



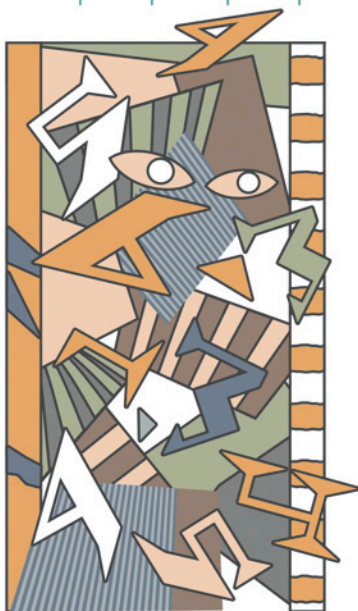
www.bhv.ru
www.bhv.kiev.ua

САМОУЧИТЕЛЬ

Никита Культин

Delphi 6

Программирование на Object Pascal



Работа в среде
Delphi 6

Графика, мультимедиа
и базы данных

Использование
MS HTML Help Workshop

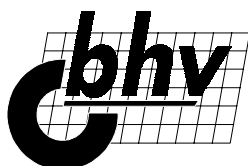


Содержит
примеры проектов

Никита Культин

Delphi 6.

Программирование на Object Pascal



Санкт-Петербург

Дюссельдорф ♦ Киев ♦ Москва ♦ Санкт-Петербург

Книга является руководством по программированию в среде Delphi 6. В ней рассматривается весь процесс создания программы: от разработки алгоритма и диалогового окна до отладки и создания справочной системы. Материал включает ряд тем, которые, как правило, остаются за рамками книг, адресованных начинающим программистам: обработка символьной информации, использование динамических структур, работа с файлами, создание справочной системы. Рассматриваются вопросы работы с графикой, мультимедиа и базами данных. Приведено описание процесса создания анимации в Macromedia Flash 5, создание справочной системы при помощи программы Microsoft HTML Help Workshop.

Книга отличается доступностью изложения, большим количеством наглядных примеров и адресована студентам, школьникам старших классов и всем изучающим программирование в учебном заведении или самостоятельно. Прилагаемая дискета содержит примеры приведенных в книге программ.

Для начинающих программистов

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зав. редакцией	<i>Наталья Таркова</i>
Редактор	<i>Михаил Кокорев</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Игоря Цырульников</i>
Зав. производством	<i>Николай Тверских</i>

Культин Н. Б.

Delphi 6. Программирование на Object Pascal. — СПб.: БХВ-Петербург, 2001. — 528 с.: ил.

ISBN 5-94157-112-7

© Н. Б. Культин, 2001

© Оформление, издательство "БХВ-Петербург", 2001

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 24.07.01.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 42,57.

Тираж 5000 экз. Заказ

"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99 от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

Содержание

Предисловие.....	11
Delphi — что это?	11
Об этой книге	12
Введение	13
Начало работы	13
Первый проект.....	16
Форма	16
Компоненты.....	20
Событие и процедура обработки события.....	28
Редактор кода.....	32
Система подсказок.....	32
Справочная система	33
Структура проекта	34
Сохранение проекта	37
Компиляция	39
Ошибки	40
Предупреждения и подсказки	41
Запуск программы	42
Ошибки времени выполнения	42
Внесение изменений	44
Окончательная настройка приложения.....	46
Создание уникального значка для приложения	46
Перенос приложения на другой компьютер.....	49
Глава 1. Основы программирования.....	51
Программа.....	51
Этапы разработки программы.....	51
Определение требований к программе.....	52
Разработка алгоритма.....	52
Кодирование	52

Отладка	52
Тестирование.....	52
Алгоритм и программа.....	53
Компиляция.....	58
Язык программирования Object Pascal.....	59
Тип данных	59
Целый тип.....	59
Вещественный тип.....	60
Символьный тип.....	60
Строковый тип.....	60
Логический тип.....	61
Переменная	61
Константы.....	62
Числовые константы	62
Строковые и символьные константы	63
Логические константы.....	63
Именованная константа.....	63
Инструкция присваивания	64
Выражение	64
Тип выражения	66
Выполнение инструкции присваивания.....	66
Стандартные функции	67
Арифметические функции	68
Функции преобразования типов	68
Использование функций.....	69
Ввод данных.....	69
Ввод из окна ввода	69
Ввод из поля редактирования.....	70
Вывод результатов	71
Вывод в окно сообщения.....	71
Вывод в поле диалогового окна	74
Процедуры и функции.....	74
Структура процедуры.....	75
Структура функции.....	76
Запись инструкций программы	77
Стиль программирования	79
Глава 2. Управляющие структуры Object Pascal	81
Условие.....	81
Выбор.....	84
Инструкция <i>IF</i>	85
Инструкция <i>CASE</i>	93
Циклы.....	103
Инструкция <i>FOR</i>	103
Инструкция <i>WHILE</i>	107
Инструкция <i>REPEAT</i>	110
Инструкция <i>GOTO</i>	113

Глава 3. Символы и строки	115
Символы.....	115
Строки	119
Операции со строками.....	120
Функция <i>LENGTH</i>	120
Процедура <i>DELETE</i>	121
Функция <i>POS</i>	121
Функция <i>COPY</i>	122
Глава 4. Консольное приложение	123
Инструкции <i>WRITE</i> и <i>WRITELN</i>	123
Инструкции <i>READ</i> и <i>READLN</i>	125
Создание консольного приложения.....	127
Глава 5. Массивы	131
Объявление массива.....	131
Операции с массивами	133
Вывод массива	133
Ввод массива	134
Использование компонента <i>StringGrid</i>	135
Использование компонента <i>Memo</i>	141
Поиск минимального (максимального) элемента массива.....	145
Поиск в массиве заданного элемента	148
Алгоритм простого перебора	148
Метод бинарного поиска	150
Сортировка массива	157
Сортировка методом прямого выбора	157
Сортировка методом обмена	160
Многомерные массивы.....	162
Ошибки при использовании массивов	169
Глава 6. Процедуры и функции	171
Функция.....	175
Объявление функции	175
Использование функции	177
Процедура	180
Объявление процедуры	181
Использование процедуры	182
Повторное использование функций и процедур	185
Создание модуля.....	185
Использование модуля.....	187
Глава 7. Файлы.....	191
Объявление файла	191
Назначение файла	192
Вывод в файл.....	192
Открытие файла для вывода.....	193

Ошибки открытия файла	195
Заккрытие файла	197
Пример программы	197
Ввод из файла	200
Открытие файла	200
Чтение данных из файла	201
Чтение чисел	202
Чтение строк	202
Определение конца файла	203
Глава 8. Типы данных, определяемые программистом	207
Перечисляемый тип	207
Интервальный тип	209
Запись	210
Объявление записи	211
Инструкция <i>WITH</i>	212
Ввод и вывод записей в файл	213
Вывод записи в файл	213
Ввод записи из файла	218
Динамические структуры данных	222
Указатели	223
Динамические переменные	224
Списки	226
Упорядоченный список	230
Добавление элемента в список	230
Удаление элемента из списка	233
Глава 9. Введение в объектно-ориентированное программирование	237
Класс	237
Объект	238
Метод	240
Инкапсуляция и свойства объекта	240
Наследование	243
Директивы <i>Protected</i> и <i>Private</i>	244
Полиморфизм и виртуальные методы	245
Классы и объекты Delphi	250
Глава 10. Графические возможности Delphi	251
Холст	251
Карандаш и кисть	252
Карандаш	252
Кисть	254
Вывод текста	257
Методы вычерчивания графических примитивов	259
Линия	259
Ломаная линия	262
Окружность и эллипс	265

