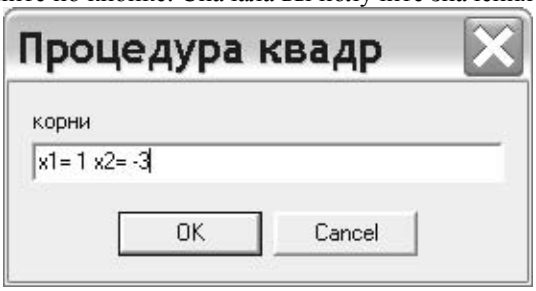


Рис. 7.1. Расчет корней квадратного уравнения

А затем, после щелчка на OK, увидите, что значения a1, b1 и c1 остались прежними (рис. 7.2), несмотря на то, что в процедуре KvadrKoren мы присваиваем a:=10000; b:=10000; c:=10000.

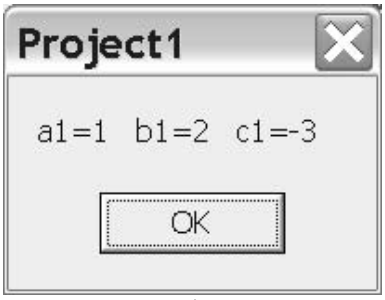


Рис. 7.2 Неизмененные процедурой значения a1, b1 и c1

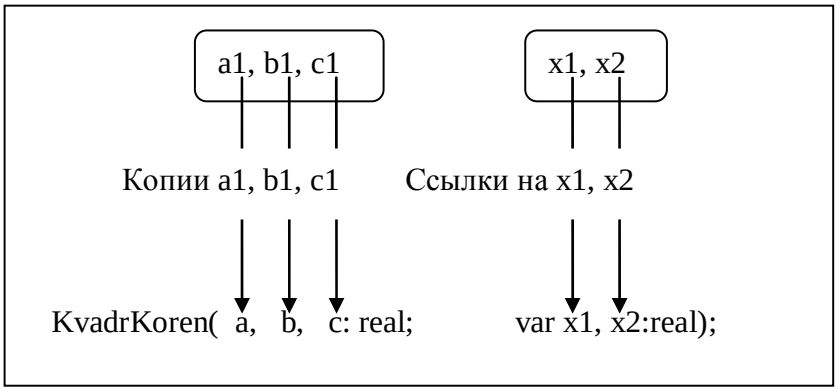
Таким образом, значения a1, b1, c1 не изменяются процедурой. **Вернитесь в режим программирования.**

При этом можно просто передавать значения, в процедуру не вводя лишних переменных, слегка изменим код для обработчика щелчка на Button1:

Вместо KvadrKoren(a1, b1, c1, x1,x2) напишите там KvadrKoren(1, 2, -3, x1,x2).

Программа будет работать точно так же. Проверьте это. **Вернитесь в режим программирования.**

Совершенно противоположное действие, **когда в процедуре перед параметрами пишется ключевое слово Var. В этом случае процедура может изменять значение переданного ей параметра. Этот способ передачи параметров называется передачей по ссылке.** Поэтому значения x1 и x2, предваренные словом var, могут, и в данном случае будут изменяться процедурой KvadrKoren. Проверим это.



Исходные аргументы a1, b1, c1 процедуре не доступны и меняться не могут (нет var) <u>передача по значению</u>. Исходные аргументы x1, x2 процедуре доступны и меняться могут (есть var) <u>передача по ссылке.</u>

Рис. 7.3 Схема передачи параметров по значению и по ссылке

Для большей убедительности измените обработчик щелчка Button1 следующим образом:

var a1,b1,c1, x1,x2:real;

```
begin
a1:=1; b1:=2; c1:=-3;
x1:=10000; x2:=10000; { Устанавливаем начальные значения X1 и X2}
InputBox('', 'начальные значения', 'x1= '+FloatToStr(x1)+' x2= '+FloatToStr(x2));
{ KvadrKoren(a1, b1, c1, x1,x2); //не может менять a1, b1, c1, может менять x1 и x2}
KvadrKoren(1, 2, -3, x1,x2); //может менять x1 и x2
InputBox('Процедура Квадр', 'корни', 'x1= '+FloatToStr(x1)+' x2= '+FloatToStr(x2));
end;
```

Запустите программу и щелкните по кнопке. Сначала вы получите начальные значения переменных x1 и x2, равные 10000 (рис. 7.4), так как в начале мы специально устанавливаем их в это значение.

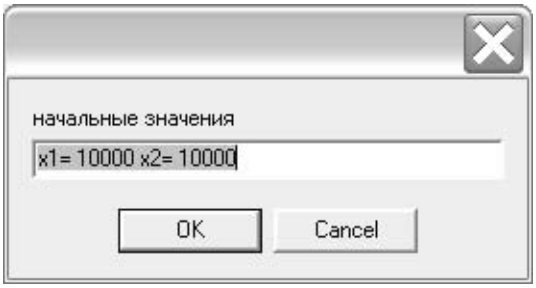


Рис. 7.4 Начальные значения переменных x1 и x2 равные 10000

Затем, после щелчка на OK, вы обнаружите, что x1 и x2 поменялись процедурой (рис. 7.5). В данном случае процедура KvadrKoren рассчитала корни и присвоила их значения к переменным x1 и x2, объявленным в обработчике щелчка Button1.

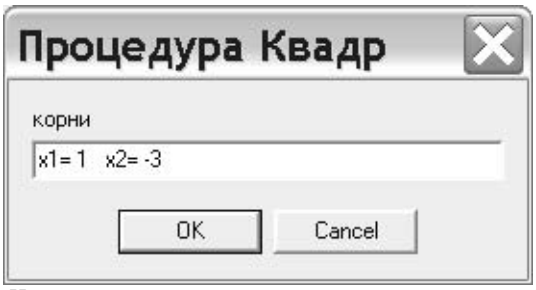


Рис. 7.5 Измененные значения переменных x1 и x2 равные 10000

Вернитесь в режим программирования.

В Delphi функция отличается от процедуры тем, что всегда в результате своей работы возвращает некоторое значение определенного типа.

Все рассуждения относительно параметров, переданных процедуре, присутствия или отсутствия ключевого слова Var также распространяются и на функцию.

Задание 7.1.2. Создадим функцию, возвращающую значения факториала 10! Будем вызывать ее из обработчика щелчка на Button2 и выведем десять ее значений в Memo1.

В этом же проекте сбросьте Button2 и Memo1 на форму (рис. 7.6).

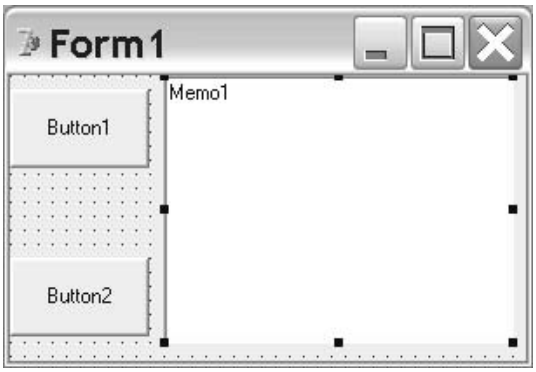


Рис. 7.6 Приложение вычисляющее факториал n!

Факториал n обозначается n! и равен n!=n·(n-1) (n-2) …·3·2·1.

В окне редактора кода в секции Public класса TForm1 объявите **прототип вашей функции**, все должно выглядеть примерно так:

```
type
 TForm1 = class(TForm)
  Button1: TButton;
  Button2: TButton;
  Memo1: TMemo;
  procedure Button1Click(Sender: TObject);

 private
  { Private declarations }
 public
  procedure KvadrKoren(a,b,c:real;var x1,x2:real);
 function Faktorial(n:word):word; {Прототип функции}
 { Public declarations }
 end;
```

Точно так же, как и для процедуры, обязательно поставьте курсор на данную строчку описания процедуры в любое место записи function Faktorial(n:word):word;

и нажмите Ctrl+Shift+C. Перед вами снова появится сформированная заготовка описания вашей функции, измените ее следующим образом:

```
function TForm1.Faktorial(n: word): word;
var i,j:word;
begin
j:=1;
for i:=1 to n do j:=j*i; //вычисляем факториал в переменную j
Result:=j; { ключевое слово Result для функции, означающее конечный результат ее действия. Хотя можно было бы написать вместо Result:=j инструкцию Faktorial:=j}
end;
```

Так как в передаваемых параметрах нет слова var, то функция является “безопасной” и менять значение переданного ей параметра не будет.

Ключевое слово Result в функции содержит возвращаемый ею результат определенного типа. Вместо этого слова можно писать имя самой функции.

В отличие от процедуры, для функции всегда указывается не только тип переданных в нее параметров, но и тип возвращаемого функцией значения. В данном случае функция будет возвращать значение типа word.

Далее в обработчике Button2 напишите следующее:

```
var i:integer; b:word;
begin
for i:=1 to 10 do
Memo1.Lines.Add('Факториал n='+IntToStr(i)+' равен '+IntToStr(Faktorial(i)));
b:=Faktorial(10); { просто присваиваем некоторой переменой типа word
значение функции Faktorial(10);}
ShowMessage(’10! = ’+IntToStr(b));
end;
```

Запустите программу и щелкните на кнопке Button2. На экране появится сообщение с результатом расчета функцией факториала 10! В**ернитесь в режим программирования.**

Нужно помнить, что в данном случае процедура TForm1.KvadrKoren(a, b, c: real; var x1, x2: real)

и функция TForm1.Faktorial(n: word): word являются методами класса TForm1.

Мы их объявили в разделе Public, это означает, что названные функции и процедуры доступны наследникам данного класса, также их можно вызывать и из другого модуля, например Unit2, подключенного к данному Unit1.

Delphi предоставляет **очень полезное окно Module Explorer (Ctrl+Shift+E) – в нем содержится основная информация о текущем программном модуле, о перечне классов, о других модулях, которые он использует, а также методах классов, процедурах и функциях.**

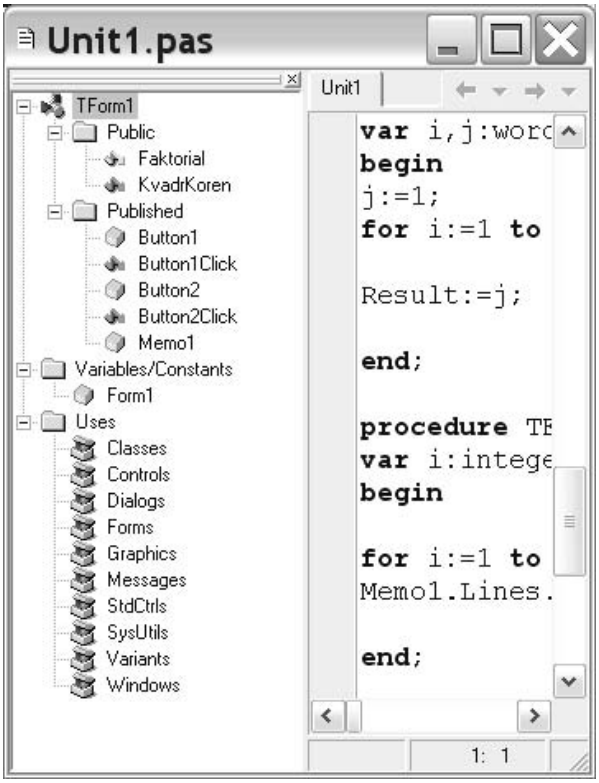


Рис. 7.7 Окно Module Explorer

Задание 7.1.3. Создадим функцию, подсчитывающую количество букв «f» во введенной пользователем строке.

Создайте новый проект. Как и в предыдущем случае, сбросьте на форму Button1, Button2 и Memo1.

Строку будем вводить при помощи оператора InputBox('Input Box', 'Prompt', 'Default string'), данный оператор является, по существу, функцией и возвращает строковое значение 'Default string' типа string.

Так же в секции Public объявите прототип функции Funkfff(s:string):word. Таким образом, функция будет получать в качестве параметра значение строковой переменной s, а возвращать количество букв «f», содержащихся в данной строке. После того, как вы объявили прототип функции, нажмите Ctrl+Shift+C и перед вами появится заготовка метода:

```
function TForm1.Funkfff(s: string): word;
begin
end;
```

Измените данную заготовку следующим образом (фактически мы описываем реализацию функции, наполняя заготовку программным кодом):

```
function TForm1.Funkfff(s: string): word;
var i,sum:word;
begin
sum:=0;
for i:=1 to length(s) do
if s[i]='f' then sum:=sum+1;
result:=sum;
end;
```

Здесь length(s) – стандартная функция, возвращающая длину строки s, то есть количество всех символов в строке.

В обработчике щелчка на Button1 напишите следующее:

```
var s:string;
begin
s:=inputBox('Ввод строки','Введите строку','');
messageDlg('Число букв f равно '+IntToStr(Funkfff(s) ),mtInformation,[mbOk],0);
end;
```

Запустите программу. Щелкните на Button1, в появившееся окно введите некоторую строчку, которая содержит несколько букв f. Затем нажмите OK, и вы получите количество букв f из введенной вами строчки. **Вернитесь в режим программирования.**

Задание 7.1.4. Здесь же создадим процедуру по имени Mas (метод Класса TForm1), получающую в качестве параметра некоторый одномерный массив, и постоянное число N, которое будет определять: сумму N элементов массива, среднее арифметическое N его элементов, а также менять четные элементы массива на -1.

Так как процедура должна менять четные элементы массива на -1, а значит должна иметь доступ к оригинальному массиву, то в процедуре перед переданным параметром-массивом должно быть слово var (передача по ссылке).

В секции Public объявите прототип и сформируйте процедуру

Mas(var a: array of integer; const N:integer; var Sum:integer; var Sred:real);

Код функции должен быть следующим:

```
procedure TForm1.Mas(var a: array of integer; const N:integer;
var Sum:integer; var Sred:real);

var i:integer;
begin
sum:=0;
for i:=0 to N-1 do sum:=sum+a[i];
sred:=sum/N;
for i:=0 to N-1 do if (a[i] mod 2)=0 then a[i]:=-1;
end;
```

Здесь в процедуре индексация массива идет с нулевого индекса, поэтому первый элемент – это a[0], второй – a[1] и т.д. Мы будем вызывать процедуру Mas из обработчика щелчка кнопки Button2, поэтому в обработчике Button2 напишите следующее:

```
const N=5;
var i:integer; a:array[1..N] of integer; Sum:integer; Sred:real;
begin
memo1.Clear;
memo1.Lines.Add('Выводим начальный массив');
for i:=1 to N do begin
a[i]:=Random(9); memo1.Lines.Add( IntToStr(a[i]) );
                    end;
mas(a,N,Sum,Sred);// Вызываем процедуру

Memo1.Lines.Add('Sum='+IntToStr(Sum)+' Sred='+FloatToStr(Sred));
memo1.Lines.Add('');
memo1.Lines.Add('Выводим измененный массив');
for i:=1 to N do memo1.Lines.Add( IntToStr(a[i])  );
end;
```

Запустите программу, убедитесь в правильности ее работы. Результат должен быть примерно следующим: (рис. 7.8); (здесь 0 считается четным числом). **Вернитесь в режим программирования.**

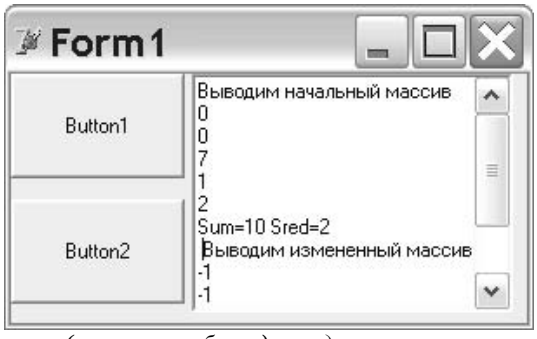


Рис. 7.8 Окно приложения (в режиме работы) определяющего сумму элементов массива, среднее арифметическое элементов и т. д.

Нужно отметить, что и процедуры и функции можно прописывать совершенно обособленно от конкретного класса, то есть они могут существовать «сами по себе», и поэтому их уже нельзя называть методами класса. При этом не нужно описывать их прототип, а достаточно описать их один раз.

Задание 7.1.5. Создадим простую, отдельную от какого либо класса функцию с именем Bush(X,Y:integer):integer возвращающую значение произведения X*Y.

Создайте новый проект.

Опишем функцию в окне редактора кода, в разделе реализации после ключевого слова **implementation** и директивы {$R *.dfm}, это должно выглядеть следующим образом:

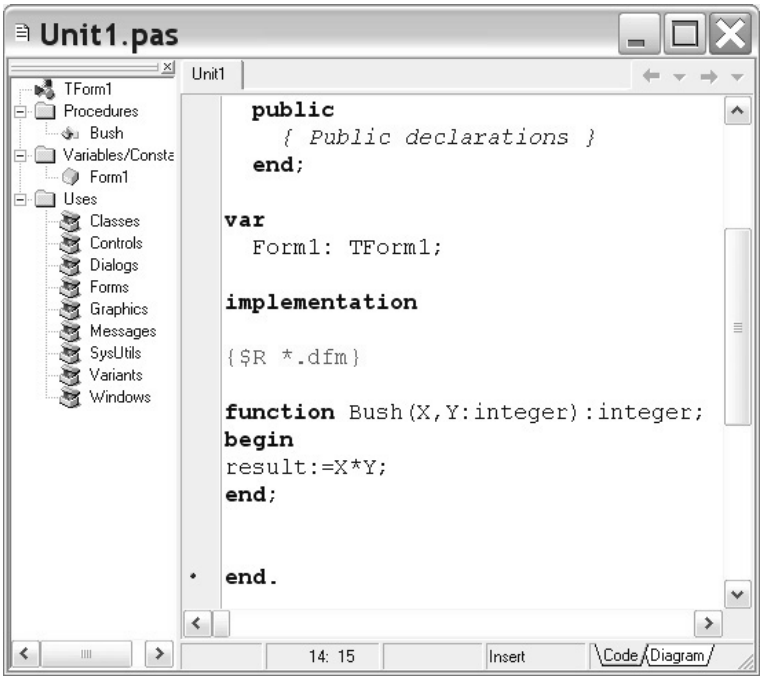


Рис. 7.9 Описание функции с именем «Bush» отдельной от какого-либо класса

Сбросьте Button1 и RichEdit1 из палитры Win32 на форму. Компонент RichEdit1 очень похож на Memo1, но по функциональности RichEdit1 значительно лучше.

Сначала, как и ранее, будем вызывать процедуру Bush из обработчика щелчка на Button1 и выводить возвращаемое ей значение в RichEdit1. Напишите в обработчике щелчка на Button1 следующее:

```
var a:integer;
begin
RichEdit1.Clear; //очищаем RichEdit1
a:=bush(3,5);{помещаем возвращаемое функцией значение в переменную a}
RichEdit1.Lines.Add('Bush ='+IntToStr(a)); {выводим это значение в Richedit1}
end;
```

Запустите программу, убедитесь в правильности ее работы, а затем **вернитесь в режим программирования.**

7.2. **Перегрузка функций**

В Delphi две или более функций могут иметь одинаковое имя, отличаясь либо типом входных параметром, либо их числом, либо и тем и другим.

Таким образом, если две или более функции имеют одинаковое имя, то говорят, что они **перегружены**. Это воплощение одного из базовых принципов программирования под названием – полиморфизм [5,6] (от греческого polymorphos) – свойство, которое позволяет использовать одно и то же имя для решения схожих задач. Особенно этот принцип востребован при работе с виртуальными методами [5] объектов Delphi.

Задание 7.2.1 Напишем еще две перегруженные функции с тем же именем Bush.

Bush(X, Y, Z:word):word; {будет возвращать значение выражения (X+Y)*Z
типа word}
Bush(S, C:string):string; {будет возвращать строку «склеенную» из S и C типа string}

Каждая перегруженная процедура или функция помечается ключевым словом **overload** справа от ее объявления. Конечно, функции и процедуры можно перегружать и когда они принадлежат некоторому классу, то есть можно перегружать методы класса.

В окне редактора кода допишите следующие две дополнительные функции, отмеченные ключевым словом overload:

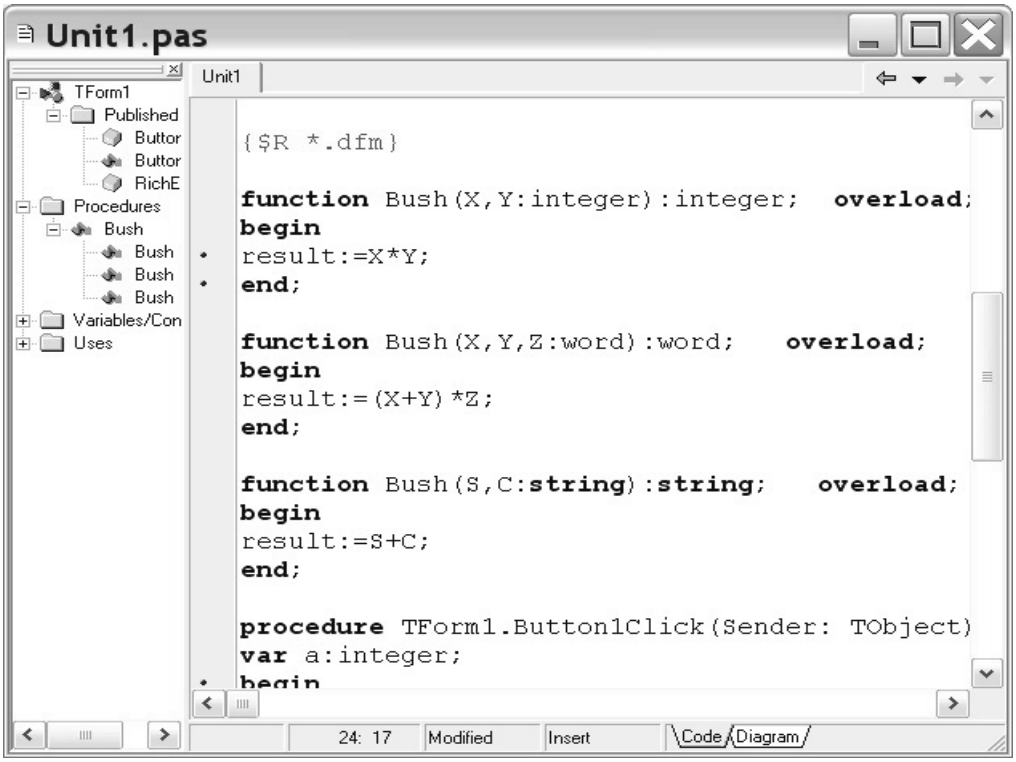


Рис. 7.10 Описание перегруженных функций с одинаковым именем «Bush»

Теперь, как и прежде, вызовем из обработчика Button1 все варианты функции Bush.
Измените код в обработчике щелчка на Button1 на следующий:

```
var a:integer; b:word; Stroka:string;
begin
RichEdit1.Clear; //очищаем RichEdit1
a:=Bush(3,5);
RichEdit1.Lines.Add('Bush(X,Y:integer):integer; ='+IntToStr(a));
b:=Bush(1,3,5);
RichEdit1.Lines.Add('Bush(X,Y,Z:word):word; ='+IntToStr(b));
Stroka:=Bush('Мощный ','Процессор');
RichEdit1.Lines.Add('Bush(S,C:string):string; ='+Stroka);
end;
```

Запустите программу, щелкните на кнопке Button1, вы получите следующий результат:

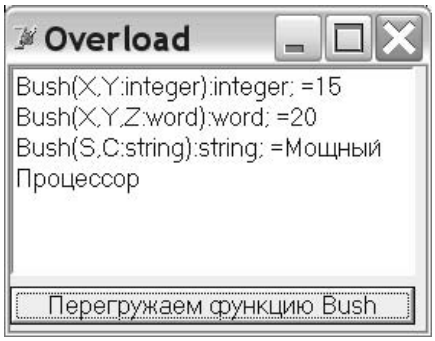


Рис. 7.11 Три перегруженных функции с одинаковым именем Bush, но разными параметрами

Таким образом, в обработчике по очереди выполняются три разных однименных функции с разными параметрами и могут возвращать (в нашем случае делают это) разные значения. Это очень удобно, когда в рамках одного имени нужно делать немного различные вещи. Не приходится заново создавать функцию с другим, хоть и похожим именем. Например, если бы механизм перегрузки отсутствовал, в нашем случае пришлось бы создавать функции: bush1, bush2 и bush3 с разными именами и параметрами.

Вернитесь в режим программирования.

Задание 7.2.2 Кратко изучим структуру модуля unit.

Обратим внимание на структуру модуля Unit1.pas.

Вообще структура модуля представлена следующим образом [4]:

unit «имя модуля»

interface {Секция – интерфейс модуля. Здесь указывается набор подключаемых модулей, объявляются классы, типы, константы и переменные, которые будут доступны из других модулей, а также объявляются прототипы методов классов.}

implementation {Секция реализации. Здесь находятся реализации всех методов классов, объявленных своими прототипами в секции interface, а также доплонительные процедуры и функции, ранее не объявленные в секции interface. Также сюда можно поместить набор подключаемых модулей, объявления типов, констант и переменных, которые не будут доступны другим модулям.}

Под реализацией понимается непосредственно тело того или иного метода, прототип которого объявлен в секции interface.

initialization {Секция инициализации. Программа, помещенная в этой секции, выполняется только один раз при запуске приложения. Данная секция не является обязательной**}.**

finalization { Секция финализации. Программа, помещенная в этой секции, будет выполнена один раз при завершении работы модуля приложения. Данная секция не является обязательной.**}**

end.

Создайте проект. Покажем как работают секции **initialization** и **finalization**.

В файле проекта закомментируете строчку Application.CreateForm(TForm1, Form1) (рис. 7.12) чтобы форма не появлялась при запуске приложения и не мешала.

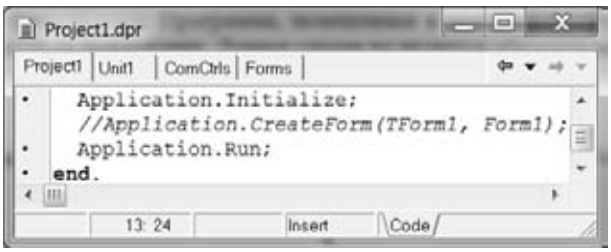


Рис. 7.12 содержимое файла проекта

В секции **interface** объявите класс TClass вместе с методом-конструктором Create и методом Free.

Рядом с ключевым словом var объявим переменную -указатель на объект данного класса Obj.

В секции **implementation** реализуем методы данного класса.

Для демонстрации работы секций **initialization** и **finalization**, поместим в эти секции соответственно, код создания объекта методом-конструктором Create и код освобождения памяти от объекта методом free.

Таким образом полностью код должен выглядеть так:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

type
  TClass = class(TObject)
  public
  constructor Create;
  procedure free;
  end;

var
  Form1: TForm1; Obj:TClass;

implementation
{ TClass }

constructor TClass.Create;
begin
ShowMessage('Секция initialization-выполняюсь 1 раз при запуске приложения');
end;

procedure TClass.free;
begin
ShowMessage('Секция finalization- выполняюсь 1 раз при завершении приложения');
inherited free; // Освобождаем объект вызывая метод free базового класса
end;

initialization
Obj:=TClass.Create; // Создаем объект

finalization
Obj.Free; // Освобождаем объект

end.
```

Запустите данный код и вы увидите сначала результат выполнения кода в секции initialization

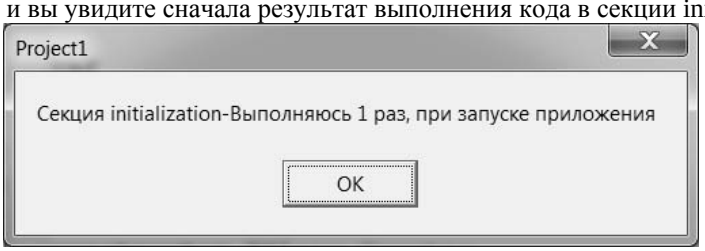


Рис. 7.13 Результат выполнения кода секции initialization

а затем , после щелчка мыши на кнопке "Ок" - резальтат выполнения кода в секции finalization

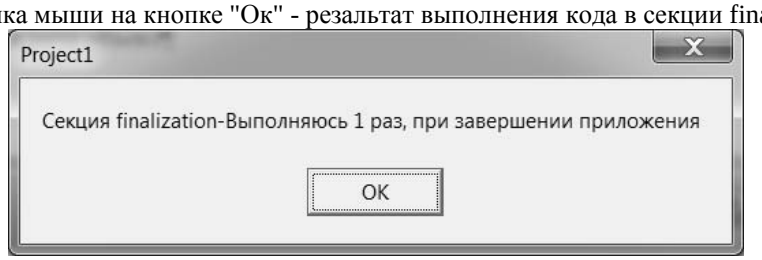


Рис. 7.14 Результат выполнения кода секции finalization

Ещё раз проанализируете результат работы программы.
Вернитесь в режим программирования.

Можно кстати запустить некоторый код и без секций **initialization** и **finalization.** Найдите завершающее модуль unit.pas - ключевое слово "end" с точкой и поставьте перед ним "begin". Таким образм в данном блоке begin ... end можно писать код. Самостоятельно проверьте это.

ПРАКТИЧЕСКАЯ РАБОТА № 8

8.1. Создание приложения «МикроПроводник». Создание заставки

В Windows имеется стандартное приложение «Проводник», однако в Delphi имеются компоненты, позволяющие построить из них аналогичное приложение.

Delphi предоставляет программисту 4 компонента [2], которые можно использовать для работы с файлами и каталогами, которые располагаются на палитре Win3.1:

Компонент DirectoryListBox1 Отображает информацию о текущем каталоге и предоставляет пользователю изенять данные установки. В большинстве случаев данный компонент используется совместно с компонентом FileListBox1

Компонент DriveComboBox1 Отображает все дисководы в системе, позволяет выбрать один из них. В большинстве случаев данный компонент используется совместно с компонентом DirectoryListBox1, поэтому пользователь имеет возможность перейти к другому устройству и просмотреть его каталоги.

Компонент FileListBox1 Отображает файлы в текущем каталоге, может либо выводить информацию обо всех файлах, либо выбирать для отображения группу файлов, используя фильтры, например *.pas или *.txt. Данный компонент обычно связывается с элементом Edit, предоставляя пользователю возможность вводить и редактировать имена файлов.

Компонент FilterComboBox1 Содержит фильтры, которые пользователь может выбирать для того, чтобы ограничить набор файлов, отображаемых компонентом FileListBox1.

Задание 8.1.1 Разработка приложения «Микропроводник» на основе стандартных компонентов DirectoryListBox1, DriveComboBox1, FileListBox1 и FilterComboBox1

Создайте новый проект. Сбросьте DirectoryListBox1 на форму справа, а FileListBox1 слева, а также DriveComboBox1 справа и FilterComboBox1 слева. А также кнопку Button1. Все должно выглядеть следующим образом:

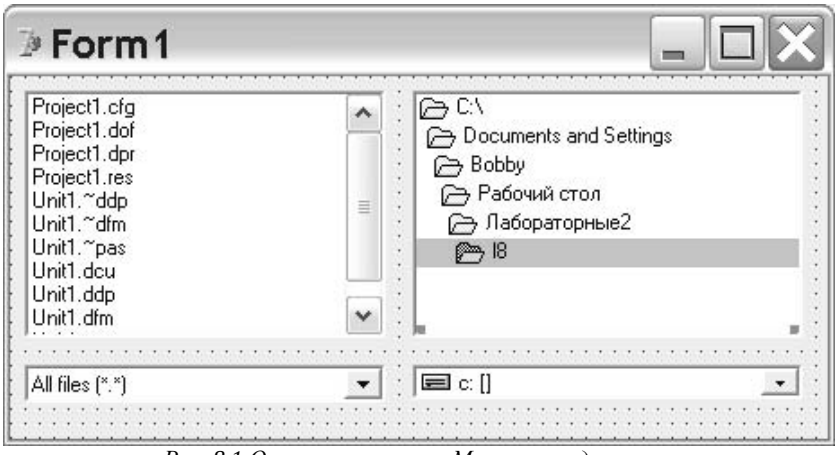


Рис. 8.1 Окно приложения «Микропроводник»

Запустите программу, пощелкайте мышкой в DirectoryListBox1. Вы можете легко перемещаться по папкам, однако файлы, содержащиеся в текущем каталоге компонента DirectoryListBox1, не отображаются в FileListBox1. Исправим это. **Вернитесь в режим программирования.**

Настроим компонент DirectoryListBox1 на FileListBox1 так, чтобы при навигации по папкам в FileListBox1 отображались файлы, находящиеся в текущем каталоге DirectoryListBox1.

В обработчике FormCreate формы напишите:

DirectoryListBox1.FileList:=FileListBox1.

Запустите программу, пощелкайте мышкой в DirectoryListBox1. Вы обнаружите, что файлы, содержащиеся в текущем каталоге компонента DirectoryListBox1, начали отображаться в FileListBox1. Однако вы не можете перемещаться по жестким дискам. **Вернитесь в режим программирования.**

Для навигации по жестким дискам, дисководам и т.д. следует настроить компонент DriveComboBox1 на DirectoryListBox1.

В обработчике FormCreate формы к уже написанному допишите следующее:

DriveComboBox1.DirList:=DirectoryListBox1;

Запустите программу, пощелкайте мышкой в DirectoryListBox1. Вы обнаружите, что можете перемещаться по всем каталогам всех дисков. **Вернитесь в режим программирования.**

Далее поместите на форму несколько компонентов Label, а также Edit1 над компонентом FileListBox1 (рис 8.2)

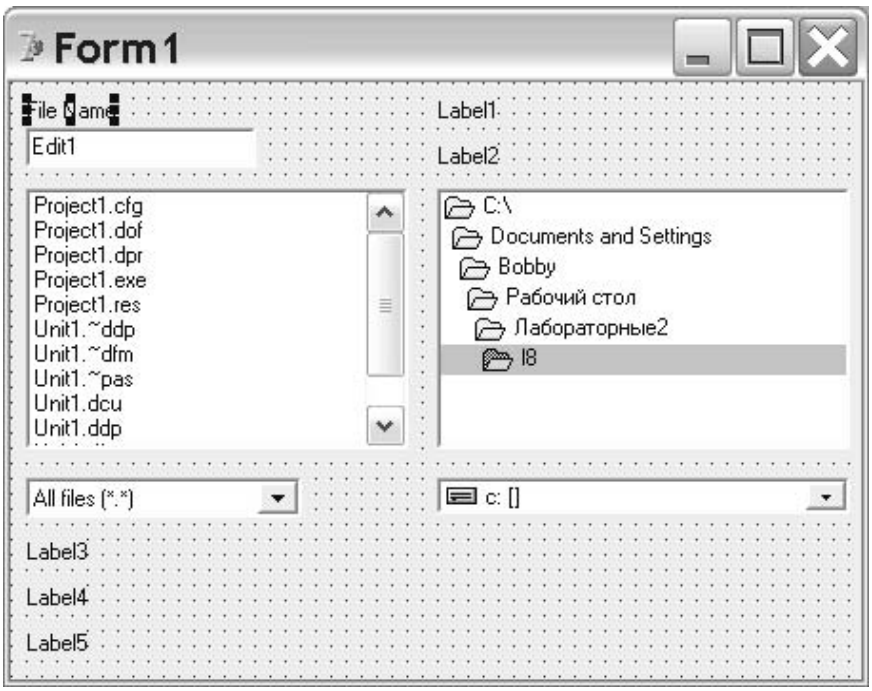


Рис. 8.2 Добавляем компоненты TLabel для отображения полной информации при навигации по папкам с файлам

Будем в Label1 выводить обрезанный путь к файлу, а в Label2 – полный путь.

В обработчике FormCreate формы к уже написанному допишите следующее:

DirectoryListBox1.DirLabel:=Label1; {Настраиваем DirectoryListBox1.DirLabel на Label1}

Label2.Caption:= DirectoryListBox1.Directory; { просто выводим путь в Caption}

Запустите программу. Убедитесь в правильности ее работы. **Вернитесь в режим программирования.**

При навигации по файлам часто используется окно редактирования, здесь в его качестве используется обычный компонент Edit1. После запуска программы в окне отображается фильтр * *, который выбирает все файлы каталога, а также отображает имена выбранных файлов.

В обработчике FormCreate формы к уже написанному допишите следующее:

FileListBox1.FileEdit:=Edit1; {Настраиваем FileListBox1.FileEdit на Edit1}

Запустите программу. Убедитесь в правильности ее работы. **Вернитесь в режим программирования.**

Конечно же, часто требуется выбор файлов по заданному расширению, который обеспечивает компонент FilterComboBox1. Зайдите в его свойство Filter и укажите следующие типы расширений:

Рис. 8.3 Задаем расширения файлов которые будут отображаться в FileListBox1

В обработчике FormCreate формы к уже написанному воспроизведите следующее:
FilterComboBox1.FileList:=FileListBox1;

Запустите программу. Убедитесь, что изменение расширения влияет на отображаемые файлы в FileListBox1. **Вернитесь в режим программирования.**

В общем случае, чтобы обратиться к имени выбранного файла, нужно обратиться к свойству FileName компонента FileListBox1.

Войдите в обработчик щелчка FileListBox1Click и напишите там следующее:

ShowMessage(FileListBox1.FileName);

Запустите программу. Убедитесь, что при выборе файла его имя отобразится на экране. **Вернитесь в режим программирования.**

Теперь здесь же в обработчике данного щелчка закомментируйте эту инструкцию (*поставьте перед ней комментарии // или {}*), чтобы компилятор ее не воспринимал:
// ShowMessage(FileListBox1.FileName);

Если не требуется информация о полном пути, содержащемся в строке FileListBox1.FileName, то ее можно обработать одной из функций Delphi для управления файлами: ExtractFileExt, ExtractFileName и ExtractFilePat.
Сделаем это путем присвоения значений работы этих функций соответствующим свойствам Caption компонентов Label3, Label4 и Label5.
Здесь же в обработчике данного щелчка допишите следующее:

```
with FileListBox1 do begin
Label3.Caption:=ExtractFileExt(FileName);
Label4.Caption:=ExtractFileName(FileName);
Label5.Caption:=ExtractFilePath(FileName);
end;
```

Запустите программу. Самостоятельно исследуете назначение каждой из этих трех функций. **Вернитесь в режим программирования.**

Далее, чтобы можно было выбирать несколько файлов в FileListBox1, установите значение его свойства MultiSelect в True.
Теперь при запуске программы можно мышкой выделять несколько файлов, а если удерживать Ctrl, можно выбирать мышкой разные файлы.
Запустите программу. Убедитесь, что можно выбирать несколько файлов. **Вернитесь в режим программирования.**

Интерес представляет выявление списка выбранных файлов, например, чтобы потом их список передать приложению. Поэтому сбросьте ListBox1 (палитра Standard), а также Button1 (установите Caption:= “Run”) на форму

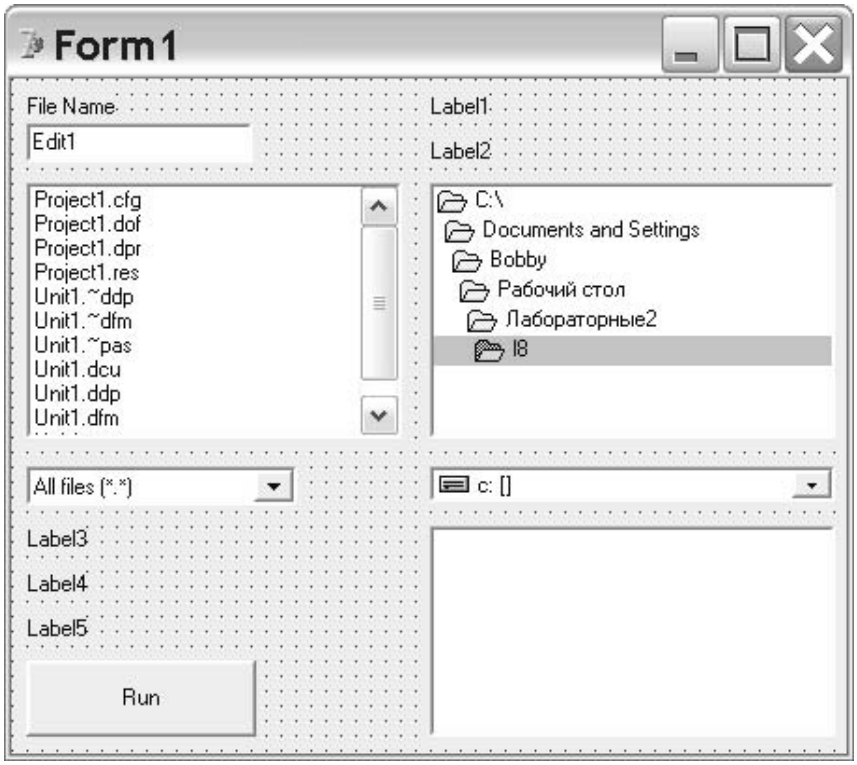


Рис. 8.4 Добавляем компонент ListBox1 для отображения в нем выбранных файлов в FileListBox1

Задание 8.1.2 Будем определять все выбранные файлы и помещать их в список компонента ListBox1

Здесь же в обработчике щелчка FileListBox1Click допишите следующее:

```
ListBox1.Clear; { Каждый раз очищаем ListBox1}
with FileListBox1, Items do begin{Работаем с массивом Items компонента FileListBox1}
If SelCount>0 then {Если выбран один или более элемент массива (файл )}
for i:=0 to Count-1 do, {то пробегаем по индексам массива Items и если}
If Selected[i] then {попался выбранный (выделенный) элемент массива}
ListBox1.Items.Add(Items[i]); { Добавляем его в ListBox1}
                                        end;
```

Не забывайте, что в данный обработчик следует добавить ключевое слово Var, так как появилась переменная i типа integer. В общем виде весь обработчик должен выглядеть так:

```
procedure TForm1.FileListBox1Click(Sender: TObject);
var i:integer;
begin
//ShowMessage(FileListBox1.FileName);
with FileListBox1 do begin
Label3.Caption:=ExtractFileExt(FileName);
Label4.Caption:=ExtractFileName(FileName);
Label5.Caption:=ExtractFilePath(FileName);
                    end;

ListBox1.Clear;
with FileListBox1, Items do begin
If SelCount>0 then
for i:=0 to Count-1 do
If Selected[i] then
```

```
ListBox1.Items.Add(Items[i]);
                                end;

end;
```

Запустите программу. Убедитесь, что при выборе нескольких файлов они отображаются в ListBox1. **Вернитесь в режим программирования.**

8.2. Запуск Программ

Одной навигации по файлам мало. Интерес представляет возможность запустить любое внешнее приложение из своей собственной программы.

В Windows существует такое понятие, как «**WIN API» – Application Program Interface – Интерфейс Прикладного Программирования представляющий из себя большой набор системных процедур и функций операционной системы Windows.**

Одна из часто используемых и полезных функций WIN API является функция **ShellExecute**, она объявлена следующим образом:

ShellExecute(Wnd: HWnd; Operation, FileName, Parameters, Directory: PChar; ShowCmd: Integer): THandle;

Функция возвращает дескриптор (*в данном случае разновидность указателя*) открытого приложения. Если возвращаемое значение меньше или равно 32, это указывает на ошибку.

Wnd – Дескриптор родительского окна, из которого запускаются внешние приложения

Operation – Указатель на строку с нулевым символом в конце, строка может содержать «open» или «print». Если данный параметр равен nil, по умолчанию выполнится операция «open»

FileName – имя файла

Parameters – передаваемые параметры, часто nil

Directory – каталог

ShowCmd – режим открытия файла, может иметь 12 значений, перечислим 4:

SW_HIDE	**Окно делается невидимым**
SW_MINIMIZE	**Свертывает окно**
SW_SHOW	**Активизирует и отображает окно в его текущей позиции с текущими размерами**
SW_SHOWMINNOACTIVE	**Отображает окно в виде пиктограммы**

Функция ShellExecute [2,4] позволяет выполнить любое приложение Windows. Можно также открыть файл, и, если с ним сопоставлено любое приложение, оно запустится.

Чтобы иметь возможность работать со многими API функциями, нужно подключить специальный модуль ShellApi.

Задание 8.2.1 Запрограммируем так, чтобы при выборе конкретного файла в FileListBox1 и двойном щелчке на нем данный файл открывался сопоставленным с ним приложением.

В этом же проекте подключите модуль ShellApi, отвечающий за вызов Api функций. В обработчике события двойного клика (onDblClick) на FileListBox1 напишите следующее:

```
ShellExecute(form1.Handle,nil,PChar(FileListBox1.FileName),nil,
PChar(DirectoryListBox1.directory),SW_SHOW);
```

Запустите программу. Убедитесь, что выбранный файл открывается соответствующим приложением при щелчке на нем. **Вернитесь в РП.**

Задание 8.2.2 С помощью функции WinExec (которая используется реже, чем ShellExecute) также запустим приложение «Калькулятор».

В этом же проекте в обработчике щелчка на кнопке Button1 напишите:

```
WinExec( 'Calc',SW_RESTORE); // вызываем калькулятор
```

Запустите программу. Убедитесь, что калькулятор запускается. **Вернитесь в РП.**

Задание 8.2.3 Самостоятельно, с **помощью API функции WinExec запустите MsPaint и Command.Com.**

8.3. **Окно заставки**

Задание 8.3.1 Покажем, как сформировать окно заставки появляющееся на 3 секунды, а затем исчезающее

Часто при запуске программ требуется не сразу показать рабочее окно приложения, а сначала показать некоторую заставку с информацией о приложении. Сама заставка часто также является стандартным окном Windows.

Создайте новый проект. На экране у вас автоматически появляется форма Form1, описываемая модулем Unit1.pas . Добавьте еще Form2, просто щелкнув на пиктограмме «New Form». У вас в проект добавится также Form2 с соответствующим модулем Unit2.

В данном случае в качестве заставки у нас будет использоваться Form2. То есть логика программы будет следующая: при запуске программы на экране 3 секунды будет находиться Form2, а затем она исчезает и появляется Form1.

Таим образом, измените заголовок Form2 на «Заставка», а заголовок Form1 – на «Основное приложение».

Рис. 8.5 Вид окна основного приложения

Поместите компонент Image1 на Form1, установите значение его свойства Align в alClient, чтобы он растянулся по всей поверхности формы Form2. Затем, используя Object Inspector, зайдите в его свойство Picture, щелкните на кнопке «Load» и загрузите рисунок, выбрав какой-нибудь файл-картинку, например с расширением *.jpg.

Чтобы картинка в компоненте Image1 масштабировалась под размеры формы, установите у него свойство Stretch в значение True.

Рис. 8.6 Вид окна заставки

Таким образом, мы сформировали внешний вид заставки.

Теперь осталось запрограммировать.

Создайте обработчик события onCreate формы Form1, который должен выглядеть следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
var c:DWORD;
begin
c:=GetTickCount div 1000;
repeat
sleep(1);
until (GetTickCount div 1000)>c+3;
end;
```

Данное событие происходит в момент формирования формы. Чтобы задержать ее появление на экране, мы специально затягиваем по времени выполнение кода внутри обработчика этого главного события для формы Form1.

Здесь функция GetTickCount возвращает значение (целое) времени в миллисекундах после загрузки Windows. Данная функция является системной функцией ОС Windows и находится в файле ядра операционной системы Windows kernel32.dll, то есть сначала мы в переменной с целого типа DWORD сохраняем значение числа секунд в начальный момент времени выполнения данного обработчика. Чтобы получить из миллисекунд – секунды, надо разделить число миллисекунд на 1000.

Затем мы организовываем цикл, где сравниваем текущее значение времени в секундах (GetTickCount div 1000) с временем начала работы обработчика плюс 3 секунды.

Как только оно его превысит, цикл заканчивается и на экране появляется Form1.

Далее, как и обычно, в окне редактора кода откройте файл проекта Project1.dpr.

Измените его следующим образом:

```
program Project1;

uses
 Forms,
 Unit1 in 'Unit1.pas' {Form1},
 Unit2 in 'Unit2.pas' {Form2};

{$R *.res}

begin
 Application.Initialize;
 Form2:=TForm2.Create(Application);
 Form2.Show; Form2.Update;

  Application.CreateForm(TForm1, Form1);
 {Application.CreateForm(TForm2, Form2);}
 Form2.Close;
 Application.Run;
end.
```

В данном случае, как только произошла инициализация главного объекта приложения Application, методом-конструктором своего класса TForm2 создается объект класса TForm2, на который указывает переменная-указатель Form2. При этом владельцем данного объекта Form2 назначается объект Application.

Затем методом Show заставка Form2 начинает отображается на экране. По ряду причин, связанных с обработкой сообщений перерисовки окна, мы дополнительно вызываем метод Update.

И далее сразу же мы создаем объект Form1 (считаем его основным окном), используя метод CreateForm главного объекта приложения Application, что влечет за собой событие onCreate формы Form1 и вызов его обработчика. Таким образом, как только Form2 уже выведено на экран и отображается там, окно Form1 еще только создается и не может быть выведено на экран, так как 3 секунды выполняется обработчик события onCreate компонента Form1. Как только работа обработчика завершена, Form1 появляется на экране, а Form2 мы просто закрываем методом Close и у нас остается лишь окно приложения Form1, которое мы считаем основным.

Задание 8.3.2 Самостоятельно создайте еще один вариант предыдущей программы, используя стандартный компонент Timer1.

ПРАКТИЧЕСКАЯ РАБОТА № 9

9.1. Виртуальные и ASC II коды

Каждый символ внутри ПК кодируется определенным кодом, представляющим собой целочисленное значение. Например, в кодировке ASCII – код английской буквы ‘A’ равен 65, ‘B’ равен 66 и т.д. В Pascal код получают с помощью функции ORD(Simvol:Char); здесь параметр Simvol является переменной символьного типа Char и может содержать только один символ (память, отведенная для переменной типа char, составляет 1 байт). Если требуется получить символ по заданному коду, можно написать CHR(65) или CHAR(65), в первом случае CHR – это функция, а во втором – приведение к символьному типу. Также существуют и виртуальные коды специально для всех клавиш. Все символы в кодировке ASCII имеют соответствующие виртуальные коды, но не все виртуальные коды имеют соответствующие коды в ASCII [2]. (Память, отведенная для хранения значения виртуального кода 2 байта (word)).

Нужно отметить, что ASCII коды разработаны для кодирования символов, а виртуальные коды относятся именно к клавишам клавиатуры или мышки, просто многим клавишам (не всем) соответствует символ, поэтому можно провести соответствие между ASCII и виртуальными кодами.

Задание 9.1.1. Выведем в Memo1 несколько символов используя их ASCII коды

Создайте новый проект. Скиньте на форму Memo1 и кнопку Button1 с заголовком «GO».
Обработчик щелчка на Button1 должен выглядеть следующим образом:

```
var Simvol:Char; Kod:integer;
begin
memo1.Clear; { очищаем Memo1}
Simvol:=Chr(65);{Присваиваем переменной Simvol символ ’A’ по его ASCII
коду 65. То же самое получили бы, если бы написали просто Simvol:=’A’ }
memo1.Lines.Add(Simvol); { Выводим символ в Memo1}
end;
```

Запустите программу. Проверьте ее работу. **Вернитесь в режим программирования.**

Теперь, если хотим получить код символа ‘A’, в переменную целого типа здесь же в обработчике щелчка на Button1 допишите к уже написанному следующее:

```
Simvol:=’A’;{Та же инструкция что и Simvol:=Chr(65) в более простом виде}
Kod:=ORD(Simvol); {определяем код символа функцией ORD}
Memo1.lines.add(‘Код символа ’+Simvol+’ равен ’+IntToStr(Kod));
```

Запустите программу. Проанализируйте полученный результат. **Вернитесь в режим программирования.**

Задание 9.1.2. Теперь выведем в Memo1 весь латинский алфавит и ряд других символов.

Ради разнообразия будем использовать цикл Repeat, хотя в данном случае удобнее использовать For.
Здесь же в разделе var добавьте переменную i типа integer и переменную строкового типа stroka:string. Здесь же в обработчике щелчка допишите к уже написанному следующее:

```
i:=0; {тут будет ASCII код символа, начнем с нулевого номера}
stroka:=''; {будем формировать строку из символов, сначала строка пустая}
REPEAT {Начало цикла}
Stroka:=Stroka+' '+Chr(i); {формируем строку из символов типа Char}
i:=i+1; {изменяем код символа на единицу выше (можно и так Inc(i))}
if (i mod 50=0) then { Если в строке накопилось 50 символов}
begin
Memo1.lines.add(Stroka); {то выводим ее в Memo1}
Stroka:=''; {Снова очищаем строку для нового накопления символов}
end;
UNTIL i=220; {Окончание цикла если условие i=220 окажется истинным}
```

Проанализируйте приведенный программный код. Запустите программу. Проверьте ее работу. **Вернитесь в режим программирования.**

Задание 9.1.3. Будем по нажатию клавиш выполнять различные действия.

У виртуальных кодов более широкий диапазон, чем у ASCII кодов в отношении клавиш. В Delphi при программировании обработки нажатия клавиш или щелчков мыши нажатые клавиши возвращаются соответствующими виртуальными кодами, которые задаются целым значением (Key:Word), в отличие от ASCII кодов, которые нужно получать из возвращаемых переменных символьного типа (Key:char).

В этом же проекте скиньте на форму 4 компонента Label, и компонент Panel1. Все должно выглядеть примерно так:

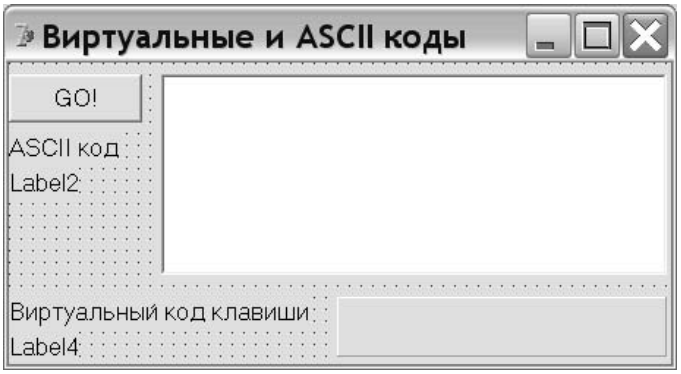


Рис. 9.1 Окно, отображающее ASCII и виртуальный код при нажатии клавиши

При нажатии клавиши выведем в label2 ее ASCII код, а в label4 – ее виртуальный код.