Дуга	266
Прямоугольник.....	267
Многоугольник.....	268
Сектор.....	269
Точка.....	270
Вывод иллюстраций.....	274
Битовые образы	279
Мультипликация.....	281
Метод базовой точки	284
Использование битовых образов	287
Загрузка битового образа из ресурса программы	291
Создание файла ресурсов.....	291
Подключение файла ресурсов	294
Просмотр "мультика"	297
Глава 11. Мультимедийные возможности Delphi	301
Компонент <i>Animate</i>	301
Компонент <i>MediaPlayer</i>	307
Воспроизведение звука	309
Запись звука	314
Просмотр видеороликов и анимации	316
Создание анимации	319
Глава 12. Рекурсия	325
Понятие рекурсии	325
Примеры программ.....	328
Поиск файлов	328
Кривая Гильберта	332
Поиск пути.....	335
Поиск кратчайшего пути	342
Глава 13. Отладка программы.....	345
Классификация ошибок.....	345
Предотвращение и обработка ошибок.....	346
Отладчик.....	349
Трассировка программы	350
Точки останова программы.....	351
Добавление точки останова	351
Изменение характеристик точки останова.....	352
Удаление точки останова	353
Наблюдение значений переменных.....	353
Глава 14. Справочная система	357
Файл документа справочной информации.....	357
Создание справочной системы.....	360
Использование справочной системы	367
HTML Help Workshop	368

Подготовка справочной информации.....	369
Использование редактора Microsoft Word.....	369
Использование HTML Help Workshop.....	371
Создание файла справки	374
Компиляция	380
Вывод справочной информации.....	380

Глава 15. Примеры программ 385

Система проверки знаний	385
Требования к программе	385
Файл теста	386
Форма приложения	389
Вывод иллюстрации	392
Загрузка файла теста	394
Текст программы	396
Усовершенствование программы.....	406
Игра Сапер 2001	416
Правила	416
Представление данных.....	417
Форма приложения	419
Начало игры.....	420
Игра.....	423
Справочная информация.....	425
Информация о программе.....	426
Листинги.....	428

Глава 16. Компонент программиста 439

Выбор базового класса.....	439
Создание модуля компонента.....	440
Тестирование модуля компонента.....	444
Установка компонента.....	447
Ресурсы компонента	447
Установка	449
Ошибки при установке компонента.....	451
Тестирование компонента	451
Удаление компонента	454
Настройка палитры компонентов.....	456

Глава 17. Базы данных..... 459

Классификация баз данных	459
Локальная база данных	459
Удаленная база данных.....	460
Структура базы данных.....	461
Модель базы данных в Delphi.....	462
Псевдоним базы данных.....	462
Создание базы данных.....	463
Создание каталога	463

Создание псевдонима	464
Создание таблицы	466
Программа управления базой данных.....	473
Доступ к файлу данных (таблице).....	475
Просмотр базы данных	477
Режим формы.....	477
Режим таблицы	484
Выбор информации из базы данных.....	489
Динамически создаваемые псевдонимы	495
Перенос программы управления базой данных на другой компьютер	498
Заключение.....	500
Приложение 1. Язык Pascal (краткий справочник)	501
Зарезервированные слова и директивы	501
Структура модуля	502
Основные типы данных.....	502
Строки	503
Массив.....	503
Запись.....	504
Инструкции выбора	504
Инструкция <i>if</i>	504
Инструкция <i>case</i>	505
Циклы.....	506
Инструкция <i>for</i>	506
Инструкция <i>repeat</i>	507
Инструкция <i>while</i>	507
Безусловный переход.....	507
Инструкция <i>Go To</i>	507
Объявление функции.....	508
Объявление процедуры	508
Стандартные функции и процедуры	509
Приложение 2. Кодировка символов в Windows	511
Приложение 3. Представление информации в компьютере	514
Десятичные и двоичные числа	514
Память компьютера.....	515
Приложение 4. Рекомендуемая дополнительная литература	517
Приложение 5. Описание дискеты	518
Предметный указатель.....	524

Предисловие

Delphi — что это?

Интерес к программированию постоянно растет. Это связано с развитием и внедрением в повседневную жизнь информационных технологий. Если человек имеет дело с компьютером, то рано или поздно у него возникает желание, а иногда и необходимость, научиться программировать.

Среди пользователей персональных компьютеров в настоящее время наиболее популярно семейство операционных систем Windows и, естественно, что тот, кто собирается программировать, стремится писать программы, которые будут работать в этих системах.

Несколько лет назад рядовому программисту оставалось только мечтать о создании собственных программ, работающих в среде Windows, т. к. единственным средством разработки был Borland C++ for Windows, явно ориентированный на профессионалов, обладающих серьезными знаниями и опытом.

Бурное развитие вычислительной техники, потребность в эффективных средствах разработки программного обеспечения привели к появлению систем программирования, ориентированных на так называемую "быструю разработку", среди которых можно выделить Borland Delphi и Microsoft Visual Basic. В основе систем быстрой разработки (RAD-систем, Rapid Application Development — среда быстрой разработки приложений) лежит технология визуального проектирования и событийного программирования, суть которой заключается в том, что среда разработки берет на себя большую часть генерации кода программы, оставляя программисту работу по конструированию диалоговых окон и функций обработки событий. Производительность программиста при использовании RAD-систем — фантастическая!

Delphi — это среда быстрой разработки, в которой в качестве языка программирования используется Object Pascal. В основе идеологии Delphi лежит

технология визуального проектирования и методология объектно-ориентированного событийного программирования.

Примечание

Delphi 6 может работать в среде операционных систем от Windows 98 до Windows 2000. Особых требований к ресурсам компьютера система не предъявляет: процессор должен быть типа Pentium или Celeron, оперативной памяти — не менее 32 Мбайт, достаточное количество свободного дискового пространства — порядка 200 Мбайт.

Об этой книге

В книге, которая посвящена программированию в конкретной среде разработки, необходим баланс между тремя линиями: язык программирования, программирование как таковое, среда разработки. Уже при первом знакомстве со средой разработки, представления ее возможностей у автора возникает проблема: чтобы объяснить процесс разработки программы, описать, как работает программа, нужно оперировать такими терминами, как объект, событие, свойство, понимание которых на начальном этапе изучения программирования весьма проблематично. Как поступить? Сначала дать описание языка, а затем приступить к описанию среды разработки и процесса программирования в Delphi? Очевидно, что это не лучший вариант. Поэтому при изложении материала принят подход, в основу которого положен принцип соблюдения баланса между Object Pascal, методами программирования и средой разработки. В начале книги некоторые понятия, без которых просто невозможно изложение материала, даются на уровне определений.

Книга, которую вы держите в руках, — это не описание языка Object Pascal или среды разработки Delphi. Данная книга представляет собой учебное пособие по программированию на языке Object Pascal в среде Delphi 6. В ней рассмотрен весь процесс создания программы: от разработки диалогового окна и функций обработки событий до создания справочной системы и установочной дискеты.

Цель создания книги может быть сформулирована так: научить программировать в среде Delphi, т. е. создавать законченные программы различного назначения: от игровых до профессиональных утилит.

Научиться программировать можно, только программируя, решая конкретные задачи. При этом достигнутые в программировании успехи в значительной степени зависят от опыта. Поэтому, чтобы получить максимальную пользу от книги, вы должны работать с ней активно. Не занимайтесь просто чтением примеров, реализуйте их с помощью вашего компьютера. Не бойтесь экспериментировать — вносите изменения в программы. Чем больше вы сделаете самостоятельно, тем большему вы научитесь!

Введение

На примере программы вычисления скорости бега демонстрируется технология визуального проектирования и событийного программирования, вводятся основные понятия и термины.

Начало работы

Запускается Delphi обычным образом (рис. 1), т. е. выбором из меню **Borland Delphi 6** команды **Delphi 6**.

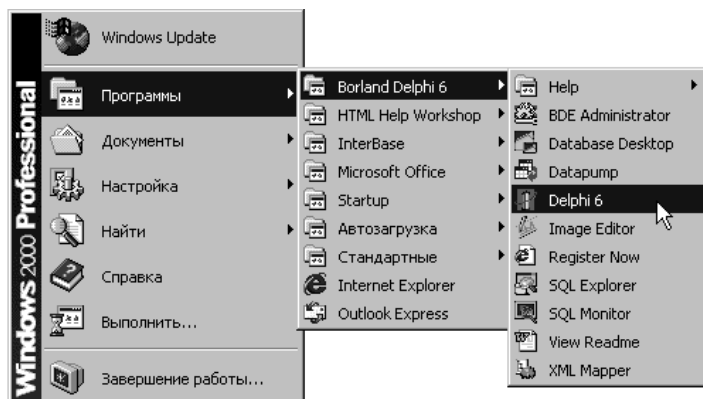


Рис. 1. Запуск Delphi

Вид экрана после запуска Delphi несколько необычен (рис. 2). Вместо одного окна на экране появляются пять:

- ☐ главное окно (**Delphi 6**);
- ☐ окно стартовой формы (**Form 1**);

- ☐ окно редактора свойств объектов (**Object Inspector** (Инспектор объектов));
- ☐ окно просмотра списка объектов (**Object TreeView** (Просмотр дерева объектов));
- ☐ окно редактора кода (**Unit1.pas**).

Окно редактора кода почти полностью закрыто окном стартовой формы.

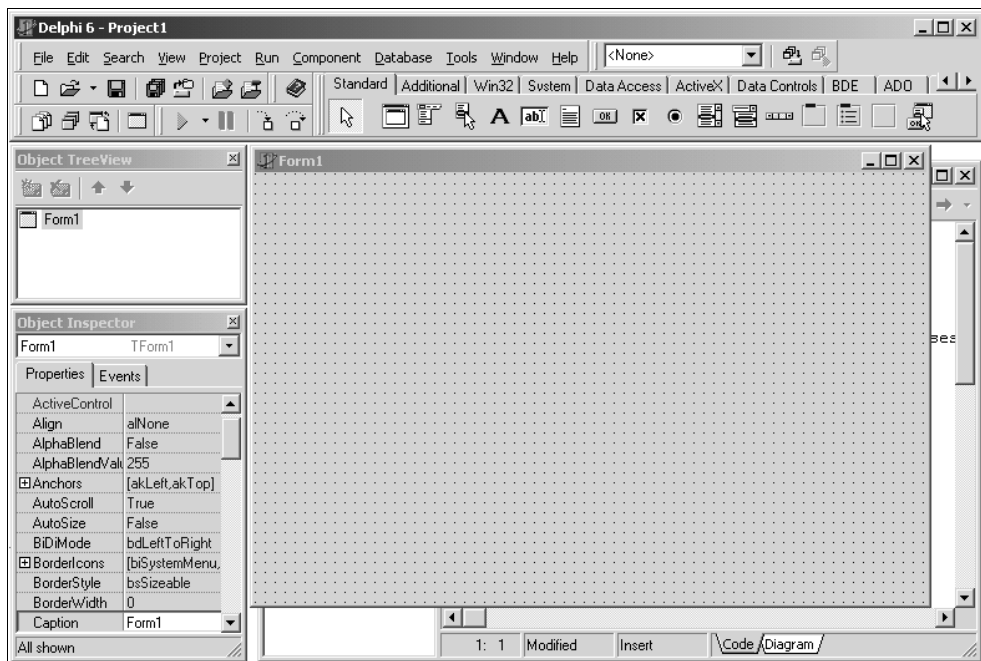


Рис. 2. Вид экрана после запуска Delphi

В главном окне (рис. 3) находится меню команд, панели инструментов и палитра компонентов.

Окно стартовой формы представляет собой заготовку главного окна разрабатываемого приложения.

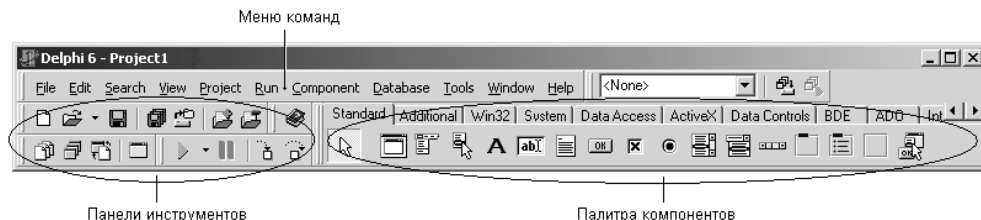
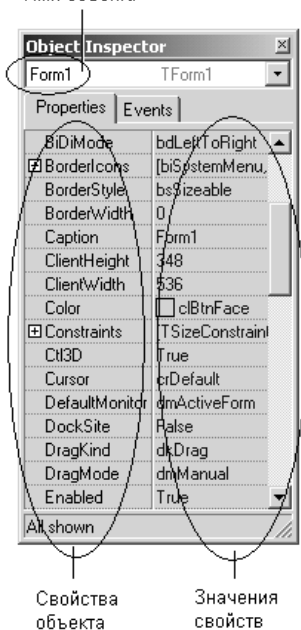


Рис. 3. Главное окно

Примечание

Программное обеспечение принято делить на системное и прикладное. Системное программное обеспечение — это все то, что составляет операционную систему. Остальные программы принято считать прикладными. Для краткости прикладные программы называют приложениями.