Создайте обработчик события **onKeyPress** формы. Оно происходит каждый раз, когда пользователь нажимает клавишу. Прототип данного события выглядит следующим образом – procedure FormKeyPress(Sender: TObject; **var Key: Char**);

Здесь параметр Key – возвращает символ нажатой клавиши и имеет тип char. Напишите в данном обработчике следующее:

```
label2.Caption:= IntToStr(ORD(Key));{Просто выводим ASCII код нажатой клавиши в Label2 }
```

Далее создайте обработчик события **onKeyDown** формы. Оно также происходит каждый раз, когда пользователь нажимает клавишу. Его прототип выглядит следующим образом – procedure TForm1.FormKeyDown(Sender: TObject; **var Key: Word; Shift: TShiftState**).

Здесь параметр Key – возвращает виртуальный код нажатой клавиши и имеет тип word, а параметр Shift является множеством типа TshiftState, который определен как TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble) и содержит константные значения системных клавиш Shift, Ctrl и т.д.

Внимание! Не путать клавишу Shift и множество Shift возвращаемое данным обработчиком.

Таким образом, клавише Shift соответствует значение ssShift множества Shift, клавише Ctrl – значение ssCtrl также множества Shift и т.д.

Напишите в данном обработчике следующее:

```
label4.Caption:=IntToStr(Key);{ Выводим виртуальный код нажатой клавиши в Label4 }
```

Кроме того, необходимо, чтобы форма всегда первой получала сообщение о нажатой клавише и обрабатывала это событие раньше других компонентов, находящихся на форме.

Поэтому в обработчике создания формы onCreate напишите:

```
form1.KeyPreview:=true;
```

Запустите программу. Нажимайте на клавиши. Обратите внимание, что ASCII коды и виртуальные коды имеют разное значение, при этом Виртуальные коды имеются почти у всех клавиш в отличие от ASC кодов. Нажмите, например, на F2 или на Ctrl.

Вернитесь в режим программирования.

Задание 9.1.4. В этом же проекте будем выводить в Panel1 надпись об одной или нескольких из трех нажатых системных клавиш Shift, Ctrl и Alt.

В обработчике события onKeyDown формы добавьте ключевое слово var и опишите там переменную строкового типа s:string, затем там же допишите к уже написанному следующее:

s:='';
if ssShift in Shift then s:=s+' +Shift'; {если в множество Shift возвращаемое обработчиком входит значение ssShift то формируем строку, в которую добавляем запись ‘ Shift’}
if ssAlt in Shift then s:=s+' +Alt';
if ssCtrl in Shift then s:=s+' +Ctrl';
panel1.Caption:=s; {Выводим сформированную строку в Panel1}

Данное событие будет генерироваться при каждом нажатии клавиш. Запустите программу. Нажимайте на клавиши Shift, Ctrl и/или Alt. Обратите внимание, что, когда вы отжимайте клавиши, надпись в Panel1 не обновляется. **Вернитесь в режим программирования.**

Чтобы надпись обновлялась, нужно просто повторить этот же алгоритм только при отпускании клавиши. Поэтому, используя Object Inspector настройте событие onKeyUp на onKeyDown (просто справа от onKeyUp, выберите из раскрывающегося списка onKeyDown).

Запустите программу. Теперь надпись в Panel1 обновляется. Заодно можно проследить значение виртуальных кодов данных клавиш.

Убедитесь, что:
клавише Shift соответствует виртуальный код 16
клавише Ctrl соответствует виртуальный код 17
клавише Alt соответствует виртуальный код 18
клавише F1 соответствует виртуальный код 112
Вернитесь в режим программирования.

Задание 9.1.5. Запрограммируем обработчик, обрабатывающий нажатия на клавишах Shif, Alt, Ctrl, и клавишах управления курсором так, чтобы:
при нажатии на клавише Shift на форме рисовался красный квадрат;
при нажатии на клавише Alt на форме появлялась новая кнопка Button2;
при нажатии на клавише Ctrl на форме появлялась вторая форма;
при нажатии на клавишах Вверх, Вниз, Вправо, Влево двигалась панель
Пока Клавиша Нажата…

Здесь же в данном проекте в Object Inspector отключите настройку события onKeyUp на onKeyDown, которую вы уже сделали ранее (справа от onKeyUp должно теперь быть пусто), просто обработчик события onKeyUp пока не нужен и будет только мешать, дублируя обработчик события onKeyDown.

Добавим еще одну форму под названием AboutBox. Это совершенно такая же форма класса Tform, только скомпонованная в виде шаблона.

Щелкните File->New-> Other->Forms и выберете AboutBox. Появится форма с именем AboutBox и соответствующий ей модуль unit2. Сохраните этот файл unit2.pas в вашей папке, просто щелкнув Save All. Можете изменить надпись на компонентах Label этой формы, поставив свое имя, версию и т.д.

Теперь в модуле unit1, чуть ниже ключевого слова, **implementation** напишите
uses unit2; // Настраиваем модуль unit1 на unit2

Это нужно для того, чтобы иметь возможность вызвать эту вторую форму AboutBox, которая описывается модулем unit2.

В общем инструкция uses unit2 говорит о том, что теперь в модуле unit1 доступны переменные, объекты, процедуры и функции, объявленные в модуле unit2, и доступ к ним можно получить из процедур и функций модуля unit1. Кстати, из модуля unit2 тоже можно сослаться на unit,1 получится взаимная ссылка модулей друг на друга.

Чтобы перемещаться между файлами модуля (соответственно между формами) и главным файлом проекта, следует пользоваться сочетанием клавиш «Ctrl+F12».

при нажатии на клавише Shift на форме должен рисоваться красный квадрат.

Далее в этом же обработчике события **onKeyDown** добавьте, допишите к уже написанному следующее:
if Key=16 then // Если нажата клавиша Shift

```
with canvas do begin // то, используя метод рисования Canvas
Pen.Width:=5; // устанавливаем ширину обрамляющей линии
Pen.Color:=clGreen; // устанавливаем цвет обрамляющей линии
Brush.Color:=clRed; // устанавливаем цвет внутренней заливки фигуры
Rectangle(72,216,132,256); { выбираем фигурой квадрат с координатами углов где x1,y1,x2,y2 – координаты соотв. Верхнего левого и Нижнего правого углов}
                end;
```

Запустите программу. Нажмите на клавишу Shift. Программа должна работать правильно. **Вернитесь в режим программирования.**

Нужно отметить, что виртуальные коды могут представляться в виде готовых констант, что оказывается очень удобным. Вместо численных значений есть готовые константы. Например:

виртуальному коду 16 клавиши Shift соответствует VC_SHIFT
виртуальному коду 18 клавиши Alt соответствует VC_MENU
виртуальному коду 17 клавиши Ctrl соответствует VK_CONTROL
виртуальному коду 112 клавиши F1 соответствует VK_F1

Поэтому вместо **if key=16 then …**
удобнее писать **if key=VC_SHIFT then …**

Чтобы увидеть все соответствующие виртуальным кодам константы, запустите HELP Delphi, в разделе поиск наберите Virtual key codes и выберите ‘Показать’ (рис. 9.2)

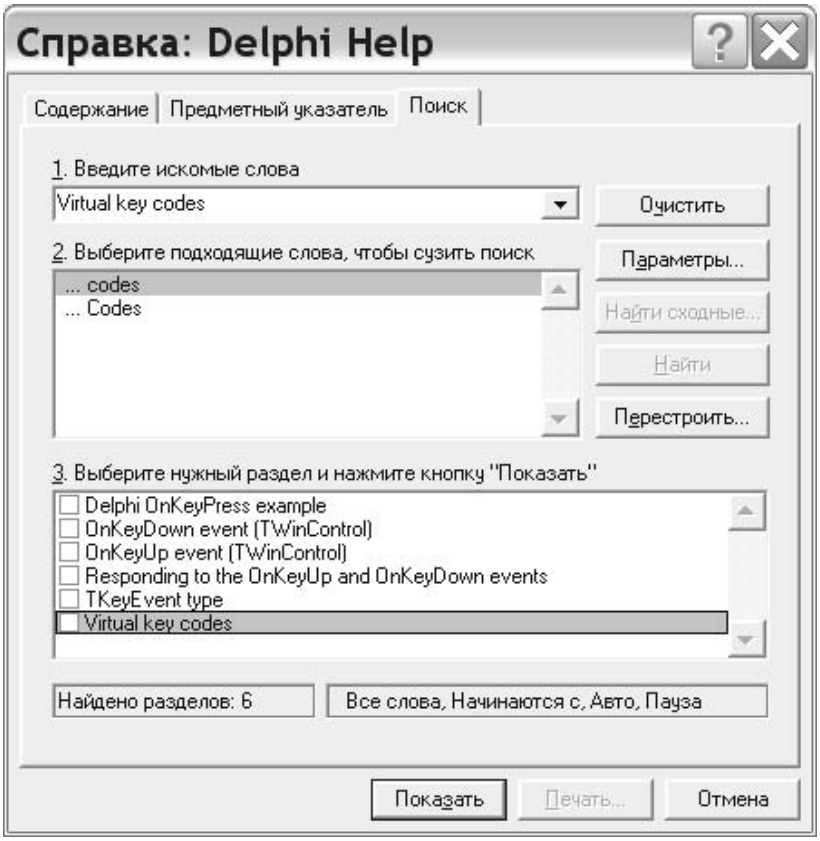


Рис. 9.2 Окно поиска справки Delphi

Теперь, **при нажатии на клавише Alt на форме должна появляться новая кнопка Button2**.

Здесь же, в этом же обработчике в разделе var, напишите Button2:TButton, таким образом определяем тип (класс) объекта, который будет создан. А затем допишите в этом же обработчике следующее:

```
if Key=VK_MENU then begin { Если нажали клавишу ALT}
Button2:=TButton.Create(Form1); { то формируем в памяти объект Button2 и устанавливаем владельцем данного объекта форму }
```

```
Button2.Parent:=Form1; {устанавливаем родителем объекта Button также форму}
Button2.Left:= 152; Button2.top:= 216; {устанавливаем координаты кнопки}
Button2.Width:=150; {и ее длину}
Button2.Caption:='УРА!!! Я Создалась!!!'; { устанавливаем заголовок на ней}
Button2.Show; {показываем ее на форме}
        end;
```

Запустите программу. Нажмите на клавишу Alt. Программа должна работать правильно, и вы должны получить следующий результат:

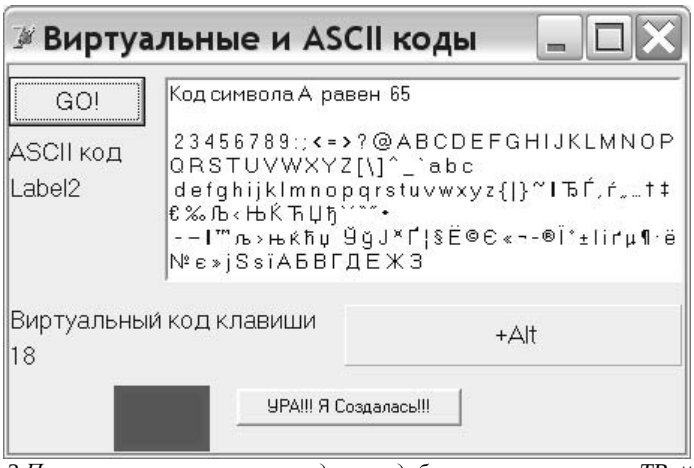


Рис. 9.3 Программно нарисовали квадрат и добавили кнопку класса TButton

Вернитесь в режим программирования.

Затем, **при нажатии на клавише Ctrl, на форме должна появиться вторая форма AboutBox**
Здесь же, в этом же обработчике, допишите к уже написанному, следующее:

```
if Key=VK_CONTROL then AboutBox.Show; {если нажали Ctrl то показать форму AboutBox}
```

Запустите программу. Нажмите на клавишу Alt. Программа должна работать правильно.
Вернитесь в режим программирования.

Теперь **запрограммируем так, чтобы при нажатии на клавише Вверх, Вниз, Вправо или Влево двигалась панель.**

Часто, особенно в компьютерных играх, нужно что-то выполнять **Пока Клавиша Нажата…** Например, стрелять из автомата, пока нажата клавиша.
Создадим совершенно аналогичную программу, где очень просто раскрывается алгоритм Пока Клавиша Нажата…

У клавиши «Влево» виртуальный код VK_LEFT
У клавиши «Вправо» виртуальный код VK_RIGHT
У клавиши «Вверх» виртуальный код VK_UP
У клавиши «Вниз» виртуальный код VK_DOWN

Создайте новый проект. Скиньте на форму Timer1 и Panel1.

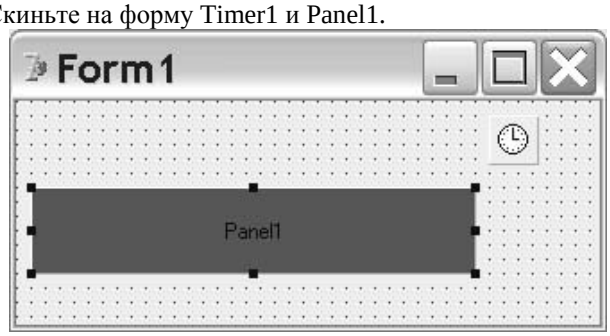


Рис. 9.4 При нажатии и удержании клавиш ←,→, ↑ или ↓ панель будет равномерно двигаться в указанном направлении

Установите у таймера значение интервала 10, а у Panel1 поставьте, например, красный цвет, для этого в свойстве Color выберете clRed.

В разделе описания типа формы TForm1 в секции Public объявите переменную целого типа Klavisha:word; она будет содержать в себе виртуальный код нажатой клавиши.

Создайте обработчик события onKeyDown формы и напишите там следующее:

```
if (Key=VK_LEFT) or (Key=VK_RIGHT)
or (Key=VK_UP) or (Key=VK_DOWN) {Если нажата хоть одна клавиша типа: или Влево или Вправо или Вверх или Вниз }
then {то}
begin
Klavisha:=Key; {запомнить виртуальный код данной клавиши в переменную Klavisha, ее передаем в обработчик таймера, так как она видна во всех методах класса TForm1 (так как находится в секции private данного класса)}
timer1.Enabled:=true; {Активировать таймер}
end;
```

Далее создайте обработчик таймера и напишите в нем следующее:

```
{Если нажата Влево, то сдвигаем панель на 1 влево}
if (Klavisha=VK_LEFT) then Panel1.Left:=Panel1.Left-1;
{ Если нажата Вправо, то сдвигаем панель на 1 вправо}
if (Klavisha=VK_RIGHT) then Panel1.Left:=Panel1.Left+1;
if (Klavisha=VK_UP) then Panel1.Top:=Panel1.Top-1;
if (Klavisha=VK_DOWN) then Panel1.Top:=Panel1.Top+1;
```

Теперь осталось только отключать таймер каждый раз, когда вы отпускаете клавишу.

В обработчике события формы FormKeyUp просто напишите:

```
timer1.Enabled:=false;
```

Все. Программа готова. Запустите ее на выполнение. Управляйте движением панели.

Программа должна работать правильно. **Вернитесь в режим программирования.**

9.2. Самостоятельные задания по написанию программ использующих ASCII и виртуальные коды клавиш

Задание 9.2.1 Самостоятельно измените код предыдущей программы так, чтобы при одновременном нажатии на 2 клавиши, например, «влево»+«вверх» панель двигалась по диагонали влево-вверх, при нажатии «вправо»+«вниз» панель бы двигалась по диагонали вправо-вниз и т.д. для всех направлений.

Задание 9.2.2 Самостоятельно запрограммируете так, чтобы при нажатии на клавише F1 запускался калькулятор.

Задание 9.2.3 Самостоятельно запрограммируете так, чтобы при нажатии кнопки мыши на форме выдавалось сообщение, какая кнопка мыши нажата, левая, правая или средняя.

Логика обработки щелчка мыши на форме точно такая же, как и у обычного нажатия клавиатуры, просто в обработчике, помимо множества Shift, присутствует еще переменная Button типа TMouseButton.

Изучите тип TMouseButton, для этого сначала создайте обработчик события нажатия кнопки мыши на форме onMouseDown, а затем, удерживая клавишу Ctrl, щелкните по надписи TMouseButton, которая будет присутствовать в параметрах обработчика.

Сделайте это двумя способами: используя множество Shift и используя тип TMouseButton, работая с Button (это, по сути, массив).

ПРАКТИЧЕСКАЯ РАБОТА № 10

10.0. Сообщения и их обработка

Фактически вся операционная система Windows организована на посылке и приеме сообщений. Приложения Windows постоянно принимают и отсылают множество различных сообщений. Приложения постоянно взаимодействуют между собой, отправляя и принимая сообщения. Источниками этих сообщений могут быть: операционная система, пользователь, работающий с клавиатурой, другие приложения и т.д. [2,4]

Само сообщение представляет из себя запись типа TMessage, содержащую поля. Поле Msg наиболее важное и содержит информацию о самом сообщении, о его виде. Оно идентифицирует сообщение.

Остальные параметры также несут информацию о данном сообщении, а поле Result несет информацию об обработке приложением полученного сообщения.

```
TMessage = packed record
  Msg: Cardinal;
  case Integer of
   0: (
    WParam: Longint;
    LParam: Longint;
    Result: Longint);
   1: (
    WParamLo: Word;
    WParamHi: Word;
    LParamLo: Word;
    LParamHi: Word;
    ResultLo: Word;
    ResultHi: Word);
 end;
```

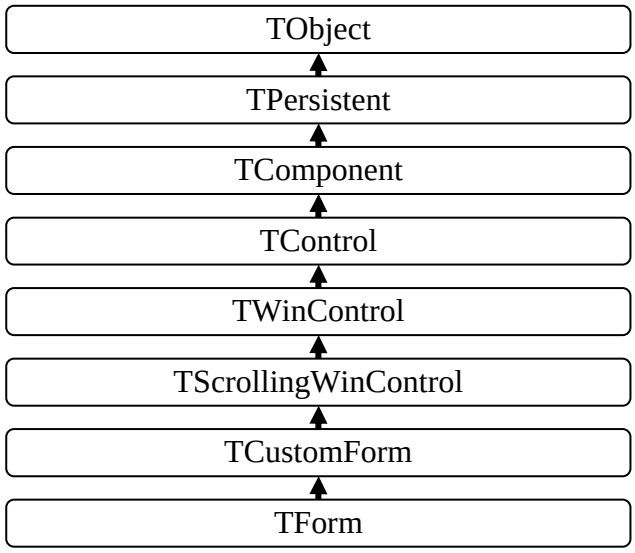
В Delphi стандартные оконные приложения являются потомками класса TWinControl и TControl, и, когда они создаются, они регистрируются в Windows, получая уникальный идентификатор – дескриптор (Handle).

Сначала при запуске программы на выполнение в оперативной памяти компьютера создается объект Application класса TApplication (который является потомком класса TComponent), а затем уже он своим методом Application.CreateForm(TForm1, Form1) инициирует создание в памяти объекта Form1 класса TForm1.

```
procedure TApplication.CreateForm(InstanceClass: TComponentClass; var Reference);
var Instance: TComponent;
begin
 Instance := TComponent(InstanceClass.NewInstance);
 TComponent(Reference) := Instance;
 try
  Instance.Create(Self);
 except
  TComponent(Reference) := nil;
  raise;
 end;
 if (FMainForm = nil) and (Instance is TForm) then
 begin
  TForm(Instance).HandleNeeded;
  FMainForm := TForm(Instance);
 end;
end;
```

Класс TForm1 является наследником класса TForm и перенимает все методы, свойства и поля класса TForm (это записывается инструкцией Form1=class(TForm)), а класс TForm является наследником TCustomForm и т.д. Каждый класс-наследник дополнят своими возможностями свой родительский класс.

Приведем иерархию наследования классов в виде диаграммы классов:



Класс TForm, таким образом, является наследником класса TWinControl и Tcontrol, и **в момент, когда объект класса TControl или TWinControl** (в данном случае Form1) **сформирован в памяти и зарегистрировался в операционной системе, получив уникальный дескриптор – Handle, сразу же в операционной системе регистрируется** специальная для данного объекта – **оконная процедура (window procedure).**

Оконная процедура – это подпрограмма, которая обрабатывает сообщения для окна или другого объекта класса TWinControl или TControl.

В большинстве случаев оконная процедура содержит оператор выбора для большого количества сообщений, которые она должна обработать.

Иначе говоря, операционная система получает указатель на процедуру в памяти, которая автоматически будет вызываться, когда формой или любым другим объектом класса TWinControl или TControl будет получено сообщение. И уже эта процедура будет отвечать за обработку конкретного сообщения. Эта процедура называется MainWndProc.

Нужно также отметить, что наследниками классов TWinControl и, соответственно, TControl является не только форма Form1, но и достаточно большое количество компонентов Delphi, например, класс TButton, TRichEdit, TProgressBar, TRichEdit, TMemo, TListBox, TRadioButton, TCheckBox и др.

Таким образом, у каждого компонента-объекта наследника класса TWinControl или TControl, который создается в памяти, будет своя оконная процедура Wndow Procedure.

Также нужно отметить, что каждый компонент Delphi полностью наследует всю систему, ответственную за посылку и прием сообщений. Система отправки сообщений обрабатывается по умолчанию. Вы определяете обработчики только для тех сообщений, которые требуется обработать специально.

Так, когда окном получено конкретное сообщение, например, “закрыть” WM_CLOSE и вы желаете обработать данное сообщение, то создаете для него специальный обработчик, в этом же приложении затем пишите разные инструкции, которые выполнятся, как только приложение получит это сообщение, но после того как вы описали реакцию вашего приложения на данное сообщение, далее, как правило, желательно положиться на родительский стандартный метод для обработки данного конкретного сообщения и вызвать этот метод прямо из написанного вами обработчика директивой **inherited.**

Данная директива, написанная в методе данного класса, вызывает метод с тем же именем, но принадлежащим классу-предку, то есть директива inherited вызывает родительский метод.

Все компоненты Delphi постоянно находятся в библиотеке компонентов, которая включает Visual Component Library (VCL) – визуальную библиотеку компонентов и библиотеку компонентов для разработки кросс – платформенных (CLX) приложений.

VCL регистрирует **метод MainWndProc как оконную процедуру, вызываемую для каждого типа компонента в приложении.** MainWndProc содержит блок обработки особых ситуаций, передавая структуру сообщения специальному виртуальному методу – **WndProc,** и обрабатывает любые исключения, вызывая метод прикладного класса HandleException [10].

Таким образом, именно виртуальный метод **WndProc, вызываемый из MainWndProc, отвечает за обработку сообщений**, и программист для каждого объекта (компонента) наследника-класса TControl может **переопределить** виртуальный метод WndProc, исходя из надобности.

Метод WndProc состоит в основном из условных операторов if и case, анализирующих идентификатор полученного сообщения Message содержащийся в поле Msg. Прототип метода WndProc следующий: procedure TControl.WndProc(var Message: TMessage);

Также, благодаря методу WndProc, можно отлавливать нежелательные сообщения

и обрабатывать их по-своему. Например, можно запрограммировать так, чтобы в момент перемещения компонента по форме он игнорировал события клавиатуры, и таким образом метод WndProc класса TWinControl будет пропускать события клавиатуры, только если компонент не перетаскивают.

Фактически метод WndProc получает и обрабатывает большое количество сообщений, которые «шныряют» по всей операционной системе Windows [2,5]. В конце концов, WndProc вызывает невиртуальный метод Dispatch – procedure TObject.Dispatch(var Message), унаследованный от прародителя всех классов класса TObject, определяет, какой метод вызывать, чтобы обработать конкретное полученное сообщение.

Метод Dispatch использует поле Msg, описанного выше класса TMessage, структуры сообщения, чтобы определить, как послано специфическое сообщение.

Если в компоненте определен обработчик для данного специфического сообщения, то метод Dispatch вызывает данный метод обработки этого сообщения, а иначе метод Dispatch вызывает обработчик по умолчанию DefaultHandler.

Таким образом, в общем случае схема алгоритма обработки сообщений для большинства объектов следующая:

Схема 10.0. *Схема алгоритма обработки сообщений объектами.*

Для переопределения тех методов-обработчиков сообщений, которые уже определены в стандартных обработчиках компонентов Delphi, нужно просто написать свой собственный обработчик конкретного сообщения или переопределить его с директивой override, если метод был объявлен виртуальным.

Если данное сообщение не предусмотрено в обработчике компонента, нужно перегрузить (переопределить) метод **WndProc** и там описать реакцию объекта на это сообщение**.**

10.1. Исследуем стандартный виртуальный метод WndProc

Задание 10.1.1 С помощью переопределения метода WndProc обработаем сообщение двойного щелчка мыши, в неклиентской области формы (область-полоска, за которую тянем окно мышкой, когда перемещаем окно-форму).

Создайте новый проект. Теперь просто переопределим метод WndProc. Напишите в разделе public класса TForm1 следующее:

```
procedure WndProc(var Message:TMessage); override; {объявили прототип}
```

В данном случае мы объявили прототип. Здесь ключевое слово override указывает на то, что мы переопределяем виртуальный метод WndProc [2], который будет относиться именно к классу TForm1, по отношению к методу WndProc, унаследованному классом TForm1 от его ближайшего потомка – класса TCustomForm.

По значению поля Message.Msg будем идентифицировать полученное сообщение, а затем по собственному желанию можем обрабатывать его нужным способом. Как только обработали данное сообщение, следует вызвать родительский метод WndProc (с помощью директивы inherited) для обработки остальных сообщений в стандартном режиме.

Нажмите Ctrl+Shift+C, и у вас сформируется следующая заготовка процедуры:

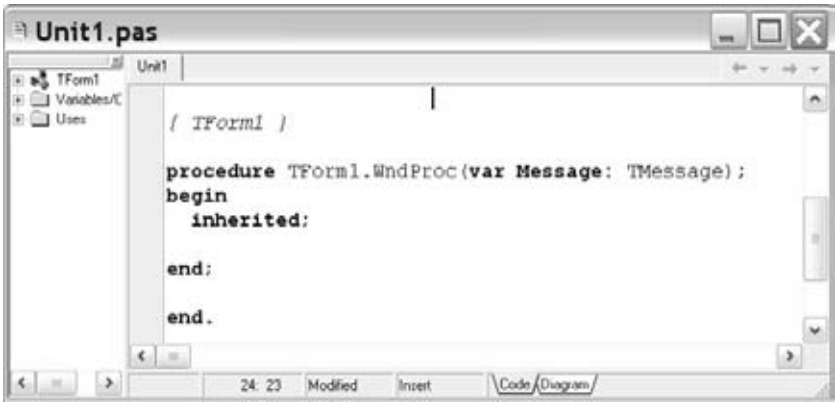


Рис. 10.1 Окно редактора кода содержащее заготовку обработчика WndProc

Измените ее следующим образом:

procedure TForm1.WndProc(var Message: TMessage);
begin
if Message.Msg=WM_NCLButtonDBlClk then
MessageDlg('Получено сообщение о двойном щелчке левой кнопкой мыши в неклиентской области. '+#10#13+'Это полученное сообщение обработано благодаря стандартному методу: '+#10#13+#10#13+'procedure WndProc(var Message:TMessage); override;'
+#10#13+#10#13+'Умеющему работать с полученными сообщениями Windows.'
+#10#13+#10#13+'Ключевое слово override означает переопределенный (измененный) метод', mtInformation,[mbOk],0);
inherited; {вызываем стандартный родительский обработчик сообщений, для обработки сообщений в стандартном режиме}
end;

Здесь с помощью директивы inherited вызываем стандартный родительский метод WndProc с тем же именем объявленный в ближайшем предке класса TForm1 классе TCustomForm, который, в свою очередь, унаследовал этот метод от класса-предка TWinControl, при этом класс TWinControl тоже наследует данный метод WndProc от класса-предка TControl. Именно в классе TControl данный метод является виртуальным и объявлен впервые следующим образом:

procedure WndProc(var Message: TMessage); virtual;

Запустите программу. Щелкните двойным щелчком в верхней неклиентской области формы. Программа должна работать правильно, в результате вы должны получить соответствующую информацию.

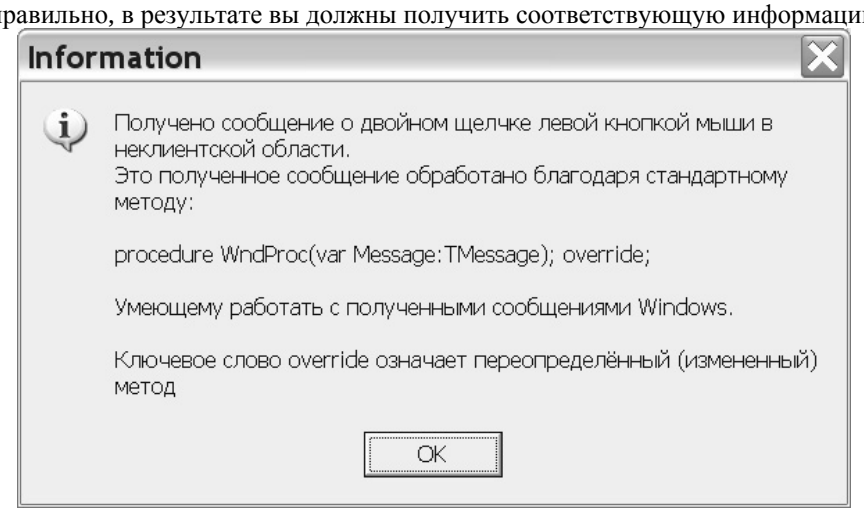


Рис. 10.2. Результат работы программы после двойного щелчка в неклиентской области окна

Вернитесь в **режим программирования.**

Задание 10.1.2. С помощью переопределения метода WndProc обработаем сообщение движения Формы.

Здесь же внутри данного обработчика поместите ключевое слово inherited в самый конец обработчика перед словом end, так как во многих случаях требуется обрабатывать сообщения в стандартном режиме после вмешательства в метод WndProc.

Кроме того, скиньте на форму Shape1 и немного его растяните (рис. 10.3)

Рис. 10.3. Окно приложения «Обработка сообщений»

В разделе public укажите переменную x integer и в обработчике onCreate проинициализируйте ее нулем x:=0, а на форму сбросьте label1, будем просто выводить туда увеличенное на 1 значение переменной x, при каждом получении формой сообщения двигаться. При этом пусть форма мигает случайными цветами и сжимается по вертикали, а фигура Shape1 будет менять свой вид случайно от круга до квадрата.

В обработчике WndProc допишите (ключевое слово inherited должно быть в самом конце обработчика, перед последним end) следующее:

```
if Message.Msg=WM_MOVE then {ежели получено сообщение Двигаться}
begin // То…
inc(x); // увеличиваем x на 1
label1.Caption:=IntToStr(x); // выводим его в label1
color:=random(100000); // Придаем форме случайный цвет
form1.Height:=form1.Height-x;//заставляем форму сжиматься по вертикали
//меняем вид фигуры через каждые 10 сообщений WM_MOVE
if x mod 10 =0 then Shape1.Shape:=TShapeType(Random(6)+1);
end;
```

Запустите программу. Двигайте форму. При движении она станет вертикально сжиматься и мелькать случайными цветами. А фигура будет меняться. В переменной label1 будет число фактически полученных формой сообщений WM_MOVE.

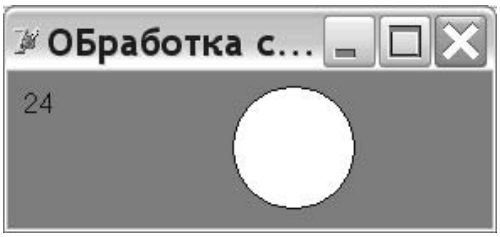


Рис. 10.4. Результат работы приложения «Обработка сообщений» при получении им сообщений о движении WM_MOVE.

Вернитесь в режим программирования.

Задание 10.1.3 С помощью переопределения метода WndProc обработаем сообщение получения формой нажатия клавиши и получим значения виртуального и ASCII кода нажатой клавиши.

В обработчике WndProc допишите к уже написанному (ключевое слово inherited все так же должно быть в самом конце обработчика, перед последним end) следующее:

```
if Message.Msg=WM_CHAR
then
MessageDlg('WM_CHAR'+'  ASCII = '+IntToStr(Message.WParam),
mtInformation,[mbOk],0);

if Message.Msg=WM_KEYDOWN then
MessageDlg('WM_KEYDOWN'+'  Виртуальный код клавиши '+IntToStr(Message.WParam), mtInformation,[mbOk],0);
```

Дело в том, что если принято сообщение WM_CHAR, то в поле WParam переменной Message как раз находится ASCII код. Очевидно, что для другого сообщения в поле WParam будет содержаться уже другая информация, специфичная для этого другого сообщения.

Поэтому, когда принимается сообщение WM_KEYDOWN, в поле WParam структуры сообщения Message, будет помещен виртуальный код клавиши.

Запустите программу. Нажимаете клавиши клавиатуры. Например, если вы нажали клавишу пробел, то вы получите следующий результат:

а) ASCII код пробела *б) Соответствующий пробелу виртуальный код*

Рис. 10.5. Благодаря обработке сообщений WM_CHAR и WM_KEYDOWN, можно получить ASCII и виртуальный коды клавиш

А если нажмете только одну клавишу F5, то результат будет таким:

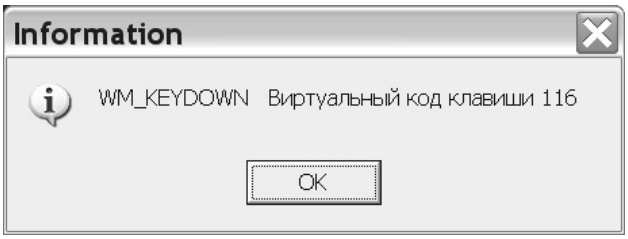


Рис. 10.6. У клавиши F5 имеется только виртуальный код, поэтому при нажатии клавиши получено только одно сообщение _KEYDOWN

Вернитесь в режим программирования.

Каждому сообщению соответствует целое число, удерживайте Ctrl и щелкните на надписи WM_KEYDOWN в окне редактора кода, где вы пишите программу. Вы попадете в модуль Messages, в котором каждому сообщению соответствует целое число, записанное в шестнадцатеричном формате. Так, например, WM_KEYDOWN = $0100. Поэтому в обработчике вместо if Message.Msg=WM_KEYDOWN then… можно было бы написать:

if Message.Msg=$0100 then …

Проверьте это. Программа должна работать правильно. **Вернитесь в режим программирования.**

Задание 10.1.4. С помощью переопределения метода WndProc обработаем сообщение получения формой щелчка мыши. Покажем, что если у объекта есть стандартные предусмотренные обработчики событий, то обработка этих же событий в стандартных предусмотренных обработчиках иногда вызывается позже, чем этих же событий, но в методе WndProc.

Нужно сказать, что любое событие это всегда, по сути, сообщение. Поэтому в некоторых случаях сообщение и событие можно считать синонимами.

Перейдите в обработчик события onMouseDown щелчка на форме. Напишите там следующее:

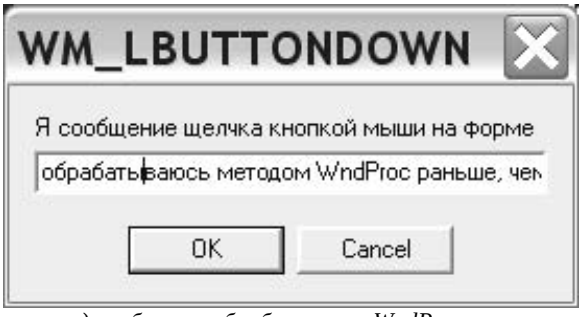
InputBox('onMouseDown', 'Я событие щелчка кнопкой мыши на форме, обрабатываюсь предусмотренным обработчиком FormMouseDown’, 'Однако обрабатываюсь обработчиком FormMouseDown позже, чем в методе WndProc ');

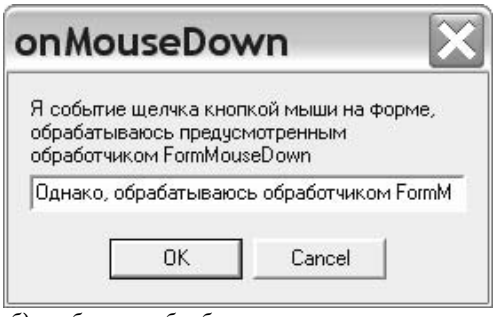
Затем вернитесь в метод WndProc и допишите там, к уже написанному, следующее:

if Message.Msg=WM_LBUTTONDOWN then

InputBox('WM_LBUTTONDOWN','Я сообщение щелчка кнопкой мыши на форме', ‘обрабатываюсь методом WndProc раньше, чем предусмотренным обработчиком FormMouseDown ');

Запустите программу. Щелкайте мышью на форме. Программа должна работать правильно и выдавать ожидаемые результаты.





а) сообщение обрабатывает WndProc *б) сообщение обрабатывается по умолчанию*

Рис. 10.7 Обработка сообщения щелчка мыши левой кнопкой WM_LBUTTONDOWN

Таким образом, первым сообщение WM_LBUTTONDOWN обработает метод WndProc, а уже затем обработает специально предусмотренный стандартный обработчик FormMouseDown. **Вернитесь в режим программирования.**

10.2. Класс TApplication

В Delphi существует 3 наиболее очевидных варианта обработки сообщений [2], один из них – метод WndProc – мы рассмотрели. Также существует класс Tapplication, поддерживающий генерирование множества событий, для которых можно создавать обработчики. Таким образом, созданный объект Application класса TApplication может отлавливать и обрабатывать сообщения Windows. Как и остальные объекты, он имеет обработчики событий по умолчанию.

В его арсенале имеется обработчик полученного события – **Application.OnMessage**, который в данном случае является свойством процедурного типа, которое объявлено в классе TApplication следующим образом:

property OnMessage: TMessageEvent read FOnMessage write FOnMessage;

а тип TMessageEvent имеет следующий вид :

TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;

Поэтому гипотетический прототип метода обработки события onMessage объекта Application выглядит так:

procedure Application.Message (var Msg: TMsg; var Handled: Boolean);

Таким образом, чтобы отловить любое шныряющее к вашему приложению сообщение Windows, достаточно создать свою собственную процедуру с любым именем, например, ApplicOnMessage с теми же параметрами.

Иными словами, процедуру ApplicOnMessage(var Msg: TMsg; var Handled: Boolean) следует описать, а затем просто установить на нее ссылку для обработчика события Application.OnMessage. Это делается простой инструкцией Application.OnMessage:= ApplicOnMessage.

Теперь при получении любого сообщения объектом Application будет вызываться ваш метод обработки сообщений ApplicOnMessage(var Msg: TMsg; var Handled: Boolean).

В этом случае можно идеально выполнить настройку системы реагирования приложения на поступающие сообщения и даже создать целую службу отлова неизвестных сообщений.

```
{ Message structure } {Структура типа TMSG. Она описана в модуле Windows.}
 PMsg = ^TMsg;

 tagMSG = packed record
  hwnd: HWND;
  message: UINT;
  wParam: WPARAM;
  lParam: LPARAM;
  time: DWORD;
  pt: TPoint;
 end;

 {$EXTERNALSYM tagMSG}
 TMsg = tagMSG;
```

```
MSG = tagMSG;

{$EXTERNALSYM MSG}
```

Запись **MSG типа TMsg** включает информацию о дескрипторе окна, посылающего сообщения, идентификаторе сообщения, параметрах сообщения, времени отправления сообщения (в миллисекундах) и координатах курсора. Значения всех этих переменных зависят от специфики получаемого сообщения.

Параметр **Handled.** Если обработчик события предназначен для обработки сообщений, тогда этому параметру нужно присвоить True, а если не предполагается их обрабатывать, тогда False.

Задание 10.2.1. С помощью объекта Application попробуем отловить и обработать некоторые сообщения. Будем определять, какой кнопкой мыши щелкнули на форме и выведем координаты указателя мыши относительно всего экрана, основываясь на информации поля Pt в сообщении типа TMsg.

Создайте новый проект. Скиньте 2 компонента Label, туда будем выводить информацию о координатах мыши. В секции Public создайте прототип обработчика сообщений

```
procedure ApplicOnMessage(var Msg: TMsg; var Handled: Boolean);
и нажмите Ctrl+Shift+C
```

Перед вами появится заготовка данного обработчика. Но прежде чем там что-то писать, нужно настроить метод Application.OnMessage на ApplicOnMessage. Лучше это сделать в самом начале при создании формы. Поэтому в обработчике onCreate напишите:

```
Application.OnMessage:= ApplicOnMessage;
{настраиваем метод Application.OnMessage на свой обработчик ApplicOnMessage}
```

Затем перейдите в обработчик ApplicOnMessage, он должен выглядеть следующим образом:

```
procedure TForm1.ApplicOnMessage(var Msg: TMsg; var Handled: Boolean);
var Koord:TPoint;
begin

if Msg.message=WM_LBUTTONDOWN { Если щелкнули левой кнопкой, то}
then            begin
label1.Caption:='Щелкнули левой кнопкой';
Koord:=Msg.pt; { присваиваем переменной Koord поле pt сообщения Msg }
label2.Caption:=IntToStr(Koord.X)+'  '+IntToStr(Koord.Y); {Выводим координаты указателя мыши относительно экрана!!!, а не формы!}
                        end;
end;
```

Запустите программу. Щелкайте левой кнопкой мыши на форме. На другие кнопки мыши форма не должна реагировать. Вы получите примерно такой результат.

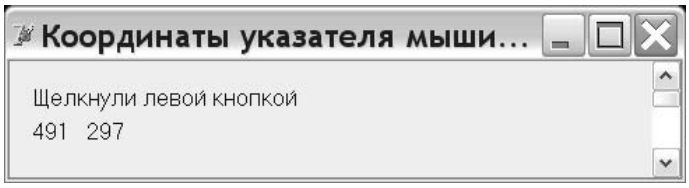


Рис. 10.8 После щелчка мыши отображаем информацию о координатах указателя мыши относительно экрана

В данном случае мы выводим координаты указателя мыши относительно экрана, а не относительно формы. Получение координат указателя мыши относительно формы будет дано ниже в самостоятельном задании. Таким образом, мы отловили сообщение WM_LBUTTONDOWN и извлекли из него требуемую информацию.

Вернитесь в режим программирования.

Задание 10.2.2. Запрограммируем обработку сообщения колесика мыши. Будем при вращении колесика уменьшать или увеличивать видимую площадь картинки.

Подключите модуль Jpeg в разделе описания модулей для правильного отображения графических файлов.

Сбросьте на форму компонент Image1.

Далее загрузите в него какой-нибудь средних размеров рисунок. Для этого щелкните на его свойстве Picture, появится окошко (рис. 10.3)

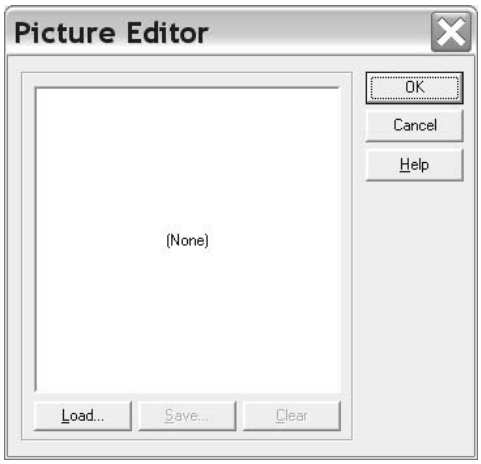


Рис. 10.9 Picture Editor компонента Image1

А затем щелкните Load. Загрузите туда средних размеров рисунок.

Таким образом, окно приложения будет выглядеть примерно так:

Рис. 10.10. Окно приложения «Координаты указателя мыши» в режиме разработки

Нужно сказать, что для события WM_MOUSEWHEEL параметры wParam и lParam записи Msg имеют:

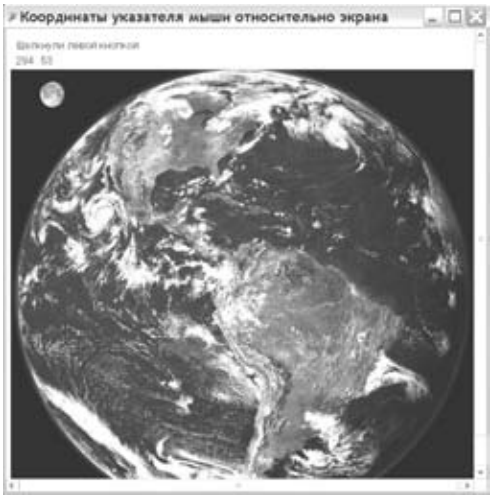
во – первых: одинаковое значение;

во – вторых: если вращаем колесико вперед , то положительное значение, а если назад – отрицательное.

Далее в обработчике ApplicOnMessage допишите к уже написанному следующее:

```
if Msg.message=WM_MOUSEWHEEL then begin {если крутим колесико, то}
if Msg.wParam>0 then {если в поле положительное значение (крутим вперед), то}
                begin
Image1.Width:=Image1.Width+7; {увеличиваем размеры видимой площади картинки}
Image1.Height:=Image1.Height+7;
                end
else {иначе, если в поле отрицательное значение, крутим назад}
                begin
Image1.Width:=Image1.Width-7; {уменьшаем размеры видимой площади картинки}
Image1.Height:=Image1.Height-7;
                end;
                                end;
```

Запустите программу. Крутите колесико мыши. Видимая площадь картинки будет либо расширяться, либо уменьшаться в зависимости от направления движения колесика мыши.



а) крутим колесико назад *б) крутим колесико вперед*

Рис. 10.11. Видимая площадь картинки зависит от направления вращения колесика мыши.

Вернитесь в режим программирования.

10.3. Собственные обработчики событий. Функции SendMessage и PostMessage

Существует третий вариант обработки сообщений. Написание своих собственных обработчиков сообщений. Это можно осуществить двумя способами. Либо заменить стандартный обработчик стандартного сообщения Windows своим собственным обработчиком, либо создать свое собственное оригинальное сообщение и написать оригинальный обработчик для него.

Задание 10.3.1. Запрограммируем обработку сообщения на закрытие окна WM_CLOSE. Пусть при получении данного сообщения форма медленно “растворяется”.

В этом же проекте у Form1 установите значение 255 свойства AlphaBlendValue, отвечающего за ее прозрачность, а также значение True свойства AlphaBlend, отвечающего за возможность “упрозрачнения” формы.

Далее скиньте Timer1 на форму. Установите его свойство interval равным 60, а свойству Enabled присвойте значение False (таймер не должен быть активирован сначала).