Имя объекта



Окно **Object Inspector** (рис. 4) — окно редактора свойств объектов предназначено для редактирования значений свойств объектов. В терминологии визуального проектирования *объекты* — это диалоговые окна и элементы управления (поля ввода и вывода, командные кнопки, переключатели и др.). *Свойства объекта* — это характеристики, определяющие вид, положение и поведение объекта. Например, свойства `Width` и `Height` задают размер (ширину и высоту) формы, свойства `Top` и `Left` — положение формы на экране, свойство `Caption` — текст заголовка.

Рис. 4. Окно редактора свойств объекта

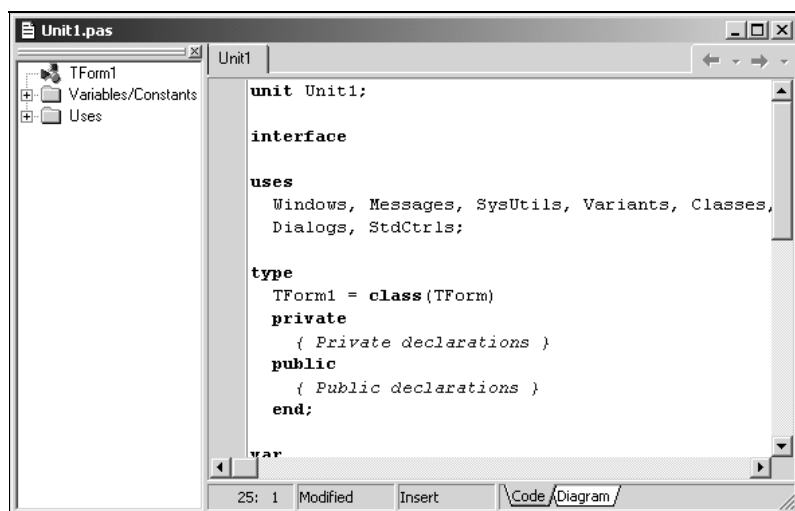


Рис. 5. Окно редактора кода

В окне редактора кода (рис. 5), которое можно увидеть, отодвинув в сторону окно формы, следует набирать текст программы. В начале работы над новым проектом это окно редактора кода содержит сформированный Delphi шаблон программы.

Первый проект

Для демонстрации возможностей Delphi и технологии визуального проектирования разработаем программу, вычисляющую скорость, с которой спортсмен пробежал дистанцию. Окно программы во время ее работы приведено на рис. 6.

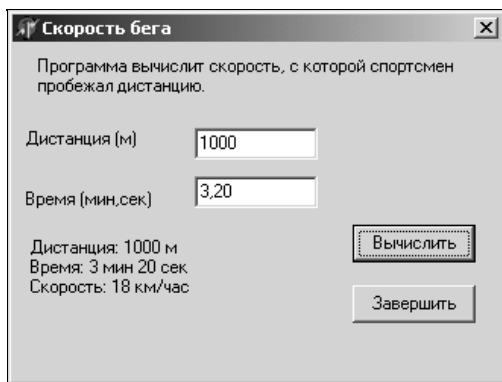


Рис. 6. Окно программы вычисления скорости бега

Для начала работы над новой программой запустите Delphi. Если вы уже работаете в среде разработки и у вас загружен другой проект, выберите из меню **File** (Файл) команду **New/Application** (Создать/Приложение).

Форма

Работа над *новым проектом*, так в Delphi называется разрабатываемое приложение, начинается с создания стартовой формы.

На этапе разработки программы диалоговые окна называют *формами*.

Стартовая форма создается путем изменения значений свойств формы **Form1** и добавления к форме необходимых компонентов (полей ввода и вывода текста, командных кнопок).

Свойства формы (табл. 1) определяют ее внешний вид: размер, положение на экране, текст заголовка, вид рамки.

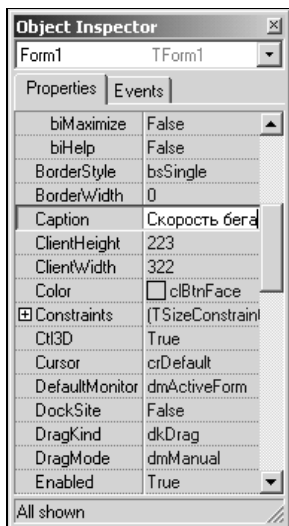
Для просмотра и изменения значений свойств формы и ее компонентов используется окно **Object Inspector**. В верхней части окна **Object Inspector** указано имя объекта, значения свойств которого отображается в данный мо-

мент. В левой колонке вкладки **Properties** (Свойства) перечислены свойства объекта, а в правой — указаны их значения.

Таблица 1. Свойства формы

Свойство	Описание
Name	Имя формы. В программе имя формы используется для управления формой и доступа к компонентам формы
Caption	Текст заголовка
Width	Ширина формы
Height	Высота формы
Top	Расстояние от верхней границы формы до верхней границы экрана
Left	Расстояние от левой границы формы до левой границы экрана
BorderStyle	Вид границы. Граница может быть обычной (biZizeable), тонкой (biSingle) или отсутствовать (biNone). Если у окна обычная граница, то во время работы программы пользователь может при помощи мыши изменить размер окна. Изменить размер окна с тонкой границей нельзя. Если граница отсутствует, то на экран во время работы программы будет выведено окно без заголовка. Положение и размер такого окна во время работы программы изменить нельзя
BorderIcons	Кнопки управления окном. Значение свойства определяет, какие кнопки управления окном будут доступны пользователю во время работы программы. Значение свойства задается путем присвоения значений уточняющим свойствам biSystemMenu, biMinimize, biMaximize и biHelp. Свойство biSystemMenu определяет доступность кнопки Свернуть и кнопки системного меню, biMinimize — кнопки Свернуть , biMaximize — кнопки Развернуть , biHelp — кнопки вывода справочной информации
Icon	Значок в заголовке диалогового окна, обозначающий кнопку вывода системного меню
Color	Цвет фона. Цвет можно задать, указав название цвета или привязку к текущей цветовой схеме операционной системы. Во втором случае цвет определяется текущей цветовой схемой, выбранным компонентом привязки и меняется при изменении цветовой схемы операционной системы
Font	Шрифт. Шрифт, используемый "по умолчанию" компонентами, находящимися на поверхности формы. Изменение свойства Font формы приводит к автоматическому изменению свойства Font компонента, располагающегося на поверхности формы. То есть компоненты наследуют свойство Font от формы (имеется возможность запретить наследование)

При создании формы в первую очередь следует изменить значение свойства `Caption` (Заголовок). В нашем примере надо заменить текст `Form1` на "Скорость бега". Чтобы это сделать, нужно в окне **Object Inspector** щелкнуть мышью в строке `Caption`, в результате чего будет выделено текущее значение свойства, в строке появится курсор, и можно будет ввести текст "Скорость бега" (рис. 7).



Аналогичным образом можно установить значения свойств `Height` и `Width`, которые определяют высоту и ширину формы. Этим свойствам надо присвоить значения 250 и 330, соответственно.

Рис. 7. Изменение значения свойства `Caption`

Примечание

Размер формы и ее положение на экране, а также размер других элементов управления и их положение на поверхности формы, задают в пикселах, т. е. точках экрана.