Создайте обработчик таймера, он должен выглядеть следующим образом:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var R:TRect; S:string;

begin
AlphaBlendValue:=AlphaBlendValue-3;{увеличиваем прозрачность}
if AlphaBlendValue<=5 then close; {закрываем приложение стандартной процедурой close}
R:=Form1.ClientRect; {устанавливаем размер области для вывода текста}
canvas.Font.Name:='Times New Roman'; {устанавливаем шрифт}
canvas.Font.Color:=ClRED; {и цвет}
canvas.Font.Style:=[fsItalic]; {а также стиль}
canvas.Font.Size:=100-AlphaBlendValue; {и меняющийся размер}
S:='РАСТВОРЯЯЯЮЮСЬ'; {сам текст}
canvas.TextRect(R,30,30,S); {вывод текста S в область R с координатами}
end;
```

Теперь переопределим стандартный обработчик события WM_CLOSE своим собственным. В разделе public напишите:

```
procedure WMCLOSE(var Message:TWMCLOSE); message WM_CLOSE;
```

затем нажмите Ctrl+Shift+C.

В появившемся обработчике просто напишите:

if Application.MessageBox('или стандартным способом? – НЕТ',
'Я закроюсь с растворением – ДА', mb_YESNO)=IdYES then
begin
timer1.Enabled:=true; {активируем таймер}
Message.Result:=0; {указываем что приложение обрабатывает это сообщение. Для сообщения WM_CLOSE, если оно нормально обрабатывается, должен возвращаться 0}
end
else {иначе}
inherited; {вызываем стандартный метод обработки данного сообщения}

Таким образом, при попытке закрыть приложение, оно получит сообщение WM_CLOSE и вызовется ваш переопределенный метод, в котором вы запрограммировали вопрос, выведенный вам на экран функцией MessageBox. Если вы на вопрос ответите «Да», то запустится таймер, в котором мы запрограммировали, что форма будет становиться все более прозрачной с течением времени (визуально растворяется), а затем приложение закроется, благодаря стандартному методу Close. А если вы ответите «Нет», то вызовется родительский метод обработки сообщения WM_CLOSE, который сразу закроет ваше приложение.

Запустите программу. Затем попробуйте ее закрыть обычным способом. Программа должна работать правильно.

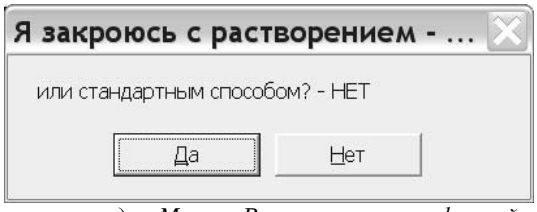


Рис. 10.12а. Вывод вопроса методом MessageBox при получении формой сообщения WM_CLOSE

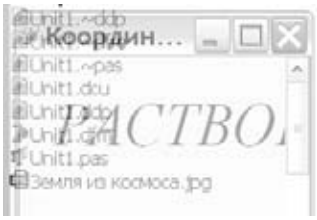


Рис. 10.12б. Работа обработчика таймера в случае ответа «Да»

Вернитесь в режим программирования.

Функции SendMessage и PostMessage.

Функция SendMessage [2,4] создана для отправки сообщения другим объектам операционной системы Windows. В Delphi она имеет следующий вид:

function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
hWnd – дескриптор объекта которому передается сообщение
Msg – само сообщение, его идентификатор
wParam – содержит дополнительную информацию
lParam – содержит дополнительную информацию

Она возвращает значение, когда приложение обработало сообщение. Этим она отличается от такой же функции PostMessage, которая возвращается сразу, как только сообщение отправлено.

Задание 10.3.2 Используя функцию SendMessage отправим сообщения: закрыться, свернуться или развернуться стандартным приложениям Windows: Калькулятору и Блокноту.

Здесь же поместите на форму 3 кнопки: Button, 2 кнопки с зависимой фиксацией TradioButton. Установите у одной из кнопок TradioButton значение checked, равное True. Установите заголовок кнопки RadioButton1 в «Свернуть», а для RadioButton2 в «Развернуть».

Все должно выглядеть примерно так:

Рис. 10.13 Приложение запускающее, а затем сворачивающее или разворачивающее стандартные приложения «Калькулятор» или «Блокнот»

В обработчике кнопки “Запуск Калькулятора и Блокнота” напишите следующее:

```
WinExec('Calc', SW_Restore); {Запускаем Калькулятор}
WinExec('Notepad', SW_Restore); {Запускаем Блокнот}
```

Обработчик кнопки “Закрыть Калькулятор и Блокнот” должен выглядеть так:

```
var hWnd: THandle;
begin
hWnd:=FindWindow('SciCalc',nil); {Определяем дескриптор окна класса 'SciCalc'}
SendMessage(hWnd,WM_Close,0,0);{Отправляем сообщение WM_CLOSE окну с данным дескриптором}
hWnd:=FindWindow('Notepad',nil); { то же для Блокнота}
SendMessage(hWnd,WM_Close,0,0);
end;
```

Обработчик кнопки “Свернуть или развернуть Все” должен выглядеть так:

```
var hWnd: THandle;
begin
hWnd:=FindWindow('Notepad',nil); {Определяем дескриптор окна класса
' Notepad ' с любой надписью заголовка}
if radiobutton1.Checked=true then {если выбрана первая кнопка}
SendMessage(hWnd,WM_SYSCOMMAND,SC_MINIMIZE,0)
{ то сворачиваем}
else            {иначе}
SendMessage(hWnd,WM_SYSCOMMAND,SC_MAXIMIZE,0);
{разворачиваем}
hWnd:=FindWindow('SciCalc',nil); {Определяем дескриптор окна класса
'SciCalc' с любой надписью заголовка}
if radiobutton1.Checked=true then
SendMessage(hWnd,WM_SYSCOMMAND,SC_MINIMIZE,0)
else
SendMessage(hWnd,WM_SYSCOMMAND,SC_MAXIMIZE,0);
end;
```

Запустите программу. Щелкните на кнопке «Запуск калькулятора и блокнота». Данные приложения должны запуститься. Затем установите точку на надпись «Свернуть» и нажмите на кнопку «Свернуть или развернуть», калькулятор и блокнот должны свернуться. Если затем поставить точку на надпись «Развернуть», приложения соответственно развернутся. Обратите внимание, что у калькулятора в обычном режиме недоступно расширение, но мы можем заставить его расшириться, отправив ему сообщение развернуться. В развернутом виде он выглядит непривычно, занимая всю верхнюю часть экрана по всей ширине:

Рис.10.14. Приложение «Калькулятор» в развернутом виде благодаря отправленному ему сообщению функцией SendMessage

Вернитесь в режим программирования.

10.4. Создание своего собственного сообщения

В Delphi существует возможность создать свое собственное сообщение и установить на него обработчик.

Задание 10.4.1. Запрограммируем так, чтобы первое приложение имело возможность посылать второму собственное сообщение, придуманное программистом, а второе приложение реагировало бы на него.

В Delphi существует специальная константа WM_USER, благодаря которой можно создать множество собственных сообщений. Например, величина WM_USER + 1 уже будет относиться ко второму сообщению, WM_USER + 2 – к третьему и т.д.

Создайте новый проект и Папку назовите Priloj_1. Здесь будет **первое приложение отправитель.**

В свойстве Caption формы напишите: «Приложение1»
Скиньте button1 и напишите там текст: Отправить сообщение второй форме
Скиньте два компонента Edit, там будут значения, которые будут отправляться второму приложению (рис. 10.15)

Рис. 10.15 Приложение1 отправляющее «пользовательское» сообщение другому приложению

Затем объявите имя вашего сообщения в интерфейсной части, например, WM_MyMessage, используя константу WM_USER, то есть WM_MyMessage= WM_USER.

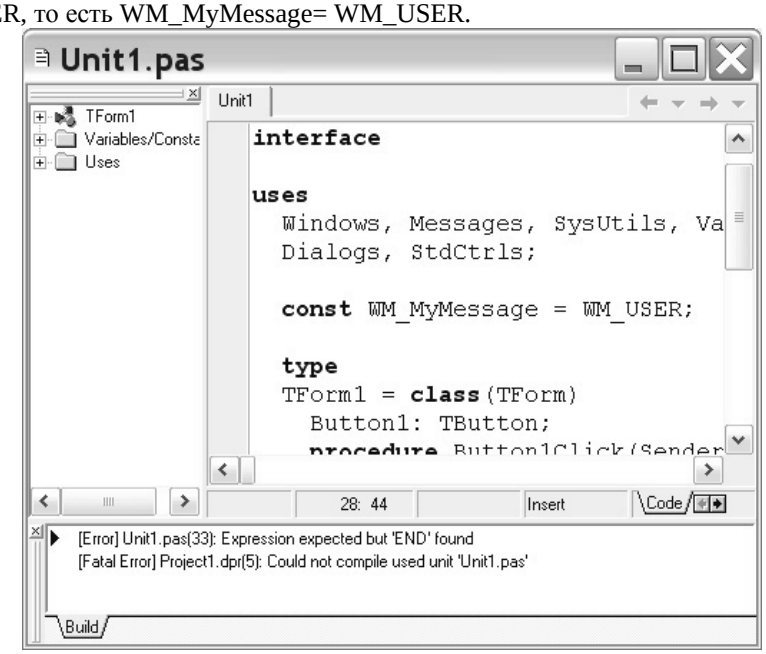


Рис. 10.16 Объявляем свое собственное сообщение WM_MyMessage

Итак, вы объявили свое собственное «пользовательское» сообщение WM_MyMessage.

Далее в обработчике щелчка кнопки запишите var HWND:THandle;
А затем напишите следующее:
begin
HWND:=FindWindow('TForm1','Приложение2'); {Определяем дескриптор второго приложения по его классу и надписи на заголовке}

SendMessage(HWND,WM_MyMessage,StrToInt(Edit1.Text),StrToInt(Edit2.Text));
{Отправляем сообщение WM_MyMessage второму приложению со значениями двух введенных параметров}
end;

Далее откомпилируйте ваш проект, нажав Ctrl+F9 или F9 на предмет ошибок и сохранив, закройте его.

Теперь нужно создать приложение-получатель.
Создайте новый проект и Папку назовите Priloj_2. Здесь будет приложение получатель. В свойстве Caption формы напишите: «Приложение2» (рис. 10.17)

Рис. 10.17 Приложение-получатель

В окне редактора кода точно также объявите ваше сообщение с тем же именем WM_MyMessage= WM_USER; и в public укажите прототип обработчика вашего сообщения:

procedure WMMyMessage(var Message:TMessage); message WM_MyMessage;

затем нажмите Ctrl+Shift+C.
В появившемся обработчике вашего сообщения напишите:

```
MessageDlg('Получено сообщение WM_MyMessage'
+#10#13+'с параметрами '
+#10#13+IntToStr(Message.wParam)+' '+IntToStr(Message.LParam)
,mtInformation, [mbOk],0);
Form1.Height:=Message.wParam;{Устанавливаем высоту Второй формы по
полученному параметру}
Form1.Width:=Message.LParam; {Устанавливаем ширину Второй формы по
полученному параметру}
```

Теперь откомпилируйте ваш проект на предмет ошибок и закройте его.
Закройте саму среду Delphi.
Далее запустите ваше первое и второе приложение, не используя среду Delphi, а просто по щелчку на *.exe-файле. Попробуйте отправить значения второй форме-получателю, используя первую-отправителя. Программа должна работать правильно.

Задание 10.4.2. Самостоятельно. С помощью переопределения метода WndProc узнайте координаты мыши относительно формы при щелчке на ней и выведите их на экран, используя ShowMessage().

Указание. Используйте проверку получения сообщения WM_LBUTTONDOWN щелчка мыши на форме, а затем выводите информацию на экран, например так:

```
ShowMessage ('X= '+IntToStr(Message.Какое_то_поле)+
                    ' Y= '+IntToStr(Message.Какое_то_поле));
```

Вы сами должны догадаться, какое поле нужно поставить после оператора «точка» у сообщения Message.

ПРАКТИЧЕСКАЯ РАБОТА № 11

11.0. Базы Данных

На сегодняшний день использование баз данных приняло повсеместный характер. Любая серьезная фирма обязательно имеет собственную базу данных о всех сотрудниках, товаре и проч. Поэтому очень часто в обязанности программиста той или иной компании входит работа и программирование баз данных.

База данных (БД) – это набор однородной, как правило, упорядоченной по некоторому критерию информации. В математической энциклопедии [7] дается следующее опредление БД – организованная совокупность данных, предназначенная для длительного хранения (обычно во внешней рамяти ЭВМ) и постоянного применения.

Компьютерная база данных представляет собой файл (или набор связанных файлов), содержащий информацию. Компьютерная Б.Д. в своей работе опирается на ту или иную файловую систему, которая обеспечивает хранение данных, их поиск во внешней памяти и выдачу на внешние устройства ЭВМ [7].

База данных состоит из записей. Каждая запись содержит информацию об одном экземпляре.

Базу данных удобно представить в виде таблицы. При этом заголовок столбца таблицы – это имя поля, а номер строки таблицы – номер записи.

Delphi предоставляет очень широкие возможности по разработке БД, кратко их рассмотрим.

11.1. Работа с мастером Баз Данных

Задание 11.1.1 Научимся работать с матером Баз Данных, работающим с уже готовыми, созданными ранее базами данных.

Создайте новый проект. В начале. как обычно, на экране у вас должна быть отображена Form1. Затем выберите File -> New -> Other -> Business -> Database From Wizard и нажмите OK (рис. 11.1).

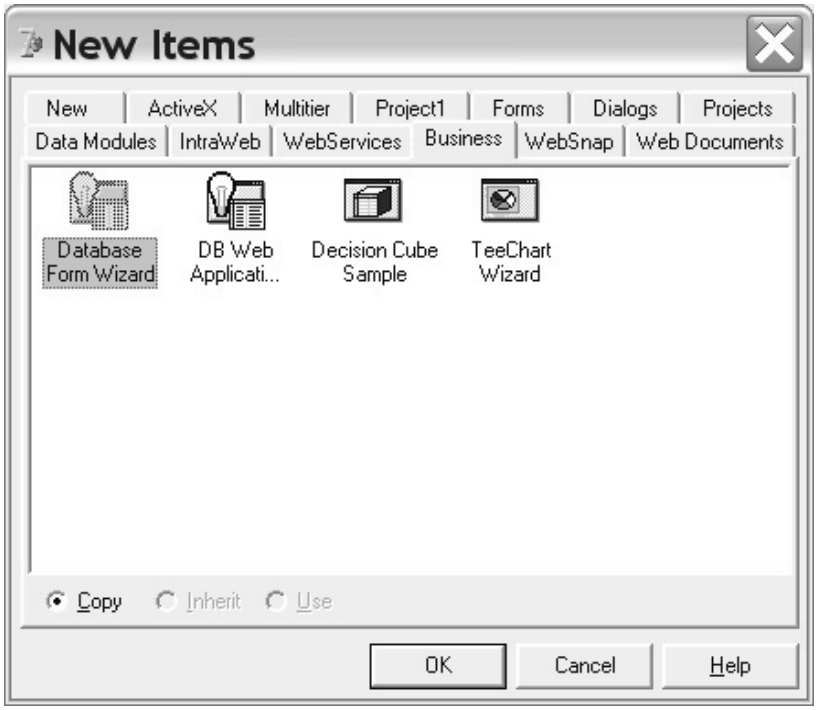


Рис. 11.1. Окно выбора Мастера БД

В данном случае потребуется создание простой формы (simple for, а БД должна быть представлена объектом класса TTable.

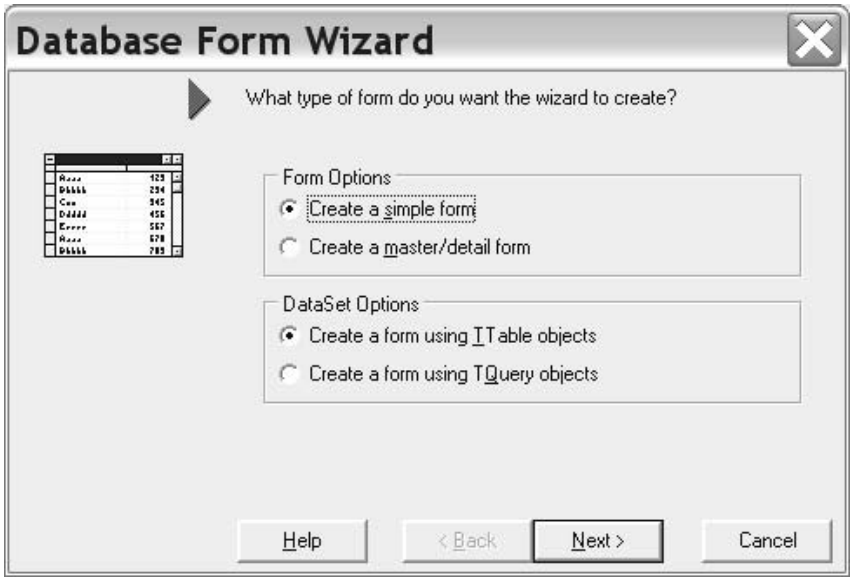


Рис. 11.2 Начальное диалоговое окно матера БД

Это установки по умолчанию щелкните Next.

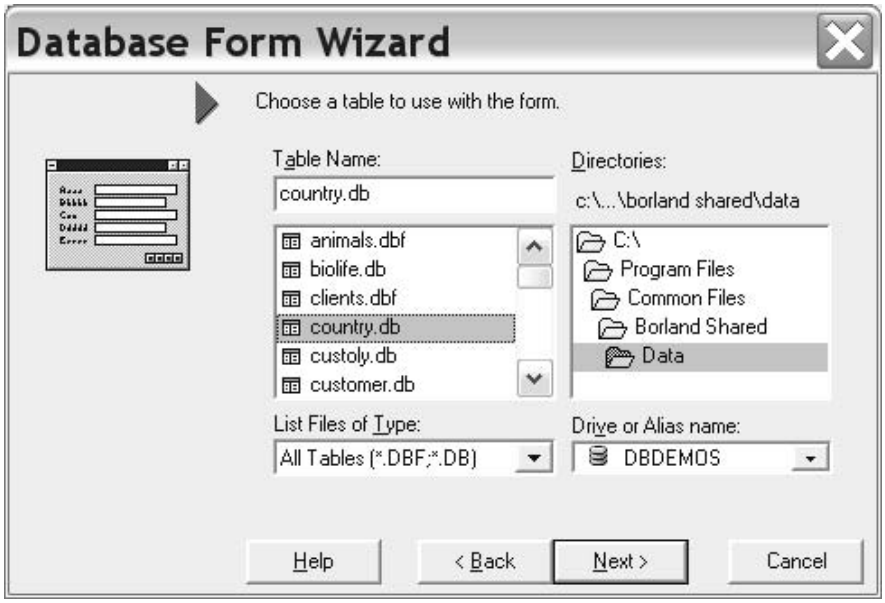


Рис. 11.3 Окно выбора псевдонима БД (Alias) и таблицы в мастере форм БД

Теперь требуется указать путь к БД или ее **Псевдоним (Alias) – Это имя, зарегистрированное в Borland Database Engine(BDE), под которым скрывается название жесткого диска и путь к файлам БД.** Для обращения к файлам БД желательно использовать псевдонимы вместо физического пути и имен файлов [2]. Так как если имеется псевдоним, то файлы БД можно перенести в другой каталог, а все приложения, использующие эти файлы, будут работать без изменений.

Здесь **BDE** – является, по сути, системной утилитой или инструментом фирмы Borland, поставляемым в комплекте с Delphi7, обеспечивающим полный набор программных средств для работы с таблицами большинства наиболее известных форматов баз данных, таких, как dBase, MSACCESS, Paradox и т.д.

Выберите псевдоним DBDEMOS в вкладке «Drive or Alias name», а затем таблицу country.db в списке Table Name. Данная БД с псевдонимом DBDEMOS является готовой стандартной БД, поставляемой вместе с Delphi в учебных целях.

Нажмите Next. Перед вами появятся поля, которые будут отображаться в вашей базе данных. В части Available будут отображены все поля данной БД, а в Ordered Selected Fields – те, которые будут отображены в вашей таблице.

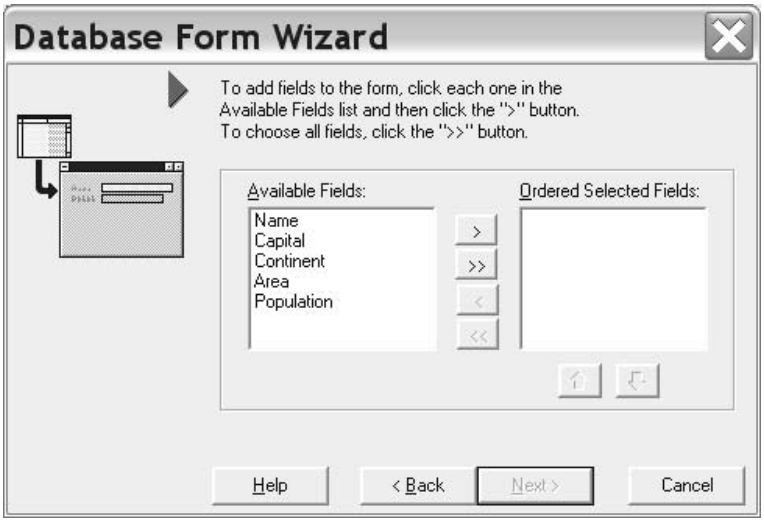


Рис. 11.4 Окно выбора требуемых полей

Сделаем так, чтобы в приложении были доступны все поля. Просто щелкните на кнопке >> и нажмите Next.

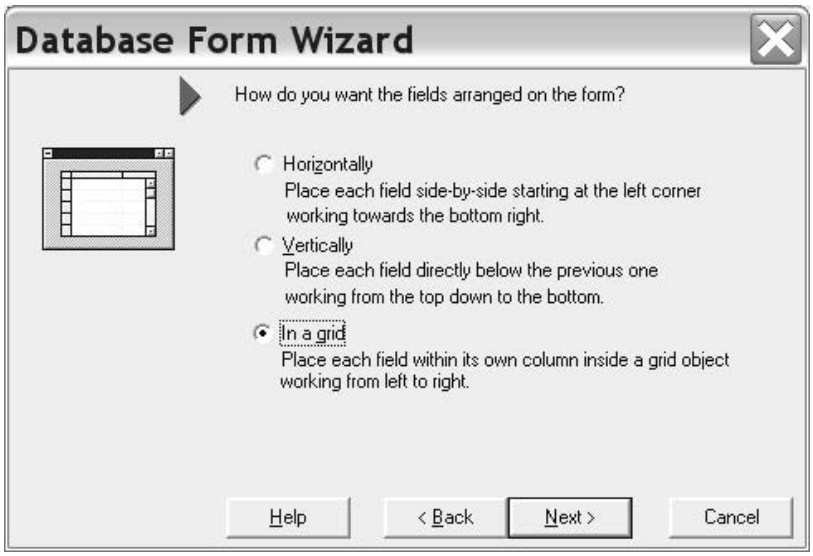


Рис. 11.5 Окно выбора представления БД

Выберите опцию In a grid – отображать в сетке и нажмите Next. В появившемся окне поставьте галочку напротив “Generate a main form” и точку напротив “Form Only”.

(*Если бы галочка напротив “Generate a main form” была сброшена, то в главный файл проекта пришлось бы вставлять оператор Show или ShowModal, чтобы отобразить форму*)

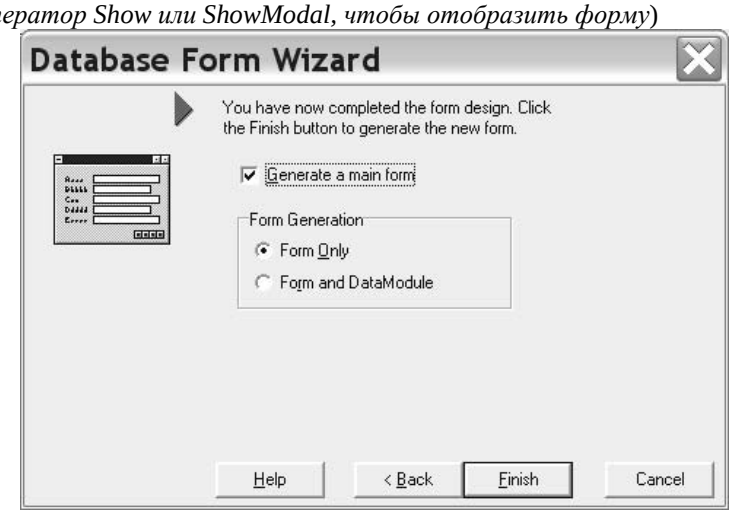


Рис. 11.6 Заключительное диалоговое окно мастера БД

Щелкните на кнопке Finish. Перед вами появится готовая форма, содержащая минимальное количество компонентов (Table1, DataSource1, DBNavigator и DBGrid1) для работы с уже созданной, готовой базой данных.

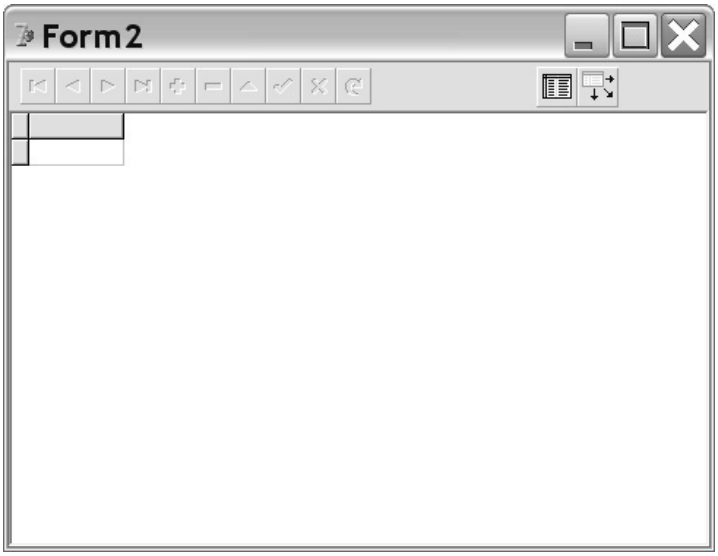


Рис. 11.7 Форма с минимальным набором компонентов для работы с базой данных

Теперь просто запустите программу, нажав F9.

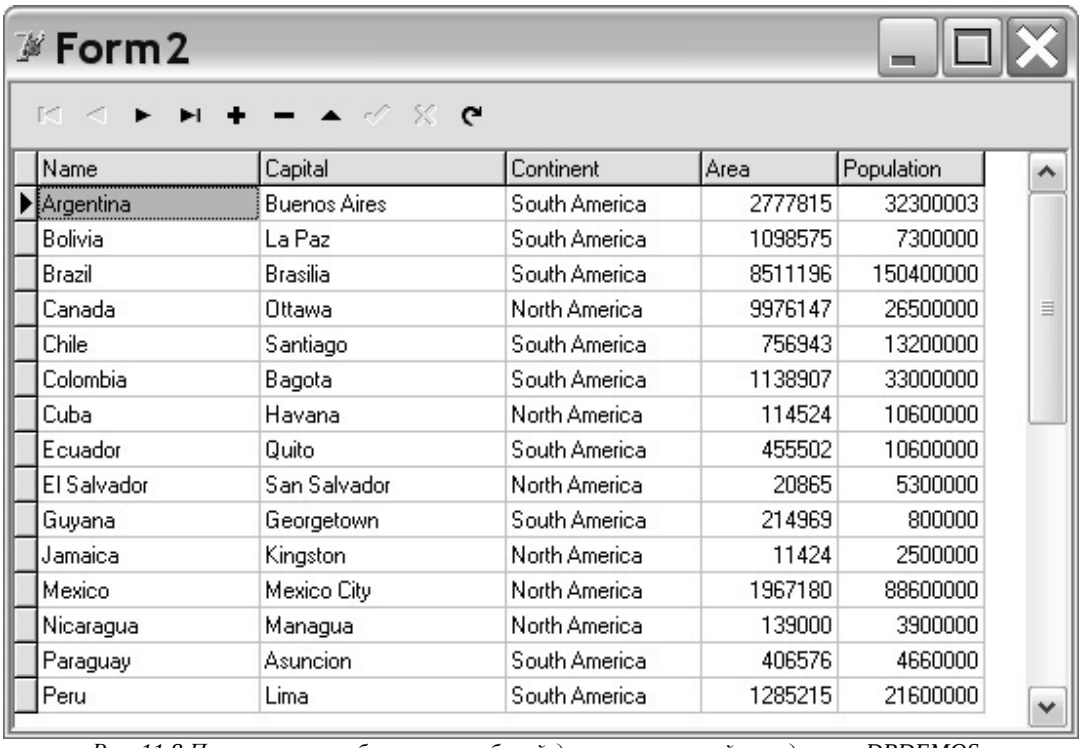


Рис. 11.8 Приложение, работающее с базой данных, имеющей псевдоним «DBDEMOS»

Получилось вполне работоспособное приложение, работающее с Базой Данных без единой строчки кода.

При программировании БД в Delphi активно используется компонент DBNavigator1, который является мощным средством управления БД. Им следует пользоваться осторожно, чтобы случайно не изменить или не испортить данные в БД.

В следующей таблице приводится описание его кнопок.

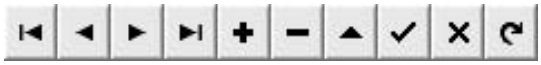


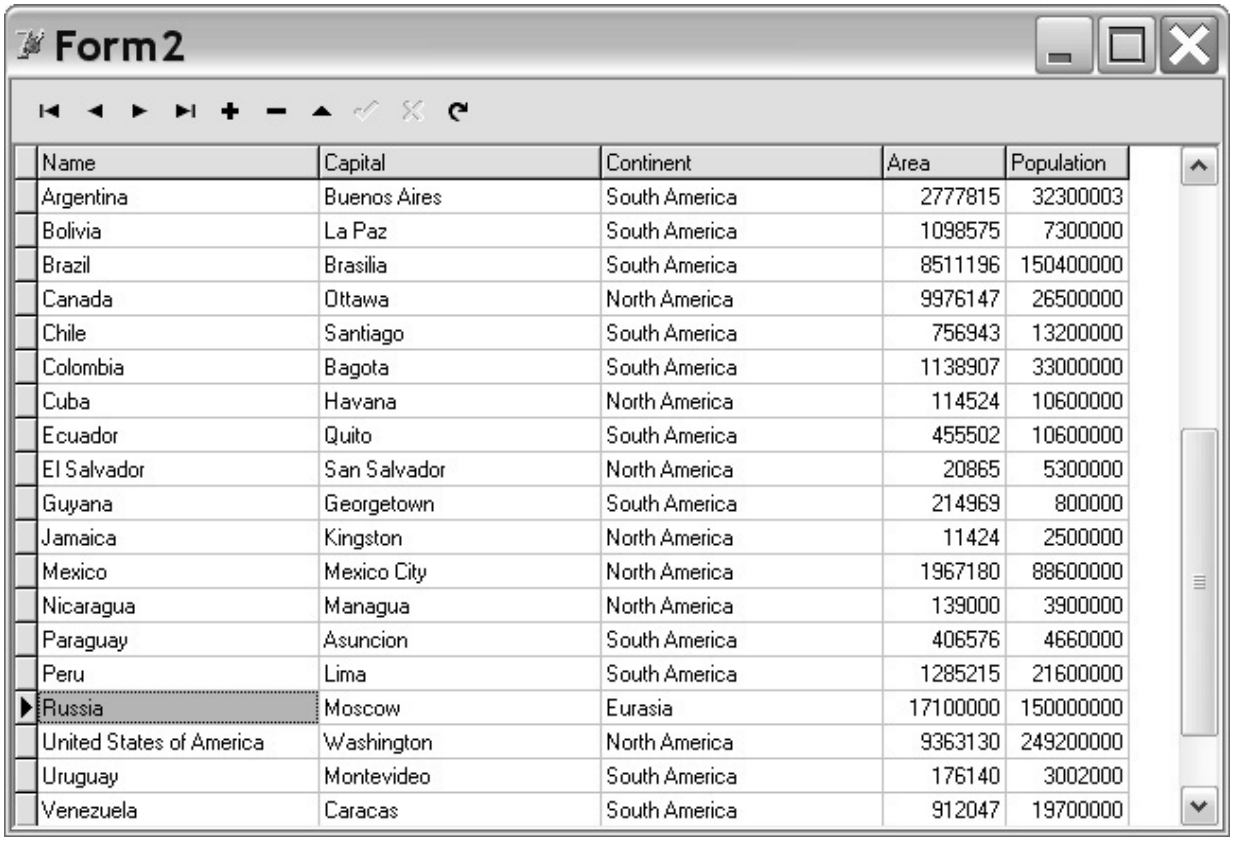
Рис. 11.9 Кнопки компонента DBNavigator

Кнопка	Обозначение	Действие
К первой	nbFirst	Указатель текущей записи перемещается к первой записи файла данных
К предыдущей	nbPrior	Указатель текущей записи перемещается к предыдущей записи файла данных
К следующей	nbNext	Указатель текущей записи перемещается к следующей записи файла данных
К последней	nbLast	Указатель текущей записи перемещается к последней записи файла данных
Добавить	nblnsert	В файл данных добавляется новая запись
Удалить	nbDelete	Удаляется текущая запись файла данных
Редактирование	nbEdit	Устанавливает режим редактирования текущей записи
Сохранить	nbPost	Изменения, внесенные в текущую запись, записываются в файл данных
Отменить	Cancel	Отменяет внесенные в текущую запись изменения
Обновить	nbRefresh	Записывает внесенные изменения в файл

Далее в **режиме работы программы** потренируйтесь перемещаться по таблице, просто нажимая любые первые 4 кнопки компонента DBNavigator. Например, переместитесь в конец таблицы, нажав кнопку с изображением . Затем установите куда-нибудь с левого края в таблице курсор и нажмите на кнопку – это значит, что вы собирайтесь добавить в этом месте таблицы в БД новую запись.

Добавьте новую запись со следующими значениями полей:

Russia, Moscow, Eurasia, 1710000000, 150000000, а затем нажмите кнопку или просто щелкните в другом месте (например, на другой записи), чтобы сохранить введенные данные.



Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300003
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000
Mexico	Mexico City	North America	1967180	88600000
Nicaragua	Managua	North America	139000	3900000
Paraguay	Asuncion	South America	406576	4660000
Peru	Lima	South America	1285215	21600000
Russia	Moscow	Eurasia	17100000	150000000
United States of America	Washington	North America	9363130	249200000
Uruguay	Montevideo	South America	176140	3002000
Venezuela	Caracas	South America	912047	19700000

Рис. 11.10 Добавляем новую запись в базу

Вернитесь в **режим программирования,** а затем снова запустите программ. Вы обнаружите, что ваша БД содержит введенную вами прежде информацию.

Добавьте еще какую-нибудь запись с произвольными значениями полей, например: Страна, Столица, Континент, 100000, 100000. Обратите внимание, что, когда вы редактируйте запись, кнопка (Редактировать) находится уже в нажатом режиме, а чтобы отменить редактирование и вернуть все изменения назад, нужно до нажатия кнопки (Сохранить *Post*) нажать кнопку (Cancel). Чтобы отредактировать запись, вы можете просто дважды щелкнуть на запись, вместо нажатия кнопки (Редактировать *Edit*).

Теперь удалите данную запись, предварительно выделив любое ее поле, а затем нажмите кнопку (удалить). Появится сообщение «Delete Record?», ответьте «Да» – и запись будет удалена из БД. **Вернитесь в режим программирования.**

Для успешного и продуктивного программирования БД нужно четко представлять и понимать следующие основные термины:

- **Таблица (Table).** Это отдельный файл данных, состоящий из строк (записей) и столбцов (полей). Одно из полей таблицы должно содержать первичный ключ, по которому индексируется информация в файле. Таблица также может индексироваться и по вторичным ключам. Таблицу часто называют набором данных dataset
- **Запрос (Query).** Во многом подобен таблице, представляет из себя набор данных SQL в формате с полосой прокрутки и навигатором.
- **База Данных (Database).** Это одна или несколько таблиц. Если одна таблица связана с другой по значению ключа в определенном поле, то в данном случае будет реляционная база данных.
- **Псевдоним (Alias)** имя, зарегистрированное в Borland Database Engine(BDE), под которым скрывается название жесткого диска и путь к файлам БД. Для обращения к файлам БД желательно использовать псевдонимы вместо физического пути и имен файлов. Так как если имеется псевдоним, то файлы БД можно перенести в другой каталог, а все приложения, использующие эти файлы будут работать без изменений.

11.2. Создание Базы Данных с использованием стандартного приложения Database Desktop

На Delphi можно написать приложение, которое будет самостоятельно создавать базу данных. Но разработчики Delphi позаботились, чтобы создание БД было проще и разработали приложение **Database Desktop**, которое поставляется в комплекте с Delphi.

Вы можете запустить Database Desktop:

либо из среды Delphi выбрав: Tools-> Database Desktop,

либо автономно, не используя Delphi:
Пуск->Программы->Borland Delphi7->Database Desktop.

Задание 11.2.1. Создадим Псевдоним для Базы Данных «Сотрудники», используя приложение Database Desktop.

С самого начала определим место, где будут храниться файлы вашей БД. Создайте на жестком диске папку с путем D:\Database\Sotrudink.

Закройте Delphi. Запустите Database Desktop. Выберете Tools -> Alias Manager (рис. 11.11).

Далее для создания нового псевдонима (Алиаса) щелкните New и в поле «Database Alias» напишите Sotrudink.

Затем щелкните кнопку «Keep New».

Теперь нужно поставить в соответствие ваш псевдоним Sotrudink и файлы БД, которые будут находиться в каталоге D:\Database\Sotrudink.

Используя кнопку Browse, настройте Path на каталог D:\Database\Sotrudink. У Вас должно получиться следующее:

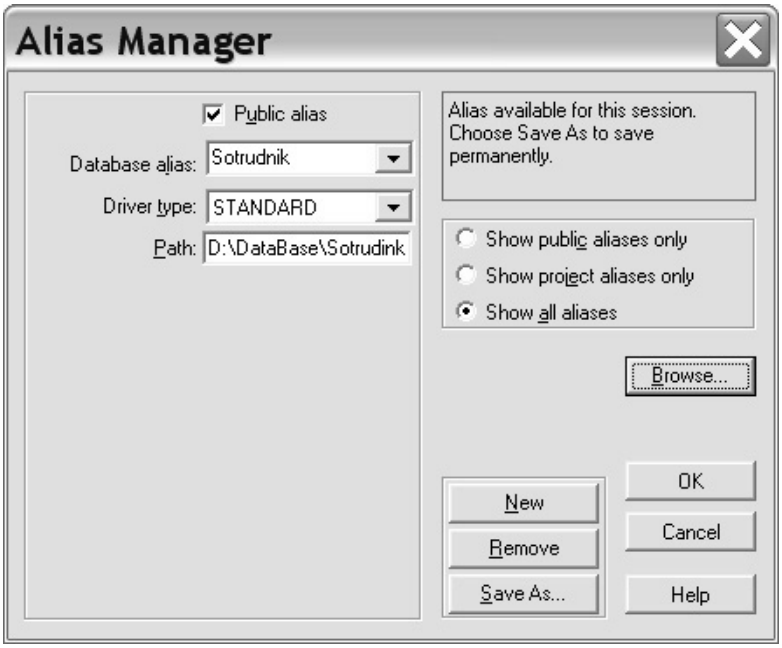


Рис. 11.11 Приложение Database Desktop регистрирует псевдоним Sotrudink

Затем сохраните данные настройки, нажав Save As.

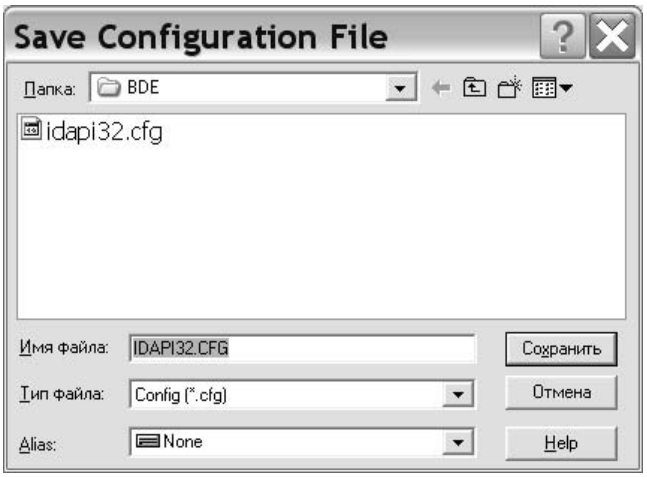


Рис. 11.12 Сохраняем псевдоним в файле конфигурации IDAPI32.CFG

Нажмите сохранить.

Затем на вопрос, сохранить (заменить) ли файл IDAPI32.CFG, ответьте ДА.

Теперь нажмите OK.

Теперь все приложения Delphi под псевдонимом Sotrudink будут понимать имя диска и физический путь к этой базе данных.

Задание 11.2.2 Непосредственно создадим Базу Данных «Сотрудники» на основе уже созданного для нее псевдонима Sotrudink, используя приложение Database Desktop.

Здесь же в Database Desktop создадим файлы БД для отображения их в табличном виде. Для этого выберете File->New->Table. Из раскрывающегося списка форматов БД выберете Paradox 7 и нажмите OK.

Затем определите поля Name, Surname, Age, Growth, Profession и Weight с конкретным типом и размером элемента поля. Также двойным щелчком поставьте звездочку в области Key для поля Name. Таким образом, мы индексируем первое поле Name (делаем его ключевым).

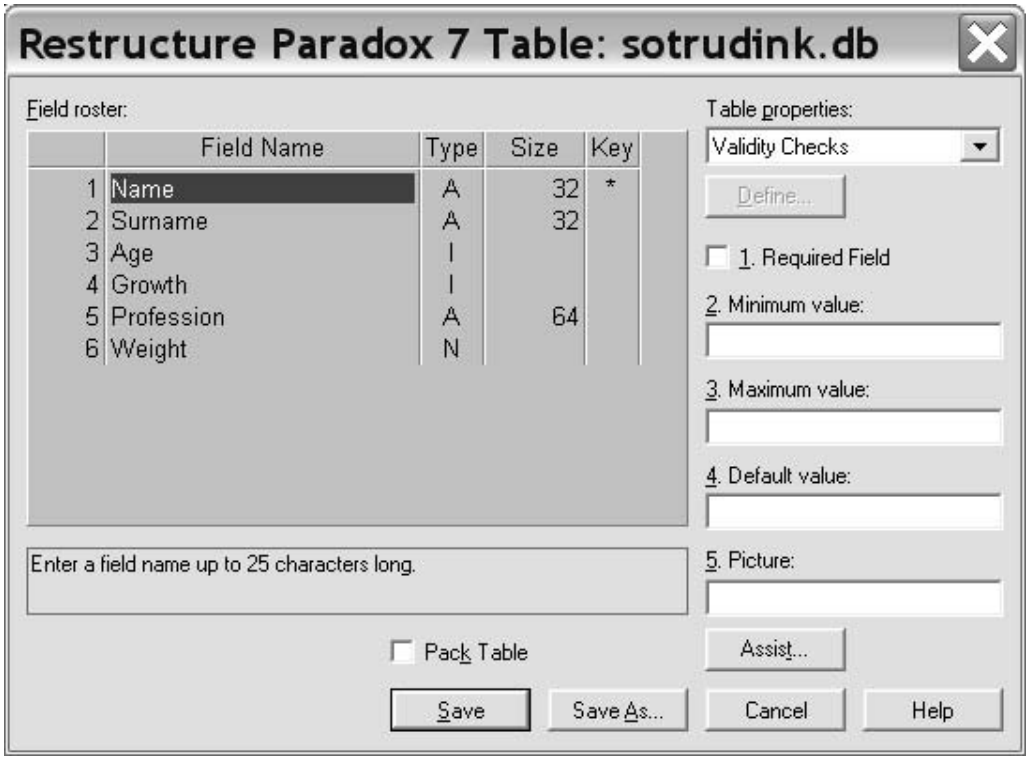


Рис. 11.13 Структура таблицы БД в Database Desktop. Здесь Type: l – означает длинное целое, A – текстовый формат и т. д. Size – означает размер в байтах элемента данного поля. Key – индексирует поле.

Теперь нажмите Save As. Выберите в поле Alias псевдоним Sotrudink, а в поле имени файла введите sotrudnik и щелкните на кнопке «Сохранить».

Рис. 11.14 Сохраняем таблицу с именем sotrudnik базы данных с псевдонимом Sotrudnik

Теперь в папке D:\Database\Sotrudink появился файл БД sotrudink.db. Можете это проверить.

Далее непосредственно будем вносить данные в эту БД.

Выберите File->Open->Table либо просто щелкните на пиктограмме OpenTable.

Затем аналогично предыдущим действиям выберите в поле Alias псевдоним Sotrudink, а в поле имени файла введите sotrudnik и щелкните на кнопке «Открыть».

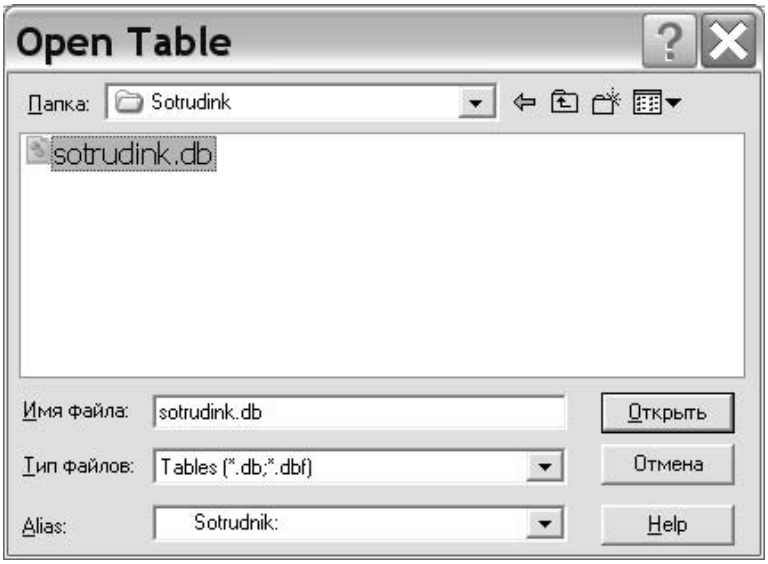


Рис. 11.15 Открываем файл-таблицу БД для ввода туда информации

Перед вами откроется сама БД «Sotrudnik» в виде таблицы с полями, которые вы уже определили ранее (рис. 11.16). Теперь щелкните крайнюю кнопку справа или нажмите F9, таким образом Вы перейдете в режим редактирования БД.

Внесите данные в таблицу, она должна выглядеть примерно так.

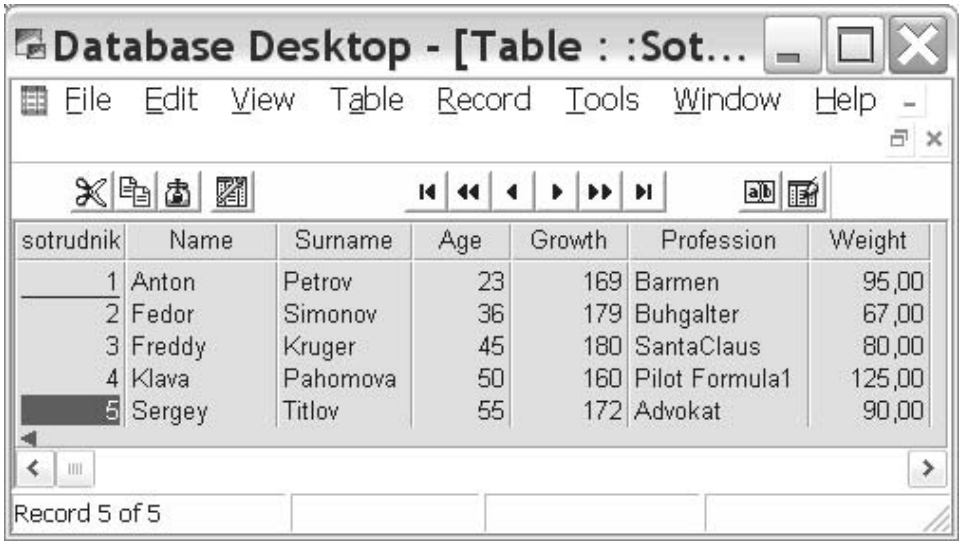


Рис. 11.16 Вносим информацию в файл-таблицу

Вот и все. Теперь База данных готова и ее можно обрабатывать приложением созданным в Delphi.

Закройте Database Desktop.

11.3. **Работа с Базой Данных непосредственно в среде** Delphi

Чтобы создать элементарное приложение в Delphi, работающее с БД, требуется 2 основных компонента: Table и DataSource, а также дополнительные компоненты элементов управления [2]. Ниже приведенная схема наглядно объясняет взаимодействие приложения Delphi и базы данных.

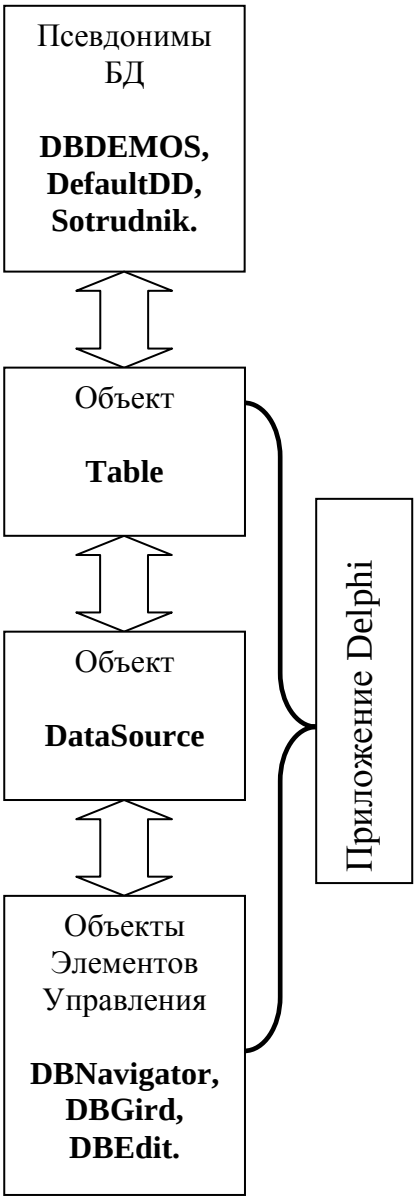
Объект **Table** соединяет ваше приложение Delphi с псевдонимом базы данных.

Объект **DataSource** является посредником между Table и остальными объектами в программе. Он получает данные из Table и передает их объектам, и, наоборот, может передавать данные из объектов в БД также через Table. Взаимодействие Table и DataSource координируется программой BDE, которая производит операции чтения и записи БД в нужном формате.

Объекты элементов управления данными в БД **DBNavigator, DBGird** и т.д. через DataSource могут получать и управлять информацией из БД.

DBGrid – это таблица для отображения базы данных в табличном виде.

Задание 11.3.1 Средствами Delphi откроем созданную Базу Данных «Сотрудники».



Запустите Delphi7. Создайте новый проект.

Первое, что нужно сделать, – это скинуть на форму компонент Table1. В Delphi6 и в более ранних версиях он находился на вкладке Data Access, а в Delphi7 находится на вкладке BDE.

Затем сбросьте на форму DataSource1 с вкладки DataAccess, а также DBNavigator1 и DBGird1 с вкладки DataControls (рис. 11.17).

Теперь настройте объект Table1 на БД через ее псевдоним. Для этого установите его свойство DatabaseName на псевдоним Sotrudnik, а свойство TableName установите на файл sotrudink.DB.

Далее подключим компонент DataSource1 к объекту (таблице) Table1. Установите для DataSource1 его свойство DataSet на Table1.

Осталось последнее: подключить объекты элементов управления к БД. Для этого установите для DBNavigator1 его свойство DataSource в DataSource1, то же самое сделайте для DBGird1. Вот и все, теперь и объекты управления настроены на БД и могут на нее влиять.

Рис. 11.17 Без помощи мастера БД создаем приложение, работающее с БД- Sotrudnik

Теперь, даже не запуская программу, можно отобразить данные из БД в компоненте DBGird1. Для этого у Table1 установите значение свойства Active в True. Ваша таблица DBGird1 заполнится данными из БД.

Name	Surname	Age	Growth	Profession	Weight
Anton	Petrov	23	169	Barmen	95
Fedor	Simonov	36	179	Buhgalter	67
Freddy	Kruger	45	180	SantaClaus	80
Klava	Pahomov	50	160	Pilot Formula	125
Sergey	Titlov	55	172	Advokat	90

Рис. 11.18а. В режиме программирования, не запуская приложение, отображаем данные в DBGird1 установив Table1.Active в True

Для того чтобы Table1 активизировался, можно не только установить Table1.Active в True, но и применить оператор Table1.open. Проверьте это. Для этого у Table1 установите значение свойства Active снова в False. Данные перестанут отображаться, а в обработчике onCreate формы напишите:

Table1.open; {открываем БД и таблицу}

Запустите программу. Данные из БД будут отображаться в таблице. **Вернитесь в режим программирования.**

Затем сбросьте на форму кнопки Button1 с заголовком «Открыть таблицу» и Button2 с заголовком «Закрыть таблицу».

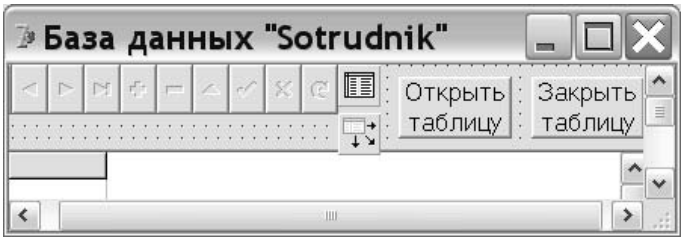


Рис. 11.18б. Временно добавляем 2 кнопки TButton чтобы показать действие Table1.open и Table1.close

В обработчике щелчка на Button1 напишите: Table1.open, а в обработчике щелчка на Button2 напишите: Table1.close.

Запустите программу. Данные из БД сначала будут отображаться в таблице. Щелкните по кнопке «Закрыть таблицу» – и данные исчезнут, затем щелкните по кнопке «Открыть таблицу» – и данные снова появятся. **Вернитесь в режим программирования.**

Мы показали эффект от применения инструкций Table1.open и Table1.close, поэтому можете удалить кнопки «Открыть таблицу» и «Закрыть таблицу»: они больше не нужны.

Снова для Table1 установите Active в True.

Теперь скиньте еще 6 компонентов label на каждом из них напишите имена полей БД и напротив каждого компонента label поместите по одному компоненту класса TDBEdit (элемент управления, палитра Data Controls), позволяющему ввод, вывод и редактирование информации в БД (рис. 11.19).

Причем для каждого компонента класса TDBEdit установите его свойство DataSource в DataSource1, а также свойство DataField на соответствующее поле Name или Surname или Age и т.д.

Рис. 11.19 Добавляем компоненты класса TDBEdit

Запустите программу. Перемещайтесь по БД с помощью навигатора DBNavigator1. Постарайтесь не менять данные. Выбранные Данные будут отображаться также и в компонентах Edit. **Вернитесь в режим программирования.**

Задание 11.3.2 В Базе Данных «Сотрудники» создадим вычисляемое виртуальное поле.

Виртуальное поле можно создать в вашем приложении и отображать его в таблице. Реально оно не будет существовать в самой БД. Виртуальное поле требуется для отображения результатов вычислений проведенных с данными из реальных полей БД.

Щелкните двойным щелчком на объекте Table1. Откроется редактор полей.

Здесь приводится список полей, которые будут доступны Table1.

Рис. 11.20 Список полей доступных компоненту Table1

Можно поменять расположение полей, перетащив их друг относительно друга. Их можно удалить, просто выбрав их и щелкнув Del. Не нужно бояться, что пострадает информация в самой БД, здесь мы просто определяем, какие поля будут доступны нашему приложению.

Если вы удалили несколько полей или все поля, так что в редакторе полей пусто, щелкните правой кнопкой мыши и выберите Add all fields, все поля снова будут доступны (рис. 11.20).

(При данных действиях часто у Table1 устанавливается значение свойства Active в False, поэтому меняйте его на True).

Создадим виртуальное поле с именем Summa. Ради простоты в него просто будем заносить сумму записей полей Age и Weight. Щелкните двойным щелчком на объекте Table1 – и откроется редактор полей, а затем щелкните правой кнопкой и выберите New Field. Запишите в поле ввода «Name» имя виртуального поля Summa, как показано ниже на рисунке, при этом автоматически в поле ввода «Component» будет записано Table1Summa. Затем в поле ввода «Type» установите значение Float.

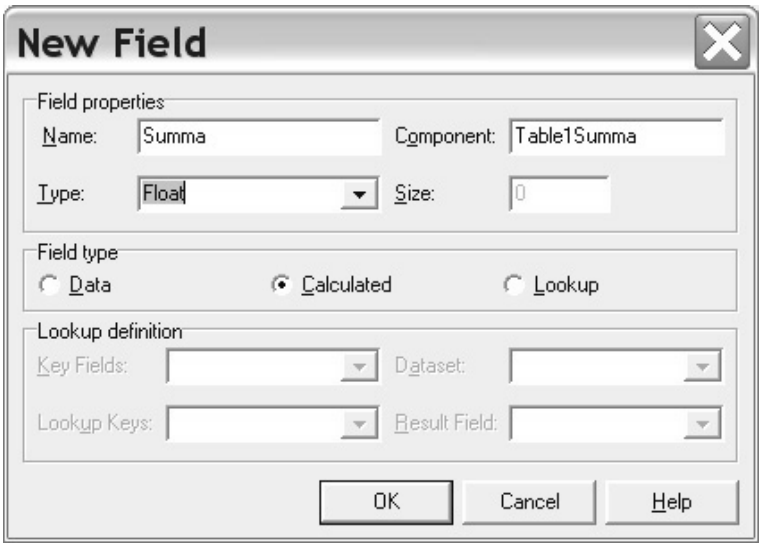


Рис. 11.21 Создаем вычисляемое поле Summa с использованием диалогового окна New Field

Здесь Summa – это имя поля, а компонент, представляющий это поле в приложении, имеет имя Table1Summa. Calculated – означает, что поле вычисляемое. Нажмите OK. Теперь в редакторе полей будет отображаться данное поле.

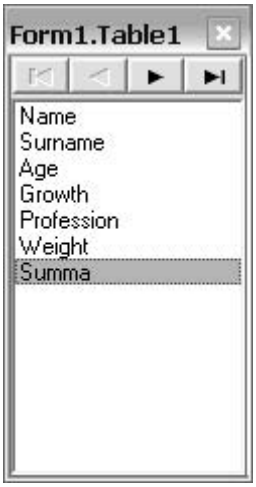


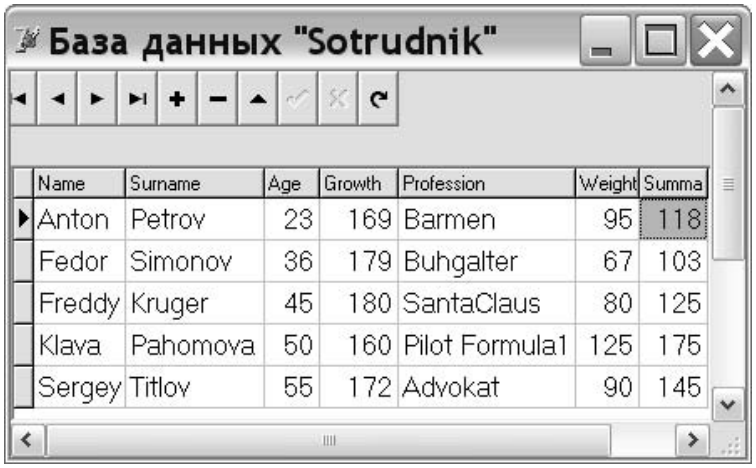
Рис. 11.22. Теперь в редакторе полей отображается вычисляемое поле Summa

Для того чтобы оно также стало отображаться в таблице DBGird, еще раз установите значение свойства Table1.Active в True.

Далее выберите в Object Inspector созданный компонент Table1Summa и создайте у него обработчик события onGetText, который должен выглядеть следующим образом:

```
procedure TForm1.Table1SummaGetText(Sender: TField; var Text: String;
 DisplayText: Boolean);
begin
Text:=FloatToStr(Table1Weight.Value+ Table1Age.Value);
end;
```

Запустите программу. Посмотрите на поле Summa, там должна быть сумма значений полей Weight и Age.



Name	Surname	Age	Growth	Profession	Weight	Summa
Anton	Petrov	23	169	Barmen	95	118
Fedor	Simonov	36	179	Buhgalter	67	103
Freddy	Kruger	45	180	SantaClaus	80	125
Klava	Pahomova	50	160	Pilot Formula1	125	175
Sergey	Titlov	55	172	Advokat	90	145

Рис. 11.23 В поле Summa содержится сумма значений полей Weight и Age

Можете изменять значения в полях Weight и Age, эти изменения сразу же отобразятся в поле Summa. **Вернитесь в режим программирования.**

Событие onGetText наступает при обращении к свойствам DisplayText или Text объекта наследника класса TDataSet, в данном случае объекта Table1. В данном случае в параметрах обработчика: Sender является объектом класса TField, то есть полем, Text является отображаемой строкой текста в таблице, а DisplayText имеет булевский тип и определяет, будет ли строка Text использоваться для отображения или редактирования.

То есть все значения поля компонента Table1Summa отображаются свойством text, но с помощью обработки события onGetText мы просто меняем значение этого свойства по своему усмотрению. В данном случае туда выводим сумму из двух других полей.

Таким образом, когда пользователь вводит в таблицу текст, инициируется данное событие. Можете это проверить.

Здесь инструкция Table1Weight.Value означает, что в свойстве Value, поля Weight таблицы Table1 содержится текущее значение определенного типа.

Данный обработчик Table1SummaGetText позволяет изменить форму отображения значения поля Value, то есть изменить значение свойства Text, обычно соответствующего свойству AsString конкретного поля, представленного компонентом, например Table1Summa.

Далее сбросьте на форму компонент Memo1, а также 7 кнопок класса TButton со следующими заголовками на них:

Кнопка класса TButton	***Значение свойства Caption кнопки TButton***
Button1	**Имена полей БД -> Memo1**
Button2	**Все данные БД -> Memo1**
Button3	**Форматы поля БД -> Memo1**
Button4	**Программно меняем поле Growth в БД**
Button5	**Программно вставим запись в БД**
Button6	**Программно удалим выбранную запись из БД**
Button7	**Найти данные в БД**

Разместите данные кнопки класса TButton и компонент Memo1 на форме примерно так, как показано на рисунке:

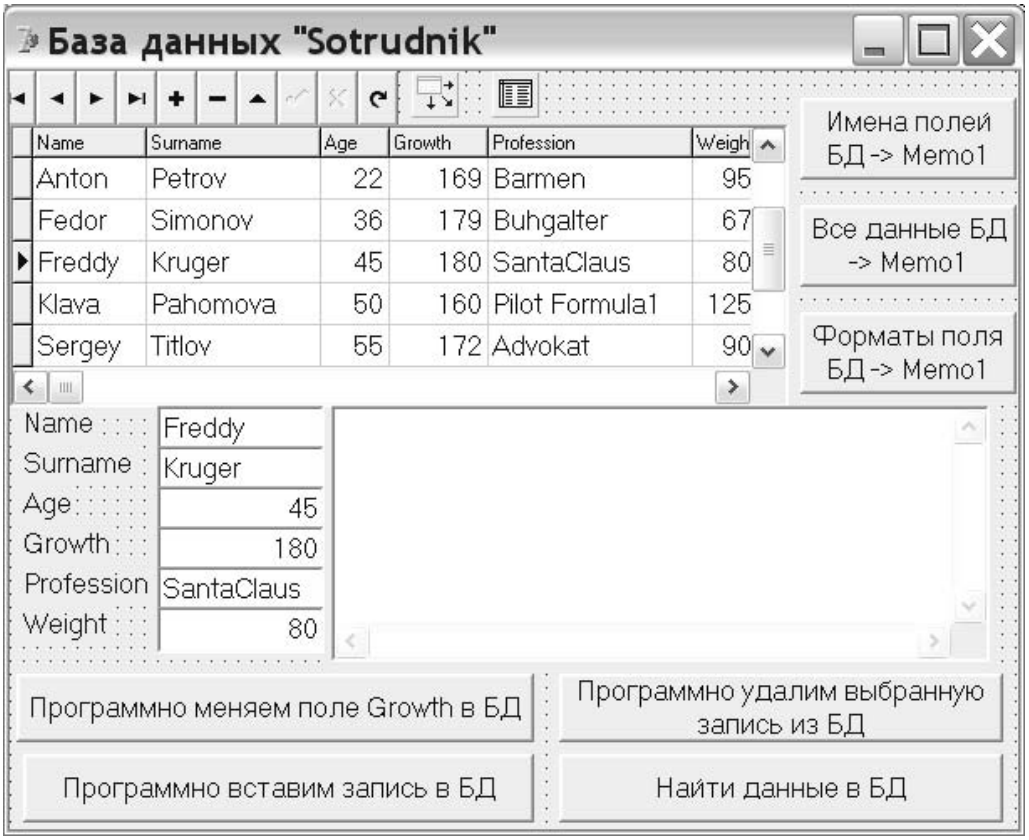


Рис. 11.24. Окно приложения базы данных «Sotrudnik» в окончательном виде

Задание 11.3.3 Программно выведем информацию о всех именах полей БД в Memo1.

Запрограммируем это в обработчике щелчка на кнопке Button1 с заголовком «Имена полей БД -> Memo1». Здесь в заголовке выражение «->» означает направление, то есть мы все имена полей базы данных намерены вывести в Memo1.

При щелчке на Button1 выведем в Memo1 названия всех полей. Обработчик щелчка на Button1 должен быть таким:

```
procedure TForm1.Button1Click(Sender: TObject);
var  i:integer;
begin
Memo1.Clear; {очищаем Memo1}
```

```
for i:=0 to Table1.FieldCount-1 do {цикл по количеству FieldCount всех полей}
Memo1.Lines.Add(Table1.Fields[i].FieldName);
end;
```

Здесь компонент Table1 имеет свойство Fields представленное как
property Fields[Index: Integer]; Index означает номер поля.

Свойство FieldCount возвращает количество полей в текущей структуре записи. FieldName – в данном случае возвращает имя поля по заданному индексу поля. Первое поле индексируется 0, второе 1 , третье 2 и т.д. Запустите программу. Щелкните Button1. Программа должна выводить имена полей.

Рис. 11.25. Результат щелчка на кнопке «Имена полей БД -> Memo1»

Вернитесь в режим программирования.

Задание 11.3.4 Программно выведем всю информацию из БД в Memo1.

Запрограммируем это в обработчике щелчка на кнопке Button2 с заголовком «Все данные БД -> Memo1». Обработчик щелчка на ней должен выглядеть следующим образом:

```
procedure TForm1.Button2Click(Sender: TObject);
var i,j:integer; s:string;
begin
memo1.Clear; {очищаем Memo1}
Table1.First; {Переходим на первую запись (строку) в Table1}
with Table1 do {Для объекта Table1}
for i:=0 to RecordCount-1 do begin s:=''; {Цикл по количеству записей то есть строк}
for j:=0 to FieldCount-1 do {Цикл по количеству полей то есть столбцов}
s:=s+Fields[j].AsString+'-'; {Вывод значения элемента записи в строковом формате и наращивание строки}
next; {переход на следующую запись от текущей}
Memo1.Lines.Add(s);
end;
end;
```

Запустите программу. Щелкните Button1. Программа должна выводить информацию из БД в Memo1, то есть результат будет таким:

Рис. 11.26. Результат щелчка на кнопке «Все данные БД -> Memo1»

Вернитесь в режим программирования.

Здесь RecordCount – является свойством Table1 и содержит количество записей или строк в таблице.

Для перемещения по таблице имеются следующие методы:
Table1.**First**- переход на первую запись
Table1.**Last**- переход на последнюю запись
Table1.**Prior**-переход на предыдущую запись
Table1.**Next** – переход на следующую запись
procedure **MoveBy(N: Integer)**;

Процедура **MoveBy перемещает указатель в объекте Table1 на N записей вперед или назад в таблице. Нет никакого функционального различия между вызовом метода Table1.Next и вызовом Table1.MoveBy(1).**

Вызов Table1.Prior имеет тот же результат, что и вызов Table1.MoveBy(-1). Например: перемещение на 3 записи вперед – MoveBy(3), а перемещение на 3 записи назад: MoveBy(-3).

property **BOF**: Boolean read FBOF; – возвращает True, если указатель в начале Table1
property **EOF**: Boolean read FEOF; – возвращает True, если указатель в конце Table1

Необходимо отметить и свойство **FieldDefs** (класса TFieldDefs) – для существующей таблицы содержит информацию обо всех полях таблицы. Число полей определяется свойством **Count**, а доступ к элементам массива осуществляется через свойство Items:
property **Items[Index: Integer]**: TFieldDef;

Задание 11.3.5 Разными операторами выведем информацию о двух полях БД в Memo1.

Для доступу к полю мы можем использовать его имя и тогда значение записи в данном поле будем получать, используя метод, требующий в качестве параметра имя поля.

Например, оператор **Table1.FieldByName('Age').Value** содержит значение записи поля Age. Относительно функции **FieldByName** поле Value имеет универсальный тип Variant и может содержать данные разных стандартных типов Object Pascal.

У нас в таблице поле Age является вторым по счету, следовательно, его индекс 1; нумерация идет от нуля, поэтому оператор, совершенно функционально-эквивалентный предыдущему, будет иметь вид – **Table1.Fields[0].Value.**

Здесь Value содержит значение, имеющее конкретный тип – целый, вещественный, текстовый и т.д. В данном случае Value для поля Age имеет целый тип, поэтому можно спокойно присвоить некоторой переменной целого типа **I:integer** одно из значений:
I:= **Table1.FieldByName('Age').Value или** I:= **Table1.Fields[0].Value.**

При этом Delphi также легко преобразовывает формат вывода, поэтому, если нужно значение целого типа в поле Age вывести в переменную строкового типа **S:string** или в Memo1, следует проделать следующее: **S:=Table1.FieldByName('Age').AsString**

Преобразования форматов вывода в классе TField следующие:
AsBoolean, AsFloat, AsInteger, AsString, AsDateTime

Далее в этом же проекте выведем информацию в Memo1 о имени первого поля БД «Name», используя свойство FieldName, а также о его текущем значении, используя свойство AsString.

Наряду с этим, выведем информацию о поле «Age», используя сначала свойство строкового типа AsString, а затем – свойство Value универсального типа Variant.

Запрограммируем это в обработчике щелчка на кнопке Button3 с заголовком «Форматы поля БД -> Memo1».

Обработчик щелчка кнопки Button3 должен выглядеть так:

```
procedure TForm1.Button3Click(Sender: TObject);
var s:string;
begin
memo1.Clear;
s:=Table1.Fields[0].AsString;
Memo1.Lines.Add('Значение поля '+
Table1.Fields[0].FieldName+' в формате AsString: '+s);

s:=Table1.FieldByName('Age').AsString;
```

```
Memo1.Lines.Add('Значение поля '+
Table1.FieldByName('Age').FieldName+' в формате AsString: '+s);

Memo1.Lines.Add('Значение поля Age в формате Value:');
Memo1.Lines.Add(Table1.FieldByName('Age').Value);
end;
```

Запустите программу. Щелкните Button3. Вы получите следующий результат:

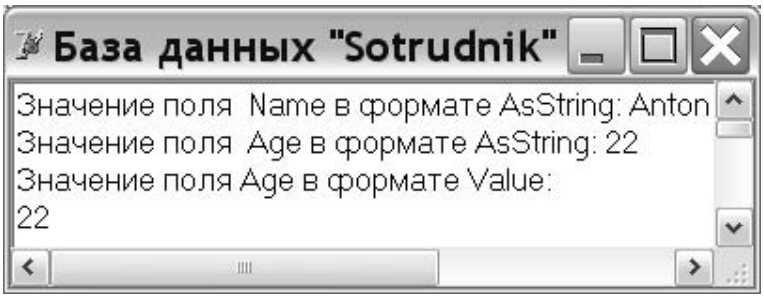


Рис. 11.27. Результат щелчка на кнопке «Форматы поля БД -> Memo1»

Еще раз проанализируйте код. **Вернитесь в режим программирования.**

Задание 11.3.6 Программно поменяем значения записей поля Growth(рост) в таблице Table1.

Чтобы программно поменять любую текущую запись, нужно сначала перевести объект Table1 в режим редактирования специально предназначенным для этого методом Table1.Edit.

Просто поменяем в цикле все значения поля Growth (рост) на 160, 161, 162, 163, и 164.

Запрограммируем это в обработчике щелчка на кнопке Button4 с заголовком «Программно меняем поле Growth в БД».

Обработчик щелчка кнопки Button4 должен выглядеть следующим образом:

```
procedure TForm1.Button4Click(Sender: TObject);
var i:integer;
begin
Table1.First; {переход на начало записи (первая строка)}
for i:=0 to Table1.RecordCount-1 do begin {Делаем цикл по записям}
Table1.Edit; {Переводим Table1 в режим редактирования}
Table1.FieldByName('Growth').value := 160+i;{Меняем записи поля Growth}
Table1.Post; {сохраняем данные на диск}
Table1.Next; {переход на следующую запись}
end;
end;
```

Мы всегда после изменения данных, **чтобы сохранить данные в БД, должны вызывать метод Table1.Post. Для того чтобы перейти на следующую запись в базе данных, требуется вызвать метод Table1.Next.**

Запустите программу. Щелкните Button4. Значения поля «Growth» в базе данных изменятся.

Рис. 11.28. Результат щелчка на кнопке «Программно меняем поле Growth в БД»

Еще раз проанализируйте код. **Вернитесь в режим программирования.**
Восстановите старые данные в поле «Growth» самостоятельно вручную.

Задание 11.3.7 Программно вставим еще одну запись в таблицу Table1 с помощью метода Insert

Это можно сделать с помощью метода Insert или Append компонента Table1.
Продемонстрируем это в обработчике щелчка на кнопке Button5 с заголовком «Программно вставим запись в БД».

Обработчик щелчка на кнопке Button5 должен выглядеть следующим образом:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
Table1.Insert;{переводим Table1 в режим вставки}
Table1.FieldByName('Name').AsString := 'STEPAN';{В поле Name пишем имя STEPAN}
Table1.FieldByName('Age').Value := 33; { Указываем возраст 33}
Table1.Post; {Сохраняем данные на диск}
end;
```

Запустите программу. Щелкните на Button5, но только один раз, иначе возникнет ошибка. Программа должна работать правильно.

Рис. 11.29. Результат щелчка на кнопке «Программно вставим запись в БД»

Еще раз проанализируйте код. **Вернитесь в режим программирования.**
Также существуют аналогичная процедура Table1.Cancel(Отмена)

Задание 11.3.8 Программно удалим эту запись из базы данных

Запрограммируем это в обработчике щелчка на кнопке Button6 с заголовком «Программно удалим выбранную запись из БД».

В обработчике щелчка на ней напишите всего одну строчку:

Table1.Delete; {удалить запись}

Запустите программу. Мышкой в таблице выберите созданную вами запись STEPAN, а затем щелкните только 1 раз на кнопке Button6 . Запись должна удалиться. Больше не щелкайте, чтобы не удалить остальные записи из базы. **Вернитесь в режим программирования.**

Задание 11.3.9 Программно определим, находится ли конкретная запись в таблице Table1 для конкретного поля?

Фактически требуется найти некоторую нужную информацию в базе данных. Программно определим, имеется ли сотрудник по фамилии Pahomova в поле Surname.

Будем использовать **function TBDEDataSet.Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions): Boolean.**
Здесь **KeyFields – имя поля, KeyValues – значение поля, Options – опции.**

TLocateOption = (loCaseInsensitive, loPartialKey); – класс опций

Запрограммируем данное действие в обработчике щелчка на кнопке Button7 с заголовком «Найти данные в БД».

В обработчике щелчка на ней напишите:

```
if Table1.Locate('Surname','Pahomova',[loCaseInsensitive])
then showmessage('Pahomova найдена!!!') else
showmessage('Pahomova ненайдена')
```

Здесь Table1.Locate('Surname','Pahomova',[loCaseInsensitive]) возвращает True, если Pahomova присутствует в поле Surname, а иначе возвращает False.

Запустите программу. Щелкните на кнопке Button7. Если все было в порядке. вы получите следующий результат:

Рис. 11.30. Результат щелчка на кнопке «Найти данные в БД»

Вернитесь в режим программирования.

11.4. Самостоятельные задания по работе с базами данных в Delphi

11.4.1 Самостоятельное задание. Программным путем выведите в Memo1 самого толстого и самого старого человека из базы данных «Sotrudnik».

В вашей базе данных поле Weight – означает вес, а поле Age – возраст.

11.4.2 Самостоятельное задание. В Delphi с помощью мастера БД откройте учебную БД с псевдонимом DBDEMOS и выберите файл с именем animals.dbf, создайте отображение БД в виде сетки (Gird) Нужно отображать рисунки в компоненте DBImage1 из данной БД.

Указание. Поместите компонент DBImage1 на форму, настройте DBImage1 на базу данных, используя свойство DataSource, а также настройте DBImage1 на отображение рисунков животных из данной базы данных используя его свойство DataField.

ПРАКТИЧЕСКАЯ РАБОТА № 12

12.0. Библиотеки

В современном программировании понятие «библиотека» играет большую роль. Большинство серьезных программ создается с их использованием. В операционной системе Windows библиотеки применяются очень широко.

В математической энциклопедии [7] дается следующее определение библиотеки в информатике: **библиотека – это организованная совокупность программ или вспомогательных алгоритмов**. Обычно библиотека алгоритмов или программ хранится во внешней памяти ЭВМ, в рамках той или иной файловой системы, обеспечивающей автоматизированный доступ к отдельным алгоритмам и программам [7].

Можно сказать, что **библиотека представляет собой файл (расширение .dll) с машинным кодом, который исполняет процессор, содержащий процедуры, функции** и ресурсы (например, рисунки и т.д.). Может возникнуть справедливый вопрос: зачем нужны библиотеки, если все функции и процедуры можно хранить в модуле?

Ответ можно дать на примере. Пусть у вас два разных проекта. Каждый из них использует некоторые одинаковые процедуры и функции. И если вы одновременно загрузите исполняемые exe-файлы этих двух проектов, в память ЭВМ будут дважды загружены одни и те же процедуры и функции. Таким образом, оперативная память будет использоваться неэкономично. К тому же и на жестком диске будут два разных модуля, имеющих внутри себя одинаковые процедуры и функции. Потому и память жесткого диска будет неэффективно тратиться.

А если загружено 5, 6 или более подобных проектов, то затраты памяти могут быть существенными.

На помощь приходит библиотека DLL-Dynamic Link Library. Она, как правило, один раз загружается в оперативную память и все процедуры и функции, которые она содержит в единственном экземпляре, могут использоваться несколькими различными приложениями. Таким образом, одну и ту же функцию, без ее тиражирования в памяти, могут использовать 2, 3, 4 и сколько угодно приложений. Это снимает проблему с расходами на оперативную память. То же самое и с жестким диском, так как все процедуры и функции хранятся в одном файле библиотеки, а не дублируются в модулях.

Кроме того, **библиотеки созданные на одном языке программирования можно использовать при написании программ на других языках программирования.** В этом смысле библиотеки универсальны.

12.1. Создание библиотеки с использованием Dll Wizard

Задание 12.1.1. Создадим библиотеку в виде файла MainDll.dll, опишем в ней 3 функции: суммы двух чисел, произведения двух чисел а также функцию, возвращающую строку направлением задом-наперед.

Библиотека в Delphi создается очень просто – путем компиляции уже привычного вам, файла-проекта с расширением *.dpr. Зайдите в «File» → «New» → «Other» и дважды щелкните по значку с надписью Dll Wizard (рис. 12.1).

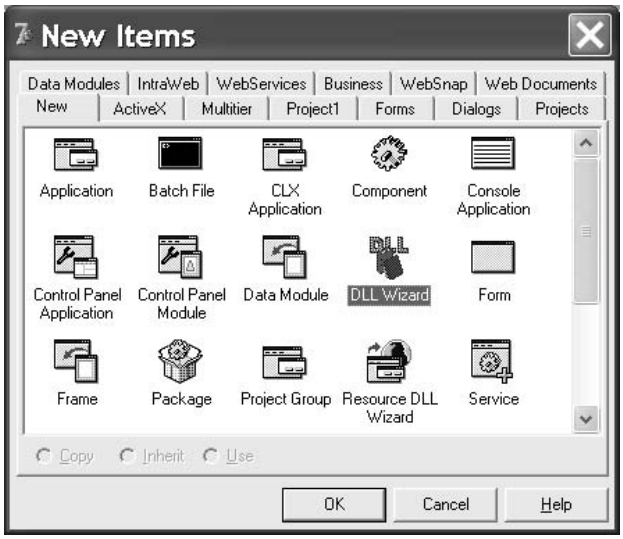


Рис. 12.1 Выберите мастера создания библиотек DLL Wizard

Перед вами в окне редактора кода появится заготовка файла-библиотечного проекта (рис. 12.2):

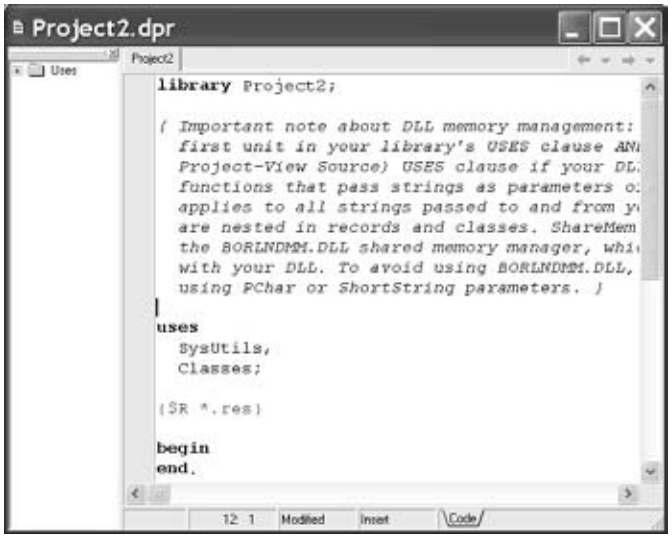


Рис. 12.2 Заготовка библиотечного проекта

Сохранитесь в конкретной папке проекта, заменив имя файла проекта с Project2.dpr на MainDll.dpr. Таким образом, у вас появится такое же окно библиотечного проекта, как и ранее, только после ключевого слова library и в заголовке окна будет написано MainDll.

Комментарии здесь говорят о том, что если процедуры или функции, которые будут экспортироваться во внешние приложения, будут получать или возвращать переменные строкового типа String, то в самом библиотечном файле-проекте, а также в файле проекта приложения, которое будет использовать данную библиотеку, требуется первым подключать модуль ShareMem.

Одна из наших функций будет работать со строками типа String, поэтому в секции **uses** первым из подключаемых модулей напишите **ShareMem**.

Удалите комментарии, так как в данном случае они лишние.

По умолчанию подключено два модуля SysUtils и Classes.

SysUtils – модуль, в котором обрабатываются исключения. Исключение, или исключительная ситуация, – это необычная или неожиданная ситуация (ошибка), прерывающая нормальное выполнение программы. Например, деление на ноль. Если этот модуль будет отключен, то исключение вызовет аварийную остановку программы и выгрузит задачу из памяти. Если он присутствует, то данное исключение будет передано для обработки в вызывающую программу для дальнейшей обработки. Желательно, чтобы он присутствовал.

После секции uses начинается блок **begin…end**.

Все, что находится внутри этого блока, будет выполнено в первую очередь и только один раз, при загрузке библиотеки в оперативную память. Если этого делать не требуется, то можно ничего не писать внутри блока или можно просто удалить одно слово begin, оставив end.

Теперь будем непосредственно описывать функции библиотеки.

Функции и процедуры библиотеки описываются между директивой {$R *.res} и ключевым словом exports.

Строчкой ниже директивы {$R *.res} напишите следующее:

```
function SummaX_Y(X,Y:real):real; stdcall; {функция возвращающая сумму двух чисел X и Y}
begin
result:=X+Y;
end;
```

Далее ниже еще напишите:

```
function ProizvX_Y(X,Y:real):real; stdcall; {функция, возвращающая произведение двух чисел X и Y}
begin
result:=X*Y;
end;
```

А также напишите снова:

```
function Revers(S:string):string; stdcall;{функция, возвращающая строку S задом-наперед}
var i,L:byte; S_:string;
begin
L:=length(S);{ length(s:string)-стандартная функция, возвращающая длину строки s}
S_:='';
for i:=1 to L do S_:=S_+S[L-i+1];
result:=S_;
end;
```

Здесь мы закончили описание функций и нужно указать компилятору, какие функции могут быть доступны внешнему приложению. Это делается с помощью ключевого слова **exports**.

Ключевое слово exports находится сразу после всех описанных процедур и функций или перед началом блока begin end.

Ключевое слово stdcall желательно указывать всегда. Оно определяет стандарт передачи параметров функции справа – налево. Данный стандарт принят многими языками программирования.

Сразу после описания последней функции Revers и перед блоком begin end укажем, какие функции будут доступны (экспортироваться) внешнему приложению.

Напишите там следующее (рис. 12.3):

```
exports
 SummaX_Y,
 ProizvX_Y,
 Revers;
```

Таким образом, приложению будут доступны функции : SummaX_Y, ProizvX_Y и Revers, которые оно будет использовать.

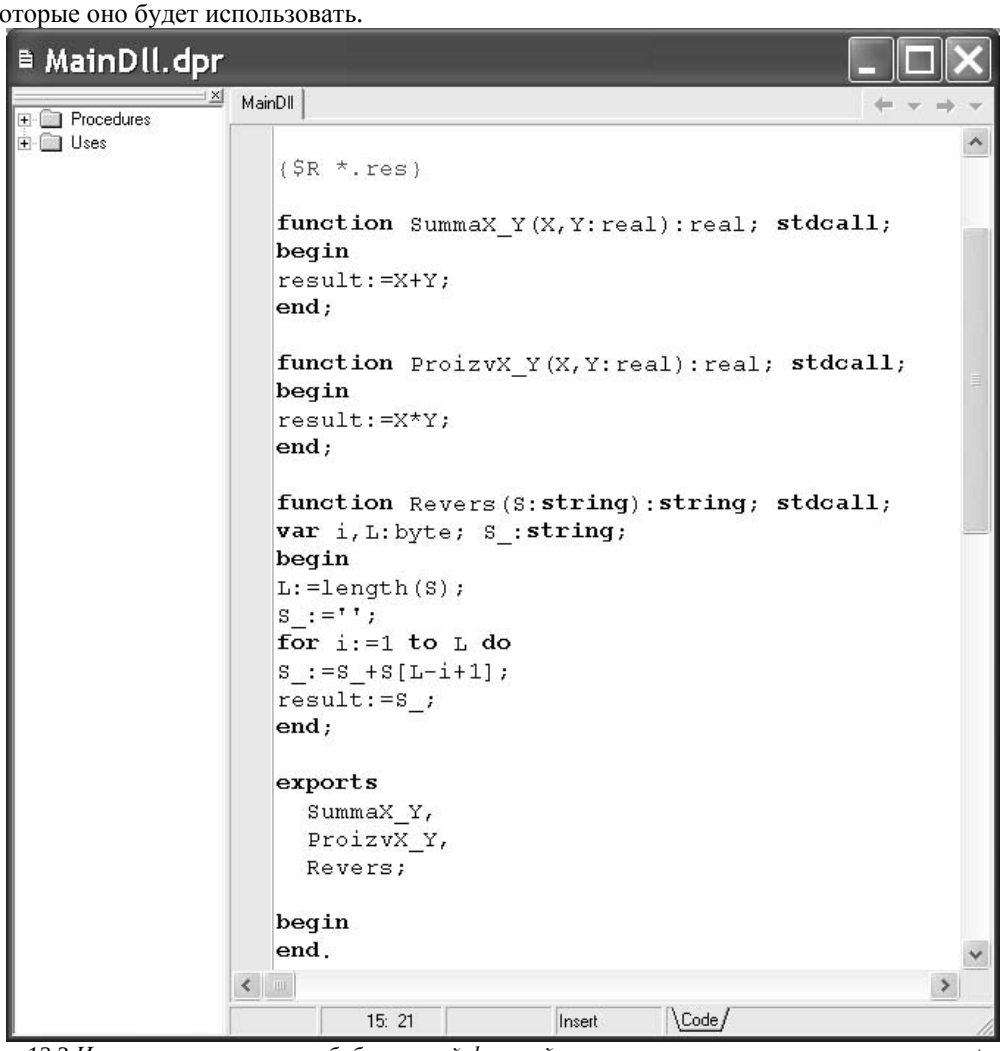


Рис. 12.3 Имена экспортируемых библиотекой функций записываем после ключевого слова exports

Заметьте, что в секции exports указываются только имена функций, без параметров.

Осталось последнее. Требуется, **чтобы набранный вами код скомпилировался в исполняемый файл библиотеки с расширением *.dll**. Это очень легко сделать, просто **нажмите «Ctrl+F9».**

Вот и все. Библиотека готова. Посмотрите в папку вашего проекта, и вы обнаружите там файл MainDll.dll.

Теперь закройте данный проект File -> Close All, так как библиотека создана.

Задание 12.1.2 Создадим внешнее приложение Project1.exe, которое использовало бы функции созданной библиотеки MainDll.dll.

Создайте новый проект и сохранитесь (имя проекта по умолчанию project1.dpr, имя модуля тоже unit1.pas) в той же папке, где и файл созданной библиотеки MainDll.dll.

Первое, что нужно сделать, так это подключить библиотеку к вашему проекту. Поэтому в окне редактора кода, в модуле, сразу после директивы {$R *.dfm} нужно прописать все функции и процедуры, экспортируемые из библиотеки. Причем, нужно прописывать функции и процедуры полностью, указывая их параметры и тип возвращаемого значения, как и при создании библиотеки.

Строчкой ниже директивы {$R *.dfm}, в окне редактора кода напишите следующее:

```
function SummaX_Y(X,Y:real):real; stdcall; external 'MainDll.dll';
function ProizvX_Y(X,Y:real):real; stdcall; external 'MainDll.dll';
function Revers(S:string):string; stdcall; external 'MainDll.dll';
```

Обратите внимание на спецификатор **external** – именно он определяет, что конкретная функция будет импортирована в ваше приложение из конкретного файла.

Здесь после external предполагается полный путь к файлу библиотеки, но если дано только одно имя файла (как в нашем случае), то предполагается, что файл библиотеки находится в той же папке, где и приложение, которое его использует.

Так как ваше приложение использует функции библиотеки для передачи строковых переменных, то в головном файле приложения *.dpr вы должны первым написать модуль ShareMem. Сделайте это. Должно быть так, как на рисунке:

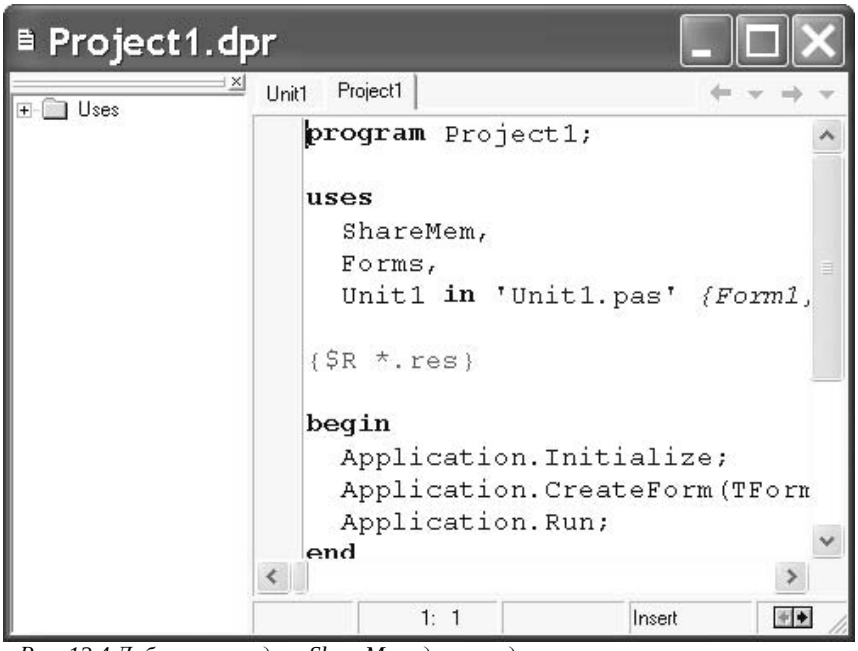


Рис. 12.4 Добавляем модуль ShareMem для передачи строковых параметров

Далее скиньте на форму Button1, RichEdit1, а также 3 компонента label, 3 компонента Edit. Разместите их следующим образом (рис. 12.5):

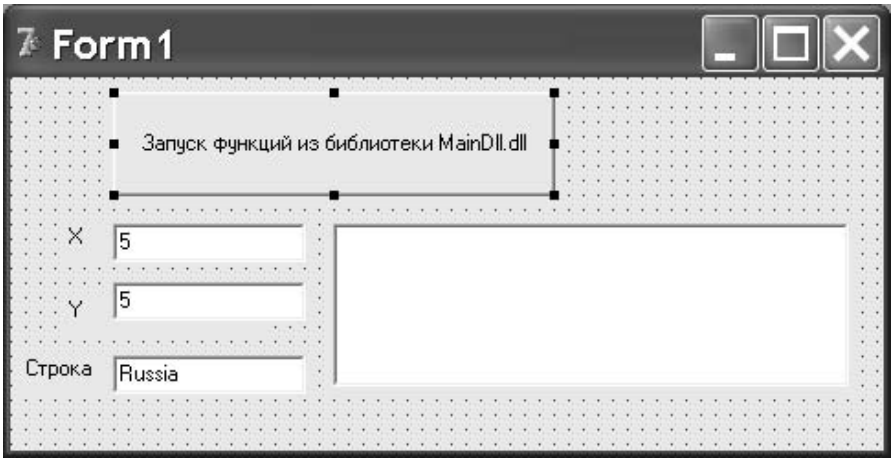


Рис. 12.5 Приложение использующее библиотеку MainDll.dll

Создайте обработчик щелчка кнопки Button1. Он должен выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var X,Y:real; S:string;
begin

RichEdit1.Clear;
X:=StrToFloat(Edit1.Text);//определяем X
Y:=StrToFloat(Edit2.Text); //определяем Y
S:=Edit3.Text; //определяем строку S

ShowMessage('Библиотека MainDll.dll загружена, можете использовать'+#10#13+'нужные вам библиотечные функции');
MessageBeep(1000);

{Вызываем библиотечную функцию SummaX_Y и результат выводим в RichEdit1}
RichEdit1.Lines.Add('Сумма X и Y равна '+ FloatToStr(SummaX_Y(X,Y) ) );
RichEdit1.Lines.Add('');

{Вызываем библиотечную функцию ProizvX_Y и результат выводим в RichEdit1}
RichEdit1.Lines.Add('Произведение X и Y равно '+FloatToStr(ProizvX_Y(X,Y)) );
RichEdit1.Lines.Add('');

{Вызываем библиотечную функцию Revers и результат выводим в RichEdit1}
RichEdit1.Lines.Add('Строка перевернута – > '+Revers(S)) ;
end;
```

Запустите программу. Введите значения для X и Y, а также строку в соответствующие поля и щелкните по Button1. Программа должна работать правильно и библиотечные функции должны сработать верно, выдав результат в компонент Richedit1.

Вернитесь в режим программирования.

Не удаляйте папку с данным проектом: он понадобится в дальнейшем.

Таким образом, мы научились создавать библиотеку, а затем подключать ее из внешнего приложения. Вообще-то, в данном контексте смысл понятия 'внешнее приложение' не вполне уместен, так как библиотека и предназначена для внешних приложений, ее использующих, но, чтобы подчеркнуть сам факт внешности приложения по отношению к библиотеке, автор и далее будет его использовать данное понятие. Это делается в учебных целях.

12.2. Библиотека хранящая объекты

Нужно отметить, что библиотека – это довольно универсальная вещь, в которой можно хранить большой объем информации, объекты [2], в частности, саму форму.

Задание 12.2.1 Создадим библиотеку MyForm.dll, которая, в свою очередь, будет создавать форму, заключающую в себе кнопку SpeedButton1, пульсирующую при ее нажатии.

Как и в первом задании, с помощью мастера Dll Wizard создайте новый проект файла библиотеки. В новой папке. Сохраните проект под именем MyForm.dpr.

Таким образом, и имя вашей библиотеки в окне редактора кода после слова library изменится на MyForm.

Строчные функции мы использовать не будем, поэтому не подключайте модуль ShareMem в проект библиотеки.

Так как одна из функций библиотеки будет создавать форму класса TForm1, то нужно подключить к библиотеке модуль, содержащий описание класса формы, добавив в секции **uses** имя модуля. описывающего форму. Проще это сделать следующим образом.

Просто щелкните на пиктограмме New Form или выберите последовательно File -> New -> Form, а затем, когда появится форма, сохраните ее модуль под именем UnitForm.pas в той же папке, где и ваш проект.

Обратите внимание, что в файле проекта появилась запись

UnitForm in 'UnitForm.pas' {Form1}; говорящая о том, что файл модуля формы подключен к библиотеке.

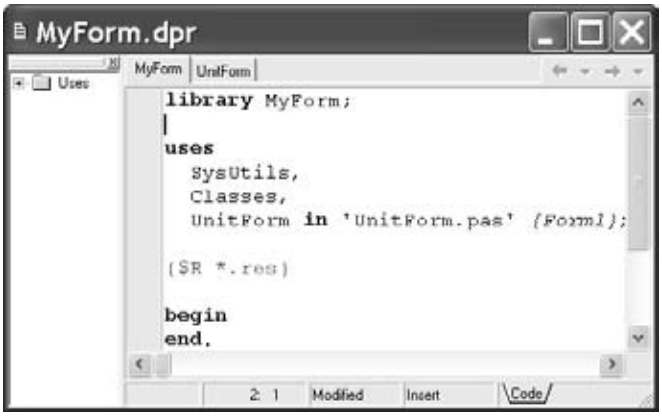


Рис. 12.6 В файле проекта библиотеки подключаем модуль UnitForm содержащий описание формы

Далее переключитесь на форму, описываемую UnitForm.pas и скиньте на нее кнопку SpeedButton1 со значениями свойств Width = Height = 100 и Caption = ‘PULS’, а также Timer1 (рис. 12.7)

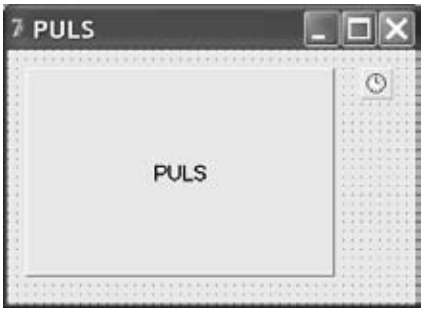


Рис. 12.7 Форма соответствующая модулю UnitForm.pas

С помощью Object Inspector установите свойство Interval компонента Timer1 равным 100, а свойство enabled = False.

В модуле формы в секции public класса TForm1 определите переменные

dx,dy:integer;

Затем в обработчике события onCreate напишите:

dx:=1; dy:=1;

В обработчике щелчка на SpeedButton1 напишите следующее:

```
Timer1.Enabled:=not(Timer1.Enabled);
if Timer1.Enabled=true then SpeedButton1.Caption:='СТОП'
else SpeedButton1.Caption:='ПУСК'
```

А после этого, как завершающий этап, в обработчике onTimer напишите:

```
if (SpeedButton1.Width<=10) or (SpeedButton1.Width>=150)
then dx:=-dx;
```

```
if (SpeedButton1.Height<=10) or (SpeedButton1.Height>=150)
then dy:=-dy;

SpeedButton1.Width:=SpeedButton1.Width+dx;
SpeedButton1.Height:=SpeedButton1.Height+dy;

Form1.Width:=Form1.Width-dx;
Form1.Height:=Form1.Height-dy;
```

Таким образом, мы описали класс TForm1. Теперь осталось определить, опять же в модуле UnitForm, процедуру которая будет доступна внешнему приложению через библиотеку.

В модуле UnitForm в окне редактора кода, сразу чуть выше перед ключевым словом **implementation,** опишем прототип экспортируемой процедуры

procedure CreateMyForm(Cvet:TColor); stdcall; а также, здесь же, сразу чуть ниже директивы **{$R *.dfm},** опишем саму экспортируемую функцию (рис. 12.8).

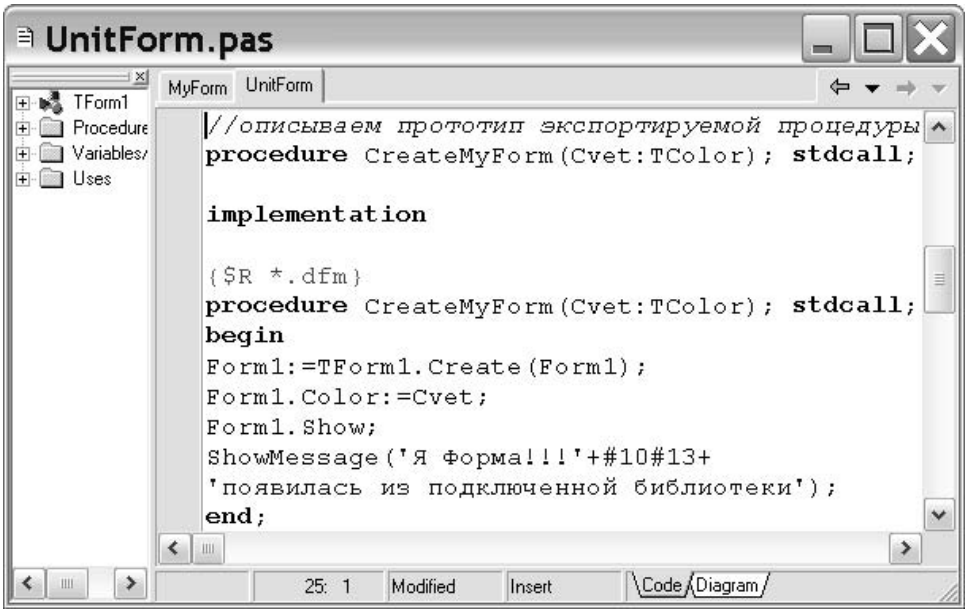


Рис. 12.8 Окно редактора кода с описанием прототипа и непосредственно самой экспортируемой процедуры CreateMyForm

Теперь переключитесь на файл библиотечного проекта MyForm.dpr. Добавьте туда небольшую строчку, определяющую exports CreateMyForm (рис. 12.9):

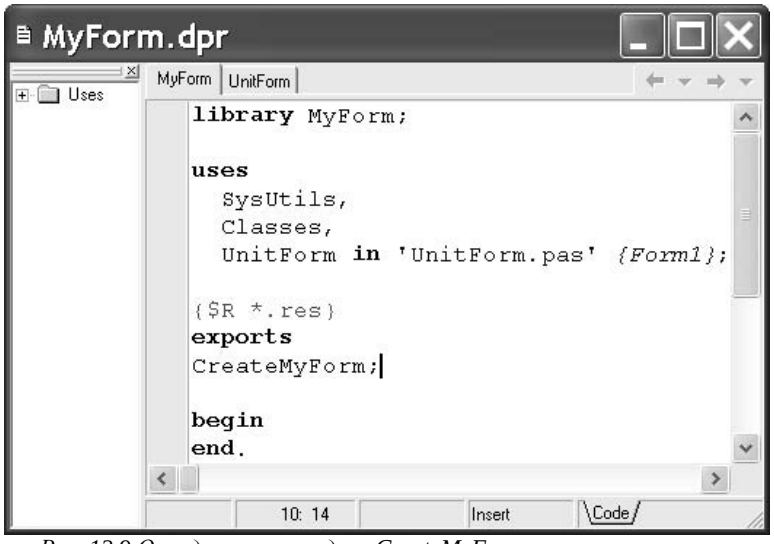


Рис. 12.9 Определяем процедуру CreateMyForm как экспортную

Вот и все. Теперь, как и прежде, можете скомпилировать библиотеку, нажав Ctrl+F9. Взгляните в папку проекта. Там должен появиться файл “MyForm.dll”. Закройте проект, нажав Close All.

Задание 12.2.2 Далее, как и в задании 1.2., создадим внешнее приложение, вызывающее функцию создания формы и передадим ей в качестве параметра цвет, который должна иметь создаваемая форма.

Снова создайте новый проект и сохранитесь в той же папке (можете согласиться с именами файлов проекта project1.dpr и unit1.pas по умолчанию), где находится созданная ранее библиотека “MyForm.dll”.

В окне редактора кода модуля проекта unit1.pas подключите используемую библиотеку. Как вам уже известно, сразу после ключевого слова implementation нужно написать следующее:

```
procedure CreateMyForm(Cvet:TColor); stdcall; external 'MyForm.dll';
```

Затем сбросьте на форму одну кнопку Button1, установите значение ее свойства Caption в “Запустить функцию CreateMyForm(Cvet:TColor), создающую форму из библиотеки MyForm.dll”.

Также скиньте компонент ColorBox1 с палитры Additional. В нем будем выбирать передаваемый цвет.

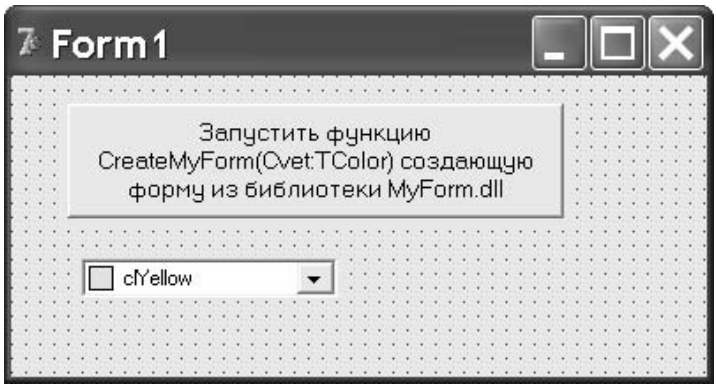


Рис. 12.10 Окно приложения использующее библиотеку MyForm.dll

А обработчик щелчка на Button1 должен быть следующим:

```
procedure TForm1.Button1Click(Sender: TObject);
var cvet:TColor;
begin
cvet:= ColorBox1.Selected;//определяем выбранный цвет
CreateMyForm(cvet);//создаем форму с этим цветом
end;
```

Вот и все. Можно запускать проект на выполнение. Программа должна работать правильно. **Вернитесь в режим программирования.**

12.3. Динамически подключаемые библиотеки

Часто во время выполнения программы бывают случаи, когда требуется загрузить библиотеку в оперативную память, попользоваться ею некоторое время, а затем выгрузить ее из оперативной памяти, так как в ней уже нет необходимости.

В этом случае пользуются тремя системными функциями операционной системы Windows, которые определены в модуле Windows.pas [2,4].

function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;

Функция загружает библиотеку с именем lpLibFileName в оперативную память и в качестве значения возвращает дескриптор загруженной библиотеки.

function GetProcAddress(hModule: HMODULE; lpProcName: LPCSTR): FARPROC; stdcall;

Зная по значению дескриптора hModule местонахождение в оперативной памяти загруженной библиотеки, функция возвращает адрес (указатель) конкретной процедуры или функции с именем lpProcName типа LPCSTR. (Здесь LPCSTR = PAnsiChar).

function FreeLibrary(hLibModule: HMODULE): BOOL; stdcall;

Функция выгружает из оперативной памяти загруженную библиотеку. Возвращает False, как только библиотека выгружена.

Нужно отметить то важное обстоятельство что, данные функции являются частью ядра операционной системы Windows, так как они содержатся в базовом модуле ядра ОС – Kernel, то есть находятся в библиотеке Kernel32.dll.

Данный модуль управляет базовыми вещами: файловым вводом/выводом, памятью, загрузкой и выполнением программ, потоками синхронизации объектов, поддержкой сетевой и файловых систем.

Таким образом, динамический вызов подпрограмм, процедур и функций конкретной библиотеки возможен только через посредника – операционную систему Windows. В ОС Linux предусмотрены другие названия подобных системных функций.

Задание 12.3.1. Подключим динамически библиотеку MainDll.dll к проекту и вызовем функцию суммы SummaX_Y(X,Y:real):real;

Будем использовать созданный вами ранее первый проект и библиотеку.

Скопируйте все файлы вашего первого проекта (вместе с готовой библиотекой MainDll.dll), который вы делали в первом задании, в новую папку.

Откройте этот новый проект в Delphi в режиме программирования.

Учитывая, что нам уже не нужна статическая загрузка библиотеки в момент загрузки внешнего приложения (т.е. данного проекта), в окне редактора кода модуля удалите все объявления импортируемых процедур и функций с ключевым словом external.

Первое, что нужно сделать, – это определить переменную процедурного (procedure) или функционального (function) типа.

Объявите глобальную переменную в секции var с именем SummaX_Y функционального типа function(X,Y:real):real.

Таким образом, в окне редактора кода должна быть следующая запись (рис. 12.11).

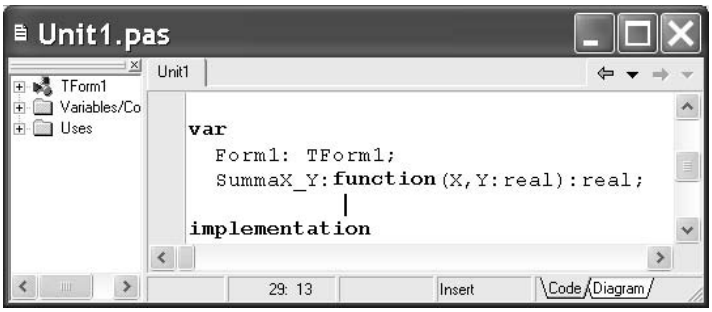


Рис. 12.11 Объявляем глобальную переменную SummaX_Y типа function

Далее измените надпись на кнопке на следующую:

«Запуск функции SummaX_Y из динамически подключенной библиотеки MainDll.dll»

Измените обработчик щелчка этой кнопки так, чтобы он выглядел следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var H:THandle;  X,Y:real; S:string;
begin
RichEdit1.Clear;
X:=StrToFloat(Edit1.Text);
Y:=StrToFloat(Edit2.Text);
S:=Edit3.Text;

H:=LoadLibrary('MainDll.dll');
{Определяем дескриптор подключенной библиотеки MainDll.dll}

if H <> 0 then begin
{Если дескриптор не равен нулю значит библиотека успешно загрузилась}

@SummaX_Y := GetProcAddress(H, 'SummaX_Y');
```

{Функцией GetProcAddress определяем адрес функции с именем SummaX_Y содержащийся в библиотеке и назначаем этот адрес переменной функционального типа SummaX_Y нашего проекта }

```
if @SummaX_Y<>nil then
Richedit1.Lines.Add(FloatToStr( SummaX_Y(X,Y)) );
{если адрес @ библиотечной функции успешно получен, то есть, не равен нулю, то вызываем функцию SummaX_Y}

FreeLibrary(H); {Выгружаем библиотеку из памяти}
                end;
end;
```

Вот и все! Можете запустить программу на выполнение. Программа должна работать правильно. **Вернитесь в режим программирования.**

Обратите внимание на оператор @ он возвращает адрес некоторого объекта в данном случае (процедуры или функции). Его применение эквивалентно функции Addr(X): Pointer.

В данном примере мы назначаем новый адрес переменной функционального типа из нашего проекта. Именно по этому адресу в оперативной памяти находится код функции из библиотеки.

Задание 12.3.2 (Самостоятельно). Добавьте на форму еще 2 кнопки Button, при щелчке на которые аналогично будут вызываться остальные 2 функции из динамически подключенной библиотеки MainDll.dll.

Указание. Должны вызываться следующие 2 функции:

function ProizvX_Y(X,Y:real):real; – функция произведения двух чисел X и Y.

function Revers(S:string):string; – функция возвращающая строку S задом-наперед.

ПРАКТИЧЕСКАЯ РАБОТА №13

13.1. Создание собственного видеопроигрывателя

В среде Delphi имеется очень полезный компонент (палитра System), представляющий собой компактную мини-студию и благодаря которому можно прослушивать звуковые файлы, например в форматах *.wav или *.mp3, а также просматривать видеофайлы различных известных форматов, например *avi.

Сам компонент состоит из управляющих кнопок, напоминает вид обычного видеомагнитофона (рис. 13.1) с возможностью проигрыша, паузы, остановки, записи, и т.д. [2]

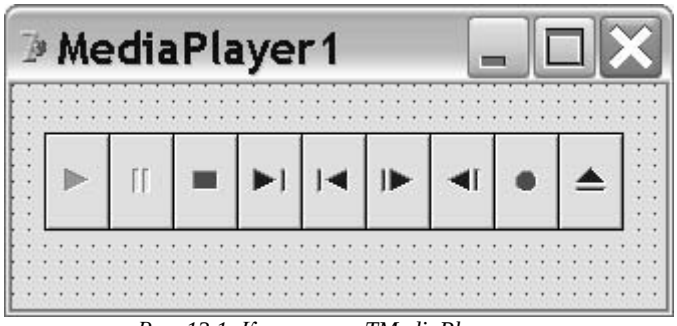


Рис. 13.1. Компонент TMediaPlayer

Задание 13.1.1 Определим общую схему проекта программы «Мой Проигрыватель», позволяющей просматривать видео файлы с расширением *avi.