Форма — это обычное окно. Поэтому его размер можно изменить точно так же, как любого другого окна, т. е. захватом и перемещением (с помощью мыши) границы. По окончании перемещения границ автоматически изменятся значения свойств `Height` и `Width`. Они будут соответствовать установленному размеру формы.

Положение диалогового окна на экране после запуска программы соответствует положению формы во время ее разработки, которое определяется значением свойств `Top` (отступ от верхней границы экрана) и `Left` (отступ от левой границы экрана). Значения этих свойств также можно задать путем перемещения окна формы при помощи мыши.

При выборе некоторых свойств, например, `BorderStyle`, справа от текущего значения свойства появляется значок раскрывающегося списка. Очевидно, что значение таких свойств можно задать путем выбора из списка (рис. 8).

Некоторые свойства являются сложными, т. е. их значение определяется совокупностью значений других (уточняющих) свойств. Перед именами

сложных свойств стоит значок "+", при щелчке на котором раскрывается список уточняющих свойств (рис. 9). Например, свойство `BorderIcons` определяет, какие кнопки управления окном будут доступны во время работы программы. Так, если свойству `biMaximize` присвоить значение `false`, то во время работы программы кнопки **Развернуть** в заголовке окна не будет.

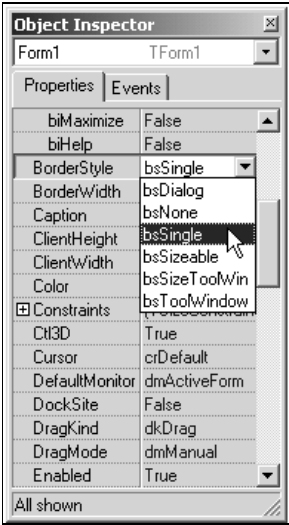


Рис. 8. Установка значения свойства путем выбора из списка

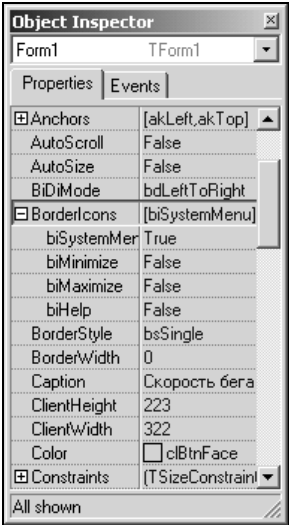


Рис. 9. Раскрытый список вложенных свойств сложного свойства `BorderIcons`

Рядом со значениями некоторых свойств отображается командная кнопка с тремя точками. Это значит, что для задания значения свойства можно воспользоваться дополнительным диалоговым окном. Например, значение сложного свойства `Font` можно задать путем непосредственного ввода значений уточняющих свойств, а можно воспользоваться стандартным диалоговым окном выбора шрифта.

В табл. 2 перечислены свойства формы разрабатываемой программы, которые следует изменить. Остальные свойства оставлены без изменения и в таблице не приведены.

Таблица 2

Свойство	Значение
Caption	Скорость бега
Height	250

Таблица 2 (окончание)

Свойство	Значение
Width	330
BorderStyle	BsSingle
BorderIcons.biMinimize	False
BorderIcons.biMaximize	False
Font.Size	10

В приведенной таблице в именах некоторых свойств есть точка. Это значит, что надо задать значение уточняющего свойства. После того как будут установлены значения свойств главной формы, она должна иметь вид, приведенный на рис. 10.

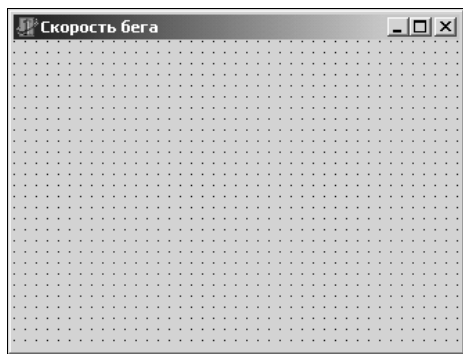


Рис. 10. Так выглядит форма после установки значений свойств

Компоненты

Программа вычисления скорости бега должна получить от пользователя исходные данные — длину дистанции и время, за которое спортсмен пробежал дистанцию. В подобных программах данные с клавиатуры, как правило, вводят в поля редактирования. Поэтому в форму надо добавить компонент `Edit` — поле редактирования.

Наиболее часто используемые компоненты находятся на вкладке **Standard** (рис. 11).

Для того чтобы добавить в форму компонент, необходимо в палитре компонентов выбрать этот компонент, щелкнув левой кнопкой мыши на его пиктограмме, далее установить курсор в ту точку формы, в которой должен быть левый верхний угол компонента и еще раз щелкнуть левой кнопкой мыши. В результате в форме появляется компонент стандартного размера.

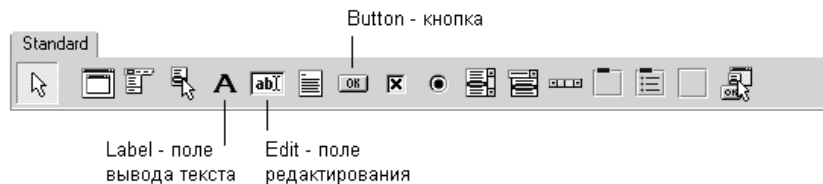


Рис. 11. Вкладка **Standard** содержит наиболее часто используемые компоненты

Размер компонента можно задать в процессе его добавления к форме. Для этого надо после выбора компонента из палитры поместить курсор мыши в ту точку формы, где должен находиться левый верхний угол компонента, нажать левую кнопку мыши и, удерживая ее нажатой, переместить курсор в точку, где должен находиться правый нижний угол компонента, и отпустить кнопку мыши. На форме появится компонент нужного размера.

Каждому компоненту Delphi присваивает имя, которое состоит из названия компонента и его порядкового номера. Например, если к форме добавить два компонента `Edit`, то их имена будут `Edit1` и `Edit2`. Программист, путем изменения значения свойства `Name`, может изменить имя компонента. В простых программах имена компонентов, как правило, не изменяют.

На рис. 12 приведен вид формы после добавления двух компонентов `Edit` полей редактирования, предназначенных для ввода исходных данных. Один из компонентов выделен. Свойства выделенного компонента отображаются в окне **Object Inspector**. Чтобы увидеть свойства другого компонента, надо щелкнуть левой кнопкой мыши на изображении нужного компонента. Можно также выбрать имя компонента в окне **Object TreeView** или из находящегося в верхней части окна **Object Inspector** раскрывающегося списка объектов.

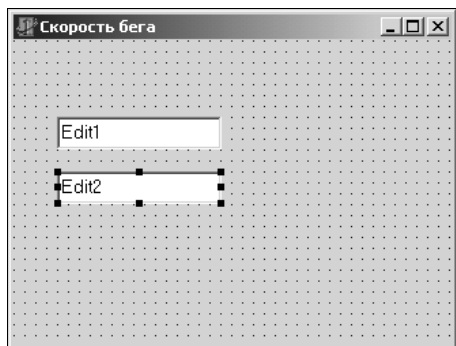


Рис. 12. Форма после добавления двух компонентов `Edit`

В табл. 3 перечислены основные свойства компонента `Edit` — поля редактирования.

Таблица 3

Свойство	Описание
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам, в частности — для доступа к тексту, введенному в поле редактирования
Text	Текст, находящийся в поле ввода и редактирования
Left	Расстояние от левой границы компонента до левой границы формы
Top	Расстояние от верхней границы компонента до верхней границы формы
Height	Высота поля
Width	Ширина поля
Font	Шрифт, используемый для отображения вводимого текста
ParentFont	Признак наследования компонентом характеристик шрифта формы, на которой находится компонент. Если значение свойства равно True, то при изменении свойства Font формы автоматически меняется значение свойства Font компонента

Delphi позволяет изменить размер и положение компонента при помощи мыши.

Для того чтобы изменить положение компонента, необходимо установить курсор мыши на его изображение, нажать левую кнопку мыши и, удерживая ее нажатой, переместить контур компонента в нужную точку формы, затем отпустить кнопку мыши. Во время перемещения компонента (рис. 13) отображаются текущие значения координат левого верхнего угла компонента (значения свойств Left и Top).

Для того чтобы изменить размер компонента, необходимо его выделить, установить указатель мыши на один из маркеров, помечающих границу компонента, нажать левую кнопку мыши и, удерживая ее нажатой, изменить положение границы компонента. Затем отпустить кнопку мыши. Во время изменения размера компонента отображаются текущие значения свойств Height и Width (рис. 14).

Свойства компонента так же, как и свойства формы, можно изменить при помощи **Object Inspector**. Для того чтобы свойства требуемого компонента были выведены в окне **Object Inspector**, нужно выделить этот компонент (щелкнуть мышью на его изображении). Можно также выбрать компонент

из находящегося в верхней части окна **Object Inspector** раскрывающегося списка объектов (рис. 15) или из списка в окне **Object TreeView** (рис. 16).

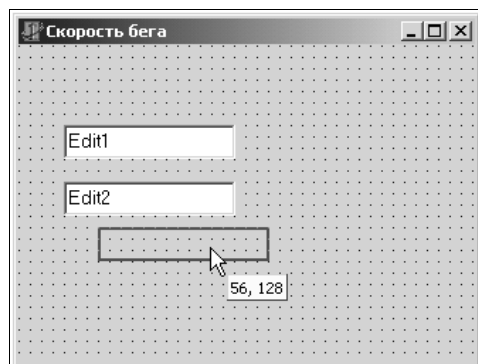


Рис. 13. Отображение текущих значений свойств *Left* и *Top* при изменении положения компонента

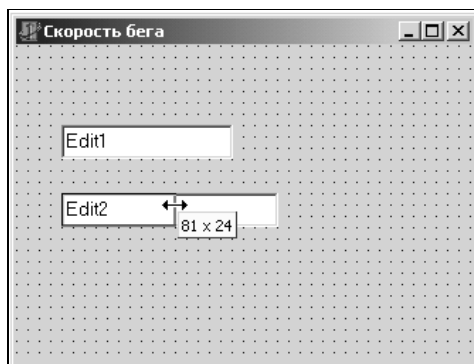


Рис. 14. Отображение текущих значений свойств *Height* и *Width* при изменении размера компонента

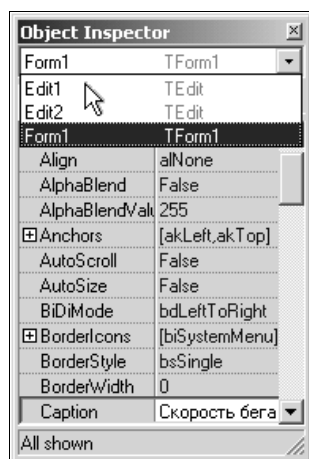


Рис. 15. Выбор компонента из списка в окне **Object Inspector**

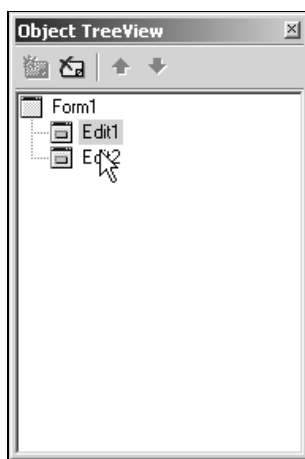


Рис. 16. Выбор компонента в окне **Object TreeView**

В табл. 4 приведены значения свойств полей редактирования *Edit1* и *Edit2*. Компонент *Edit1* предназначен для ввода длины дистанции, *Edit2* — для ввода времени.

Обратите внимание на то, что значением свойства *Text* обоих компонентов является пустая строка.

Таблица 4

Name	Edit1	Edit2
Text		
Top	56	88
Left	128	128
Height	21	21
Width	121	121

Помимо полей редактирования в окне программы должна находиться краткая информация о программе и назначении полей ввода. Для вывода текста в форму используют поля вывода текста. Поле вывода текста — это компонент Label. Значок компонента Label находится на вкладке **Standard** (рис. 17). Добавляется компонент Label в форму точно так же, как и поле редактирования.



Рис. 17. Компонент Label — поле вывода текста

В форму разрабатываемого приложения надо добавить четыре компонента Label. Первое поле предназначено для вывода информационного сообщения, второе и третье — для вывода информации о назначении полей ввода, четвертое поле — для вывода результата расчета — скорости.

Свойства компонента Label перечислены в табл. 5.

Таблица 5

Свойство	Пояснение
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
Caption	Отображаемый текст
Font	Шрифт, используемый для отображения текста
ParentFont	Признак наследования компонентом характеристик шрифта формы, на которой находится компонент. Если значение свойства равно True, текст выводится шрифтом, установленным для формы
AutoSize	Признак того, что размер поля определяется его содержимым

Таблица 5 (окончание)

Свойство	Пояснение
Left	Расстояние от левой границы поля вывода до левой границы формы
Top	Расстояние от верхней границы поля вывода до верхней границы формы
Height	Высота поля вывода
Width	Ширина поля вывода
WordWrap	Признак того, что слова, которые не помещаются в текущей строке, автоматически переносятся на следующую строку

Следует обратить внимание на свойства `AutoSize` и `WordWrap`. Эти свойства нужно использовать, если поле вывода должно содержать несколько строк текста. После добавления к форме компонента `Label` значение свойства `AutoSize` равно `True`, т. е. размер поля определяется автоматически в процессе изменения значения свойства `Caption`. Если вы хотите, чтобы находящийся в поле вывода текст занимал несколько строк, то надо сразу после добавления к форме компонента `Label` присвоить свойству `AutoSize` значение `False`, свойству `WordWrap` — значение `True`. Затем изменением значений свойств `Width` и `Height` нужно задать требуемый размер поля. Только после этого можно ввести в свойство `Caption` текст, который должен быть выведен в поле.

После добавления полей вывода текста (четырёх компонентов `Label`) и установки значений их свойств в соответствии с табл. 6, форма программы принимает вид, приведенный на рис. 18.