Создайте новый проект. Сбросьте на форму следующие компоненты (рис. 13.2):

1) компонент MediaPlayer1, установите значение его свойства AutoEnable равным false. Данное свойство отвечает за доступность кнопок компонента MediaPlayer1. Если оно установлено в false, то все кнопки всегда доступны.
2) OpenDialog1
3) Timer1, установите значение его свойства Enabled равным false, а Interval:=100
4) TrackBar1
5) Button1, установите значение ее свойства Caption равным OPEN

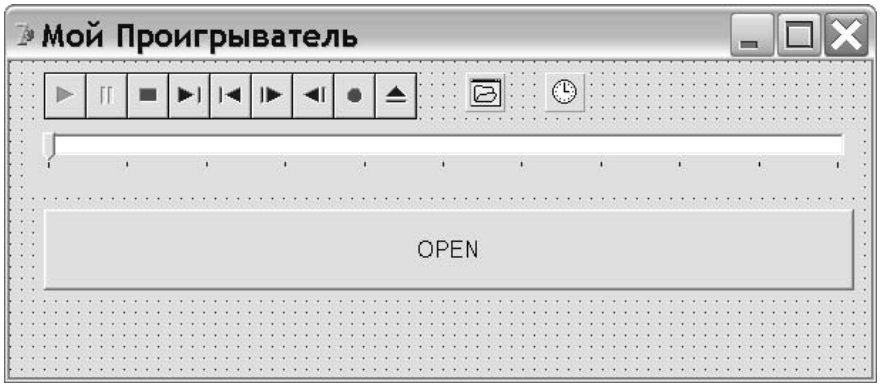


Рис. 13.2. Окно программы «Мой Проигрыватель»

Здесь кнопка «OPEN» будет содержать код, который, во-первых, активизирует диалоговое окно в виде компонента OpenDialog1, с помощью которого будет выбираться имя видео-файла, который затем будет проигрываться компонентом MediaPlayer1, а во-вторых, непосредственно запустит MediaPlayer1 на выполнение.

Указатель компонента TrackBar1 будет перемещаться в момент проигрывания файла,

то есть информировать пользователя о том, какая часть видеофайла в данный момент проигрывается, а также установкой этого указателя можно будет перемещаться на любую позицию проигрываемого файла.

В данной программе запрограммируем произвольное перемещение пользователем указателя на любую позицию только в момент «Паузы» или «Остановки» проигрывателя, как и в стандартном компоненте Windows XP – «Проигрыватель Windows Media».

Задание 13.1.2 Определим соотношения между кнопками «Play», «Pause» и «Stop» и кратко рассмотрим тип TButtonSet

В момент нажатия кнопки «Play» должно начаться воспроизведение, указатель компонента TrackBar1 должен перемещаться по мере проигрывания, данная кнопка «Play» должна перестать быть доступной, а должны быть доступны только кнопки «Stop» и «Pause».

В момент нажатия кнопки «Pause» проигрывание должно остановиться, указатель компонента TrackBar1 должен остановиться в той позиции, где он находился, данная кнопка «Pause» должна перестать быть доступной, и только кнопки «Stop» и «Play» должны быть доступны.

В момент нажатия кнопки «Stop» проигрывание должно завершиться, указатель компонента TrackBar1 должен установиться в начальную позицию, а кнопка «Stop» вместе с кнопкой «Pause» должны перестать быть доступными, зато должна быть доступна кнопка «Play».

Доступность всех кнопок компонента MediaPlayer1 определяет его свойство

EnabledButtons, которое имеет тип – множество. В этом множестве перечисляются идентификаторы кнопок компонента MediaPlayer1. Если они находятся в множестве, то данные кнопки доступны, иначе нет.

Описание данного множества имеет следующий вид:

property EnabledButtons: TButtonSet read FEnabledButtons write SetEnabledButtons
default [btPlay, btPause, btStop, btNext, btPrev, btStep, btBack, btRecord, btEject];

TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep, btBack, btRecord, btEject);
TButtonSet = set of TMPBtnType;

Из описания следует, что по умолчанию (default) доступны все имеющиеся кнопки, однако нас интересуют только первые 3 кнопки btPlay, btPause и btStop из множества типа TButtonSet. Остальные кнопки мы ради простоты программировать не будем.

Поэтому, чтобы изменять доступность той или иной кнопки, требуется просто уметь работать со множеством.

В данном случае, если множество пустое, то есть имеет вид [] , значит недоступна ни одна кнопка, если, например, множество имеет вид [btPlay, btPause], то доступны только две кнопки «Play» и «Pause».

Чтобы, например, создать множество [btPlay, btPause, btStop], можно применить простое сложение множеств [btPlay, btPause] + [btStop], аналогично, чтобы получить множество [btPlay, btPause], можно применить операцию разности множеств [btPlay, btPause, btStop] – [btStop].

Задание 13.1.3 Создадим глобальную переменную типа TbuttonSet. хранящую множество идентификаторов кнопок компонента MediaPlayer1 по умолчанию.

В окне редактора кода объявите глобальную переменную T типа TButtonSet (рис. 13.3).

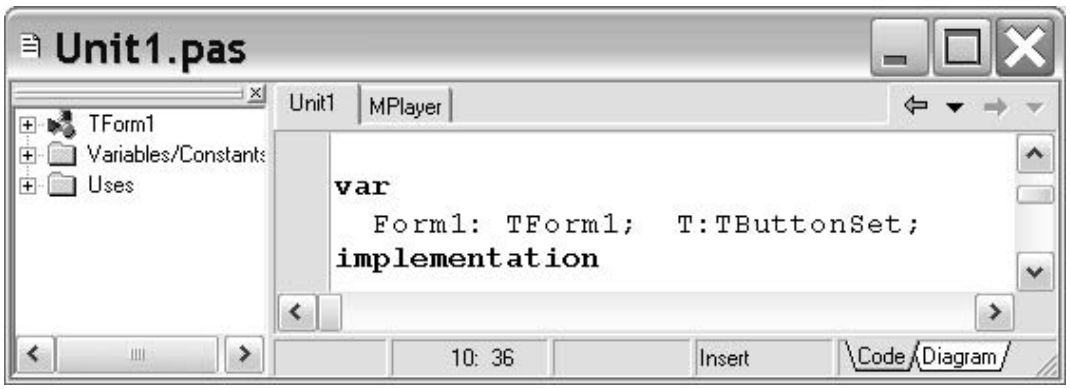


Рис. 13.3. Объявляем глобальную переменную T типа TButtonSet

Затем в обработчике события onCreate формы напишите следующее:

T:=MediaPlayer1.EnabledButtons;
{Инициализируем глобальную переменную T множеством всех кнопок MediaPlayer1.EnabledButtons в начальный момент, то есть все кнопки доступны}

MediaPlayer1.EnabledButtons:=[];
{Устанавливаем множество пустым, то есть ни одна кнопка не доступна в MediaPlayer1 в начальный момент запуск программы}

Задание 13.1.4 Создадим обработчики события onClick кнопки Button1, события onTime компонента Timer1, событий onClick и onNotify компонента MediaPlayer1.

Создадим обработчик события onClick кнопки Button1.

В обработчике события onClick кнопки Button1 напишите следующее:

MediaPlayer1.EnabledButtons:=T-[btPlay];

{Отключаем доступность кнопки «Play», просто вычитая ее из множества T, где доступны все кнопки, и присваиваем данный результат свойству типа множество EnabledButton компонента MediaPlayer1}

if OpenDialog1.Execute then

begin

MediaPlayer1.FileName:=OpenDialog1.FileName; {Указываем имя проигрываемого файла компоненту MediaPlayer1}

MediaPlayer1.Frames:=1; {Устанавливаем количество кадров, на которое будет сдвигаться видеофайл при щелчке на кнопке «Step»}

MediaPlayer1.Open; {Загружаем видеофайл в оперативную память}

MediaPlayer1.Play; {Начать воспроизведение видеофайла}

end;

MediaPlayer1.Notify:=true;{Устанавливаем уведомление о событии мультимедиа}

TrackBar1.Min:=0; {Минимально возможный номер, соответствующий позиции указателя TrackBar1}

TrackBar1.Max:=MediaPlayer1.Length; {Присваиваем максимальному значению позиции указателя TrackBar1 число, равное количеству кадров видеофайла}

Timer1.Enabled:=true; {активируем таймер для перемещения указателя}

Здесь все ясно из комментариев, метод Open загружает видеофайл в оперативную память, а Play воспроизводит видеофайл. У компонента MediaPlayer1 имеется событие onNotify – уведомление о событии мультимедиа, которое происходит всякий раз, когда нажали какую- либо кнопку компонента MediaPlayer1,или когда началось воспроизведение или прекратилось и т.д., то есть когда произошло любое мультимедийное событие.

Существует булевский флаг Notify, отвечающий за то, будет ли обрабатываться такое событие. Если он установлен в True, то всякий раз, когда генерируется мультимедийное событие, будет вызываться обработчик данного события, иначе он вызываться не будет. В описываемой будем всегда устанавливать его значение равным True.

Затем устанавливаем в минимальное и максимальное значение позиции указателя TrackBar1. Минимальное ноль, а максимальное равно полному числу кадров видеофайла.

И чтобы в реальном времени видеть, какая часть файла проигрывается, перемещаем указатель TrackBar1 через определенные промежутки времени, используя обработку события onTime таймера Timer1.

Создадим обработчик события onTime компонента Timer1

Создайте данный обработчик события и напишите там всего одно выражение:

TrackBar1.Position:=MediaPlayer1.Position;

Устанавливаем указатель TrackBar1 в позицию равную номеру кадра видеофайла.

Создадим обработчик события onClick компонента MediaPlayer1

Создайте данный обработчик события и напишите там следующее:

MediaPlayer1.Notify:=true; {Генерировать уведомление о событии мультимедиа}

if Button=btPlay then begin {Если нажали кнопку Play}

MediaPlayer1.EnabledButtons:=T-[btPlay]; {то она становится недоступной}

MediaPlayer1.Position:=TrackBar1.Position; {Устанавливаем номер текущего кадра MediaPlayer1 в позицию определяемую указателем TrackBar1}
timer1.Enabled:=true; {активируем таймер}
end;

if Button=btPause then MediaPlayer1.EnabledButtons:=T-[btPause]; {Если нажали кнопку Pause, то она становится недоступной }

if Button=btStop then MediaPlayer1.EnabledButtons:=T-[btPause,btStop]; {Если нажали кнопку Stop, то она становится недоступной вместе с кнопкой Pause}

И наконец осталось описать обработчик события мультимедиа.

Создадим обработчик события onNotify компонента MediaPlayer1

Создайте данный обработчик события и напишите там следующее:

if (MediaPlayer1.Mode=mpPaused) {Если режим воспроизведении находится в состоянии Паузы, тогда}
then
timer1.Enabled:=false; {останавливаем перемещение указателя TrackBar1 путем перевода таймера в неактивное состояние}

if (MediaPlayer1.Mode=mpStopped) {Если режим воспроизведении находится в состоянии Остановки, тогда}
then
begin
timer1.Enabled:=false; {останавливаем перемещение указателя TrackBar1 }
TrackBar1.Position:=0; {устанавливаем его позицию в ноль}
MediaPlayer1.Rewind; { переходим в начало видеофайла }
end;

if (MediaPlayer1.Mode=mpPlaying) {Если режим воспроизведении находится в состоянии Проигрывания, тогда}
then timer1.Enabled:=true; {активируем перемещение указателя TrackBar1 путем перевода таймера в активное состояние}

Вот и все. Запустите программу щелкните на кнопке Button1, затем с помощью диалогового окна выберите нужный видеофайл *.avi или музыкальный файл *.mp3 и нажмите OK. Начнется проигрывание. Щелкайте на кнопках Play, Pause и Stop. В состоянии паузы переместите указатель и снова нажмите Play. Программа должна работать правильно. **Вернитесь в режим программирования.**

13.2. Вывод видео на стандартные компоненты Delphi. Описание свойства DeviceType компонента MediaPlayer1

Задание 13.2.1 В этом же проекте перенаправим видеопоток либо на Button1 либо на Panel1.
Создайте новый проект. Растяните немножко кнопку Button1. Сбросьте на форму следующие компоненты (рис. 13.4):

Компонент	Свойство	Значение свойства
CheckBox1	Caption	Видео или в Button1 или в Panel1
Panel1	Caption	VIDEO
Label1	Caption	Выберите тип устройства
ComboBox1	Text	dtAutoSelect

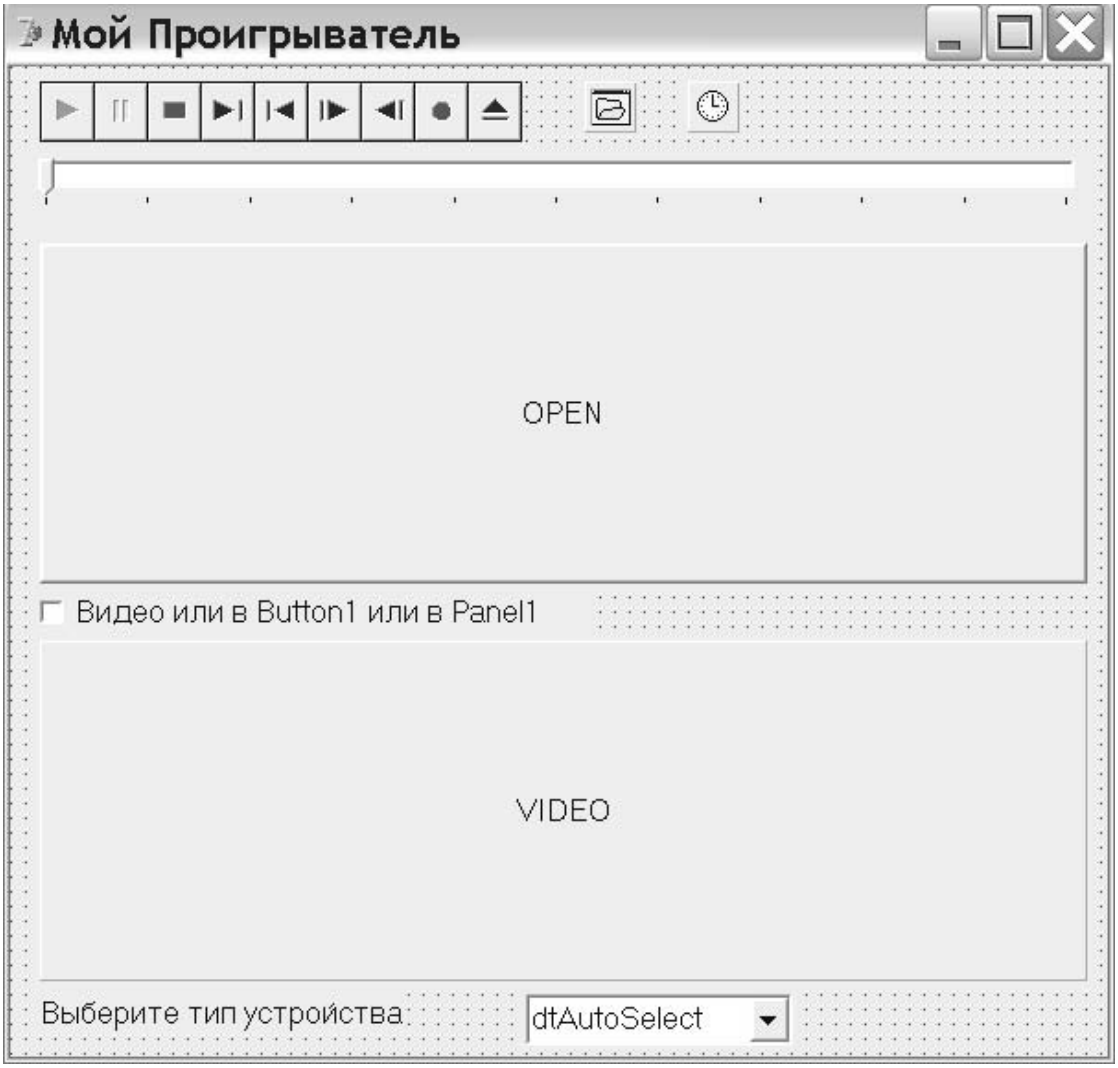


Рис. 13.4 Добавляем несколько компонентов в программу «Мой Проигрыватель»

Создайте обработчик события onClick компонента ComboBox1 и напишите там всего лишь одно выражение:

```
if CheckBox1.Checked then MediaPlayer1.Display:=Button1
else MediaPlayer1.Display:=Panel1;
```

То есть, если CheckBox1 выбран, то видеоизображение будет направлено на кнопку Button1, а иначе будет выводиться на поверхность компонента Panel1.

Здесь используется свойство Display класса TwinControl, определяющее компонент. на который будет выводиться изображение. Таким образом, любой компонент, являющийся наследником класса TwinControl, может стать «экраном» для отображения видео. Чтобы в этом убедиться, в дальнейшем попробуйте самостоятельно добавить любой компонент этого класса и присвойте его имя свойству Display.

Запустите программу. Щелкайте на CheckBox1, при каждом щелчке изображение должно перепрыгивать с одного компонента на другой (рис. 13.5). **Вернитесь в режим программирования.**

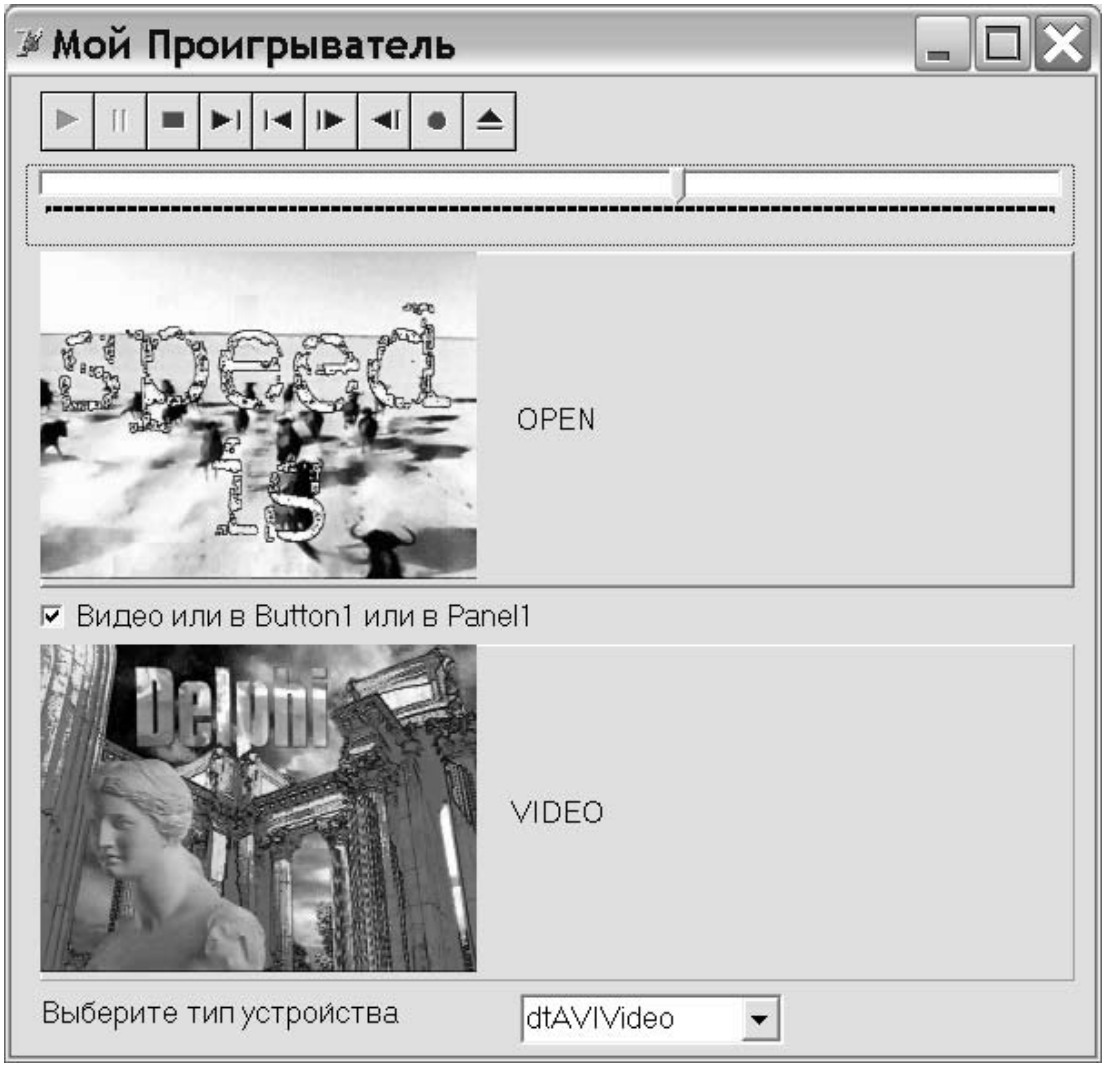


Рис. 13.5 Программа «Мой Проигрыватель» в режиме выполнения

Задание 13.2.2 В этом же проекте используя выпадающий список ComboBox1 добавим возможность пользователю выбирать тип устройства, которым будет управлять MediaPlayer1

Это можно сделать с помощью свойства DeviceType компонента MediaPlayer1. Это свойство по умолчанию имеет значение dtAutoSelect – автоматический выбор устройства, то есть автоматически определяет тип используемого драйвера на основании расширения имени файла в системном реестре Windows.

Так как свойства DeviceType имеет тип TMPDeviceTypes, то, чтобы конкретно задать тип используемого устройства, нужно просто присвоить свойству DeviceType одно из значений типа TMPDeviceTypes

Описание свойства DeviceType
property DeviceType: TMPDeviceTypes read FDeviceType write SetDeviceType default dtAutoSelect;

Описание типа TMPDeviceTypes
TMPDeviceTypes = (dtAutoSelect, dtAVIVideo, dtCDAudio, dtDAT, dtDigitalVideo, dtMMMovie, dtOther, dtOverlay, dtScanner, dtSequencer, dtVCR, dtVideodisc, dtWaveAudio);

Поэтому, чтобы произвести выбор нужного устройства, щелкните двойным щелчком на свойстве Items компонента ComboBox1 и внесите в появившееся окно String List Editor все значения типа TMPDeviceTypes, данное окно должно выглядеть так:

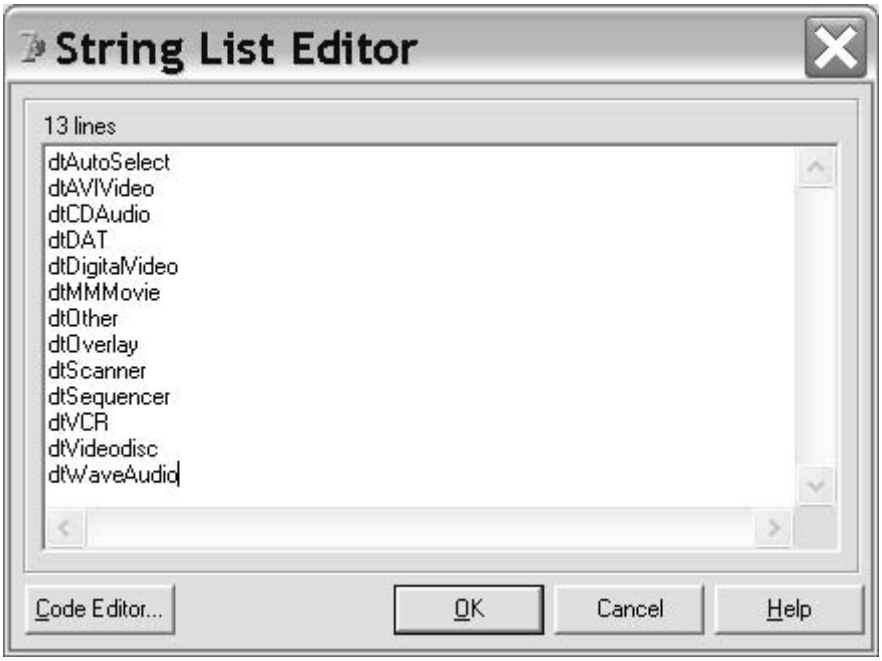


Рис. 13.6 Окно String List Editor со всеми значениями типа TMPDeviceTypes

И нажмите OK. Теперь при щелчке на ComboBox1 будет выпадать список с содержимым данного окна.

Далее требуется запрограммировать присвоение свойству DeviceType выбранное из ComboBox1 значение, поэтому создайте обработчик события onChange компонента ComboBox1 и напишите там следующий код:

```
case ComboBox1.ItemIndex of

 0: MediaPlayer1.DeviceType:=dtAutoSelect;
 1: MediaPlayer1.DeviceType:=dtAVIVideo;
 2: MediaPlayer1.DeviceType:=dtCDAudio;
 3: MediaPlayer1.DeviceType:=dtDAT;
 4: MediaPlayer1.DeviceType:=dtDigitalVideo;
 5: MediaPlayer1.DeviceType:=dtMMMovie;
 6: MediaPlayer1.DeviceType:=dtOther;
 7: MediaPlayer1.DeviceType:=dtOverlay;
 8: MediaPlayer1.DeviceType:=dtScanner;
 9: MediaPlayer1.DeviceType:=dtSequencer;
 10: MediaPlayer1.DeviceType:=dtVCR;
 11: MediaPlayer1.DeviceType:=dtVideodisc;
 12: MediaPlayer1.DeviceType:=dtWaveAudio;
 end;
```

Здесь мы используем оператор выбора case по значению свойства ItemIndex компонента ComboBox1. ItemIndex определяет номер выбранной строки свойства Items.

Вот и все. Программа готова. Запустите ее на выполнение. Выбирайте тип используемого устройства. Программа должна работать правильно. **Вернитесь в режим программирования.**

ПРАКТИЧЕСКАЯ РАБОТА №14

14.1. Классы и объекты

Нужно всегда помнить, что Delphi – это среда объектно-ориентированного программирования [2-4, 8-10]. На сегодняшний день именно объектно-ориентированная парадигма является основой программирования и максимальное свое воплощение нашла в .NET-технологии компании Microsoft, где переменная может в одних случаях вести себя как обыкновенная переменная, а в других – выступать в качестве полноценного объекта со всеми своими свойствами и методами [5].

К большой радости разработчиков, Delphi 2006 позволяет разрабатывать как стандартные объектно-ориентированные win32 приложения, так и .NET приложения.

Однако в целом данная технология достаточно сложна и для начала нужно научиться разрабатывать обыкновенные объектные windows 32 – приложения, после чего будет проще разобраться в технологии .NET.

Под объектом понимается целостная программная единица, являющаяся совокупностью данных и программного кода.

Вспомним о том, о чем кратко говорилось в начале первой практической работы. А именно о том, что каждый объект в объектном программировании (например, объект с именем Form) описывается определенным классом (TForm). **Понятие «класс» является логической абстракцией, описывающей некоторый объект реального мира. В оперативной памяти компьютера физически существует объект, созданный на основе данного класса (говорят, объект – это экземпляр класса). Объект физически занимает определенную область в оперативной памяти компьютера и производит определенные действия со своими данными (полями), используя методы класса.**

Довольно удачное сравнение приводит Холзнер [1], говоря, что класс – это форма для печенья, а объект – это само печенье. Часто объектом выступает обыкновенное окно Windows, именно такое окно в «чистом виде» и описывает класс TForm в Delphi. То есть внешний вид окна, цвет заливки, надпись заголовка в окне и т. д.

У класса имеются свои поля и методы. В сущности, поля класса – это данные любого определенного типа, принадлежащие классу, а методы класса – это процедуры и функции класса. Например некоторые поля класса TForm содержат информацию о внешнем виде окна (ширина, высота, цвет и т. д.). Поля могут содержать текстовую информацию, некоторый адрес в памяти, просто целое или вещественное число, то есть могут содержать значение почти любого типа определенного в Delphi.

Методы – это процедуры и функции, которые принадлежат объекту или классу. Методы непосредственно исполняют некоторые действия, в том числе могут работать с полями. Говоря кратко, **методы – это процедуры и функции, принадлежащие объекту (классу), а поля – это данные объекта (класса).**

Существуют статические, виртуальные и динамические, а также абстрактные методы. В Delphi По умолчанию все методы являются статическими.

Виртуальные и динамические методы имеют одинаковую природу с точки зрения их функциональности, а абстрактные методы – это почти те же виртуальные методы только не реализованные в исходном классе [4, 5].

Данное различие в названиях методов обусловлено наиважнейшей особенностью классов-**наследованием**, которое будет рассмотрено ниже.

Синтаксис объявления класса следующий [4]:

type

<имя класса> = class (<имя класса-родителя>)

public {доступно методам любого другого класса в исходном модуле, где объявлен класс и методам любого другого класса в другом модуле}

<поля, методы, свойства, события>

private {доступно методам любого другого класса, но только в исходном модуле, где объявлен класс}

<поля, методы, свойства, события>

protected { доступно только методам производных-классов даже в других модулях и методам любого класс,а но только в исходном модуле, где объявлен класс }

<поля, методы, свойства, события>

published {отображаются в Object Inspector}

<поля, методы, свойства, события>

Например, рассмотрим простейший класс, описывающий параллелепипед. Можно было бы назвать его Parallel, но лучше назвать его TParallel. По общепринятому соглашению, впереди имени класса ставится заглавная буква «T», от слова «type» – тип, так как часто класс и тип – это одно и тоже.

Ясно, что у параллелепипеда есть ширина, длина и высота.

Создадим поля, которые будут хранить соответствующие ширине, длине и высоте значения целого типа word.

Например, поле, хранящее ширину, можно было бы назвать «a», но имена полей, по негласному соглашению, предваряются заглавной буквой «F» (Field – поле), поэтому лучше назвать его «Fa». Остальные два поля назовем соответственно «Fb» и «Fc».

Осталось создать методы. Пока рассмотрим **статические методы**. По умолчанию в Delphi все методы являются статическими [8].

Например, для объекта-параллелепипеда интересно узнать его объем, а затем вывести его значение на экран. Поэтому для данного класса создадим три статических метода:

Первый метод-процедура Init будет инициализировать значения полей Fa, Fb и Fc.

Когда говорят, **проинициализировать объект, это означает, что требуется проинициализировать данные – члены объекта (поля, свойства), то есть присвоить полям объекта начальные значения.** Если этого не сделать, то эти поля будут содержать случайные значения, или, как часто говорят, «мусорные значения».

Реализуем инициализирующий метод в виде процедуры с именем Init. Пусть он принимает 3 параметра a, b, c целого типа word, а затем присваивает значения этих аргументов соответствующим полям и возвращает объем в виде целого значения типа word.

Второй метод-функция Volume будет рассчитывать и возвращать объем, равный простому произведению полей.

Реализуем его в виде функции с именем Volume. Пусть у этого метода вообще не будет параметров.

И наконец, третий метод-процедура Show будет выводить на экран значение объема и полей объекта.

У этого метода также не будет параметров.

Начинается описание класса с ключевого слова type, а заканчивается словом end.

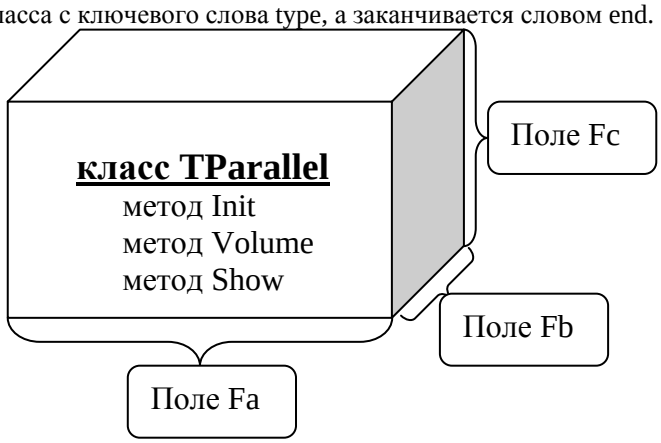


Рис. 14.1а. Схематическое описание класса TParallel

Задание 14.1.1 Опишем новый класс с именем TParallel (описывающий параллелепипед) с тремя полями Fa, Fb и Fc и тремя методами.

Создайте новый проект. Описание классов производится в секции **interface** модуля**.**

Поэтому в окне редактора кода, сразу же после описания уже имеющегося класса TForm1 напишите ключевое слово **type,** предваряющее описание класса. Затем непосредственно укажите имя класса, записав инструкцию TParallel = class.

Данная инструкция, имеющая ключевое слово **class,** говорит о том, что будет создан новый класс TParallel, являющийся наследником **базового класса TObject**. И этот новый класс TParallel унаследует все поля, методы и свойства от базового класса TObject.

Далее объявите там члены класса (поля и методы) (рис. 14.1б). Завершается описание класса привычным ключевым словом **end**.

Таким образом, описание данного класса должно иметь следующий вид:

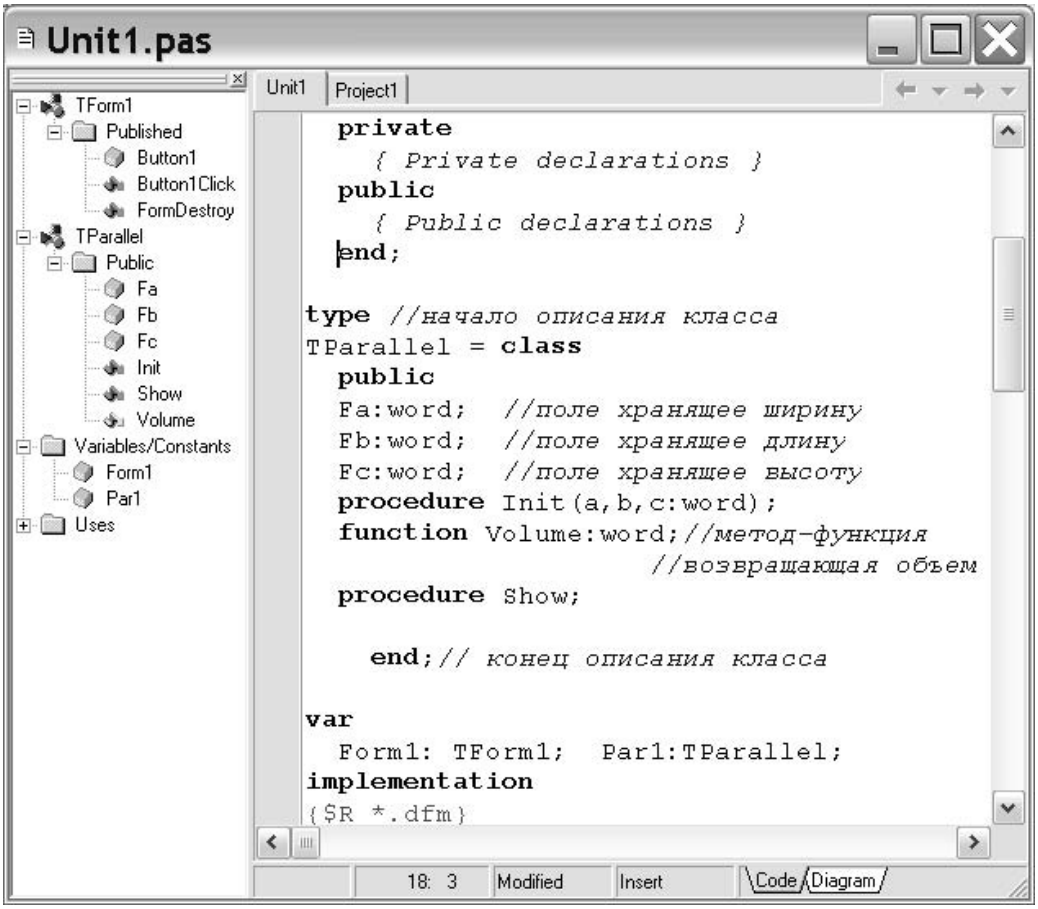


Рис. 14.1б Описание класса TParallel в окне редактора кода Delphi

Теперь требуется конкретно описать реализацию всех методов класса TParallel. Она описывается за пределами описания класса в **разделе реализации,** предваренном ключевым словом **implementation**.

Для этого установите курсор на имя любого метода класса и нажмите Ctrl+Shift+C

Перед вами сгенерируются заготовки реализации всех методов данного класса:

```
{ TParallel }
// Метод инициализирующий поля объекта класса TParallel
procedure TParallel.Init(a, b, c: word);
begin
end;

// Метод, отображающий значение объема, а также всех полей Fa, Fb и Fc
procedure TParallel.Show;
begin
end;

// Метод, возвращающий объем
function TParallel.Volume:word;
begin
end;
```

А теперь изменим заготовки методов, наполняя их программным кодом следующим образом:

```
// Метод, инициализирующий поля объекта класса TParallel
procedure TParallel.Init(a, b, c: word);
begin
Fa:=a; //присваиваем полю Fa значение a
Fb:=b; //присваиваем полю Fb значение b
Fc:=c; //присваиваем полю Fc значение c
end;
```

```
// Метод, отображающий значение объема, а также всех полей Fa, Fb и Fc
procedure TParallel.Show;
begin
ShowMessage('Объем параллелепипеда равен ' + IntToStr(Volume)+#10#13+
'Ширина – Поле Fa= '+IntToStr(Fa)+#10#13+
'Длина – Поле Fb= '+IntToStr(Fb) +#10#13+
'Высота – Поле Fc= '+IntToStr(Fc) );
end;

// Метод, возвращающий объем
function TParallel.Volume:word;
begin
result:=Fa*Fb*Fc; {возвращаем во внешнюю программу объем в виде значения типа word}
end;
```

Вот и все: мы полностью описали реализацию всех методов, входящих в данный класс.

Задание 14.1.2 Запрограммируем так, чтобы при щелчке на кнопке Button1 на основе созданного класса TParallel в памяти создавался конкретный объект с именем Par1, а при щелчке на кнопке Button2 будем проверять существование данного объекта в памяти.

Сбросьте на форму кнопку Button1. Установите надпись на ее заголовке – «Создать объект класса TParallel». Аналогично для Button2 установите надпись – «Проверить, существует ли объект в памяти».

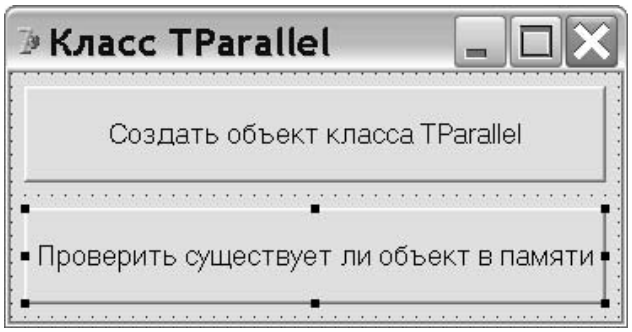


Рис. 14.2 Окно программы «Класс TParallel»

Ради простоты доступа пусть созданный объект будет глобальным, для этого объявите глобальную переменную-объект класса TParallel с именем Par1.

Для этого рядом с объявлением var Form1: TForm1 сразу перед разделом implementation просто напишите (рис. 14.1):

```
Par1:TParallel;
```

Данная инструкция говорит компилятору об объявлении лишь переменной-указателя объектного типа с именем Par1 класса TParallel, но при этом самого объекта класса TParallel в оперативной памяти еще существовать не будет, то есть необходимый объем памяти для объекта выделен не будет. В данном случае мы имеем дело с динамическим выделением памяти.

Как только объект будет сформирован (создан) в оперативной памяти с помощью вызова специального метода-конструктора Create, переменная-указатель Par1 (часто ее называют ссылкой или переменной-объектом) будет содержать адрес данного объекта (указывать на объект) в памяти. А до этого момента переменная-указатель Par1 не указывает, «ни на что». Это означает, что, пока объект не создан, ссылка Par1 указывает на область памяти с адресом, равным 0. А операционная система Windows запрещает расположение любого объекта по этому нулевому адресу. **У нулевого адреса есть псевдоним – служебное слово nil.**

Тот факт, что Par1 лишь содержит адрес объекта и не является самим объектом, не должен постоянно отвлекать программиста. В данном случае переменная-указатель Par1 ассоциирована с объектом, поэтому в некотором смысле ее можно отождествить с тем же объектом, на который она указывает и считать Par1 – объектом класса TParallel.

В Delphi объект может быть создан только динамически.

Создайте обработчик события onClick кнопки Button2, отвечающий за проверку нахождения объекта в памяти, и напишите там следующее:

```
if Par1=nil // если Par1 ни на что не указывает
then // то
Form1.Caption:='Объекта в памяти нет'
else Form1.Caption:= 'Объект существует в памяти';
```

Таким образом, данный обработчик проверяет на равенство нулю ссылки Par1 и в зависимости от результата выводит уведомление об отсутствии или присутствие объекта в памяти в заголовок формы.

Теперь **создайте обработчик события onClick кнопки Button1. отвечающей за создание объекта, и** напишите там следующий код:

```
Par1:=TParallel.Create; // Создать объект в памяти
Par1.Init(1,2,3); // Инициализировать поля объекта
Par1.Show; // Вывести на экран значение объема и полей объекта
```

Здесь первая **инструкция вызова метода-конструктора Create класса TParallel дает указание компилятору непосредственно создать объект в куче, то есть динамически выделить необходимый объем оперативной памяти, разместить там объект, и присвоить переменной Par1 адрес данной области, где расположен созданный объект. После вызова метода-конструктора Create объект начинает существовать в памяти**

Кучей называется вся динамически распределяемая оперативная память, то есть вся свободная память, не занятая системными программами.

Create это имя метода-конструктора, отвечающего за размещение и инициализацию объекта в динамической памяти.

Обратная по своему действию процедура уничтожения объекта и освобождающая память называется **деструктор**, как правило деструктор имеет имя **Destroy**. Однако часто удобно вызывать метод **free**, который аналогичен действию деструктора. Например, для того чтобы освободить объект Par1 из памяти, следовало бы написать Par1.Destroy или (что лучше) Par1.free, так как в методе free проверяется значение переменной указателя, и если оно не равно nil, то тогда вызывается destroy.

Далее с помощью метода Init(1,2,3) мы инициализируем объект-параллелепипед значениями его сторон, равными например 1, 2 и 3. То есть соответствующим полям объекта Par1 присваиваем соответствующие начальные значения 1, 2 и 3.

Доступ к методу Init объекта Par1 **достигается с помощью оператора точка «.».**

Такой оператор по-другому называется **оператором прямого доступа** к методу или полю объекта. Для сравнения в языке C++ доступ к методу может быть организован таким же образом через точку, но также можно создать переменную-указатель pPar1, содержащую адрес объекта Par1, и тогда доступ через указатель к методу Init будет осуществляться в виде инструкции со стрелкой [5]: pPar1 -> Init. Это так называемый **косвенный доступ** к методу.

В Builder C++ , по внешнему виду брате близнеце Delphi, в большинстве случаев доступ к методам именно такой.

И наконец, мы вызываем метод Show данного объекта, отображающий информацию о полях и объеме на экран. **Нужно помнить, что в оперативной памяти присутствует именно объект Par1 класса TParallel, а не сам класс TParallel.**

Запустите программу. В самом начале щелкните на кнопке «Проверить, существует ли объект в памяти», вы увидите, как поменяется заголовок формы. А затем щелкните на кнопке «Создать объект класса TParallel», вы увидите значения объема и полей объекта (рис. 14.3).

Затем снова щелкните на кнопке «Проверить, существует ли объект в памяти» и проанализируйте полученный результат. **Вернитесь в режим программирования.**

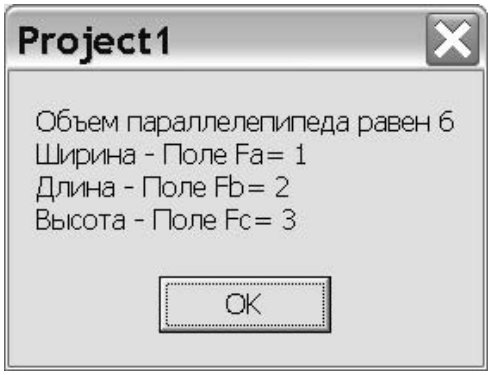


Рис. 14.3 Метод Show объекта Par1
отображает текущие значения полей объекта и объема

Конечно же, нужно отметить, что проверка истинности равенства Par1=nil не является гарантией того, что объекта в памяти нет, все зависит от программного кода.

Ведь после создания объекта можно в любой момент **установить Par1 равным nil,** просто написав Par1:=nil. Данное действие называется **сбросом указателя в ноль.**

Убедитесь в этом, добавив в самое начало обработчика Button2Click данную инструкцию. При щелчке на кнопке «Проверить, существует ли объект в памяти» в заголовке формы вне зависимости от того, создан объект или нет, всегда будет отображаться, что объекта в памяти нет.

Запустите программу. Она должна работать верно. **Вернитесь в режим программирования и закомментируйте или удалите эту привнесенную запись Par1:=nil в обработчике Button2Click.**

Задание 14.1.3 Покажем, что каждый раз при выполнении инструкции Par1:=TParallel.Create (при очередном щелчке на Button1) динамическая память под новый объект выделяется заново, а предыдущие объекты, на которые до этого указывал Par1, «сидят в памяти», занимают ее и не уничтожаются.

Таким образом, когда вы щелкаете на кнопке «Создать объект класса TParallel» при каждом следующем щелчке в памяти создается новый объект, и переменная-указатель Par1 указывает уже на этот объект, а предыдущее значение Par1, которое до этого соответствовало адресу старого объекта, теперь теряется, перезаписываясь новым.

Поэтому память, выделенная для старого объекта, безвозвратно утеряна, то есть пропадает информация о местоположении объекта (значение его адреса) и данная потерянная память уже не может быть использована ни операционной системой, ни вашей программой. В программировании такая ситуация называется **утечкой памяти,** поэтому в некоторых системах программирования, в частности, с использованием .NET-технологии предусмотрен так называемый **сборщик мусора**-специальная программа общей среды выполнения CLR [5], отслеживающая все переменные ссылающиеся на динамически размещенные объекты. Это значит, что, как только сборщик мусора обнаруживает отсутствие переменной-указателя, содержащего адрес некоторого объекта, то в дальнейшем на него невозможно будет сослаться в программе, и поэтому он автоматически удаляется из памяти [5]. В Delphi7 такого механизма нет, поэтому программист сам обязан следить за памятью.

Запрограммируем вывод адреса каждого вновь созданного объекта при очередном щелчке на Button1.

В этом же проекте сбросьте компонент Memo1 и кнопку Button3 с надписью «Вывести значения полей массива»:

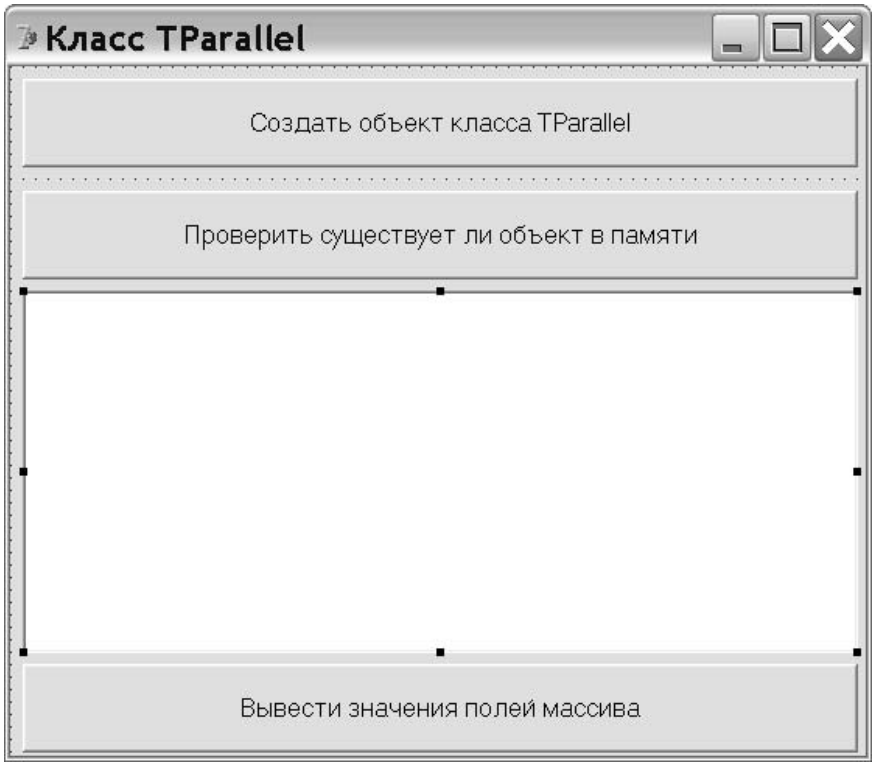


Рис. 14.4 Модифицированное окно программы «Класс TParallel»

Создайте 3 глобальные переменные x,y,z типа word следующим образом:

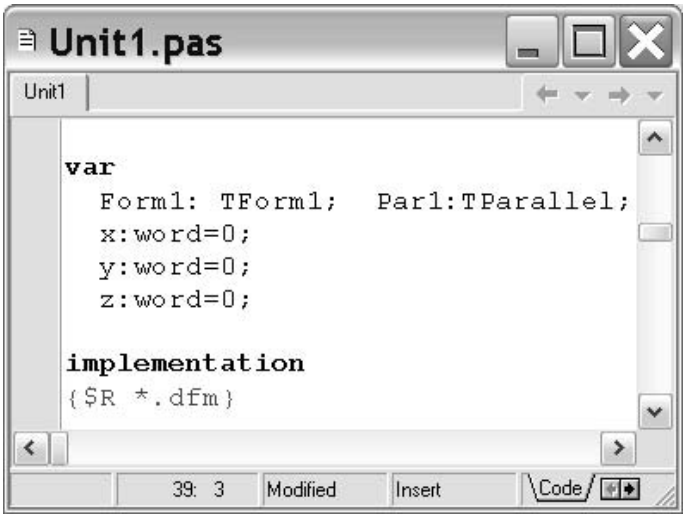


Рис. 14.5 Объявление глобальных переменных, инициализированных нулем

В данном случае объявление глобальной переменной x:word = 0 означает, что начальное значение x сразу же после запуска программы равно нулю (вместо нуля можно поставить любое входящее в диапазон типа значение).

Значения данных переменных просто будут увеличиваться на 1 при нажатии на кнопку «Создать объект класса TParallel» для инициализации полей объекта.

Измените обработчик щелчка на Button1 следующим образом:

```
x:=x+1; y:=y+1; z:=z+1; { инкрементируем переменные при очередном щелчке}
Par1:=TParallel.Create; {создаем новый объект и записываем его адрес в Par1 }
Par1.Init(x,y,z); {инициализируем поля объекта значениями глобальных переменных}
Par1.Show; {вызываем метод Show нового объекта}

{В Memo1 выводим содержимое переменной Par1, то есть адрес объекта в памяти}
Memo1.Lines.Add('Адрес в памяти объекта, содержащийся в Par1, равен '
+IntToStr(integer(Par1)) );
```

Запустите программу. Несколько раз щелкните на Button1. Каждый раз адрес объекта в памяти будет новым. Если вы щелкните 5 раз, результат будет примерно таким же, как на рис. 14.6. **Вернитесь в режим программирования.**

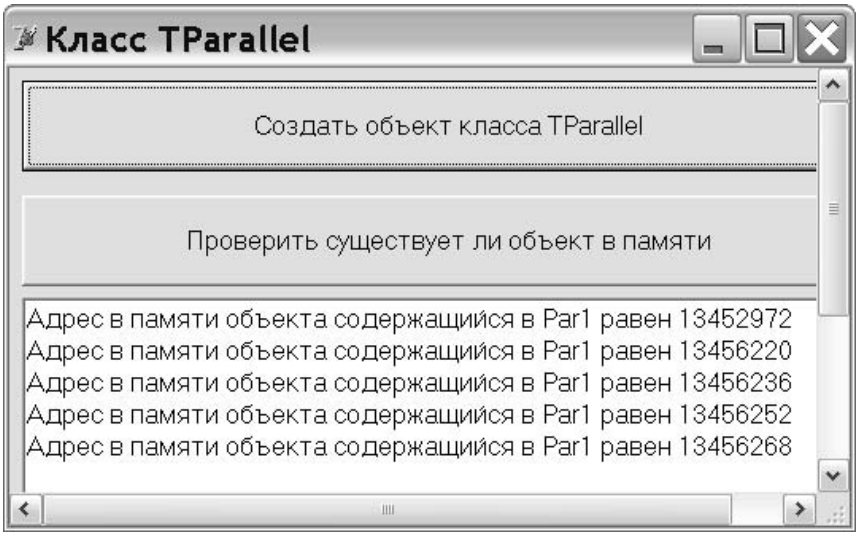


Рис. 14.6 Каждый раз при выполнении инструкции Par1:=TParallel.Create переменная Par1 инициализируется новым значением-адресом вновь созданного объекта

Таким образом, при очередном щелчке на кнопке в памяти каждый раз будет создаваться новый объект, на который указывает Par1, при этом старый объект, на который до этого указывал Par1, не уничтожится, а занимает место в памяти. После пяти щелчков будет создано 5 объектов. То есть в программе непосредственно сама переменная-указатель Par1 существует в единственном экземпляре, а объектов после пяти щелчков создано 5. Поэтому после пяти щелчков мы знаем только текущее значение, содержащееся в Par1, указывающее на 5-ый объект, то есть знаем адрес только пятого объекта, созданного на пятом щелчке, а адреса четырех предыдущих уже нет (при каждом щелчке значение в Par1 перезаписывалось новым), поэтому доступ к ним безвозвратно потерян.

Поэтому, если мы не хотим наплодить лишних объектов и избежать утечки памяти, нужно специально заботиться об этом и своевременно удалять объект, когда он не нужен методам. Удаляются объекты специальным методом-деструктором **Destroy**, или методом **Free**. Кроме того, каждый объект уже будет иметь разные значения полей. Так как при первом щелчке все переменные x, y, z имеют значение 1, то вызов метода Par1.Init(x,y,z) проинициализирует поля первого объекта как Fa=1, Fb=1 и Fc=1, при втором щелчке переменные x, y, z уже имеют значение 2, и поэтому у второго объекта поля уже проинициализируются как: Fa=2, Fb=2 и Fc=2 и т.д.

Далее покажем, что можно создавать массив объектов.