Обратите внимание, что значение свойства `Caption` вводится как одна строка. Расположение текста внутри поля вывода определяется размером поля, значением свойств `AutoSize` и `WordWrap`, а также зависит от характеристик используемого для вывода текста шрифта.

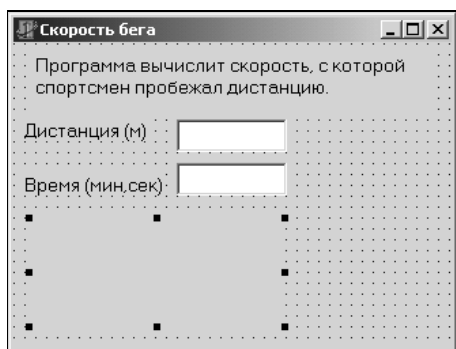


Рис. 18. Вид формы после добавления полей вывода текста

Таблица 6

Свойство	Компонент			
Name	Label1	Label2	Label3	Label4
AutoSize	False	True	True	False
WordWrap	True	False	False	True
Caption	Программа вычислит скорость, с которой спортсмен пробежал дистанцию	Дистанция (метров)	Время (мин, сек)	
Top	8	56	88	120
Left	8	8	8	8
Height	33	13	13	41
Width	209	102	832	273

Последнее, что надо сделать на этапе создания формы — добавить в форму две командные кнопки: **Вычислить** и **Завершить**. Назначение этих кнопок очевидно.

Командная кнопка, компонент `Button`, добавляется в форму точно так же, как и другие компоненты. Значок компонента `Button` находится на вкладке **Standard** (рис. 19). Свойства компонента приведены в табл. 7.



Рис. 19. Командная кнопка — компонент `Button`

Таблица 7. Свойства компонента `Button`

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 <code>True</code> , и не доступна, если значение свойства равно <code>False</code>
Left	Расстояние от левой границы кнопки до левой границы формы
Top	Расстояние от верхней границы кнопки до верхней границы формы

Таблица 7 (окончание)

Свойство	Описание
Height	Высота кнопки
Width	Ширина кнопки

После добавления к форме двух командных кнопок нужно установить значения их свойств в соответствии с табл. 8.

Таблица 8

Name	Button1	Button2
Caption	Вычислить	Завершить
Top	176	176
Left	16	112
Height	25	25
Width	75	75

Окончательный вид формы разрабатываемого приложения приведен на рис. 20.

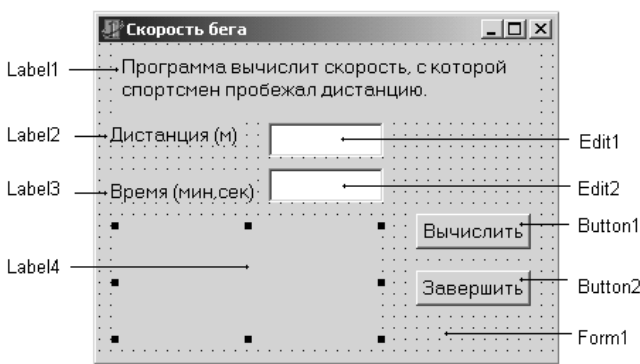


Рис. 20. Форма программы **Скорость бега**

Завершив работу по созданию формы приложения, можно приступить к написанию текста программы. Но перед этим обсудим очень важные при программировании под Windows понятия:

- ☐ событие;
- ☐ процедура обработки события.

Событие и процедура обработки события

Вид созданной формы подсказывает, как работает приложение. Очевидно, что пользователь должен ввести в поля редактирования исходные данные и щелкнуть мышью на кнопке **Вычислить**. Щелчок на изображении командной кнопки — это пример того, что в Windows называется *событием*.

Событие (Event) — это то, что происходит во время работы программы. В Delphi каждому событию присвоено имя. Например, щелчок кнопкой мыши — это событие `OnClick`, двойной щелчок мышью — событие `OnDblClick`.

В табл. 9 приведены некоторые события Windows.

Таблица 9

Событие	Происходит
<code>OnClick</code>	При щелчке кнопкой мыши
<code>OnDblClick</code>	При двойном щелчке кнопкой мыши
<code>OnMouseDown</code>	При нажатии кнопки мыши
<code>OnMouseUp</code>	При отпускании кнопки мыши
<code>OnMouseMove</code>	При перемещении мыши
<code>OnKeyPress</code>	При нажатии клавиши клавиатуры
<code>OnKeyDown</code>	При нажатии клавиши клавиатуры. События <code>OnKeyDown</code> и <code>OnKeyPress</code> — это чередующиеся, повторяющиеся события, которые происходят до тех пор, пока не будет отпущена удерживаемая клавиша (в этот момент происходит событие <code>OnKeyUp</code>)
<code>OnKeyUp</code>	При отпускании нажатой клавиши клавиатуры
<code>OnCreate</code>	При создании объекта (формы, элемента управления). Процедура обработки этого события обычно используется для инициализации переменных, выполнения подготовительных действий
<code>OnPaint</code>	При появлении окна на экране в начале работы программы, после появления части окна, которая, например, была закрыта другим окном и в других случаях
<code>OnEnter</code>	При получении элементом управления фокуса
<code>OnExit</code>	При потере элементом управления фокуса

Реакцией на событие должно быть какое-либо действие. В Delphi реакция на событие реализуется как *процедура обработки события*. Таким образом, для того, чтобы программа выполняла некоторую работу в ответ на действия пользователя, программист должен написать процедуру обработки соответ-

ствующего события. Следует обратить внимание, на то, что значительную часть обработки событий берет на себя компонент. Поэтому программист должен разрабатывать процедуру обработки события только в том случае, если реакция на событие отличается от стандартной или не определена. Например, если по условию задачи ограничений на символы, вводимые в поле `Edit`, нет, то процедуру обработки события `onKeyPress` писать не надо, т. к. во время работы программы будет использована стандартная (скрытая от программиста) процедура обработки этого события.

Методику создания процедур обработки событий рассмотрим на примере процедуры обработки события `OnClick` для командной кнопки **Вычислить**.

Чтобы приступить к созданию процедуры обработки события, надо сначала в окне **Object Inspector** выбрать компонент, для которого создается процедура обработки события. Затем, в этом же окне нужно выбрать вкладку **Events** (События).

В левой колонке вкладки **Events** (рис. 21) перечислены имена событий, которые может воспринимать выбранный компонент (объект). Если для события определена (написана) процедура обработки события, то в правой колонке, рядом с именем события, выводится имя этой процедуры.