Объявите глобальный массив с именем Par класса TParallel на 5 элементов. как показано на рис. 14.7

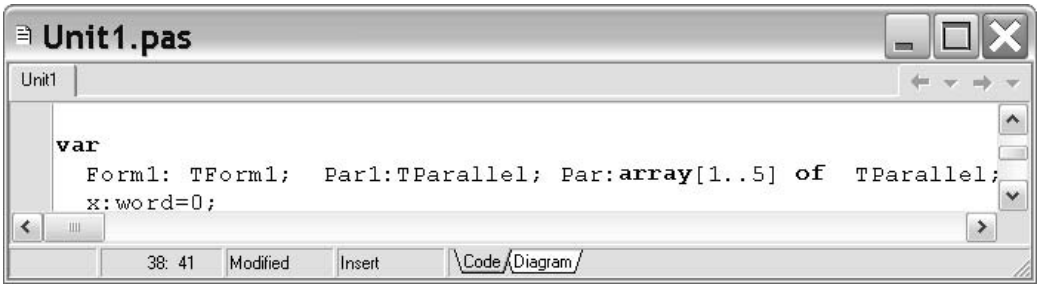


Рис. 14.7 Объявление глобального массива переменных-указателей на объекты класса TParallel

Следует обратить внимание на то, что первоначальная запись Par:array[1..5] of TParallel – это только лишь объявление массива ссылок на объекты данного класса, и никаких объектов в памяти еще нет. Это означает, что каждая из ссылок объявленного массива Par[1], Par[2], Par[3] и т.д. вначале содержит значение нулевого адреса, то есть nil.

Чтобы в памяти создать объекты вызова, к каждой ссылке в отдельности нужно применить метод-конструктор (инструкция вроде Par[i]:= TParallel.Create) или просто присвоить ей адрес уже имеющегося объекта, содержащегося в другой ссылке такого же класса (инструкция вроде Par[i]:=Par1).

Таким образом в обработчике щелчка на Button1 сразу после выражения Par1:=TParallel.Create вставьте инструкцию Par[x]:=Par1. То есть при каждом щелчке на кнопке Button1 будем присваивать очередному элементу массива Par[i] значение адреса объекта, на который указывает Par1.

В данном случае переменную x просто удобно использовать и как индекс массива, так как ее значение при каждом щелчке будет увеличиваться на 1.

Опишем работу программы при нескольких щелчках на кнопке Button1:

Первый щелчок: x=1 и Par1 указывает на первый объект, сохраняем этот адрес в элементе Par[1] – фактически выполнится Par[1]=Par1.

Второй щелчок: x=2 и Par1 указывает на новый(второй) объект, сохраняем этот адрес нового объекта в элементе Par[2] – фактически выполнится Par[2]=Par1. И так далее.

Далее после того как заполнили массив Par, при щелчке на Button3 выведем в Memo1 адреса и значения полей сохраненных объектов. Для этого создайте обработчик щелчка на Button3, который должен выглядеть следующим образом:

```
procedure TForm1.Button3Click(Sender: TObject);
var i:integer;
begin

for i:=1 to 5 do
Memo1.Lines.Add('Адрес объекта с номером '+IntToStr(i)
+' равен '+IntToStr( integer(Par[i]))+' Поле Fa = '+
IntToStr(Par[i].Fa) );

end;
```

Запустите программу. Обязательно 5 раз или более щелкните на Button1. А затем на кнопке «Вывести значение полей массива». Вы получите все сохраненные адреса объектов и их полей (в данном случае ради простоты выводим значение только одного поля Fa). **Вернитесь в режим программирования.**

Используя массив ссылок на объекты, получим значения не только полей, но и вызовем метод Show каждого объекта из массива.

Для этого измените содержимое обработчика щелчка кнопки Button3 на следующее:

```
for i:=1 to 5 do          begin
Memo1.Lines.Add('Адрес объекта с номером '+IntToStr(i)
+' равен '+IntToStr( integer(Par[i]))+' Поле Fa = '+
IntToStr(Par[i].Fa) );
Par[i].Show; //Вызываем метод Show очередного объекта из массива
                          end;
```

Запустите программу. Как и ранее, сначала создайте 5 объектов, щелкнув 5 раз на Button1, а затем один раз щелкните на Button3. Теперь для каждого объекта из массива Par будет вызван свой метод Show, работающий с полями и методами именно того объекта, для которого он вызван.

Самостоятельно добавьте методы очистки памяти от ненужных объектов. Используйте метод free.

Вернитесь в режим программирования

14.2. Наследование классов, класс TObject, назначение конструктора Create. Статические и виртуальные методы.

Классы могут наследоваться! Это один из фундаментальных принципов объектно-ориентированного программирования [2,3,5,6,8]. Сделано это по многим причинам, в том числе для уменьшения программного кода при разработке программ.

Действительно, один раз создав класс с определенными методами, намного легче объявить новый класс – наследник, который будет вбирать в себя всю функциональность родительского класса, то есть уже будет содержать все методы родительского класса, которые не нужно заново программировать.

Новый класс-наследник будет содержать все методы, поля и свойства класса родителя и иметь к ним доступ с некоторым возможным ограничением (за это отвечают спецификаторы доступа public, protected, private).

Если мы хотим объявить новый класс TForm1, являющийся наследником TForm, мы должны записать инструкцию TForm1 = class (TForm). В начале работы в окне редактора кода такая инструкция записана автоматически.

Таким образом, новый класс TForm1 вбирает в себя все свойства, возможности и всю функциональность класса TForm, а также может и скорее всего будет (специально для этого он и создан) дополнять их своими новыми.

В данном случае говорят, что класс TForm1 является производным классом от класса TForm, а класс TForm является базовым по отношению к классу TForm1.

В некоторых случаях можно говорить, что класс TForm – это родительский класс по отношению к дочернему TForm1. Говоря грубо, класс TForm – является «отцом» для класса TForm1, который является его «сыном».

Нужно отметить, что в **Delphi имеется главный базовый класс с именем TObject, а все остальные классы являются производными от него**. Данный класс TObject содержит стандартные методы, такие, как например конструктор Create и деструктор Destroy.

Таким образом, почти во всех классах Delphi имеются методы, унаследованные от TObject. Кроме того, по умолчанию, запись TParallel = class, которая использовалась выше, является совершенно эквивалентной TParallel = class(TObject), то есть класс TParallel является наследником класса Tobject, а это значит, что в Delphi невозможно создать класс, который не был бы потомком TObject. Поэтому и объект, созданный на основе класса TParallel,содержит все методы, в том числе Create, Free и деструктор Destroy, унаследованный от TObject.

Особого внимания заслуживает специальный метод конструктор, существующий в каждом классе и в большинстве случаев имеющий имя Create [8].

Метод-конструктор «Create» распределяет объект в динамической памяти и помещает в специальную переменную Self адрес этой памяти.

Одной из основных задач метода-конструктора является инициализация полей объекта. Метод-конструктор формирует объект в оперативной памяти, как правило вызывается единственный раз и производит с объектом какие-либо начальные действия. В основном эти действия сводятся к инициализации объекта, то есть установлению начальных значений полей объекта. Конструктор имеется у всех классов. Конструктор может иметь или не иметь параметры и может быть перегружен.

Например, при создании в памяти объекта - формы Form1 класса TForm1 вызывается конструктор Create данного класса TForm1. В тело конструктора можно записать какой-либо программный инициализирующий код, например, установить начальные значения определенных полей в единицу и т.д.

В общем случае конструктор может выполнять любые действия, но если он не связан напрямую с инициализацией, то считается, что это не самый лучший стиль программирования.

Чтобы создать объект, программист обязан явно вызвать метод-конструктор класса.

Также существует и **метод-деструктор «Destroy», производящий действия, обратные конструктору.**

Как правило. он освобождает ту часть динамической памяти, которую занимает объект, то есть уничтожает объект. В классе TObject имеется метод Free, проверяющий, существует ли объект в памяти, путем сравнения значения в Self с нулем, и если Self на что-то указывает (адрес ненулевой, то есть объект находится в памяти), то вызывается метод-деструктор

```
procedure TObject.Free;
begin
 if Self <> nil then
  Destroy;
end;
```

Часто, вместо вызова метода-деструктора, вызывается метод Free, который, в свою очередь, вызвает деструктор.

Задание 14.2.1 Будем инициализировать поля объекта класса TParallel с помощью стандартного метода конструктора Create, унаследованного от TObject вместо метода-функции Init

Откройте все тот же проект, в котором до сих пор работали, работа снова будет осуществляться в нем.

Первое. что требуется сделать, – это закомментируйте или вообще удалите в разделе interface из описания класса TParallel прототип метода Init и его реализацию в разделе implementation (рис. 14.8).

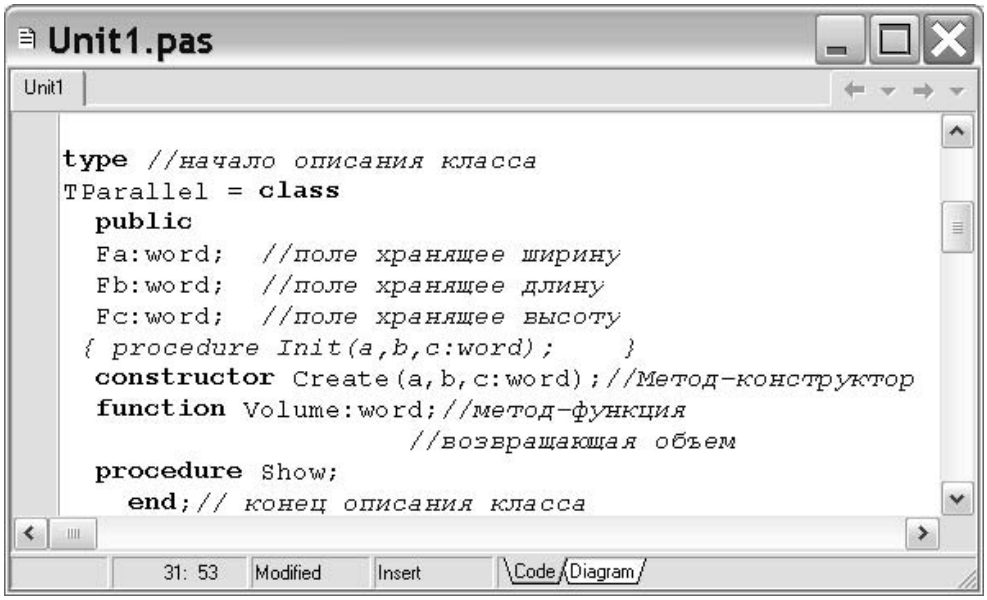


Рис. 14.8 Используем стандартный метод-конструктор Create для инициализации полей вместо функции Init

Далее, как обычно, нажмите сочетание клавиш Ctrl+Shift+C и получите заготовку метода-конструктора Create. Его содержимое будет совпадать с содержимым старой функции Init, которую уже убрали из класса, поэтому измените пустую заготовку конструктора следующим образом:

```
constructor TParallel.Create(a, b, c: word);
begin
Fa:=a; //присваиваем полю Fa значение a, введенное из внешней программы
Fb:=b; //присваиваем полю Fb значение b, введенное из внешней программы
Fc:=c; //присваиваем полю Fc значение c, введенное из внешней программы
end;
```

Теперь требуется изменить и обработчик щелчка на Button1, так как именно там мы создаем объект. Измените данный обработчик так (жирным шрифтом выделены изменения в программном коде):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
x:=x+1; y:=y+1; z:=z+1;
Par1:=TParallel.Create(x,y,z); {Методом конструктором размещаем объект в динамической памяти
и инициализируем поля объекта. Создаем объект.}
Par[x]:=Par1;
Par1.Show;
Memo1.Lines.Add('Адрес в памяти объекта, содержащийся в Par1, равен '
+IntToStr(integer(Par1)) );
end;
```

В данном случае метод инициализации Init уже не потребуется, инициализировать объект будет конструктор. В отличие от Init, метод-конструктор Create всегда будет вызываться единственный раз при создании объекта. Поэтому для инициализации лучше всего использовать конструктор, так как о методе Init можно случайно забыть, а метод-конструктор вы никогда не забудете, так как только с его помощью можно создать сам объект и заодно проинициализировать нужные поля.

Запустите программу. Ее работа должна быть совершенно такой же, как и прежде, до удаления метода Init. **Вернитесь в режим программирования.**

Задание 14.2.2 Наследование. Статические методы.

На основе класса TParallel как базового создадим производный от него класс Tbar, описывающий трехмерный брусок в форме параллелепипеда, имеющий плотность – как поле класса и массу – как метод класса

Здесь инструкция TBar = class(TParallel) означает, что класс TBar будет наследником класса TParallel, то есть унаследует всю функциональность TParallel и автоматически будет содержать и иметь доступ к полям Fa, Fb и Fc (он их унаследовал от родительского класса), а также методам volume и show, унаследованным им от базового (родительского) класса TParallel.

При этом, естественно, класс TBar будем наделять собственными полями и методами, в частности, новым методом massa, который будет возвращать значение массы. По этой причине в данном же проекте сразу же после описания класса TParallel в интерфейсной части опишите класс TBar следующим образом:

```
type
TBar = class(TParallel)// начало описания класса TBar
 public
 FRo:real; // новое поле содержит значение плотности
 constructor Create(a,b,c:word; Ro: real);// метод-конструктор
 function massa:real; //новый метод-функция, возвращающая массу
 procedure Show; { новый метод с тем же именем Show, перекрывающий старый унаследованный Show из TParallel}
 end;// конец описания класса TBar
```

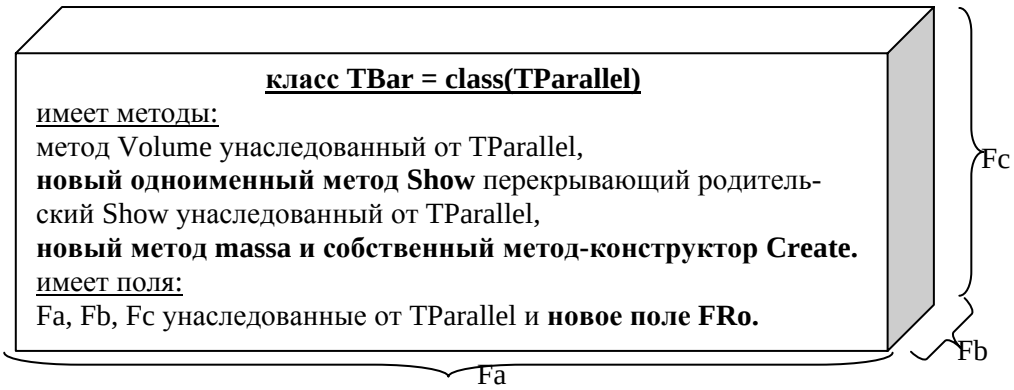


Рис. 14.9а. Схема описания класса TBar наследника класса TParallel

Затем, после того как вы описали класс TBar, как и ранее. нажмите сочетание клавиш Ctrl+Shift+C. Вы получите заготовку 3-ех методов: конструктора Create, новой принадлежащей классу TBar функции-метода Show, которая перекрывает унаследованную Show из TParallel, а также в классе TBar впервые вводится метод massa, возвращающий значение массы бруска.

Нужно обязательно уточнить что **в момент формирования объекта в памяти всегда должен вызываться сначала конструктор базового класса, инициализирующий базовую часть объекта, описанную родительским классом, а затем вызываться конструктор производного класса, отвечающий за инициализацию остальной части объекта. Деструкторы же вызываются в обратном порядке конструктором.**

То есть в новом конструкторе с тем же именем Create, но принадлежащиму производному классу Tbar, обязательно нужно сначала вызвать родительский конструктор с тем же именем Create класса TParalle для инициализации старой части объекта, описываемой классом TParallel, а затем продолжить инициализацию новой части объекта конструктором производного класса TBar.

Грубо говоря, старый конструктор будет вызываться из нового конструктора, и, когда он выполнит свою задачу, довыполнится остальная часть программного кода нового конструктора.

Вызов родительского метода предваряется ключевым словом **inherited**.

Теперь настало время сказать, что такое статический метод.

Если в производном классе объявлен метод с тем же именем, что и имеющийся в базовом классе, то этот новый одноименный метод производного класса отменяет старый метод родительского класса и для объекта производного класса будет вызываться новый метод.

В этом случае говорят, что новый одноименный метод производного класса перекрыл (заменил) старый одноименный метод родительского класса. Такие методы называются статическими. В Delphi все методы статические по умолчанию.

Если справа от объявления метода пусто и не стоит никаких директив (или единственная директива reintroduce), то это означает, что метод объявлен статическим.

Перекрытие одноименного статического метода означает его замену в классе-наследнике. То есть в классе-наследнике доступен будет уже новый метод вместо старого родительского.

Однако существуют еще виртуальные, динамические и абстрактные методы. Пока мы рассматриваем только статические методы.

В данном случае новый метод Show класса TBar будет вызываться вместо метода с тем же именем Show из класса наследника TParallel, то есть перекрывать (отрицать) старый метод. Продемонстрируем это.

Измените данные заготовки так, как показано ниже:

{**Метод конструктор Create класса TBar**}
constructor TBar.Create(a,b,c:word; Ro: real);
begin
inherited Create(a,b,c); {вызываем конструктор родительского класса TParallel и инициализируем старую часть объекта, описываемую этим классом, то есть устанавливаем значения полей Fa, Fb, Fc в значения соответственно a, b и c}

FRo:=Ro; {инициализируем новую часть объекта – поле Fro, содержащее значение плотности}
end;

{**Метод massa класса TBar** }
function TBar.massa: real;
begin
result:=FRo*Volume;{возвращаем значение массы, используя новое поле FRo и вызывая унаследованную от TParallel метод-функцию Volume для получения объема}
end;

{**Новый одноименный метод Show класса TBar будет вызываться вместо метода с тем же именем Show из класса наследника TParallel, то есть перекроет старый метод с тем же именем из TParallel**}
procedure TBar.Show;
begin
ShowMessage(‘Объем параллелепипеда равен ’ + IntToStr(Volume)+#10#13+
'Ширина – Поле Fa= '+IntToStr(Fa)+#10#13+
'Длина – Поле Fb= '+IntToStr(Fb) +#10#13+
'Высота – Поле Fc= '+IntToStr(Fc)+#10#13+
'Плотность – Поле FRo= '+FloatToStr(FRo)+#10#13+
'Масса = '+FloatToStr(massa));
end;

Таким образом, если бы в классе TBar не было нового метода с тем же именем Show, то класс TBar унаследовал бы метод Show от родительского класса TParallel и именно этот унаследованный метод вызывался бы объектом класса TBar. Но так как в класс TBar вводится новый статический метод с тем же именем Show, что имеется в родительском, то теперь в классе TBar существует именно новый метод, отменяющий старый, одноименный родительский метод класса TParallel. Покажем это на примере.

Далее сбросьте на форму еще одну кнопку Button4 и установите значение ее свойства Caption в «Создать объект класса TBar = class(TParallel)» (рис. 14.9б). Также объявите глобальный объект по имени Bar1 класса TBar. Как объявляются подобные объекты, вы уже знаете.

Рис. 14.9б Окончательный вид окна программы «Класс TParallel»

Создайте обработчик щелчка на Button4 и напишите там следующее:

Bar1:=TBar.Create(1,2,3,10.5);

Bar1.Show; {Вызывается метод Show класса TBar}

В данном случае значения 1,2 и 3 соответствуют полям старой части объекта, которые проинициализирует родительский конструктор, а значение 10.5 присвоится новому полю Fro – новой части объекта.

Запустите программу. Щелкните на Button4. В оперативной памяти создастся объект Bar1, все характеристики (значения полей, рассчитанная масса и объем) которого отобразятся на экране с помощью нового статического метода Show (рис. 14.10а) класса Tbar, заменяющего одноименный родительский статический метод Show, унаследованный от TParallel.

Вы получите следующий результат.

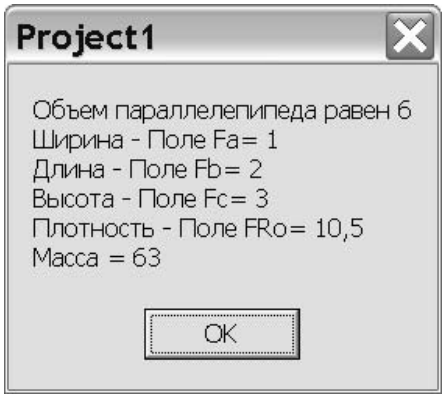


Рис. 14.10а Инструкцией Bar1.Show вызывается именно новый статический метод Show класса TBar. Этот новый статический метод перекрыл (заменил) метод с тем же именем Show из родительского класса TParallel.

Здесь мы видим значения всех полей объекта Bar1. в том числе и унаследованных от TParallel. Таким образом, новый одноименный метод в классе TBar дополнительно выводит значения нового поля-плотности FRo и значение массы, рассчитанной новым методом massa. **Вернитесь в Режим программирования.**

Задание 14.2.3 Покажем, что если убрать описание прототипа и реализацию нового статического метода Show в классе TBar, то метод Show унаследуется классом TBar от родительского TParallel и инструкцией Bar1.Show будет вызываться родительский унаследованный метод Show родительского класса TParallel.

Удалите (взяв в комментарии, он понадобится далее) прототип и реализацию метода Show класса TBar. Теперь метод Show описан только в классе TParallel и значит наследуется в полном объеме в производный класс TBar.

Запустите программу Щелкните на кнопке Button4.

Вы обнаружите, что вызывается старый, унаследованный метод Show родительского класса Tparallel, отображающий информацию только о трех полях и объеме (рис. 14.10б).

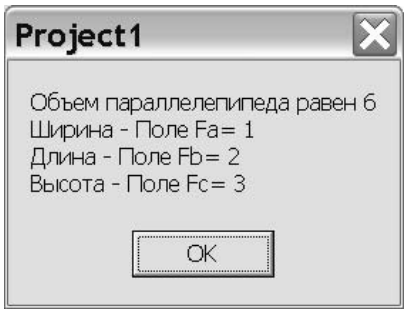


Рис. 14.10б А теперь инструкцией Bar1.Show снова вызывается старый родительский статический метод Show, унаследованный классом TBar от класса TParallel

Вернитесь в режим программирования.

Итак, если метод Show существует только в TParallel и не перекрывается одноименным методом в TBar, он просто наследуется классом TBar от TParallel.

Поэтому инструкция Bar1.Show сгенерирует вызов метода, унаследованного от TParallel.

Обратно снимите все комментарии с метода Show класса TBar, то есть возвратите новый метод Show классу TBar. Сейчас, если запустите программу, будет снова вызываться новый метод.

Теперь покажем, что можно вызывать любой метод родительского класса (в данном случае TParallel), просто имея указатель на объект производного класса (в данном случае Bar1).

В обработчике щелчка на Button4 допишите к уже написанному следующее:
TParallel(Bar1).Show;

И запустите программу. Данной инструкцией как раз и будет вызван метод Show родительского класса TParallel. **Вернитесь в режим программирования.**

Здесь мы просто даем указание компилятору рассматривать Bar1 как объект класса TParallel и вызываем его метод Show.
Запомните этот важнейший прием!

а) Сначала указываем родительский класс со скобками – TParallel();
б) Затем в скобки помещаем переменную-указатель производного класса – Bar;
в) С помощью оператора «.» получаем доступ к методу родительского класса – Show
Таким образом, мы продемонстрировали здесь наследование и перекрытие статических методов. Ниже мы более полно сформируем представление о статических методах, а пока дополним наши предыдущие рассуждения, направленные на восприятие идеи ООП, следующими.
В реальном мире определенный класс объектов обладает некоторыми общими чертами или свойствами. Например, такие объекты, как автомобиль, самолет, корабль, можно отнести к некоторому общему классу «транспорт», а объекты «автомобиль», «самолет» или «корабль» будут уже частными случаями класса «транспорт» или производными классами от класса «транспорт». С другой стороны, теперь уже класс «автомобиль» является общим для конкретных машин (производных классов от «автомобиль») определенных марок, например «Honda», «Toyota» и т.д.
Таким образом, можно составить следующую иерархию классов в виде таблицы:

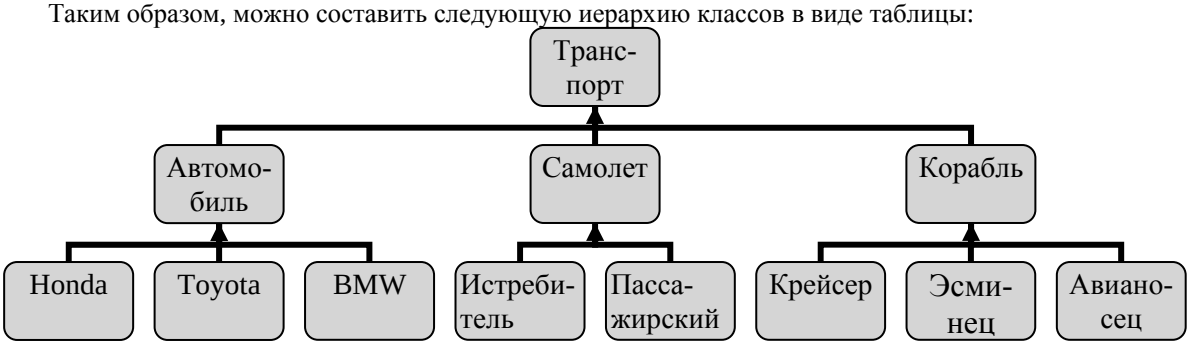


Диаграмма. 14.1 Иерархия наследуемых классов от родительского «Транспорт»

Становится ясно, что, в общем такие высказывания, как «автомобиль,- это вид транспорта», или «Toyota – это вид Автомобиля», или «Toyota – это вид вида Транспорта» справедливы. и поэтому некоторая переменная-указатель класса «Транспорт» может указывать не только на объекты своего класса «Транспорт», но и на объекты производного класса, например на объект класса «Автомобиль» или на объект класса«Toyota».
Переменная-указатель может указывать не только на объекты своего класса, но и на объекты любых производных классов!
Как же все этоэто выглядит в объектном программировании? Рассмотрим следующее описание:

```
type
TTransport = class// описание класса TTransport
end;

type
TAutomobile = class (TTransport) // описание класса TAutomobile
end;

type
TToyota = class (TAutomobile) // описание класса TToyota
end;

var     Transport:TTransport;
          Automobile: TAutomobile;
          Toyota: TToyota;
begin
```

Transport:=TTransport.Create;{создаем переменную-указатель Transport, указывающую на объект класса TTransport}

Automobile:=TAutomobile.Create; {создаем переменную-указатель Automobile, указывающую на объект класса TAutomobile }

Toyota:= TToyota.Create; {создаем переменную-указатель Toyota, указывающую на объект класса TToyota}

Transport:= Automobile;{теперь переменная-указатель Transport класса TTransport указывает на объект класса TAutomobile }

Transport:= Toyota; {теперь переменная-указатель Transport класса TTransport указывает на объект класса TToyota}

Automobile:= Toyota; {теперь переменная-указатель Automobile класса TAutomobile указывает на объект класса TToyota}

// Automobile:= Transport;{а **здесь ошибка, компилироваться не будет**}

end.

Однако обратное не всегда верно. Если попытаетнсь написать Automobile= Transport, получите ошибку, так как понятие «Транспорт» является более общим, чем «Автомобиль», и переменная-указатель типа «Автомобиль» не может указывать на объект класса своего родителя.

Другим словами, в реальном мире, фраза «Транспорт это Автомобиль» не является корректной в полном смысле этого определения. Транспорт – более общее понятие и не может являться только лишь одним автомобилем.

Задание 14.2.4 Самостоятельно. Создайте программу, добавив туда объявления только что описанных классов TTransport, TAutomobile и TToyota (пока без методов), переменных и данный программный код. Проверьте работу программы только на предмет правильной компиляции, а не на запуск.

Задание 14.2.5 Виртуальные методы. Сначала на простом примере покажем, как вообще работает механизм виртуальных методов, а затем попробуем его проанализировать

Виртуальные методы по сути реализуют концепцию полиморфизма одного из четырёх важнейших механизмов объектно-ориентированного программирования.

Полиморфи́зм (от греч. πολ - — много, и μορφή — форма) в языках программирования — возможность объектов с одинаковой спецификацией иметь различную реализацию. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций»

Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования [Википедия].

Полиморфизм – важнейший принцип объектно-ориентированного программирования, основанный на том, что некоторый элемент «общего» может проявляться по - разному в множестве «частных проявлениях» этого общего.

Допустим, что существует некое общее понятие «Движение» и его частные проявления, например, для транспорта будут выглядеть так: у автомобиля движение проявляется при вращении колес, у ракеты при выпуске струи газа, которая ее толкает, у лодки при вращении лопастей винта и т.д.

Одним словом, нечто общее проявляется по-разному в своих частных реализациях. Эта идея из объектов реального мира как раз и привнесена в объектно-ориентированное программирование. Иначе говоря, если мы устанавливаем некую переменную указатель «Транспорт» на какой либо объект производного класса, например «Корабль», а затем вызываем метод «Движение», то завращаются лопасти, а. если устанавливаем на объект «Ракета». будет выпущена струя газа. Просто указатель «Транспорт» “не знает”, как реализован метод «Движение» у того или иного объекта. на который он указывает. Он “узнает” об этом в момент фактического вызова метода объекта, то есть во время работы программы, а не на этапе компиляции.

Продолжим работу с уже готовым проектом, с которым вы работали сразу до задания 14.2.4. Откройте данный проект. Закомментируете там (но не удаляйте, он понадобится в дальнейшем) прототип и реализацию метода Show класса TBar, чтобы классом TBar наследовался именно старый метод Show из родительского класса TParallel.

В классе TParallel имеется метод с именем Volume, возвращающий значение объема, равное произведению трех полей Fa·Fb·Fc; естественно, этот метод наследуется и в класс TBar. Теперь точно так же, как и в случае с методом show, если мы определим новый метод с тем же именем Volume в классе TBar, то мы перекроем родительский метод и инструкцией Bar1.Volume вызываться будет именно новый метод.

Создадим этот новый метод Volume в классе Tbar, перекрывающий унаследованный.

Для этого создайте прототип метода Volume в классе TBar следующим образом:

type

```
TBar = class(TParallel)//начало описания класа TBar
 public
 FRo:real; // поле – плотность
 constructor Create(a,b,c:word; Ro: real);//Метод-конструктор
 function massa:real;//Метод-функция, возвращающая массу
 function Volume:word; {Новый метод, возвращающий объем, перекрывает унаследованный одноименный метод из класса TParallel }
 // procedure Show;{функция show теперь наследуется из базового класса TParallel}
  end;// конец описания класса TBar
```

И как обычно. нажмите сочетание клавиш Ctrl+Shift+C. Измените полученную заготовку реализации метода Volume класса TBar так:

```
function TBar.Volume: word;
begin
result:=Fa*Fb*Fc*100;
end;
```

То есть теперь, значение объема, возвращаемое новым методом, будет отличаться от значения, возвращаемого старым методом, в 100 раз большим значением.

Далее измените обработчик щелчка на Button4 следующим образом:

```
Par1:=TParallel.Create(1,2,3);
Par1.Show;
Bar1:=TBar.Create(1,2,3,10.5);
Bar1.Show;
```

Здесь мы специально для сравнения работы методов сначала вызовем метод Show объекта Par1 класса **TParallel**, а затем метод Show объекта Bar1 класса TBar наследника **TParallel**.

Также вспомним, что метод Show класса TParallel содержит в своем теле вызов метода Volume и выглядит так:

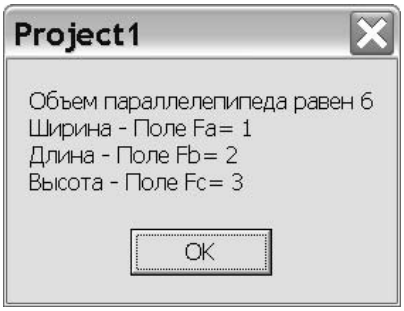
```
procedure TParallel.Show;
begin
ShowMessage('Объем параллелепипеда равен ' + IntToStr(Volume)+#10#13+
'Ширина – Поле Fa= '+IntToStr(Fa)+#10#13+
'Длина – Поле Fb= '+IntToStr(Fb) +#10#13+
'Высота – Поле Fc= '+IntToStr(Fc) );
end;
```

Для метода Show объекта Par1 класса TParallel все совершенно ясно, при щелчке на Button4, то есть при выполнении инструкции Par1:=TParallel.Create(1,2,3), поля установятся в значения: Fa=1, Fb=2, Fc=3, а при выполнении инструкции Par1.Show будет вызван метод Show класса Tparallel, и он, в свою очередь, вызовет метод Volume также класса Tparallel, возвращающий значение 1·2·3 – значение равное 6.

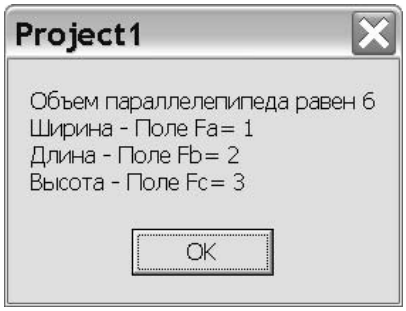
Теперь обсудим класс TBar.

Так как метод Show мы убрали из класса TBar (закомментировав его), то в данном случае при щелчке на Button4 будет вызываться старый метод Show, унаследованный из класса TParallel, но зато в класс TBar мы привнесли новый метод Volume, отменяющий унаследованный метод Volume от старого класса TParallel.

Таким образом, при щелчке на Button4, то есть при выполнении инструкции Bar1:=TBar.Create(1,2,3,10.5), поля установятся в значения: Fa=1, Fb=2, Fc=3, а при выполнении инструкции Bar1.Show мы ожидаем, что унаследованный метод Show от TParallel вызовет уже новый метод Volume, возвращающий значение 1·2·3·100, то есть значение. равное 600. Запустите программу и щелкните на кнопке Button4. Вы получите следующий результат (рис. 14.11). **Вернитесь в режим программирования.**



а) Par1.Show



б) Bar1.Show

Рис. 14.11 Результат работы программы до применения виртуальных или динамических методов

С объектом Par1 все ясно, и его метод Show работает, как и предполагалось (рис. 14.11а), выдавая значение объема, равное 6.

С объектом Bar1 дело обстоит несколько сложнее.

С функцией Show объекта Bar1 все в порядке, она вызывается унаследованной от класса Tparallel, как и должно быть, однако в случае с Volume снова вызывается метод родительского класса Tparallel, выдавая значение объема, равное 6, а не 600 (рис. 14.11б), вместо нового метода Volume, определенного в TBar.

Оказывается причина кроется в том, что мы имеем дело с **ранним связыванием** [5].

Вызов Volume в методе Show установлен раз и навсегда по версии, определенной в базовом классе TParallel. На этапе компиляции метод Show является методом базового класса и метод Volume разрешается относительно него (так как он из нее вызывается) как метод именно базового класса. Поэтому компилятор совершенно ничего не знает ни о каких других методах Volume, кроме базового класса TParallel.

В данном случае связь между Volume и Show происходит на этапе компиляции (**ранее,** чем выполняется сама программа), то есть до выполнения программы и вызов Volume из Show фиксирован.

Для того чтобы выбор метода не был фиксирован и производился во время работы программы, применяется **позднее или динамическое связывание** [5], основанное на виртуальных механизмах. То есть нам требуется, чтобы решение, о том, какой метод Volume будет вызван для конкретного объекта конкретного класса, принималось во время выполнения программы, а не на этапе компиляции. где вызов методов фиксирован.

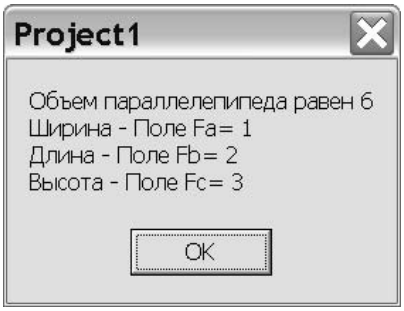
Подобные операции являются динамическими.

Для того чтобы реализовать нужное нам динамическое связывание, метод Volume базового класса нужно объявить виртуальным, используя ключевое слово **virtual,** и метод из статического превратится в виртуальный.

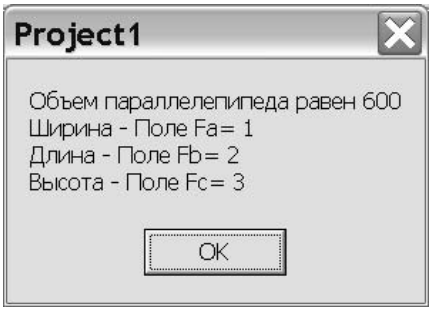
Таким образом, в базовом классе TParallel подпишите справа от прототипа метода Volume слово virtual, а в классе, в котором данная функция будет переопределена (overridding), подпишите справа от прототипа метода ключевое слово **override**, то есть описание обоих классов будет следующим:

```
type TParallel = class // Начало описания класса TParallel
 public
 Fa:word;
 Fb:word;
 Fc:word;
 { procedure Init(a,b,c:word);}
 constructor Create(a,b,c:word);
 function Volume:word; virtual; {объявляем метод Volume базового класса виртуальным}
 procedure Show;
   end; // Конец описания класса TParallel

type TBar = class(TParallel) // Начало описания класса TBar
 public
 FRo:real;
 constructor Create(a,b,c:word; Ro: real);
 function massa:real;
 function Volume:word; override; {переопределенный метод Volume}
// procedure Show;
  end; // Конец описания класса TBar
```

Запустите программу и щелкните на кнопке Button4. Вы получите следующий результат (рис. 14.12). **Вернитесь в режим программирования.**



а) Par1.Show



б) Bar1.Show

Рис. 14.12 Результат работы программы с применением виртуальных или динамических методов

Теперь все работает, и для объекта своего класса метод Show вызывает нужный метод Volume.

В данном контексте метод Volume является виртуальным в полном смысле этого слова потому, что содержащая его функция Show «понятия не имеет», какой именно вызовется метод, базового класса или производного или производного от производного и т.д.

Значит, если по факту создан объект класса TBar, то Show будет вызывать метод volume класса по факту имеющегося объекта, то есть вызовет метод volume класса TBar, а если фактически создан объект класса TParallel, то Show вызовет volume класса TParallel.

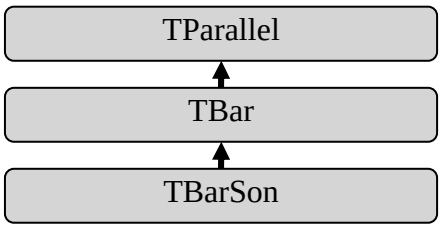
Ниже этот же метод продемонстрируем на примере возможности настраивать указатели родительских классов на объекты производных классов.

Переопределение (overriding) метода означает – расширять (продлевать) или совершенствовать метод, вместо того, чтобы заменить его. Класс – потомок может переопределить любой из его унаследованных виртуальных методов.

Переопределение метода имеет место только в том случае, если он объявлен виртуальным или динамическим. Переопределение виртуальных методов в других взаимо-наследующих классах имеет место только при полном совпадении имени метода и его параметров (одинаковая сигнатура).

Задание 14.2.6 Покажем, что механизм виртуальных методов работает и при многократном наследовании на примере 3-ех взаимонаследуемых классов

Объявите новый, третий класс TbarSon, являющийся наследником класса TBar (а следовательно. автоматически и наследником TParallel), в результате иерархия наследуемых классов станет таковой:



И в этом новом классе TbarSon, как и ранее, опишите новый метод Volume, а рядом снова подпишите ключевое слово override.

Теперь в общем описание классов будет выглядеть так:

```
type TParallel = class
//тут подробное описание класса
function Volume:word; virtual;
end;

type TBar = class(TParallel)
//тут подробное описание класса
function Volume:word; override;
end;

type TBarSon = class(TBar)
//тут подробное описание класса
function Volume:word; override;
end;
```

И как обычно нажмите Ctrl+Shift+C. Измените получившуюся заготовку реализации метода так:

```
function TBarSon.Volume: word;
begin
result:=Fa*Fb*Fc*1000;
end;
```

Затем объявите глобальную переменную с именем BarSon1 класса TBarSon и измените обработчик щелчка на Button4 так , чтобы он выглядел следующим образом:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
Par1:=TParallel.Create(1,2,3);
Par1.Show;
Bar1:=TBar.Create(1,2,3,10.5);
Bar1.Show;
BarSon1:=TBarSon.Create(1,2,3,10.5);
BarSon1.Show;
end;
```

Запустите программу и щелкните на кнопке Button4. Как и в предыдущем случае, для каждого объекта вызывается свой метод Volume. **Вернитесь в режим программирования.**

Важно запомнить, что **если в некотором базовом классе метод был объявлен как виртуальный, то он остается виртуальным во всех классах наследниках, даже наследниках классов наследников**.

Описанный механизм реализует виртуальные методы, но есть еще динамические методы, отличающиеся от виртуальных только механизмом реализации их вызова, но не функционально. Виртуальные методы более эффективны по времени выполнения, а динамические – по объему затрачиваемой памяти. С точки зрения написания программы, динамические методы реализуются так же, как и виртуальные, только вместо слова virtual, пишут ключевое слово **dynamic**.

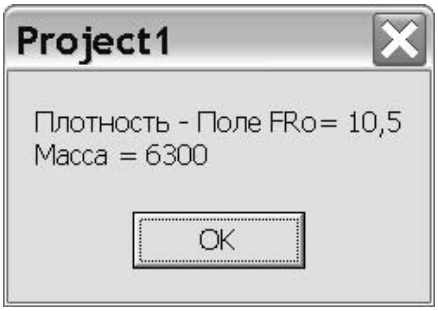
Задание 14.2.7 Продемонстрируем еще раз на примере готовых классов, что переменная-указатель может указывать не только на объекты своего класса, но и на объекты производных классов, а также то, что при этом механизм виртуальных методов работает нормально

Применим выше разобранный теоретический пример к нашим готовым классам TParallel, TBar и TBarSon.

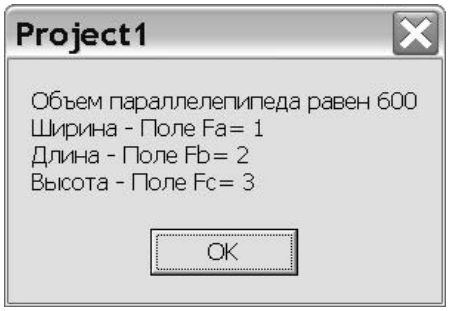
Измените обработчик щелчка на Button4 следующим образом:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
Par1:=TParallel.Create(1,2,3); {создаем переменную-указатель Par1, указывающую на объект класса TParallel }
Bar1:=TBar.Create(1,2,3,10.5); {создаем переменную-указатель Bar1, указывающую на объект класса TBar }
BarSon1:=TBarSon.Create(1,2,3,10.5);
Par1:=Bar1; {теперь переменная-указатель Par1 класса TParallel указывает на объект класса TBar}
ShowMessage('Плотность – поле FRo= '+ FloatToStr((Par1 as TBar).FRo)
+#10#13+'Масса = ' + FloatToStr((Par1 as TBar).massa) );
Par1.Show;{здесь сохраняется механизм виртуальных методов, и метод Show класса TParallel для расчета объема будет вызывать виртуальный метод Volume фактического объекта, на который указывает Par1, то есть объекта класса TBar}
end;
```

Запустите программу. Щелкайте на кнопке Button4. Вы получите следующие результаты (рис. 14.13):



а) Инструкции (Par1 as TBar).FRo и



б) Par1.Show (Par1 as TBar).massa

Рис. 14.13 Результат работы программы после инструкции Par1:=Bar1

Вернитесь в режим программирования.

После выполнения инструкции Par1:=Bar1 переменная-указатель Par1 указывает уже на объект производного класса TBar, то есть содержит адрес такой же, как и Bar1.

И когда мы стандартной функцией ShowMessage хотим вывести поле Fro, которое изначально описывается в классе-наследнике TBar, но его нет в классе-родителе Tparallel, которому принадлежит Par1, мы должны применить конструкцию (Par1 as TBar).

Эта инструкция как бы говорит компилятору: “Да, Par1 имеет тип класса TParallel, но давайте Par1 рассматривать как принадлежащий классу Tbar. В таком случае считаем, что Par1 указывает на объект класса TBar. Значит, мы имеем доступ к любому полю (Par1 as TBar).FRo объекта класса TBar и можем вызвать любой его доступный метод (Par1 as TBar).massa”.

Механизм виртуальных методов работает без изменений. Метод Par1.Show после Par1:=Bar1 по факту уже будет работать с той частью объекта класса TBar, которая описывается классом Tparallel, и сама инструкция Par1.Show подразумевает, что будет вызван метод Show класса Tparallel, если, конечно, он не является виртуальным и переопределенным (с ключевым словом override) в TBar. Таким образом, инструкцией Par1.Show вызовется метод Show базового класса TParallel, который, в свою очередь, вызовет Volume, который объявлен виртуальным во всех классах и поэтому вызовется Volume фактически имеющегося объекта, то есть объекта класса Tbar, на который по факту указывает Par1.

То есть Par1 может указывать на какой-либо объект своего или производного класса и вызвать виртуальный volume, но какой из методов volume вызовется? Своего или производного класса? И как он реализован? Указатель Par1 об этом “не знает”. А “узнает он” об этом только в момент фактического вызова метода. Если Par1 указывал на объект своего класса, то вызовется метод volume этого класса, если Par1 указывал на объект производного класса, то вызовется метод volume производного класса.

Если бы функции не были виртуальными, объем бы рассчитывался методом Volume базового класса TParallel и в данном случае был бы равен 6, а не 600.

В том случае, если бы мы не написали инструкцию Par1:=Bar1, то Par1 фактически так бы и указывал на объект своего класса TParallel, а не на объект класса TBar и при выполнении программы в реальном времени при исполнении инструкции (Par1 as TBar).FRo или (Par1 as TBar).massa сгенерировалась бы ошибка. Это и понятно, ведь у объекта класса TParallel нет поля FRo или метода massa, а мы пытаемся добраться до несуществующих методов и полей. Проверьте это.

Задание 14.2.8 Покажем, что с помощью одной инструкции можно сразу создавать переменные указатели, указывающие на объекты производного класса.

Измените обработчик щелчка на Button4 следующим образом:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
Par1:=TBar.Create(1,2,3,10.5); {переменная-указатель Par1 класса TParallel указывает на объект класса TBar }
ShowMessage('Плотность – поле FRo= '+ FloatToStr((Par1 as TBar).FRo)
+#10#13+'Масса = ' + FloatToStr((Par1 as TBar).massa) );
Par1.Show;

Par1:=TBarSon.Create(1,2,3,10.5); {а теперь Par1 указывает на объект класса TBarSon }
Par1.Show;
end;
```

Запустите программу. Щелкайте на кнопке Button4. Программа выдаст ожидаемые результаты. В данном случае инструкция Par1:=TBar.Create(1,2,3,10.5) фактически в оперативной памяти создаст объект класса TBar, а Par1 будет указывать на ту часть объекта Tbar, которая описывается классом TParallel.

Механизм виртуальных методов работает также без изменений. **Вернитесь в режим программирования.**

Задание 14.2.9 Используя виртуальный механизм, запрограммируем так, чтобы в одном случае инструкция Par1.Show вызывала метод класса TParallel, а в другом – TBar.

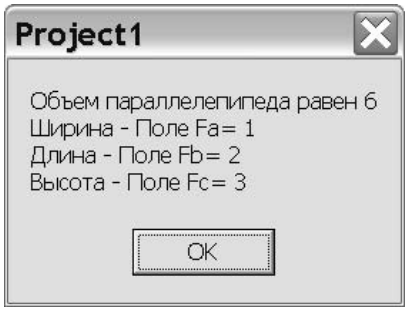
Здесь же в этом же проекте добавьте еще одну кнопку Button5 на форму и пишите программу в обработчике щелчка на ней. Верните назад (раскомментируете) метод Show в классе TBar и объявите этот метод виртуальным в классе TParallel и переопределенным в TBar. Теперь полное описание этих классов будет таким:

```
type TParallel = class public
 Fa:word;
 Fb:word;
 Fc:word;
 { procedure Init(a,b,c:word);  }
 constructor Create(a,b,c:word);
 function Volume:word; virtual; {объявляем метод Volume базового класса виртуальным}
 procedure Show;          virtual; {объявляем метод Show базового класса виртуальным}
   end;

type TBar = class(TParallel)
 public
 FRo:real;
 constructor Create(a,b,c:word; Ro: real);
 function massa:real;
 function Volume:word; override; {переопределенный метод Volume}
 procedure Show;          override; {переопределенный метод Show}
  end;
```

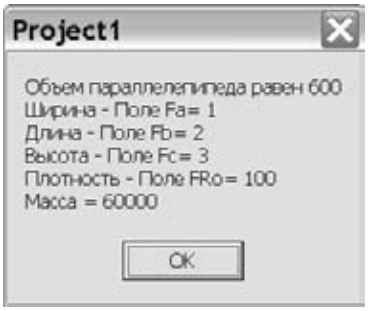
В обработчике щелчка на кнопке напишите следующее:

```
Par1:=TParallel.Create(1,2,3);{Par1 указывает на объект своего класса TParallel}
Par1.Show; {здесь вызывается метод Show класса TParallel }
Par1.Free;{освобождаем память}

Par1:=TBar.Create(1,2,3,100); {Par1 указывает на объект производного класса TBar}
Par1.Show; {а здесь вызывается метод Show класса TBar, так как метод переопределен}
Par1.Free; {освобождаем память}
```

Запустите программу на выполнение. Щелкайте на кнопке Button5. Вы получите следующие результаты (рис. 14.14):



а) Par1 указывает на объект класса Tparallel;



б) Par1 указывает на объект класса TBar

Рис. 14.14 Результат работы программы после выполнения инструкции Par1. Show

То есть будет вызываться виртуальный метод Show именно того класса, на объект которого фактически указывает Par1. Когда Par1 указывает на объект своего класса, то, естественно, вызывается Show класса TParallel, а когда Par1 указывает на объект производного класса TBar будет вызван виртуальный метод класса TBar.

Задание 14.2.10 Самостоятельно. Дополните класс TBarSon своим виртуальным, переопределенным (с ключевым словом override) методом Show и проверьте следующее верное утверждение: если

Par1 указывает на объект класса TBarSon, то инструкцией Par1.Show должен генерироваться вызов метода класса TBarSon.

Указание. Метод Show класса TBarSon должен отличаться от остальных методов с тем же именем у классов-предков. Например, пусть на экран выводится строчка “Show – override. TBarSon”.

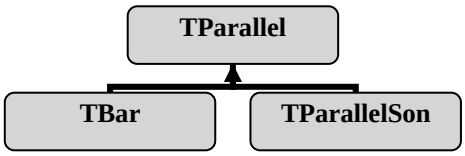
Задание 14.2.11 Покажем что замена ключевого слова virtual на ключевое слово dynamic не меняет результата работы программы

Везде в программе замените слово virtual на ключевое слово dynamic и запустите программу. Программа будет работать без изменений, как и раньше. **Вернитесь в режим программирования.** Отмените данные замены, везде вернув назад слово virtual.

Задание 14.2.12 Теперь на дополнительном примере дадим еще более полное понятие статического и виртуального методов и их различие.

Теперь, вследствие того, что мы знаем, что переменную-указатель можно настраивать на объекты производного класса, дадим более полное понятие статического и виртуального методов. Создайте новый проект. Опишем 3 новых класса; для простоты пусть имена двух классов снова будут TParallel, TBar, а имя третьего – TParallelSon.

Теперь класс TParallel будет родителем и для TBar и для TParallelSon. То есть иерархия классов станет такой:



Как и обычно, объявите данные классы в интерфейсной секции следующим образом:

```
type TParallel = class
public
procedure Show_Static;
procedure Show_Virtual; virtual;
end;

type TBar = class(TParallel)
public
procedure Show_Static;
procedure Show_Virtual; override;
end;

type TParallelSon = class(TParallel)
public
procedure Show_Virtual; override;
end;
```

Затем сбросьте на форму 2 кнопки: Button1 и Button2 и организуйте внешний вид приложения так, как показано на рисунке 14.15:

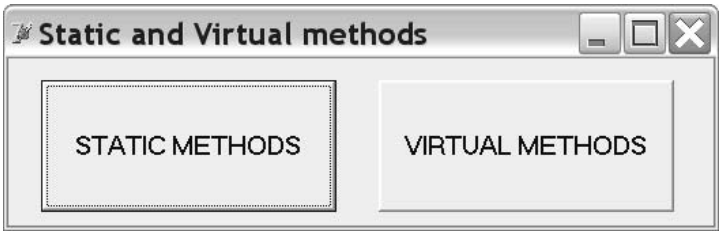


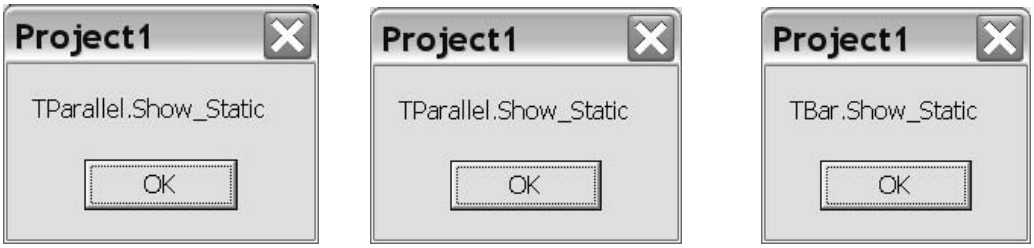
Рис. 14.15 Окно программы, иллюстрирующей возможности статических и виртуальных методов

Создайте обработчик щелчка на Button1, который должен выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var Parallel:TParallel; Bar:TBar;
begin

 Parallel := TParallel.Create; {Parallel указывает на объект своего класса TParallel}
 Parallel.Show_Static;  {вызовется TParallel.Show_Static}
```

```
  Parallel.Destroy;{очищаем память}

  Parallel := TBar.Create; {Parallel указывает на объект производного класса TBar }
  Parallel.Show_Static; {И снова вызовется TParallel.Show_Static }
  Parallel.Destroy; {очищаем память}

  Bar := TBar.Create;{Bar указывает на объект своего класса TBar}
  Bar.Show_Static;{А здесь вызовется TBar.Show_Static, так как Bar имеет класс TBar}
  Bar.Destroy; {очищаем память}
 end;
```

Внимательно вчитайтесь в комментарии, попробуйте предугадать, каков будет результат, а затем запустите программу на выполнение и последовательно щелкайте на Button1. Результат будет следующим:



а) Parallel→ TParallel *б) Parallel→ TBar* *в) Bar→ TBar*

Рис. 14.16 Результат после щелчка на кнопке «Static Methods» при очередном выполнении инструкции вызова метода Show_Static

Вначале, когда Parallel указывает на объект своего класса TParallel , то при исполнении инструкции Parallel.Show_Static естественно вызовется Show_Static класса TParallel.

Далее мы настраиваем Parallel на объект производного класса TBar, в котором также есть собственный статический метод Show_Static, и затем при исполнении инструкции Parallel.Show_Static мы ожидаем вызов именно этого нового метода. Но вызывается снова метод класса TParallel. Почему?

Ответ такой: метод является статическим. Это означает, что он не изменяется от класса к классу.

Не изменяется, когда наследуется классом-потомком и всегда будет выполнять одно и то же действие, несмотря на то, объектом какого класса он вызван, своего или производного.

Все дело в классе указателя, вызывающего метод.

При выполнении инструкции Parallel.Show_Static мы в первую очередь отталкиваемся от переменной-указателя, вернее, от класса (типа) переменной указателя, а в данном случае переменная-указатель Parallel принадлежит классу TParallel, следовательно будет вызываться Show_Static именно класса TParallel. И не важно, имеется ли в производном классе одноименный метод или нет.

Из кода с комментариями становится ясным, что для статических методов, куда бы ни указывал переменная-указатель Parallel, будь то объект класса TParallel или объект производного класса TBar, всегда будет вызываться метод Show_Static класса, к которому принадлежит Parallel, то есть метод класса TParallel.

Таким образом, **если метод является статическим, то всегда будет вызываться метод только того класса, к которому принадлежит переменная-указатель.**

Это основной вывод, который нужно хорошо уяснить.

При этом, если объявить статический метод в производном классе с тем же именем, что и статический метод в родительском классе, новый метод просто заменит унаследованный в производном классе. В данном случае, когда Bar настроен на объект своего класса TBar, который является производным от TParallel, вызовется уже новый метод Show_Static класса Tbar, заменивший одноименный метод из TParallel. Если бы собственного метода Show_Static в TBar не было бы, то вызвался бы унаследованный метод из Tparallel. Обязательно проверьте это путем удаления (закомментируйте, чтобы потом восстановить) данного метода из TBar.

В отличие от виртуального, статический метод нельзя переопределить, но можно просто заменить на другой одноименный. **Замена для статических методов означает отмену старого метода и введение в производный класс другого метода с тем же именем, что был в родительском. Эти методы никак не связаны, просто теперь в новом классе существует новый метод, заменивший старый,** старого метода как бы уже нет.

Пример статического метода можно взять из природы, например объявим класс-«обезьяна». Одному из методов этого класса можно назначить статический метод – «дышать воздухом», производный класс от «обезьяна», например, класс «человек» также унаследовал этот метод почти без изменений. То есть у человека имеется вдох, имеется выдох и т.д., поэтому этот метод можно считать статическим.

И снова введите (расскомментируйте) прежний собственный метод Show_Static в TBar, чтобы класс принял начальный вид. Создайте обработчик щелчка на Button2, который должен выглядеть следующим образом:

```
procedure TForm1.Button2Click(Sender: TObject);
var Parallel:TParallel;
begin
 Parallel := TParallel.Create; {Parallel указывает на объект своего класса TParallel это}
 Parallel.Show_Virtual; {значит, вызовется TParallel.Show_Virtual}
 Parallel.Destroy; {очищаем память}

 Parallel := TBar.Create;{Parallel указывает на объект производного класса TBar это}
 Parallel.Show_Virtual; {значит, теперь вызовется TBar.Show_Virtual }
 Parallel.Destroy; {очищаем память}

 Parallel:= TParallelSon.Create;{Parallel указывает на объект производного класса TParallelSon}
 Parallel.Show_Virtual;{значит, вызовется TParallelSon.Show_Virtual }
 Parallel.Destroy; {очищаем память}
end;
```

Снова внимательно вчитайтесь в комментарии, попробуйте предугадать, каков будет результат, а затем запустите программу на выполнение и последовательно щелкайте на Button1. Результат будет следующим:

а) Parallel→ TParallel *б) Parallel→ TBar* *в) Bar→ TParallelSon*

Рис. 14.17 Результат после щелчка на кнопке «Virtual Methods» при очередном выполнении инструкции вызова метода Show_Static

Полученный результат говорит о том, что в случае с виртуальными методами вызов метода определяется классом фактически имеющегося объекта в момент выполнения программы. Если Parallel указывает на объект своего класса TParallel, то при исполнении инструкции Parallel.Show_Virtual, естественно, вызовется Show_Virtual класса TParallel. Однако если Parallel указывает на объект производного класса TBar или TParallelSon, то будет вызван метод какого-либо из этих классов.

Из приведенного кода становится ясно, что, в отличие от статических методов, **виртуальный метод вызывается именно класса того объекта, на который фактически указывает указатель.**

Таким образом, впервые введя виртуальный метод в классе, в дальнейшем, в других классах, он может переопределяться в смысле совершенствования или расширения возможностей метода. Этот виртуальный метод как бы расширен своими переопределениями (вариациями) по взаимонаследующим классам.

Нужно помнить, что метод именно переопределен, а не заменен. В данном случае переопределение метода в очередном производном классе не является заменой, как в случае статических методов, а является именно некоторой модификацией или расширением все одного и того же целостного виртуального метода.

Пример из реального мира при использовании виртуальных методов мы приводили в задании 14.2.5**.**

Первоначально в классе объявляется виртуальный метод, а затем в следующем классе он переопределяется с директивой override. У нас имеется 2 переопределенных метода в соответствующих классах. В нашем случае один метод Show_Virtual в классе TParallel, а второй, переопределенный Show_Virtual, в классе-наследнике TBar. Можно было бы добавить и третий и т.д. взаимонаследующие классы с переопреденными в них методами.

Как и в случае со статическими методами, **имеется возможность заменить одноименный виртуальный метод новым**. Для этого требуется поставить справа от объявления одноименного метода специальную директиву **reintroduce** [8,10].

В целом в программировании виртуальные методы имеют немного более сложный механизм, по сравнению со статическими, и поэтому требуют более подробного рассмотрения. Данное рассмотрение продолжим в следующей практической работе.

ПРАКТИЧЕСКАЯ РАБОТА №15

15.1. Замена виртуальных методов. Цепочки Виртуальных методов. Директива reintroduce. Таблица виртуальных методов VMT

Нужно еще раз отметить, что в Object Pascal имеется важное ключевое слово **reintroduce** [10], означающее замену одноименного виртуального метода в производном классе. Когда мы в производном классе явно хотим **заменить** виртуальный метод базового класса и объявить новый метод с тем же именем в производном классе. мы должны использовать это ключевое слово (директиву) **reintroduce**. Дословно можно перевести: introduce- ввести, а reintroduce – повторно ввести, заново ввести.

Присутствие reintroduce подавляет предупреждения компилятора о замене предварительно заявленных виртуальных методов новым одноименным статическим или снова виртуальным методом.

В принципе, эту директиву можно также использовать не по прямому назначению, если мы хотим явно указать компилятору заменить одноименный статический метод в производном классе, хотя это в некоторых случаях просто может улучшить читабельность кода (или ухудшить), для программиста и для статических методов эта директива не предназначена.

15.1.1 Покажем, что можно использовать директиву reintroduce не по ее прямому назначению – для замены виртуального метода, а для замены одноименного статического метода в производнном классе.

То есть допустим имеется 2 класса: TFirst и его класс-наследник TSecond:

```
type
 TFirst = class
 public
 procedure Hello;
 end;

type
 TSecond = class(TFirst)
 public
 procedure Hello; {объявляем новый статический метод с тем же именем, заменяющий родительский }
 end;
```

Если описать классы указанным способом, то метод Hello в TSecond автоматически заменит старый метод Hello из TFirst. Если же добавить директиву reintroduce в TSecond, то результат будет прежним, однако мы явно сообщаем компилятору свое намерение заменить метод:

```
type
 TSecond = class(TFirst)
 public
 procedure Hello; reintroduce; {Явно для читабельности кода объявляем новый статический метод с тем же именем, заменяющий родительский}
 end;
```

Задание 15.1.2 С помощью ключевого слова reintroduce продемонстрируем, как перекрыть наследуемый виртуальный метод базового класса новым методом с тем же именем в производном классе.

Теперь применим директиву reintroduce по своему прямому назначению.

Создаете новый проект. Опишите в окне редактора кода, в интерфейсной секции модуля 7 простейших классов с соответствующими именами TA, TB, TC, TD, TE, TF и TG. То есть описание классов должно быть таким:

```
type
 TA = class
 public
 procedure show; virtual; {объявляем виртуальный метод Show}
end;
```

```
type
 TB = class(TA)
 public
 procedure show; override; {переопределяем виртуальный метод Show}
end;

type
 TC = class(TB)
 public
 procedure show; override; {переопределяем виртуальный метод Show}
end;

type
 TD = class(TC)
 public
 procedure show; reintroduce; {Заменяем (но не переопределяем) виртуальный метод Show ,унаследованный от TC новым собственным статическим методом с тем же именем }
end;

type
 TE = class(TD)
 public
 procedure show; virtual; {объявляем новый виртуальный метод Show}
end;

type
 TF = class(TE)
 public
 procedure show; override; {переопределяем виртуальный метод Show}
end;

type
 TG = class(TF)
 public
 procedure show; override; {переопределяем виртуальный метод Show}
end;
```

И как обычно, нажмите Ctrl+Shift+C. Перед вами сгенерируются заготовки всех методов данного класса. Измените их следующим образом:

```
{ TA }
procedure TA.show;
begin
ShowMessage('TA.show');
end;

{ TB }
procedure TB.show;
begin
ShowMessage('TB.show');
end;

{ TC }
procedure TC.show;
begin
ShowMessage('TC.show');
end;

{ TD }
procedure TD.show;
begin
ShowMessage('TD.show');
end;
```

```
{TE }
procedure TE.show;
begin
ShowMessage('TE.show');
end;

{ TF }
procedure TF.show;
begin
ShowMessage('TF.show');
end;

{ TG }
procedure TG.show;
begin
ShowMessage('TG.show');
end;
```

Таким образом, вызов метода Show просто укажет нам, метод какого класса был вызван фактически, путем выведения на экран информации функцией ShowMessage.

Далее объявите 7 глобальных переменных: A класса TA, B класса TB, C класса TC и т.д. следующим образом:

```
var
 Form1: TForm1;  A:TA; B:TB; C:TC; D:TD; E:TE; F:TF; G:TG;
```

Сбросьте теперь на форму кнопку Button1, изменив надпись на ней, как на рисунке 15.1.

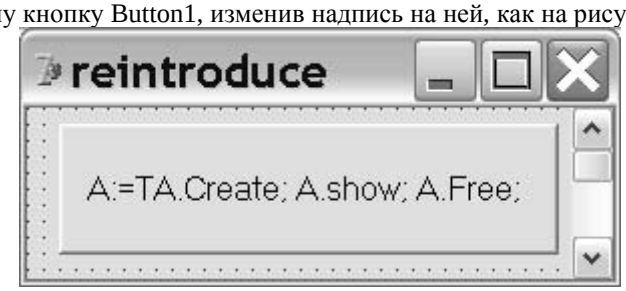


Рис. 15.1 Окно программы “reintroduce”

Запрограммируйте обработчик щелчка на ней следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
A:=TA.Create; A.show; A.Free;
A:=TB.Create; A.show; A.Free;
A:=TC.Create; A.show; A.Free;
A:=TD.Create; A.show; A.Free;
A:=TE.Create; A.show; A.Free;
A:=TF.Create; A.show; A.Free;
A:=TG.Create; A.show; A.Free;
end;
```

Запустите программу и щелкните на Button1. Вы получите следующий результат:

а)A→TA б)A→TB в)A→TC г)A→TD д)A→TE е)A→TF ж)A→TG

запись, например «A→TB» понимается так, что переменная-указатель A указывает на объект класса TB

Рис. 15.2 Результат щелчка на Button1 программы “reintroduce”

Вернитесь в **режим программирования**.

Здесь у нас везде во всех взаимо-наследующих классах метод show имеет одинаковую сигнатуру, поэтому он может переопределяться. Вернее сказать, параметров у этого метода вообще нет, но это также определенный вид сигнатуры.

Теперь попробуем подробно описать полученный результат.

Здесь инструкцией A:=TA.Create сначала создаем переменную-указатель A класса TA, указывающую на объект своего класса TA и вызываем метод Show. Все работает нормально и вызывается именно метод класса TA (рис. 15.2а). Затем мы освобождаем память, занятую этим объектом, используя метод Free и вновь аналогично устанавливаем переменную-указатель A класса TA на объект, но уже класса TB и снова вызываем метод Show. И опять все работает нормально и вызывается нужный метод Show класса TB так как там переопределен данный виртуальный метод. Далее снова устанавливаем A на объект TC и опять вызывается нужный метод Show, то есть класса TC. А вот когда A указывает на объект класса TD и во всех остальных случаях, когда A указывает на объекты классов TE, TF и TG, вызывается метод Show класса TC (рис. 15.2. г-ж.).

Дело в том, что в данном случае будет вызван виртуальный переопределенный метод класса фактического объекта, на который указывает указатель. Если в каком-то из последующих классов он был заменен (в данном примере в TD), то в этом классе и в последующих от него производных классах будет вызван последний переопределенный метод (который был в классе TC).

Здесь большую роль играет переменная-указатель. В данном случае все отсчеты производятся от его класса, то есть от TA.

В нашем случае в классе TD мы заменили (перекрыли) наследуемый виртуальный метод Show из родительского класса TC, на новый не виртуальный, обыкновенный статический метод с тем же именем Show, просто записав рядом ключевое слово reintroduce. То есть класс TD с этого момента «знает» только свой новый статический метод Show, который заменяет виртуальный метод класса-родителя TC, и поэтому сам класс TD и его последующие наследники TE, TF и TG как бы уже "не знают" (на самом деле информация о адресе метода сохраняется в таблице виртуальных методов) о том виртуальном методе Show, который переопределялся, начиная от TA до TC, так как метод Show в TD именно заменен новым статическим, а не переопределен.

При этом нужно учитывать, что в каждом классе имеется своя **таблица виртуальных методов VMT-Virtual Method Table**, **в которой описаны все виртуальные методы, как класса-наследника, так и класса-родителя, и вообще в ней хранятся адреса всех виртуальных методов класса, независимо от того, унаследованы ли они от класса-родителя либо перекрыты (заменены) в классе-наследнике, и поэтому информация о всех методах сохраняется** [8]**.**

И когда мы создаем переменную-указатель A класса TA (в котором началось объявление виртуального метода Show и последующее переопределение в TB и TC) и настраиваем A на объекты производных от TA классов, механизм виртуальных методов будет работать “относительно” именно класса TA, так как A принадлежит классу TA, а все вызовы методов мы производим используя переменную-указатель A (вот она важность класса указателя!).

Поэтому, с точки зрения класса TA, последний виртуальный метод Show (из цепочки переопределений метода, начавшейся в TA и закончившейся в TC) находится в классе TC, и именно этот класс TC является ближайшим родителем к цепочке классов TD-TG, и адрес виртуального Show класса TC будет записан в каждой VMT каждого из взаимо-наследующих классов TD-TG.

Именно поэтому инструкция A.Show, когда A указывает на объекты классов TD, TE, TF или TG будет генерировать вызов метода Show класса TC. (С точки зрения класса TA класс TC как бы является базовым классом последнего переопределения виртуального метода для цепочки TD-TG, где его уже нет, так как он заменен новым одноименным начиная с класса TD).

Другими словами, в некотором начальном базовом классе объявляется виртуальный метод со словом virtual, а в последующих взаимо-наследующих классах метод переопределяется со словом override; и как только в очередном классе встречается директива reintroduce, новый метод полностью отменяет виртуальный родительский, и следовательно данная цепочка виртуальных переопределенных методов типа virtual→ override→ …→override обрывается в этом классе (рис. 15.3).

То есть когда переменная-указатель A базового класса TA указывает на объект своего или производного класса, который еще входит в цепочку virtual→ override→ …→override, то соответствующие одноименные виртуальные методы будут вызываться в нормальном порядке. А как только переменная-указатель A указывает на класс, где впервые цепочка virtual→ override→ … →override обрывается (родительский виртуальный метод полностью заменен благодаря наличию единственной директивы reintroduce) или A указывает на последующий от него любой класс-наследник, то всегда будет вызываться виртуальный метод класса конца цепочки (метод того класса, где находится последняя директива override в цепочке).

Сам отсчет цепочки происходит относительно любого класса, который мы считаем базовым (в данном случае относительно TA, так как мы создаем переменную-указатель A этого класса с целью установки ее на объекты производных от TA классов). В нашем случае в базовом классе TA впервые был объявлен виртуальный метод Show, который, начиная от TA и вплоть до класса TC ,был переопределен (рис. 15.3), то есть

цепочка виртуальных методов virtual→ override→ override начинается в классе TA, продолжается в классах TB и TC и обрывается в классе TD, в котором наследуемый виртуальный метод заменен собственным, обычным статическим методом благодаря директиве reintroduce.

Поэтому когда A указывает на объекты классов TA, TB или TC все работает нормально и вызывается виртуальный метод того класса который соответствует объекту на который фактически указывает A.

Но как только последний виртуальный метод класса-родителя TC в классе-наследнике TD перекрыт (заменен), собственным статическим (благодаря единственной директиве reintroduce), в классе TD обрывается цепочка виртуальных методов, начавшаяся в классе TA. Поэтому инструкцией A.Show, когда A указывает на объект класса TD или классов его потомков TE, TF или TG, будет генерироваться вызов виртуального метода класса соответствующего концу цепочки, то есть вызовется метод класса TC. Именно в классе TC находится последний override из цепочки начавшейся в классе TA (отсчитываем цепочку от TA), а в TD она обрывается.

Если A указывает на объект какого-либо производного класса, где виртуальный метод заменен (классы TD – TG) другим одноименным, то это не значит, что сам A «не помнит» об этом методе. Сам A «помнит» именно только свой один метод (начавшийся в своем классе TA, рассмотрение идет относительно TA) с данным именем, который переопределялся, но закончился (был заменен) в одном из классов с введением туда нового одноименного. Этот другой метод хоть и с тем же именем указатель A «знать» не будет, так как он знает только свой метод в пределах целостной, не прерванной цепочки virtual→ override→ … →override. Именно поэтому, если указатель класса TA настроен на объекты классов TD-TG, будет вызван унаследованный и перекрытый метод последнего переопределения в классе TC. Фактически в данных классах он будет вести себя аналогично статическому, то есть в классах TD-TG он уже не меняется, а просто наследуется ими в скрытом виде.

Здесь, если рассматривать фундаментально, то фактически оба метода [10] существуют в классе TD: и виртуальный метод, унаследованный от TC, и новый скрывающий его статический метод. Просто виртуальный метод от TC унаследован в TD без переопределения там (без директивы override), при этом он одновременно скрыт новым статическим методом. Таким образом, в классе-потомке TD имеется оба метода, где имя метода теперь статически связано, благодаря перекрытию новым статическим методом, и «официально» присутствует новый метод заменяющий старый.

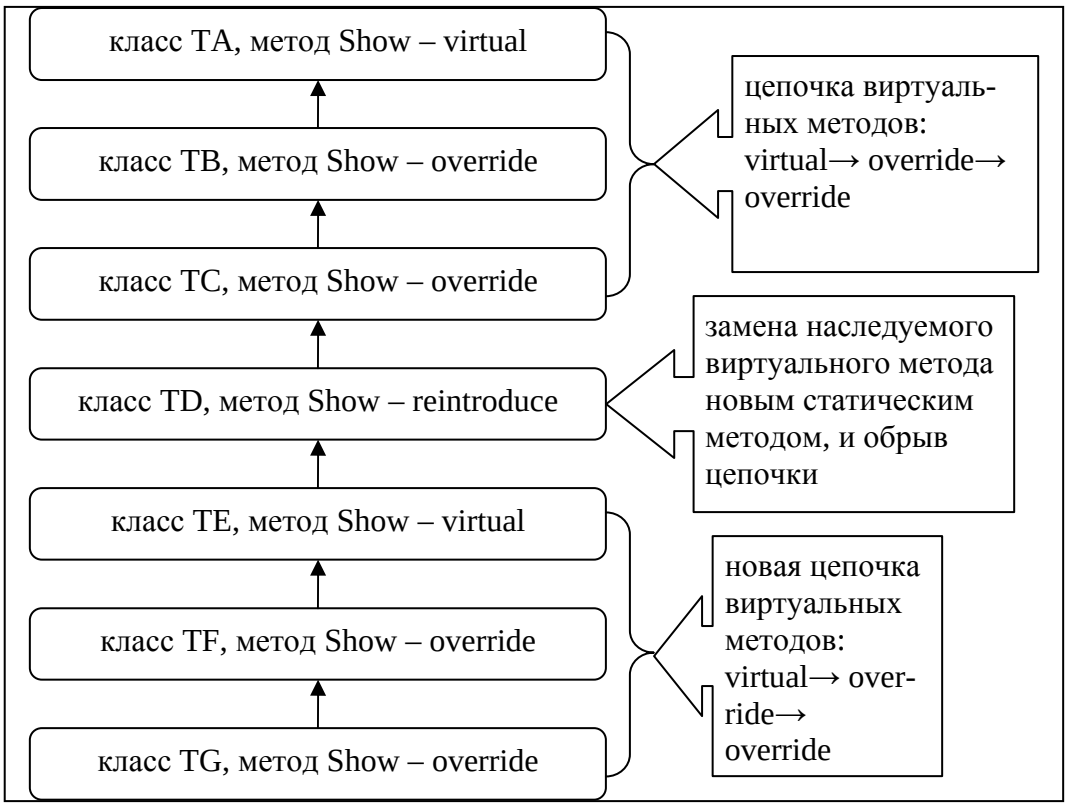


Рис. 15.3 Схема методов Show во взаимо-наследующих классах

Далее сбросьте еще 2 кнопки Button2 и Button3 на форму. Заголовок Button2 должен быть “B:=TB.Create; B.show; B.Free;” а, заголовок Button3 – “D:=TD.Create; D.show; D.Free;”.

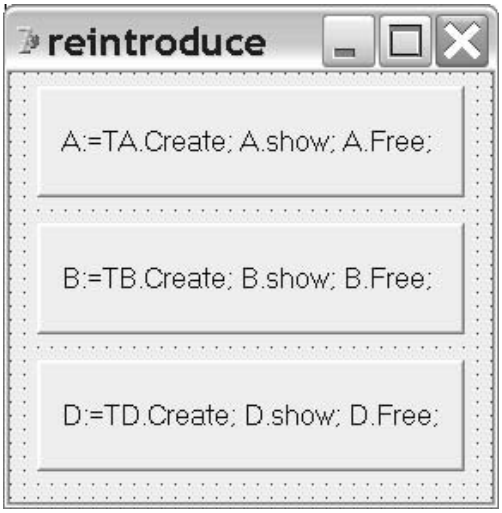


Рис. 15.4 Окно программы «reintroduce»

В обработчике щелчка на Button2 напишите следующий программный код:

```
    B:=TB.Create; B.show; B.Free;
B:=TC.Create; B.show; B.Free;
B:=TD.Create; B.show; B.Free;
B:=TE.Create; B.show; B.Free;
```

Запустите программу. Вы получите следующий ожидаемый результат:

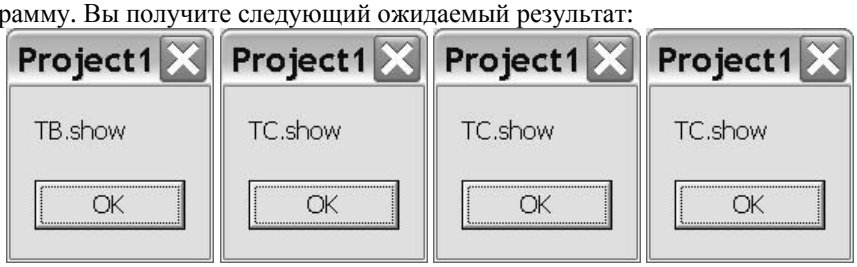


Рис. 15.5 Результат работы обработчика щелчка на Button2

Вернитесь в режим программирования.

Здесь логика работы программы та же, что и в предыдущем случае, только базовым классом выбран класс TB указателя B.

Начало отсчета виртуальной цепочки можно производить от любого класса, который мы выбрали базовым, где виртуальный метод уже переопределен с директивой override, не обязательно выбирать класс (TA или TE рис. 15.3), где метод изначально объявлен виртуальным с директивой virtual или dynamic. В классе TB мы начали отсчитывать виртуальную цепочку от слова override, и оборвалась она в классе TD.

Задание 15.1.3 Покажем, что несмотря на замену одноименного метода его вновь можно объявить виртуальным

Используя директиву reintroduce, любой одноименный виртуальный метод из родительского класса можно заменить в классе-наследнике и вновь объявить его виртуальным в этом же классе.

Измените классы TD и TC следующим образом:

```
type
 TD = class(TC)
 public
 procedure show; reintroduce; virtual;;
 end;

type
 TE = class(TD)
 public
 procedure show; override;
end;
```

И тогда цепочка методов во взаимо-наследующих классах будет выглядеть так:

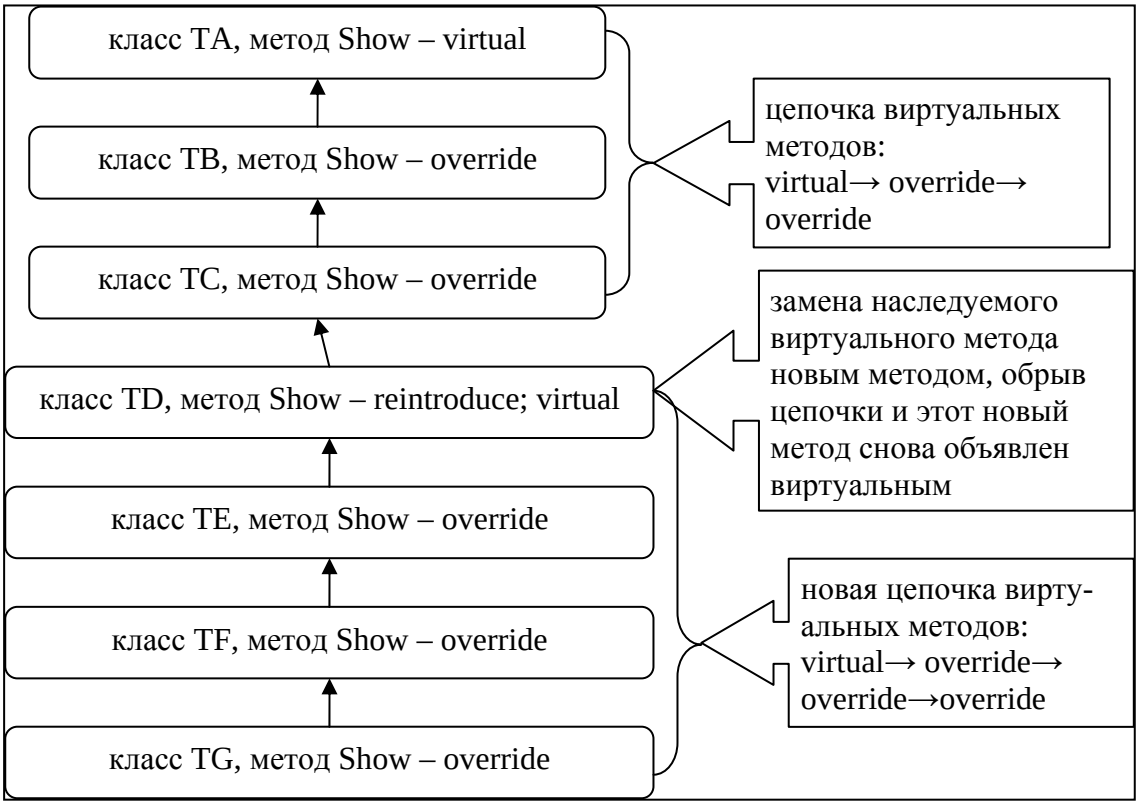


Рис. 15.6 Схема методов Show во взаимо-наследующих классах. Замененный метод в классе TD вновь объявлен в нем виртуальным.

Напишите в обработчике щелчка на кнопке Button3 следующее:
A:=TA.Create; A.show; A.Free;{вызывается Show класса TA}
A:=TB.Create; A.show; A.Free; {вызывается Show класса TB}
A:=TC.Create; A.show; A.Free; {вызывается Show класса TC}
A:=TD.Create; A.show; A.Free; {вызывается Show класса TC}
A:=TE.Create; A.show; A.Free; {вызывается Show класса TC}
D:=TD.Create; D.show; D.Free; {вызывается Show класса TD}
D:=TE.Create; D.show; D.Free; {вызывается Show класса TE}
D:=TF.Create; D.show; D.Free; {вызывается Show класса TF}
D:=TG.Create; D.show; D.Free; {вызывается Show класса TG}

Запустите программу, щелкайте на Button3. Для инструкций, связанных с переменной-указателем A класса TA, все работает так же, как и раньше (рис. 15.2а,б,в,г,д).

Несмотря на то, что в классе TD метод Show объявлен виртуальным, он, в первую очередь, является новым виртуальным замененным методом (рядом reintroduce) относительно родительского виртуального метода класса TC. И цепочка виртуальных методов, отсчитываемая от класса TA, так же как, и раньше. обрывается на TD. Результат последних 4-ех щелчков на кнопке Button3 отображен на рис. 15.7.

Создав переменную-указатель D класса TD и настраивая D сначала на объект класса TD, а затем и на производные от него TE, TF и TG, получим ожидаемый результат:

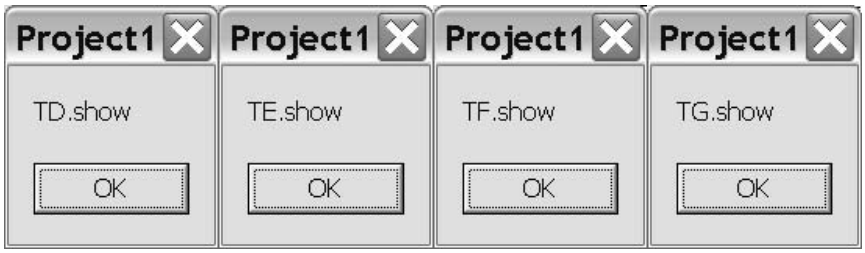


Рис. 15.7 Результат выполнения 4-ех последних строчек кода обработчика щелчка на Button3

Здесь мы создаем переменную-указатель D класса TD, по очереди настраивая ее на объекты производных классов, поэтому цепочку виртуальных методов отсчитываем от этого класса TD. **Вернитесь в режим программирования.**

Задание 15.1.4 Покажем, что если из последовательности reintroduce; virtual убрать директиву reintroduce, то это не изменит результата работы программы

Уберите директиву reintroduce из метода Show класса TD, а оставьте только virtual.

Теперь нажмите Ctrl+F9. Вы увидите предупреждение компилятора о том, что метод Show перекрывает (скрывает) виртуальный метод базового класса TC.

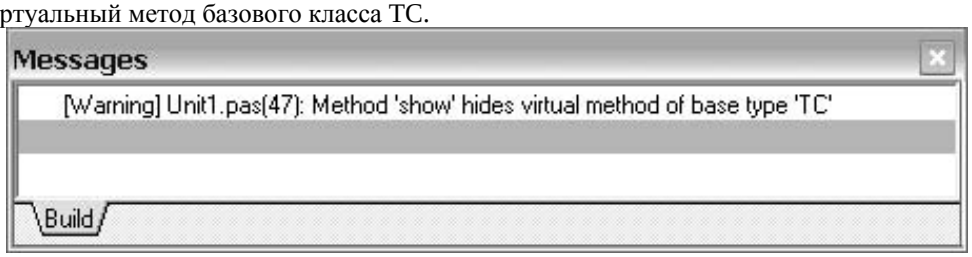


Рис. 15.8

Запустите программу. Щелкайте на Button3, программа будет работать без изменений. **Вернитесь в режим программирования.**

Дело в том, что сама по себе директива virtual (или dynamic), без reintroduce, в производном классе заменяет одноименный метод родительского класса в производном классе, разрывает предыдущую цепочку виртуальных методов и начинает новую. Одним словом, если пишем одну из директив: virtual или dynamic, компилятор предупреждает вас о том, что в классе-наследнике произойдет замена (перекрытие) метода, который вновь будет объявлен виртуальным или динамическим.

А в случае, **если пишем** одну директиву **reintroduce, то мы явно указываем компилятору на свои намерения – заменить родительский виртуальный метод новым** статическим, но если после reintroduce, сразу же еще подписать virtual, то метод будет заменен и вновь объявлен виртуальным в этом же классе, как если бы просто написали один virtual.

Если в классе-наследнике мы ничего не пишем рядом с новым одноименным методом, то это значит, что в этом классе-наследнике вводится новый статический метод, который, заменяет одноименный старый статический метод родительского класса, либо виртуальный метод родительского класса. То же будет, если написать один reintroduce, просто уже не будет предупреждений со стороны компилятора о замене метода. В этом случае компилятор не выдает никаких предупреждений о замене виртуального метода, компилятор уже проинформирован программистом, благодаря директиве reintroduce.

Директива reintroduce указывает компилятору, что программист понимает, что делает и он обойдется без предупреждений с его стороны.

Задание 15.1.5 Покажем, что если в классе-наследнике виртуальный метод заменить (с директивой reintroduce) и сразу же его переопределить (с директивой override), то метод будет считаться обыкновенным виртуальным переопределенным относительно родительского класса методом, как если бы директивы reintroduce не было. То есть последовательная запись директив reintroduce, override не разрывает цепочки виртуальных методов.

Будем использовать уже готовый код обработчика щелчка на Button1, где переменная-указатель A класса TA указывает на все объекты производного класса.

Измените описание метода Show в классах TD, TE и TC так:

```
type
 TD = class(TC)
 public
 procedure show; reintroduce; override;
 end;

type
 TE = class(TD)
 public
 procedure show; override;
end;

type
 TF = class(TE)
 public
 procedure show; virtual;
end;
```

И теперь цепочка методов во взаимо-наследующих классах будет выглядеть так:

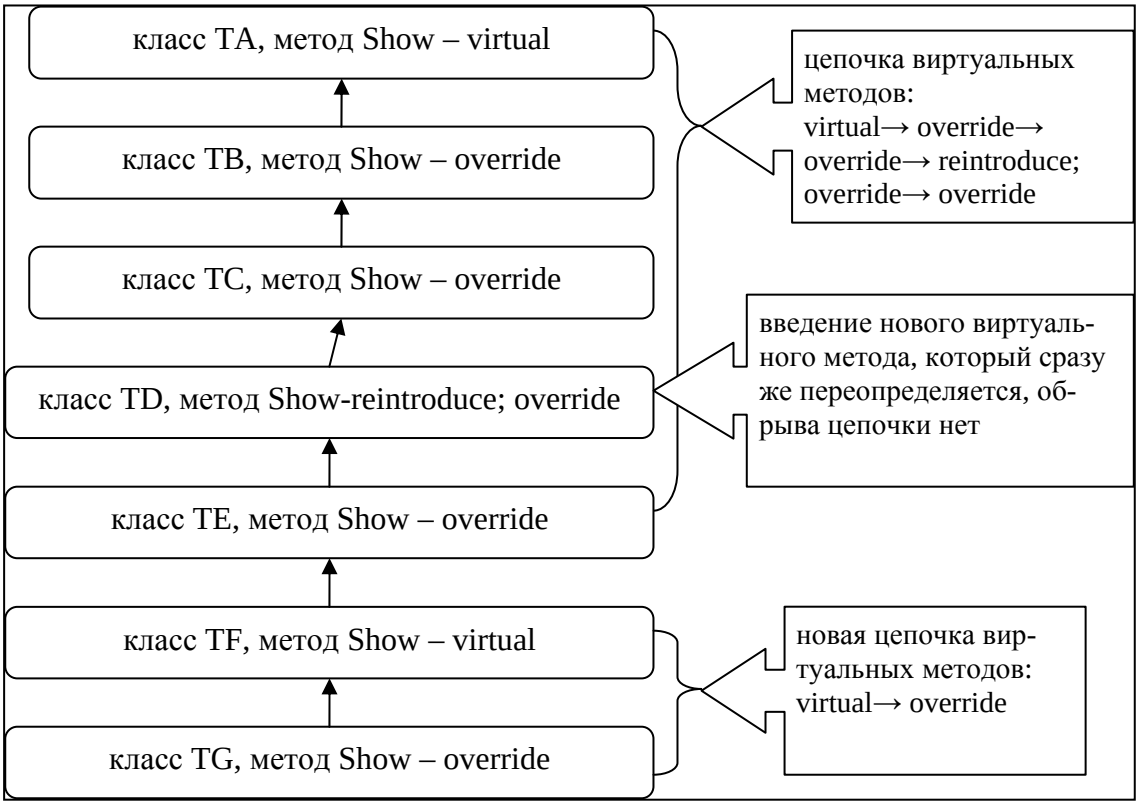


Рис. 15.9 Схема методов Show во взаимо-наследующих классах. Замененный метод в классе TD объявлен в нем переопределенным относительно метода в TC.

Запустите программу и щелкните на Button1. Вы получите следующий результат:

а)A→TA б)A→TB в)A→TC г)A→TD д)A→TE е)A→TF ж)A→TG

здесь для упрощения запись, например «A→TB» понимается так, что переменная-указатель A указывает на объект класса TB

Рис. 15.10 Результат щелчка на Button1 с учетом сделанных изменений в классах

Таким образом, начавшаяся цепочка виртуальных методов в классе TA продолжилась до класса TD, в котором комбинация слов reintroduce; override как бы говорит компилятору следующее: «Да, метод заменяется (reintroduce), но мы его сразу переопределяем как виртуальный (override) относительно родительского класса TC» – и цепочка в TD не обрывается, продолжаясь до TE включительно. То есть последовательная запись директив reintroduce, override не разрывает цепочки виртуальных методов.

15.2. Перегрузка статических и виртуальных методов в разных взаимонаследующих классах

Как известно, если в классе-наследнике введен статический метод с тем же именем (параметры при этом могут отличаться), что и у родительского класса, то он просто заменяет метод родительского класса. Однако если мы хотим вызывать методы с одинаковым именем и разными параметрами, принадлежащим разным классам (классу-наследнику и классу-родителя), мы можем перегрузить методы, как статические, так и виртуальные, используя ключевое слово **overload** как и при обычной перегрузке методов, принадлежащих одному классу.

Задание 15.2.1 Покажем, что можно перегружать статические методы в классе-наследнике с методами в классе-родителе

Создайте новый проект.

В окне редактора кода интерфейсной секции объявите 3 взаимо-наследующих класса T1, T2 и T3 следующим образом:

```
type
 T1 = class
  procedure Test(I: Integer);
 end;

 type
 T2 = class(T1)
  procedure Test(S: string);
 end;

 type
 T3 = class(T2)
  procedure Test(I: Integer);
 end;
```

И как обычно, нажмите сочетание Ctrl+Shift+C. Измените появившиеся заготовки реализации методов так:

```
{T1}
procedure T1.Test(I: Integer);
begin
ShowMessage('T1.Test(I: Integer)');
end;

{T2 }
procedure T2.Test(S: string);
begin
ShowMessage('T2.Test(S: string)');
end;


procedure T3.Test(I: Integer);
begin
ShowMessage('T3.Test(I: Integer)');
end;
```

Здесь, как и ранее, каждый метод при вызове будет обозначать себя путем вывода на экран своего имени и параметров.

Создайте 3 глобальных переменных T1_ класса T1, T2_ класса T2, и T3_ класса T3. То есть их объявление в интерфейсной части будет таким:

```
var
 Form1: TForm1;  T1_:T1; T2_:T2; T3_:T3;
```

Далее скиньте кнопку Button1 на форму и напишите в обработчике щелчка следующее:

```
T2_:=T2.Create;
T2_.
```

После того, как написан оператор доступа к объекту «.», на экране отобразится информация о том, что в классе T2 существует только один метод с именем Test и параметром S: string:

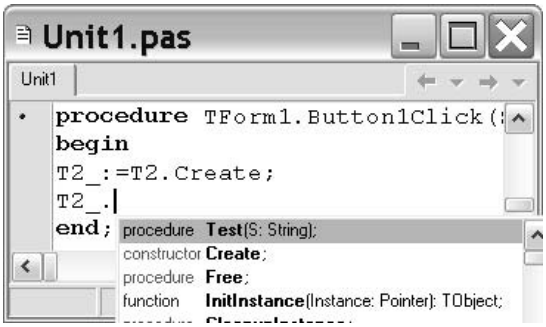


Рис. 15.11 В классе T2 имеется только один статический метод – скрывающий одноименный родительский

Это естественно, ведь теперь в классе T2 родительский статический метод заменен новым с тем же именем, хотя и с другими параметрами и «официально» класс T2 не знает ни о каком другом метод, кроме о T2.Test(S: string).

Однако мы хотим, чтобы класс T2 имел доступ к методу родительского класса T1.Test(I: Integer) в качестве перегруженного, с имеющимся своим новым T2.Test(S: string). Для этого в интерфейсной секции подпишите ключевое слово overload справа от каждого из методов соответствующего класса:

```
type
 T1 = class
  procedure Test(I: Integer);overload;
 end;

 type
 T2 = class(T1)
  procedure Test(S: string);overload;
 end;
```

Измените теперь содержимое обработчика щелчка на следующее:

```
T2_:=T2.Create;
T2_.Test('Метод вызывайся!');
T2_.Test(1000);
T2_.Free;
```

В данном случае при написании оператора доступа к объекту-«.», на экране отобразится информация о том, что в классе T2 существует уже не один, а два перегруженных метода с именем Test, но разными параметрами: String и Integer (переменная T2_ принадлежит классу T2).

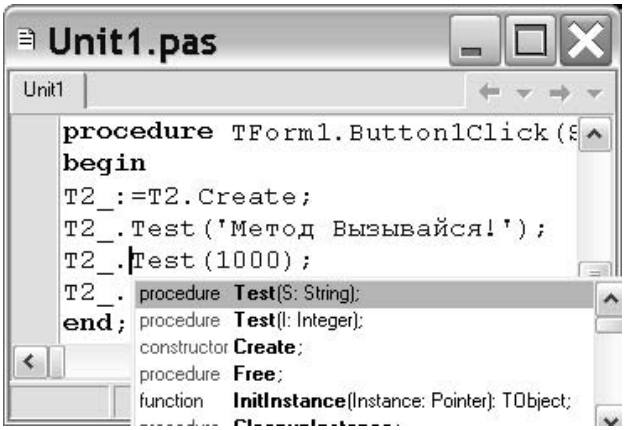


Рис. 15.12 В классе T2 имеется 2 одноименных перегруженных статических метода, один из них благодаря перегрузке унаследован без перекрытия из родительского класса T1

Запустите программу. Щелкните на Button1. По очереди вызовутся два перегруженных метода T2.Test(S: string) и T1.Test(I: Integer). **Вернитесь в режим программирования.** Закомментируйте инструкции T2_.Test('Метод вызывайся!'), и

T2_.Test(1000), чтобы, в целях дальнейшего изложения, не было сообщений об ошибке от компилятора.

Задание 15.2.2 Покажем, каким образом можно перегружать виртуальные методы в разных взаимонаследующих классах и проанализируем, как при этом происходит их наследование

При использовании метода перегрузки может возникнуть проблема, когда объявление метода перекрывает метод родительского класса с тем же названием.

Таким образом, может возникнуть конфликт между переопределением (overriding) и перегрузкой (overloading) унаследованного метода [2].

В этом же проекте удалите оба слова overload из интерфейсной секции каждого класса T1 и T2 и в классе T1 подпишете слово virtual к методу:

```
type
 T1 = class
  procedure Test(I: Integer); virtual;
  end;
```

Попробуйте скомпилировать программу и нажмите Ctrl+F9. Вы получите предупреждение: Method ‘Test’ hides virtual method of base type ‘T1’ о том, что новый статический метод Test(S: string) класса T2 перекрывает метод с тем же именем базового класса T1.

Как и в ряде других примеров, новый статический метод в T2 перекрывает старый виртуальный в T1.

Зайдите в обработчик щелчка на Button1 и снимите комментарии с закомментированных инструкций. Обратите внимание н то, чтопосле написания оператора «точка» для переменной T2_, на экране отобразится информация о том, что в классе T2 пока, как и в предыдущем случае (рис.15.11), существует только один новый замененный метод с именем Test и параметром S: string, скрывающий старый, унаследованный метод.

Перегрузка методов помогает устранить данный конфликт перекрытия виртуального родительского метода новым одноименным статическим методом [2].

Измените объявления классов следующим образом:

```
type
 T1 = class
  procedure Test(I: Integer); overload; virtual;
  end;

 type
 T2 = class(T1)
  procedure Test(S: string); overload;
 end;
```

Здесь имеется определенная сложность, которая заключается в использовании слова overload [2]. В данном случае перегружаются 2 метода, поэтому, сняв все комментарии из обработчика щелчка, весь код откомпилируется нормально и без ошибок, так как теперь компилятор отличает методы по их параметрам.

Здесь одновременно происходит следующее: классом T2 наследуется виртуальный метод из T1 и происходит изменение параметров метода при неизменном названии.

Другими словами, благодаря ключевому слову overload, виртуальный метод из T1 наследуется и не перекрывается одноименным методом в T2,благодаря возможности перегрузки в T2.

Это некоторое исключение из правил вызвано перегрузкой. Казалось бы, вроде бы заново вводим новый одноименный метод в класс T2, поэтому чисто лексически новый метод «официально» перекрывает (скрывает) старый, однако именно благодаря перегрузке старый метод не скрыт и доступен из производного класса. То есть, несмотря на то, что одноименный метод из T1 в T2 не скрыт (из-за возможности перегрузки), он еще и унаследован классом T2 том виде, в котором был объявлен в T1; значит в классе T2 будут присутствовать эти два перегруженных метода.

Можно добавить ключевое слово reintroduce перед overload в класс T2, говорящеещее компилятору о намерении замены метода (скрытие метода не будет из-за перегрузки), о том, что в дальнейшем перегрузка метода не предполагается. Теперь описание классов будет таким:

```
type
 T1 = class
  procedure Test(I: Integer); overload; virtual;
  end;

 type
 T2 = class(T1)
  procedure Test(S: string); reintroduce; overload;
 end;
```

Откомпилируйте программу еще раз, нажав Ctrl+F9. Появится предупреждение компилятора, о том, что метод Test в T3 перекрывает метод ,объявленный в базовом классе T1. Можете это проверить. Попробуйте это объяснить самостоятельно.

В приведенном выше примере в T2 объявлен тот же метод, но с другим параметром Test(S: string), то есть сигнатура методов не одинакова, поэтому как такового переопределения виртуального метода Test(I: Integer) из T1 в классе T2 – нет.

Здесь при компиляции выдается предупреждение, что метод Test в T3 перекрывает (скрывает) метод, объявленный в T1, так как он впервые вводится в T1 без последующего переопределения. Если бы мы его переопределили в T2, выдалось бы предупреждение, что метод Test в T3 перекрывает метод, объявленный в T2.

В данном случае метод в T3 объявлен без единого ключевого слова и, следовательно, является статическим скрывающим любой одноименный родительский метод (даже виртуальный), но так как его имя и параметры Test(I: Integer) полностью совпадают с объявлением метода в T1, то в классе T3 его можно изменить из статического на переопределенный виртуальный, просто добавив директиву override (пока не добавляйте ее).

Запустите программу. Щелкните на Button1. Как и ранее, по очереди вызовутся два перегруженных метода T2.Test(S: string) и T1.Test(I: Integer). **Вернитесь в режим программирования.**

Сбросьте на форму еще одну кнопку Button2 и в обработчике щелчка на ней напишите:

```
T1_:=T2.Create;{T1_ указывает на объект производного класса T2}
T1_.Test(1000);
T1_.Free;
T1_:=T3.Create; {T1_ указывает на объект производного класса T3}
T1_.Test(1000);
T1_.Free;
T2_:=T3.Create; {T2_ указывает на объект производного класса T3}
T2_.Test(1000);
T2_.Free;
```

Запустите программу. Щелкните на Button2. Вы получите следующие результаты:

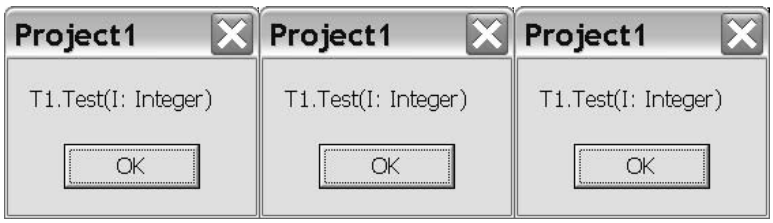


Рис. 15.13 Результат действия инструкций вызова виртуального метода

Результат понятен. При выполнении первой инструкции вызова метода T1_.Test(1000) полиморфно вызывается метод класса T1 и затем при выполнении оставшихся двух инструкций вызова методов происходит то же самое, так как в классе T3 метод объявлен статическим и в нем обрывается цепочка виртуальных методов.

Вернитесь в режим программирования.

Теперь, для сравнения, измените статический метод в T3 на переопределенный виртуальный, просто добавив директиву override. Описание класса T3 будет в этом случае выглядеть так:

```
type
T3 = class(T2)
procedure Test(I: Integer); override;
end;
```

И снова запустите программу на выполнение. Щелкните на Button2. Вы получите следующие результаты:

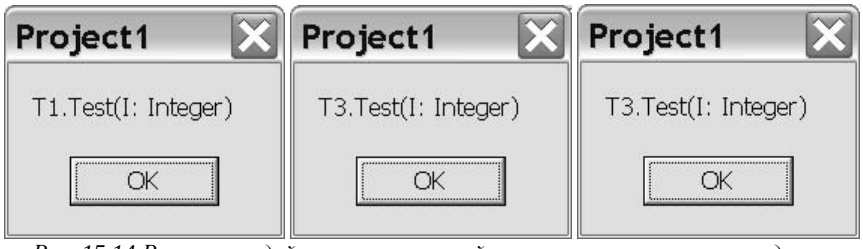


Рис. 15.14 Результат действия инструкций вызова виртуального метода

При выполнении первой инструкции вызова метода, как и ранее, полиморфно вызывается метод класса T2, унаследованный от T1, а при выполнении второй и третьей инструкции вызова метода полиморфно вызывается переопределенный виртуальный метод класса T3. **Вернитесь в режим программирования.**

Чтобы лучше уяснить вышесказанное, добавьте еще один переопределенный виртуальный метод в класс T2 с тем же именем Test и параметрами, что и в T1 и в T3, то есть теперь описание класса T2 будет таким :

```
type
T2 = class(T1)
procedure Test(S: string); reintroduce; overload;
procedure Test(I: Integer);override;
end;
```

Щелкните сочетание Ctrl+Shift+C и появившуюся заготовку реализации метода измените следующим образом:

```
procedure T2.Test(I: Integer);
```

```
begin
ShowMessage('T2.Test(I: Integer) ');
end;
```

Снова запустите программу на выполнение. Щелкните на Button2. Вы получите следующие результаты:

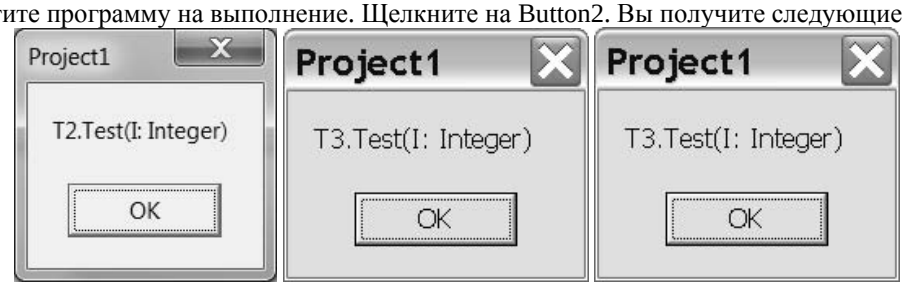


Рис.15.15 Результат действия инструкций вызова виртуального метода

Здесь при выполнении первой инструкции вызова метода T1_.Test(1000) полиморфно вызывается новый переопределенный метод класса T2, а при выполнении второй и третьей инструкции вызова метода полиморфно вызывается переопределенный виртуальный метод класса T3. **Вернитесь в режим программирования.**

А теперь уберите директиву override с метода Test(I: Integer) класса T2 и попробуйте скомпилировать программу. Нажмите Ctrl+F9. Вы получите сообщение компилятора об ошибке:

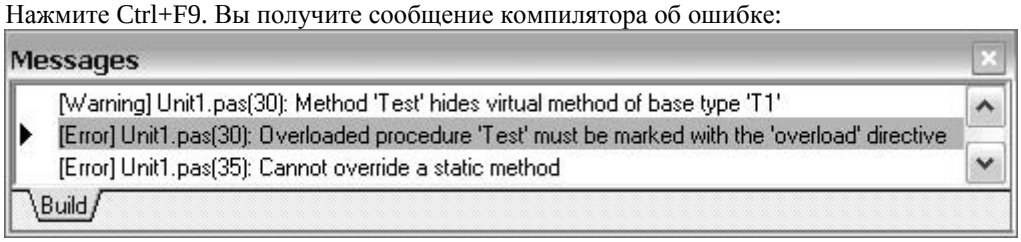


Рис. 15.16 Сообщение компилятора об ошибках

Дело в следующем. Здесь первое предупреждение о том, что теперь в классе T2 появился новый статический метод, который перекрывает одноименный виртуальный метод родительского класса T1. Второе сообщение об ошибке заключается в том, что объявление нового замененного метода должно быть помечено директивой overload. И наконец, третье сообщение об ошибке состоит в том, что компилятор не может переопределить виртуальный метод в классе T3, так как одноименный метод объявлен статическим в классе T2.

Задание 15.2.3 Самостоятельно. Здесь же добавьте директиву overload к методу Test(I: Integer) в классе T2 и уберите override из класса T3. Запустите программу и щелкните на Button2. Объясните полученный результат.

Таким образом виртуальные методы могут быть перегружены. На примере трёх взаимонаследующих классов с тремя переопределенными виртуальными методами это может выглядеть так:

```
type
  T1 = class
    procedure Test(I: Integer); overload;  virtual;
   end;

  type
  T2 = class(T1)
   procedure Test(S: string); overload;
   procedure Test(I: Integer); override;
  end;

  type
  T3 = class(T2)
    procedure Test(I: Integer);override;
    end;
```

Задание 15.2.4 Покажем, как получить доступ к адресу виртуального метода в таблице виртуальных методов VMT объекта, а также при этом кратко опишем формат хранения объекта в памяти.

Нужно хотя бы примерно представлять себе, как же хранится сам объект в оперативной памяти, иными словами – каков его формат.

Внутренний формат данных самого объекта напоминает формат данных записи (тип record). Поля объекта сохранены в порядке объявления в виде последовательности непрерывных переменных. Любые поля, унаследованные от базового класса, сохранены перед новыми полями, определенными в классе – наследнике, а первое 4-байтное поле каждого объекта содержит 32-разрядный указатель на виртуальную таблицу методов (VMT) класса, которая содержит адреса всех виртуальных методов класса, независимо от того, унаследованы ли они от класса-родителя либо перекрыты (заменены) в классе-наследнике [9,10].

Поясним данные рассуждения. Пусть имеется некоторая переменная указатель pObj класса TObj на объект этого же TObj или производного от него класса TObjDescendant, тогда будет справедлива следующая схема:

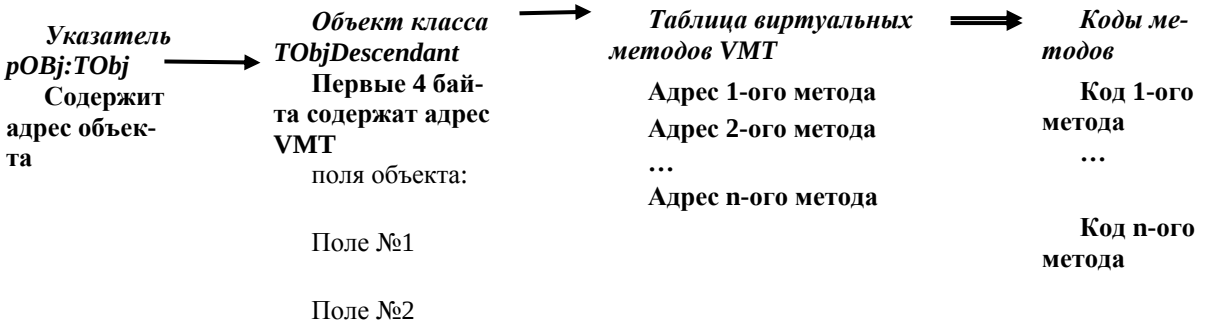


Схема существования объекта класса TObj в оперативной памяти

Таким образом, на основе данной схемы, создадим программу, где стандартным способом попытаемся вызвать виртуальный метод по его адресу, который содержится в таблице виртуальных методов, при этом мы сначала попытаемся добраться до самой таблицы виртуальных методов VMT [9,10].

Создайте новый проект. Сбросьте на форму 2 компонента Button1 и Memo1 следующим образом:

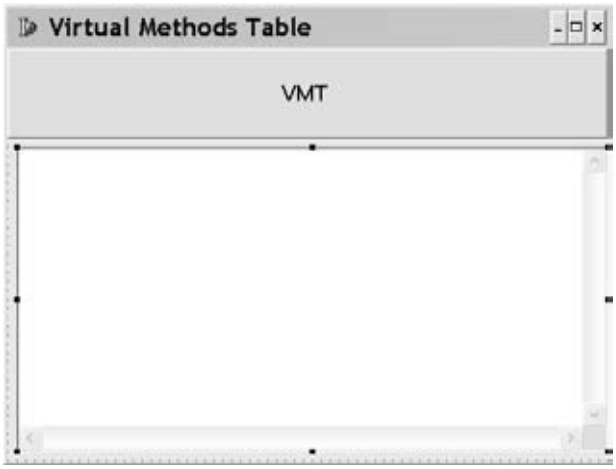
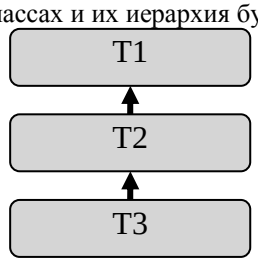


Рис. 15.17 Окно программы демонстрирующей работу с VMT

Далее в интерфейсном разделе объявим 3 новых, взаимонаслдующих класса с уже знакомыми нам именами T1, T2 и T3. Каждый будет содержать несколько виртуальных методов (Test, Fly и look), которые будут переопределяться в последующих классах и их иерархия будет выглядеть так:



Объявите данные классы следующим образом:

```
type
 T1 = class
  public
  procedure Test(I: Integer); virtual;
  procedure Fly; virtual;
  procedure look; virtual;

  end;

 type
 T2 = class(T1)

  public
  procedure Test(I: Integer); override;
  procedure Fly; override;
  procedure look; override;
  procedure Fly2; virtual;


 end;

 type
 T3 = class(T2)
 public
  procedure Test(I: Integer);
  procedure Fly;
  procedure look;
  end;
```

В классе T3 методы пока заменены одноименными статическими.
Также объявите 3 глобальные переменные T1_ класса T1, T2_ класса T2 и T3_ класса T3.

```
var
 Form1: TForm1; T1_:T1; T2_:T2; T3_:T3;
```

И как обычно, нажмите сочетание клавиш Ctrl+Shift+C, измените появившиеся заготовки методов следующим образом:

```
{T1}

procedure T1.Test(I: Integer);
begin
ShowMessage('T1.Test(I: Integer)');
end;

procedure T1.fly;
begin
ShowMessage('T1.fly');
end;

procedure T1.look;
begin
ShowMessage('T1.look');
end;

{T2}
procedure T2.Test(I: Integer);
begin
ShowMessage('T2.Test(I: Integer)');
end;

procedure T2.fly;
```

```
begin
ShowMessage('T2.fly');
end;

procedure T2.look;
begin
ShowMessage('T2.look');
end;

procedure T2.Fly2;
begin
ShowMessage('T2.Fly2');
end;

{T3}
procedure T3.Test(I: Integer);
begin
ShowMessage('T3.Test(I: Integer)');
end;

procedure T3.fly;
begin
ShowMessage('T3.fly');
end;

procedure T3.look;
begin
ShowMessage('T3.look');
end;
```

Как и ранее, вызов метода будет просто выводить информацию о методе класса фактически имеющегося объекта на экран в виде сообщения.

Создайте обработчик щелчка на Button1, который будет выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
type
 TVMT=array[0..100] of pointer;
 GoProc = procedure(Obj:T1);

var
 pVMT:^TVMT;
 pMethod:pointer;
 pObj:pointer;

 i,j,k,sum:integer;
begin
 memo1.Clear;
 T1_:=T1.Create;

 pObj:=pointer(T1_); {pObj указывает на тот же объект что и T1_, но уже имеет тип pointer}
 pVMT:=pointer(pObj^); { pVMT содержит адрес таблицы виртуальных методов VMT объекта на который указывает pObj}
 pMethod:=pVMT^[0]; { pMethod содержит адрес первого метода из VMT объекта, на который указывает pObj }

memo1.Lines.Add('pVMT указывает на VMT объекта класса '+ TObject(pObj).ClassName); memo1.Lines.Add('Адрес первого метода в VMT равен '+#9+IntToStr(longword(pVMT^[0])));
 memo1.Lines.Add('Адрес второго метода в VMT равен '+#9+IntToStr(longword(pVMT^[1])));
 memo1.Lines.Add('Адрес третьего метода в VMT равен '+#9+IntToStr(longword(pVMT^[2])));
 memo1.Lines.Add('Адрес четвертого метода в VMT равен'+#9+IntToStr(longword(pVMT^[3])));
 memo1.Lines.Add('');
 GoProc(pMethod)(T1_);
```

```
T1_.Free;
end;
```

Разберем приведенный в обработчике код.

Здесь в самом начале в секции type объявляем используемые типы данных. Запись TVMT=array[0..100] of pointer означает объявление типа TVMT, представляющего из себя массив из 100 элементов типа pointer. Тип pointer является стандартным типом указателя и может указывать на данные любого типа. Далее следует объявление типа GoProc = procedure(Obj:T1), представляющее из себя возможность вызвать процедурный метод объекта Obj класса T1 по указанному адресу.

Последующее объявление GoFunc аналогично предыдущему.

Далее в секции var мы объявляем переменные. Запись pVMT:^TVMT означает, что мы объявляем переменную pVMT, являющуюся указателем на данные типа TVMT, этот факт отражает значок ^. То есть pVMT будет содержать адрес данных типа TVMT. Затем объявляются 2 переменные pMethod и pObj типа pointer, которые будут использоваться так:

pMethod – будет содержать адрес метода, который можно будет вызвать с помощью объявленного типа GoProc или GoFunc;

pObj – будет также содержать адрес объекта какого-либо класса, просто уже приведенный к стандартному типу pointer. Использовать тип pointer в подобном случае очень удобно, ведь с его помощью можно указывать на данные любого типа или класса и не нужно специально водить тип указателя на конкретный класс.

Затем мы создаем объект класса T1, на который указывает соответствующая переменная-указатель T1_.

Из приведенных комментариев кода ясна логика работы программы. Инструкцией pObj:=pointer(T1_) мы приводим тип переменной-указаетля T1_ к типу pointer и сохраняем его в pObj, который начинает указывать на тот же объект что и T1_.

Подробней поясним инструкциею pVMT:=pointer(pObj^).

Здесь запись в скобках pObj^ означает следующее: так как pObj содержит адрес объекта (указывает на сам объект), то применение оператора ^ к указателю pObj означает (разыменование указателя) – получение значения, находящегося по этому адресу (где расположен объект). А так как тип pointer (к которому принадлежит pObj) занимает 4 байта, то в результате разыменования pObj^ будет получено некоторое целое 4-ех байтное число, которое и будет являться адресом таблицы виртуальных методов, так как именно первые 4 байта любого объекта содержат адрес этой таблицы, а pObj указывает именно на начало объекта.

Затем полученный адрес таблицы VMT, в результате разыменования указателя pObj инструкцией pObj^, дополнительно приводим к типу pointer и сохраняем значение в переменной pVMT. Таким образом, теперь в pVMT сохраняется адрес VMT в виде типа pointer.

Далее по коду программы идет инструкция pMethod:=pVMT^[0]. Здесь запись в скобках pVMT^[0] означает следующее: так как тип pVMT является типом указателя на таблицу TVMT, то, чтобы получить доступ к данным находящимся по этому адресу, нужно сначала разыменовать указатель pVMT, то есть применить оператор ^ к pVMT в виде инструкции pVMT^.

Но так как сами данные этого типа TVMT являются последовательным массивом указателей типа pointer, то, чтобы получить доступ к конкретному элементу массива (в нем содержится адрес метода), нужно указать его порядковый номер в этом массиве. Полностью это выглядит как pVMT^[0]. Итак, при выполнении инструкции pVMT^[0] мы получили адрес первого метода из таблицы VMT, а так как массив нумеруется с нуля, то адрес второго метода будет получен при выполнении инструкции pVMT^[1], третьего при pVMT^[2] и т.д.Специально сохраним адрес первого метода из таблицы VMT в переменную pMethod:=pVMT^[0].

Затем мы выводим в Memo1 информацию о том, к какому классу принадлежит созданный объект инструкцией TObject(pObj).ClassName, а также адреса всех методов, имеющихся в VMT, используя инструкцию longword(pVMT^[0]), с помощью которой просто приводим значение адреса от типа pointer к типу longword – описывающий обыкновенное 32-разрядное целое положительное число.

И наконец, инструкцией GoProc(pMethod)(T1_) мы вызываем первый виртуальный метод из VMT объекта T1, зная, что он находится по адресу, сохраненному в переменной pMethod.

Запустите программу на выполнение и щелкните на Button1. Вы получите следующий результат:

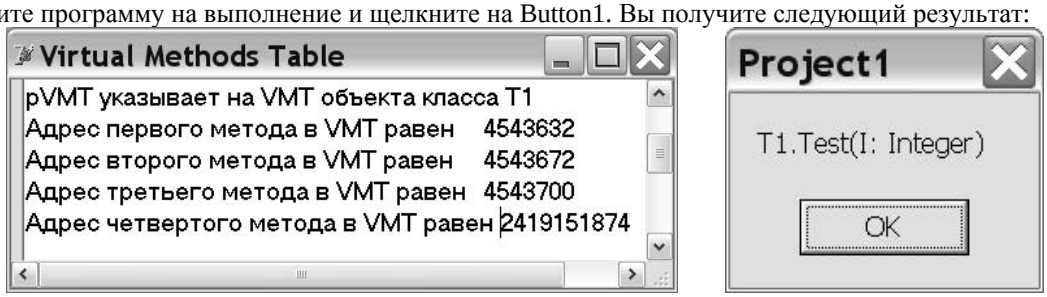


Рис. 15.18 Результат работы программы «VMT»

Вернитесь в режим программирования.

Таким образом, мы получили адреса всех виртуальных методов, а также вызвали первый виртуальный метод на выполнение, адрес которого взяли из таблицы VMT. Им оказался метод Test класса T1, который был объявлен в T1 первым. Обратим внимание, что при выводе 4-го метода выдается число, резко отличающееся от остальных. Дело в том, что значение, содержащееся в pVMT^[3], является некоторым случайным, так как у нас в классе T1 не объявлен 4-й метод, а имеется только 3 метода.

Теперь поменяйте в инструкции номер метода с 0 на 1, то есть измените инструкцию pMethod:=pVMT^[0] на pMethod:=pVMT^[1], и снова запустите программу.

У вас уже вызовется метод Fly класса T1.

Рис. 15.19 Теперь pMethod=pVMT^[1] содержит адрес второго метода

Вернитесь в режим программирования.

И конечно же, если вы напишите pMethod:=pVMT^[2], будет вызван третий объявленный виртуальный метод в T1 и т.д. Можно сделать вывод: VMT содержит адреса методов в порядке следования их объявления в классе.

Теперь закомментируйте инструкции pMethod:=pVMT^[0] и GoProc(pMethod)(T1), они пока не нужны. Просто с их помощью мы показали, что можно вызывать метод, зная только его адрес. Таким же способом можно вызывать не только виртуальный, но и статический метод.

Далее одновременно получим адреса виртуальных методов объектов класса T1 и класса T2. Для этого измените код между begin и end в обработчике на следующий:

```
memo1.Clear;
T1_:=T1.Create;
pObj:=pointer(T1_); {pObj указывает на тот же объект, что и T1, но уже имеет тип pointer}
pVMT:=pointer(pObj^); { pVMT содержит адрес таблицы виртуальных методов VMT объекта, на который указывает pObj}
memo1.Lines.Add('pVMT указывает на VMT объекта класса '+ TObject(pObj).ClassName);
memo1.Lines.Add('Адрес первого метода в VMT равен '+#9+IntToStr(longword(pVMT^[0])));
memo1.Lines.Add('Адрес второго метода в VMT равен '+#9+IntToStr(longword(pVMT^[1])));
memo1.Lines.Add('Адрес третьего метода в VMT равен '+#9+IntToStr(longword(pVMT^[2])));
memo1.Lines.Add('');

T2_:=T2.Create;
pObj:=pointer(T2_); {Теперь pObj указывает на тот же объект, что и T2_}
pVMT:=pointer(pObj^); { pVMT содержит адрес таблицы виртуальных методов VMT объекта, на который указывает pObj}
memo1.Lines.Add('pVMT указывает на VMT объекта класса '+ TObject(pObj).ClassName);
memo1.Lines.Add('Адрес первого метода в VMT равен '+#9+IntToStr(longword(pVMT^[0])));
memo1.Lines.Add('Адрес второго метода в VMT равен '+#9+IntToStr(longword(pVMT^[1])));
memo1.Lines.Add('Адрес третьего метода в VMT равен '+#9+IntToStr(longword(pVMT^[2])));
memo1.Lines.Add('Адрес четвертого метода в VMT равен'+#9+IntToStr(longword(pVMT^[3])));
memo1.Lines.Add('');

FreeAndNil(T1_); {аналогично вызову деструктора}
FreeAndNil(T2_); {аналогично вызову деструктора}

end;
```

Здесь процедура FreeAndNil модуля SysUtils просто освобождает память, занятую объектом, и устанавливает T1 равным nil.

Запустите программу. Результат будет аналогичен, только уже адреса виртуальных методов будут иными:

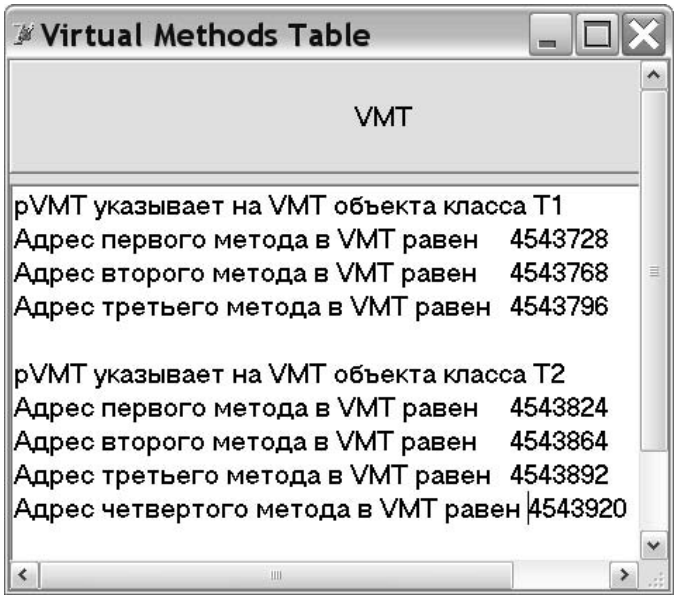


Рис. 15.20 Все адреса виртуальных методов объектов классов T1 и T2 различны

В данном случае все адреса всех методов различные. Ясно, что в VMT объекта класса T1 находятся адреса впервые введенных там виртуальных методов, а в VMT объекта класса T2 также находятся новые адреса виртуальных методов. Из них 3 первые метода переопределены в T2 (именно поэтому адреса другие), а также добавлен адрес нового виртуального метода Fly2, впервые введенный в класс T2, который объявлен там четвертым.

Вернитесь в режим программирования.

Теперь в классе T2, справа от метода Test закомментируйте только одну директиву override. Таким образом, при компиляции вы получите предупреждение компилятора о перекрытии метода новым одноименным статическим. Запустите программу на выполнение. Результат будет следующим:

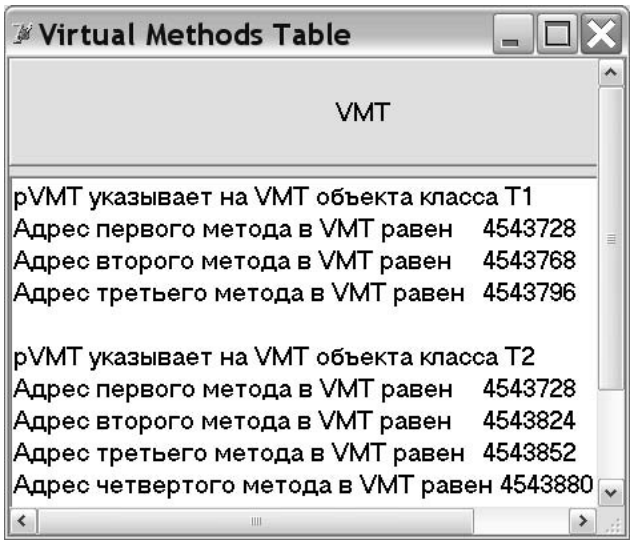


Рис. 15.21 Теперь адреса первых методов в VMT объектов обоих классов равны

Мы видим, что адрес первого метода в VMT класса T1 также присутствует и VMT класса T2. Это имеет свои объясняения.

В данном случае виртуальный метод Test из T1 перекрывается одноименным статическим в классе T2, однако, несмотря на то, что метод перекрыт и родительский виртуальный метод явно уже недоступен в T2, он все равно существует в T2, наследуясь путем записи в VMT класса T2. Таким образом, оба метода (и унаследованный и новый одноименный) существуют в T2.

Напомним, что в виртуальной таблице VMT описаны все виртуальные методы, как класса-наследника, так и класса-родителя и, вообще, в ней хранятся адреса всех виртуальных методов класса, независимо от того, унаследованы ли они от класса-родителя либо перекрыты (заменены) в классе-наследнике. Поэтому информация о всех методах сохраняется.

В оперативной памяти таблица виртуальных методов VMT для каждого класса имеется в единственном экземпляре, просто каждый объект данного класса имеет на нее ссылку. **Вернитесь в режим программирования.**

Попробуйте теперь убрать (закомментировать) директиву override еще и у метода Fly в классе T2, а затем запустите программу. Теперь в VMT классов будут совпадать значения уже двух методов. **Вернитесь в режим программирования.**

Задание 15.2.5 В классе T3 метод fly объявите с директивой vitrtual, а метод look с с директивой override. Получите адреса всех методов в VMT класса T3 и сравните их с VMT классов T1 и T2. Объясните результат.

Указание: Выполняйте задание в этом же проекте. Добавьте кнопку Button2 на форму с с заголовком «Получить VMT объекта класса T3» и в ее обработчике пишите код.

Задание 15.2.6 Кратко покажем, как получить адрес обычного статического метода

Указание: Выполняйте задание в этом же проекте. Будем работать с классом T3, так как метод Test там является статическим.

Добавьте еще одну кнопку Button3 на форму с заголовком «Получить адрес статического метода функцией addr».

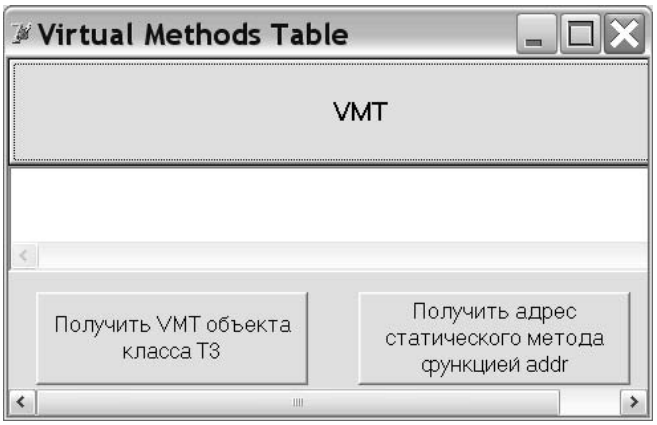


Рис. 15.22 Теперь адреса первых методов в VMT объектов обоих классов равны

Запрограммируйте обработчик щелчка на Button3 следующим образом:

```
procedure TForm1.Button3Click(Sender: TObject);
type GoProc = procedure(Obj:TObject);
var pMethod:pointer;
begin
pMethod:=addr(T3.Test);
memo1.Lines.Add('Адрес статического метода равен'+#9+IntToStr(longword(pMethod)));
T3_:=T3.Create;
GoProc(pMethod)(T3_);
T1(T3_).Test(1);
FreeAndNil(T3_);
end;
```

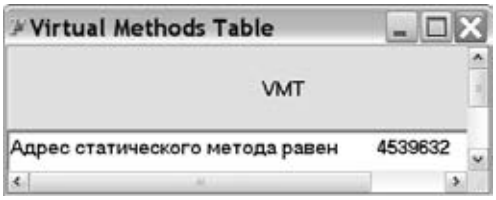
Как и ранее, объявляем тип GoProc для вызова метода объекта класса TObject (T3 его дальний наследник по иерархии) и переменную pMethod, хранящую адрес метода.

Основная задача в данном конкретном случае состоит в том, чтобы узнать адрес статического метода, а затем его вызвать. С этой целью мы пользуемся функцией addr, которой в качестве параметра передаем идентификатор функции, процедуры или метода класса, как в нашем случае. Функция возвращает адрес метода в виде указателя типа pointer. Таким образом, мы сохраняем его значение в переменой pMethod. Затем выводим его в Memo1.

Затем мы вызываем данный метод по его адресу, используя процедурный тип GoProc.

И далее для демонстрации возможностей ООП у объекта T3 мы вызываем метод Test родительского класса T1 путем инструкции T1(T3).Test(1). Здесь мы даем указание компилятору рассматривать объект, на который указывает T3 как объект класса T1, и вызываем метод этого класса.

Запустите программу. Вы получите следующий результат:



а) GoProc(pMethod)(T3_) *б) T1(T3_).Test(1)* *в) Выводим адрес метода*

Рис. 15.23 Результат выполнения кода кнопки «Получить адрес статического метода»

Мы пошли по пути от сложного к простому. Сначала узнали, как получать адреса виртуальных методов из таблицы VMT, а теперь узнали, как получать адрес метода по его имени и вызывать его.

Нужно отметить, что подобным способом можно узнать не только адрес статического, но и виртуального метода. Проверьте это.

15.3. **Инкапсуляция**. **Спецификаторы доступа** public, private, protected

Как известно, один из основополагающих принципов объектного программирования – это целостность объекта и сокрытие внутреннего устройства объекта и его данных от других объектов. Данный принцип называется **инкапсуляцией** [6]**.** Как правило, для взаимодействия с другими объектами у объекта имеется некоторый **интерфейс** в виде открытых (public) методов и полей, которые могут быть доступны из разных мест программы другим объектам (то есть доступ к интерфейсу может быть осуществлен из методов других объектов, просто из некоторой точки программы или из других модулей) [6]. Однако некоторые поля и методы (private), принадлежащие объекту, не должны быть доступны другим объектам или из определенных мест программы (например, из других модулей), то есть на них налагается ограничение доступа.

Именно с целью ограниченности доступа в синтаксис объявления класса специально были введены спецификаторы ограничения доступа к объекту: public, private, protected.

В отличие, например, от языка C++ , спецификаторы доступа в Object Pascal сильно “привязаны” к модулю, где описан исходный класс.

Напомним что синтаксис объявления класса следующий [4]:

type

<имя класса> = class (<имя класса-родителя>)

public {доступно методам любого другого класса в исходном модуле, где объявлен класс и методам любого другого класса в другом модуле}

<поля, методы, свойства, события>

private {доступно методам любого класса но только в исходном модуле где объявлен класс}

<поля, методы, свойства, события>

protected {доступно методам производных-классов даже в других модулях и методам любого класса но только в исходном модуле где объявлен класс}

<поля, методы, свойства, события>

published {отображаются в Object Inspector}

<поля, методы, свойства, события>

Задание 15.3.1 Покажем действие спецификаторов доступа: public, private и protected на примере рассмотрения доступа из производных классов к полям родительских классов с учетом расположения классов в разных местах программы.

Создайте новый проект. И после того как у вас на экране появится форма Form1, добавьте еще одну форму Form2, щелкнув на пиктограмме «New Form» либо выбрав последовательно File → New → Form. Теперь у вас в проекте будет две формы: Form1, описываемая модулем Unit1.pas, и Form2, описываемая модулем Unit2.pas.

Напомним, что для переключения между этими двумя формами и модулями необходимо применять сочетание клавиш Ctrl+F12 и Ctrl+F11.

Первое, что нужно сделать, так это настроить взаимную ссылку модулей Unit1 и Unit2 друг на друга, чтобы каждый из модулей имел доступ к другому модулю.

Для этого в окне редактора кода модуля Unit1 сразу после ключевого слова implementation напишите инструкцию uses unit2. Это будет выглядеть так:

```
implementation
uses unit2;
{$R *.dfm}
```

А в модуле Unit2 это нужно сделать в интерфейсной части после ключевого слова interface, в результате получится следующее:

```
interface
uses
 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
 Dialogs, StdCtrls, unit1;
```

Сбросьте на Form1 кнопку Button1, а на Form2 кнопку Button2.

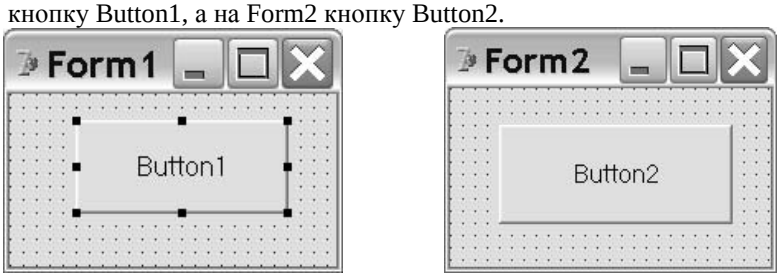


Рис. 15.24 Обработчики щелчков на этих кнопках выступают как соответствующие методы классов TForm1 и TForm2

С учетом того, что, когда вы сбрасывайте кнопки на разные формы, сначала каждой кнопке на каждой форме будет присвоено имя Button1, поэтому самостоятельно измените для второй формы значение свойства Name кнопки на Button2 и заголовок тоже.

Мы планируем воспользоваться созданными обработчиками щелчков на данных кнопках, как методами классов TForm1 и TForm2, чтобы продемонстрировать возможность доступа и из этих методов к полям объекта другого класса описанного в конкретном модуле, с учетом спецификаторов доступа к этим полям и области видимости.

При этом мы специально сделали так, чтобы TForm1 и TForm2 были в разных модулях, так как спецификаторы доступа регламентируют доступ к полям и методам в зависимости от расположения модулей.

Затем в модуле Unit1 объявите 2 класса следующим образом:

```
type

TFirstClass = class
 private
   priv:integer;
 public
   publ:string;
 protected
   prot:real;
 end;

type
 TSecondClass1 = class(TFirstClass)
 public
 procedure access;
end;
```

В данном случае мы описываем класс, где объявлены 3 поля: поле priv в секции private, поле publ в секции public и поле prot в секции protected.

Установите курсор на строчку объявлении метода access класса и, как обычно, нажмите сочетание Ctrl+Shift+C. Измените появившуюся заготовку реализации метода так:

```
procedure TSecondClass1.access;
begin
priv:=1;
publ:='publ';
prot:=100.123;
end;
```

Попробуйте скомпилировать программу нажав Ctrl+F9. Программа скомпилируется нормально.

Дело в том, что TSecondClass1 является наследником класса-родителя TFirstClass и значит имеет доступ ко всем полям и методам класса TFirstClass. То есть в любом методе класса TSecondClass1 можно изменять значения полей и вызывать методы родительского класса TFirstClass. Однако обратите внимание, что оба класса TFirstClass и TSecondClass1 находятся в одном модуле Unit1, именно поэтому доступ полный.

Опишем логику работы программы, исходя из определений спецификаторов области видимости.

Поле priv (доступно методам любого класса (в том числе и в TSecondClass1) но только в исходном модуле, где объявлен класс). Это условие для TSecondClass1 выполнено, он объявлен в том же модуле, что и TFirstClass. Доступ к полю priv есть.

Поле publ {доступно методам любого класса в исходном модуле, где объявлен класс, и методам любого другого класса в другом модуле}. Это условие для TSecondClass1 выполнено. Доступ к полю publ есть.

Поле prot { доступно только методам производных-классов даже в других модулях и методам любого класса, но только в исходном модуле. где объявлен класс }. В данном случае производным классом является TSecondClass1, это условие для TSecondClass1 также выполнено. Доступ к полю prot есть.

Теперь в модуле Unit2 опишите класс TSecondClass2, также являющийся классом-наследником TfirstClass, следующим образом:

```
type
 TSecondClass2 = class(TFirstClass)
 public
 procedure access;
end;
```

И нажмите Ctrl+Shift+C. Измените появившуюся заготовку метода следующим образом:

```
procedure TSecondClass2.access;
begin
priv:=1; {здесь ошибка, компилироваться не будет, нужно закомментировать}
publ:='publ';
prot:=100.123;
end;
```

И попробуйте откомпилировать программу. В данном случае компилятор выдаст ошибку на инструкцию priv:=1. Закомментируйте эту инструкцию и снова попытайтесь откомпилировать программу. Теперь все в порядке.

Снова опишем логику работы программы, исходя из определений спецификаторов области видимости:

Поле priv (доступно методам любого класса, но только в исходном модуле, где объявлен класс). Это условие для TSecondClass2 не выполнено, так как он объявлен в другом модуле, нежели класс TFirstClass. Поэтому компилятор выдает сообщение об ошибке. То есть доступа к полю priv нет из метода access класса TSecondClass2 и изменить его значение нельзя.

Поле publ {доступно методам любого класса в исходном модуле, где объявлен класс, и методам любого другого класса в другом модуле}. Это условие для TSecondClass2 выполнено, по отношению к TFirstClass он является другим классом в другом модуле. Доступ к полю publ есть.

Поле prot { доступно только методам производных-классов даже в других модулях и методам любого класса, но только в исходном модуле, где объявлен класс }. Это условие для TSecondClass2 также выполнено, так как TSecondClass2 является производным классом в другом модуле. Доступ к полю prot есть.

Та же логика будет и при доступе к методам, а именно: если бы в классе TFirstClass мы бы вдобавок к полям объявили методы под каждым из спецификаторов доступа, то доступ к этим методам был бы тот же, что и к полям описанный выше.

Задание 15.3.2 Теперь рассмотрим возможность доступа к полям объекта с учетом тех же спецификаторов доступа.

Для этого в модуле Unit1 объявите глобальную переменную с именем Obj класса TFirstClass. Ее объявление будет выглядеть так:

```
var Form1: TForm1; Obj:TFirstClass;
```

Так как модули взаимно ссылаются друг на друга, то глобальная переменная-указатель Obj будет видна в каждом модуле.

Для Form1 создайте обработчик щелчка на Button1, который должен выглядеть следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Obj:=TFirstClass.Create;
Obj.priv:=1;
Obj.publ:='publ';
Obj.prot:=100.123;
Obj.Free;
end;
```

Попробуйте откомпилировать программу. Компиляция выполнится нормально и без ошибок. И опять опишем логику работы программы, исходя из определений спецификаторов области видимости:

Поле priv (доступно методам любого другого класса, но только в исходном модуле, где объявлен класс). Это условие для объекта класса TfirstClass, на который указывает Obj, выполнено, так как метод Button1Click(Sender: TObject) как раз принадлежит этому другому классу TForm1, который объявлен в этом же модуле, что и TFirstClass. И из метода Button1Click(Sender: TObject) доступ к полю priv объекта Obj есть.

Поле publ {доступно методам любого класса в исходном модуле, где объявлен класс и методам любого другого класса в другом модуле}. Это условие для объекта класса TfirstClass, на который указывает Obj выполнено, так как метод Button1Click(Sender: TObject) как раз принадлежит классу TForm1, который объявлен в этом же исходном модуле что и TFirstClass. Доступ к полю publ объекта Obj есть.

Поле prot {доступно только методам производных-классов даже в других модулях и методам любого класса но только в исходном модуле, где объявлен класс }. Это условие для объекта класса TfirstClass, на который указывает Obj, выполнено. В данном случае любым классом является TForm1, а его методом является Button1Click(Sender: TObject), при этом TForm1 объявлен в том же исходном модуле, что и TFirstClass. Поэтому доступа к полю prot объекта Obj из этого метода есть.

Далее аналогично для Form2 создайте обработчик щелчка на Button2, который должен содержать тот же программный код и выглядеть следующим образом:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
Obj:=TFirstClass.Create;
Obj.priv:=1; {здесь ошибка. компилироваться не будет}
Obj.publ:='publ';
Obj.prot:=100.123; {здесь ошибка. компилироваться не будет}
Obj.Free;
end;
```

И снова попробуйте откомпилировать программу. Компилятор сообщит о наличие ошибок. И опять опишем логику работы программы, исходя из определений спецификаторов области видимости.

Поле priv (доступно методам любого другого класса, но только в исходном модуле, где объявлен класс). Это условие для объекта класса TfirstClass, на который указывает Obj, не выполнено, так как метод Button2Click(Sender: TObject) принадлежит к классу TForm2, который объявлен в другом модуле по отношению к классу TFirstClass. Из метода Button2Click(Sender: TObject) доступ к полю priv объекта Obj нет.

Поле publ {доступно методам любого класса в исходном модуле, где объявлен класс, и методам любого другого класса в другом модуле}. Это условие для объекта класса TfirstClass. на который указывает Obj, выполнено, так как метод Button2Click(Sender: TObject) как раз принадлежит к этому другому классу TForm2, который объявлен в другом модуле по отношению к классу TFirstClass. Доступ к полю publ объекта Obj есть.

Поле prot {доступно только методам производных-классов даже в других модулях и методам любого класса, но только в исходном модуле, где объявлен класс}. Это условие не выполнено. В данном случае любым классом является TForm2, а его методом является Button2Click(Sender: TObject), при этом класс TForm2 объявлен в другом модуле по отношению к классу TFirstClass. Поэтому доступа к полю prot объекта Obj из этого метода нет.

15.4. Свойства класса

Для поддержи принципа инкапсуляции, то есть сокрытия данных объекта и для улучшения его интерфейса, в класс введено такое понятие, как «свойство» (**property**). Так как некоторые поля напрямую могут быть совершенно недоступны некоторым объектам из определенных мест программы, то повлиять на них можно только через свойство.

Свойство – это переменная конкретного типа, являющаяся такой же частью исходного объекта, как и поле, но предназначенное быть посредником между скрытым полем исходного объекта и внешним объектом, который хочет поменять или получить к этому полю доступ.

Фактически свойство является частью интерфейса объекта, свойство объекта определяет, каким образом другой объект может изменить поле или получить его значение. Коротко говоря, свойство объекта диктует правила доступа одного объекта к полю другого.

Синтаксис объявления свойства следующий [4]:

property <имя свойства>:<тип> read <имя поля или метода чтения>
write <имя поля или метода записи>;

Здесь «метод чтения» и «метод записи» – это методы класса, которые имеют полный доступ ко всем полям класса (открытым и закрытым) и поэтому могут изменять их любым доступным образом. Именно за счет этих методов свойство и обеспечивается интерфейс класса.

Если в секции read находится «имя поля», то это говорит о том поле, значение которого будет считываться в свойство (присваиваться к свойству) в момент, когда свойство участвует в некоторой инструкции (кроме записи). Например, мы некоторой инструкцией просто пытаемся получить значение свойства.

А если в секции read находится «метод чтения», то это говорит о том, что, когда в некотором выражении встречается свойство, каждый раз неявно вызывается «метод чтения» (часто он оперирует некоторым скрытым полем) возвращающий свойству некоторое значение, как правило, значение скрытого поля. Так как метод чтения возвращает значение, его часто оформляют в виде функции, а не процедуры.

Если в секции write находится «имя поля», то это говорит о том, что когда встречается инструкция присвоения « := » свойству некоторого значения, то это значение сразу же запишется в указанное в секции поле.

А если в секции write находится «метод записи», то, как только в программе встретится инструкция присвоения « := » свойству некоторого значения, сразу же неявно вызовется «метод записи», который, как правило, записывает это значение в указанное в секции поле. Так как во многих случаях метод записи не должен возвращать никакого значения, то его оформляют в виде процедуры.

Вообще в синтаксисе определения свойства секция read подразумевает действия для чтения значения поля в свойство, то есть, по сути, это означает запись в свойство значения поля, а секция write подразумевает действия для изменения значения этого поля (перезапись поля) путем присвоения свойству некоторого значения.

Однако на деле можно самым различным способом варьировать действия, производимые этими методами и не по назначению их секций.

Задание 15.4.1 Объявим свойство «A» типа word, в которое считываться будет значение скрытого поля FA, а записываться свойство A будет методом записи SetA

Создайте новый проект.

Не будем создавать новых классов и сразу же воспользуемся готовым классом TForm1.

Измените класс TForm1 так:

```
type
 TForm1 = class(TForm)
 private
  FA:word; {вводим поле FA}
 public
  property A:word;
 end;
```

И как обычно, нажмите сочетание Ctrl+Shift+C. Вы обнаружите, что в модуле произошли следующие изменения (выделенные жирным шрифтом):

```
type
 TForm1 = class(TForm)
 private
  FA:word;
  procedure SetA(const Value: word);
 public
  property A:word read FA write SetA;
 end;

var
 Form1: TForm1;

implementation
{$R *.dfm}
```

```
{ TForm1 }
procedure TForm1.SetA(const Value: word);
begin
 FA := Value;
end;
```

Во-первых: изменилось объявление класса. Добавилось объявление прототипа метода SetA в классе и его реализация. Также по умолчанию изменилось объявление свойства A на property A:word read FA write SetA. Это объявление нужно интерпретировать так:

В любой инструкции, кроме записи, в свойство A будет считываться значение поля FA, а при инструкции присвоения свойству A некоторого значения будет вызываться метод записи SetA с константным параметром Value (который как раз и будет содержать это некоторое значение), присваивающий значение Value полю FA. Здесь описывается содержимое кода SetA по умолчанию, конечно же, оно может быть любым.

Далее сбросьте кнопку Button1 и компонент Memo1 на форму.

В обработчике щелчка на Button1 напишите следующее:

```
A:=100;
Memo1.Lines.Add(IntToStr(A));
```

Запустите программу. Щелкните на Button1. Вы получите следующий результат:

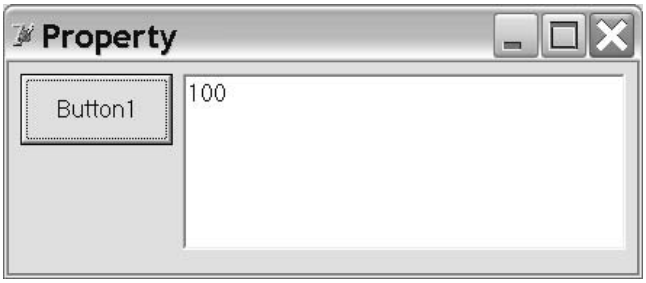


Рис. 15.25 Результат работы приложения «Property»

Вернитесь в режим программирования.

В данном случае инструкция A:=100 автоматически вызовет метод записи SetA(const Value: word), значение константного параметра Value при этом установится в 100, и значит при выполнении кода метода записи SetA, а именно FA:=Value, полю FA присвоится значение 100.

Во второй инструкции Memo1.Lines.Add(IntToStr(A)) мы просто хотим получить значение A и при выполнении этой инструкции в A будет присвоено значение FA, которое уже равно 100, благодаря предыдущей инструкции присвоения. То есть значение A будет равно 100.

Теперь слегка измените обработчик щелчка на Button1 так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
A:=100; FA:=1;
Memo1.Lines.Add(IntToStr(A));
end;
```

И запустите программу. Вы получите следующий результат:

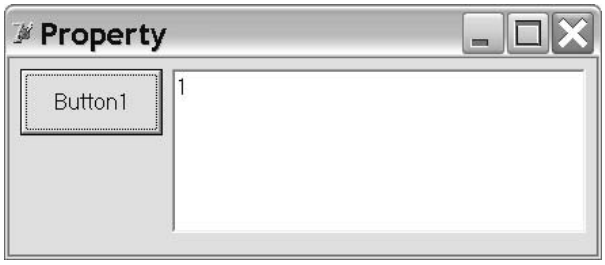


Рис. 15.26 Результат работы приложения «Property»

Вернитесь в режим программирования.

Такой результат предопределен рядом причин. Как и ранее. инструкция A:=100, благодаря методу записи SetA, установит значения FA в 100. Однако затем мы явной инструкцией FA:=100, устанавливаем значение

FA, которое становится равным 1. И когда мы выводим значение A в Memo1, в A присваивается значение FA, и следовательно A также становится равным 1.

Теперь измените метод SetA следующим образом:

```
procedure TForm1.SetA(const Value: word);
begin
 FA := Value+5000;
end;
```

А из обработчика удалите инструкцию FA:=1 и запустите программу. Щелкните на Button1. Результат будет таким:

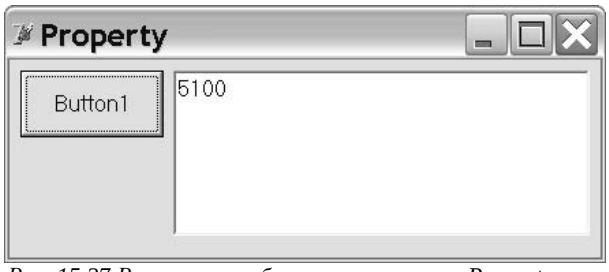


Рис. 15.27 Результат работы приложения «Property»

И снова при выполнении инструкции A:=100 вызовется SetA с его параметром Value, который будет равен 100, но теперь внутри метода SetA полю FA присваивается сумма Value и 5000, и значит при выводе A в Memo1 выведется 5100.

Задание 15.4.2 Объявим еще одно свойство «B» типа integer, в которое методом чтения GetA будет считываться значение, полученное с использованием поля FB, а записываться свойство B будет полем FB.

Допишите в секцию private еще одно поле FB типа integer, и секция private теперь будет выглядеть так:

```
 private
  FA:word;
  FB:integer.
```

А в секции public объявите еще одно свойство B и секция public теперь будет выглядеть так:

```
 public
  property A:word read FA write SetA;
  property B:integer read GetB write FB;
```

Таким образом, у свойства B имеется метод чтения GetB и поле для записи FB.

Теперь для того, чтобы у вас сформировалась заготовка метода чтения GetB , снова нажмите Ctrl+Shift+C.

Перед вами появится заготовка метода GetB, в виде функции:

```
function TForm1.GetB: integer;
begin
 Result := FB;
end;
```

По умолчанию возвращается значение поля FB, то есть программный код этого метода по умолчанию, эквивалентен тому, как если бы мы объявили свойство B в виде property B:integer read FB write FB.

Измените обработчик щелчка на Button1 так (введенные изменения отмечены жирным шрифтом):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
A:=100;
Memo1.Lines.Add(IntToStr(A));
B:=300;
Memo1.Lines.Add(IntToStr(B));
end;
```

Запустите программу на выполнение. Результат будет следующий.

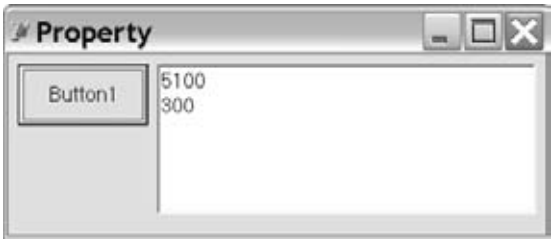


Рис. 15.28 Результат работы приложения «Property»

В данном случае при выполнении инструкции B:=300 в поле FB запишется значение, равное 300. И когда мы выводим значение B в Memo1, автоматически вызывается метод GetB, возвращающий свойству B значение, равное полю FB, то есть значение B тоже будет равно 300.

Заметьте, что мы постоянно объявляем поля FA и FB в секции private. Это сделано для того, чтобы они стали недоступными из других мест программы, в других модулях из методов объектов других классов, то есть скрыть их от влияния других объектов. Но так как свойства A и B объявлены в секции public, то доступ к ним останется в любом месте, даже в другом модуле и из методов других классов. Поэтому другой объект может влиять на поля FA и FB исходного объекта, но только через предоставленные ему свойства этого объекта.

Задание 15.4.3 Покажем, что необходимо тщательнейшим образом следить за программным кодом в методах записи или чтения свойства, так как эти методы имеют неограниченный доступ к любому полю объекта.

Дело в том, что когда в программе встречается выражение, содержащее свойство B, у которого имеется метод чтения GetB, то, по сути, само появление названного свойства в выражении является **инструкцией действия**, которая скрыто от программиста автоматически вызывает метод GetB, возвращающий значение свойству B. Это аналогично инструкции объявления переменной объектного типа в языке C++, при которой автоматически и скрыто вызывается конструктор класса, инициализирующий объект.

Продемонстрируем это в два этапа.

Сначала измените обработчик щелчка на Button1 так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
A:=100;
Memo1.Lines.Add(IntToStr(A));
B:=300;
Memo1.Lines.Add(IntToStr(B+B));
end;
```

Запустите программу. Щелкните на Button1. Так как метод GetB просто возвращает свойству B значение поля FB, то результат будет совершенно очевидным.

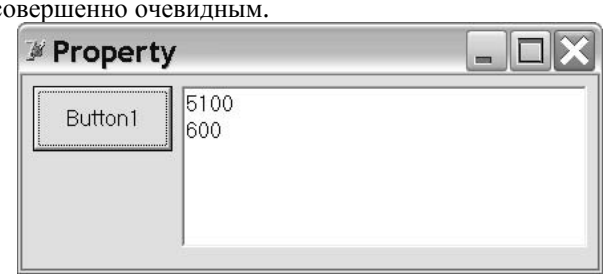


Рис. 15.29 Результат работы приложения «Property»

Вернитесь в режим программирования.

Действительно, внутри оператора IntToStr имеется выражение B+B. следовательно, когда B встречается в выражении первый раз, то вызывается GetB, возвращающий B значение FB, которое уже равно 300, благодаря написанной ранее инструкции B:=300.

То же самое происходит, когда B встречается в выражении второй раз, и следовательно GetB также вызовется второй раз в одном выражении. То есть сколько раз встречается в одном выражении B, столько же раз неявно автоматически будет вызван метод чтения GetB.

Таким образом, окончательно выражение IntToStr(B+B) будет эквивалентно выражению IntToStr(300+300); в Memo1 выведется 600.

А теперь измените метод GetB, добавив в него всего одну строчку, следующим образом:

```
function TForm1.GetB: integer;
begin
 FB:=FB+5;
 Result := FB;
end;
```

При каждом неявном вызове метода GetB значение поля FB будет постоянно увеличиваться на 5 и это значение поля будет возвращаться свойству B.

Запустите программу. Щелкните на Button1. Результат будет таким:

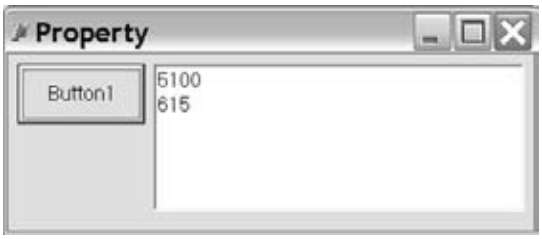


Рис. 15.30 Результат работы приложения «Property»

Вернитесь в режим программирования.

Начнем с инструкции B:=300, в результате чего полю FB присвоено значение 300. А затем, когда в выражении B+B, B встречается первый раз, то сразу вызывается метод GetB, в котором значение поля FB увеличивается на 5 и теперь равно 305, по сравнению со старым значением 300, а так как GetB возвращает значение FB свойству B, то теперь свойство B равно 305.

Когда же в выражении второй раз встречается B, снова происходит вызов GetB и снова значение поля FB увеличивается на 5 и теперь равно 310, по сравнению со старым значением 305. То есть B теперь также становится равно 310.

Следовательно, в данном контексте выражение B+B эквивалентно 305+310 и в результате выводится значение 615.

Таким образом, от программиста может быть скрыто то, что происходит на самом деле при интерпретации свойства как обыкновенной переменной, участвующей в некотором выражении. Это может явиться потенциальным источником ошибок, поэтому надо тщательным образом анализировать содержимое кода в методах чтения и записи.

Задание 15.4.4 Самостоятельно. Объявите свойство C типа string в виде property C:string read GetC write SetC. В каждом из методов GetC и SetC должно фигурировать поле FC, объявленное в секции private. Присвойте свойству C некоторое значение и выведите его в Memo1. Сделайте так, чтобы свойство участвовало в некотором выражении несколько раз. Объясните полученный результат.

Указание. Поэкспериментируйте с содержимым кода в методах GetC и SetC.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Холзнер, С. Visual C++ 6. Учебный курс. / С. Холзнер – Спб.: Питер, 2007.
2. Сван, Т. Delphi 4. Библия разработчика. / Т. Сван. – СПб.: Диалектика, 1998. – 672 с.
3. Культин, Н. Delphi 3. Программирование на Object Pascal. / Н. Культин – СПб.: BHV – Санкт-Петербург, 1998.
4. Архангельский, А. Я. Delphi 7. Справочное пособие. / А. Я. Архангельский – М.: ООО «Бином-Пресс», 2003 г.
5. Хортон, А. Visual C++ 2005: базовый курс.: / А. Хортон – М.: ООО «И.Д. Вильям», 2007.
6. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. / Г. Буч. – М.: «Издательство Бином», СПб.: «Невский диалект», 2000.
7. Математика: Энциклопедия / Под ред. Ю.В. Прохорова. – М.: Большая Российская энциклопедия, 2003.
8. Пачеко, К. Delphi 5. Руководство Разработчика. Т. 1. Основные методы и технологии программирования / К. Пачеко, С. Тейксейр. – М.: ООО «И.Д. Вильямс», 2002.
9. Вальвачев, А. Н. Программирование на языке Delphi. Учебное пособие / А. Н. Вальвачев, К. А. Сурков, Д. А. Сурков [и др.]. – Russian Software Developer Network 19.11.2005.
10. Borland® Delphi™ for Windows 95 and Windows NT.

Е. В. Санников

Курс практического программирования в Delphi. Объектно – ориентированное программирование

Ответственный за выпуск
В. Митин

Обложка
СОЛОН-ПРЕСС

ОО «СОЛОН-ПРЕСС»
123001, г. Москва, а/я 82
Телефоны:
(499) 254-44-10, (499) 252-36-96, (499) 795-73-26
E-mail: avtor@coba.ru,
www.solon-press.ru

По вопросам приобретения обращаться:
ООО «АЛЬЯНС-БУКС»
Тел: (499) 725-54-09, 725-50-27, **www.alians-kniga.ru**

По вопросам подписки на журнал «Ремонт & Сервис» обращаться:
ООО «СОЛОН-ПРЕСС»
тел.: (499) 795-73-26, **e-mail: rem_serv@coba.ru**
www.remserv.ru

ООО «СОЛОН-ПРЕСС»
103050, г. Москва, Дегтярный пер., д. 5, стр. 2
Формат 60×88/8. Объем 23,5 п. л. Тираж 400 экз.

Заказ №