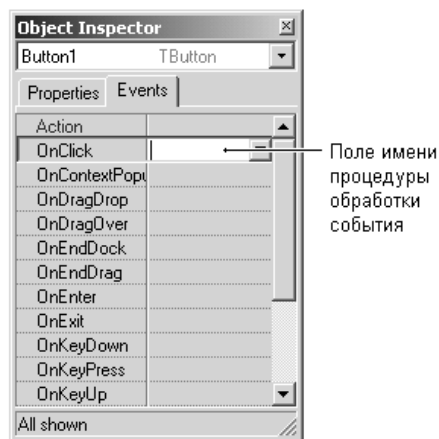


Рис. 21. На вкладке **Events** перечислены события, которые может воспринимать компонент

Для того чтобы создать функцию обработки события, нужно сделать двойной щелчок мышью в поле имени процедуры обработки соответствующего события. В результате этого откроется окно редактора кода, в которое будет добавлен шаблон функции обработки события, а в окне **Object Inspector**, рядом с именем события, появится имя функции его обработки (рис. 22).

Delphi присваивает функции обработки события имя, которое состоит из двух частей. Первая часть имени идентифицирует форму, содержащую объект (компонент), для которого создана процедура обработки события. Вторая часть имени идентифицирует сам объект и событие. В нашем примере имя формы — `Form1`, имя командной кнопки — `Button1`, а имя события — `Click`.

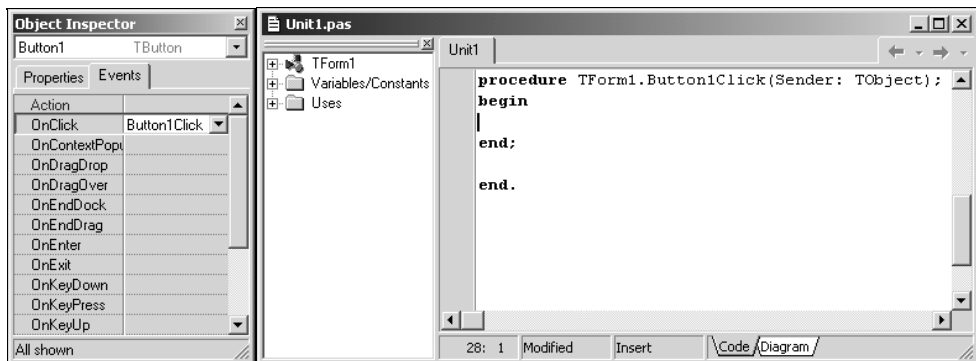


Рис. 22. Пример процедуры обработки события, сгенерированной Delphi

В окне редактора кода между словами `begin` и `end` можно печатать инструкции, реализующие функцию обработки события.

В листинге 1 приведен текст функции обработки события `OnClick` для командной кнопки **Вычислить**. Обратите внимание на то, как представлена программа. Ее общий вид соответствует тому, как она выглядит в окне редактора кода: ключевые слова выделены полужирным, комментарии — курсивом (выделение выполняет редактор кода). Кроме того, инструкции программы набраны с отступами, в соответствии с принятыми в среде программистов правилами хорошего стиля.

Листинг 1. Процедура обработки события `OnClick` на кнопке **Вычислить**

```
// нажатие кнопки Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    dist : integer; // дистанция, метров
    t:     real;    // время, как дробное число

    min : integer; // время, минуты
    sek : integer; // время, секунды

    v: real;       // скорость
begin
    // получить исходные данные из полей ввода
    dist := StrToInt(Edit1.Text);
    t := StrToFloat(Edit2.Text);

    // предварительные преобразования
    min := Trunc(t); // кол-во минут — это целая часть числа t
    sek := Trunc(t*100) mod 100; // кол-во секунд — это дробная часть
                                // числа t
```

```
// вычисление
v := (dist/1000) / ((min*60 + sek)/3600);

// вывод результата
label4.Caption := 'Дистанция: ' + Edit1.Text + ' м' + #13 +
                  'Время: ' + IntToStr(min) + ' мин ' +
                  IntToStr(sek) + ' сек ' + #13 +
                  'Скорость: ' + FloatToStrF(v, ffFixed, 4, 2) +
                  ' км/час';

end;
```

Функция `Button1Click` выполняет расчет скорости и выводит результат расчета в поле `Label4`. Исходные данные вводятся из полей редактирования `Edit1` и `Edit2` путем обращения к свойству `Text`. Свойство `Text` содержит строку символов, которую во время работы программы введет пользователь. Для правильной работы программы строка должна содержать только цифры. Для преобразования строки в числа в программе используются функции `StrToInt` и `StrToFloat`. Функция `StrToInt` проверяет символы строки, переданной ей в качестве параметра (`Edit1.Text` — это содержимое поля `Edit1`), на допустимость, и, если все символы верные, возвращает соответствующее число. Это число записывается в переменную `dist`. Аналогичным образом работает функция `StrToFloat`, которая возвращает дробное число, соответствующее содержимому поля `Edit2`. Это число записывается в переменную `t`.

После того как исходные данные будут помещены в переменные `dist` и `t`, выполняются подготовительные действия и расчет. Первоначально с использованием функции `Trunc`, которая "отбрасывает" дробную часть числа, выделяется целая часть переменной `t` — это количество минут. Значением выражения `Trunc(t*100) mod 100` является количество секунд. Вычисляется это выражение так. Сначала число `t` умножается на 100. Полученное значение передается функции `Trunc`, которая возвращает целую часть результата умножения `t` на 100. Полученное таким образом число делится *по модулю* на 100. Результат деления по модулю — это остаток от деления.

После того как все данные готовы, выполняется расчет. Так как скорость должна быть выражена в км/час, то значения дистанции и времени, выраженные в метрах и минутах, преобразуются в километры и часы.

Вычисленное значение скорости выводится в поле `Label4` путем присваивания значения свойству `Caption`. Для преобразования чисел в строки используются функции `IntToStr` и `FloatToStr`.

В результате нажатия кнопки **Завершить** программа должна завершить работу. Чтобы это произошло, надо закрыть, убрать с экрана, главное окно программы. Делается это при помощи метода `Close`. Процедура обработки события `OnClick` для кнопки **Завершить** приведена в листинге 2.

Листинг 2. Процедура обработки события OnClick на кнопке Завершить

```
// нажатие кнопки Завершить
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.Close; // закрыть главное окно программы
end;
```

Редактор кода

Редактор кода выделяет ключевые слова языка программирования (`procedure`, `var`, `begin`, `end`, `if` и др.) полужирным шрифтом, что делает текст программы более выразительным, и облегчает восприятие структуры программы.

Помимо ключевых слов редактор кода выделяет курсивом комментарии.

Система подсказок

Во время набора текста редактор выводит справочную информацию о параметрах процедур и функций, о свойствах и методах объектов.

Например, если в окне редактора набрать `ShowMessage`, то на экране появится окно, в котором будут перечислены параметры функции `ShowMessage` и указан их тип (рис. 23). Один из параметров выделен полужирным. Так редактор подсказывает программисту, какой параметр он должен вводить. После набора параметра и запятой в окне подсказки будет выделен следующий параметр. И так до тех пор, пока не будут указаны все параметры.

Для объектов редактор кода выводит список свойств и методов. Как только программист наберет имя объекта (компонента) и точку, так сразу в окне подсказки появится список свойств и методов этого объекта (рис. 24). Перейти к нужному элементу списка можно, используя клавиши перемещения курсора или набирая с клавиатуры несколько первых букв имени свойства или метода. После того как будет выбран нужный элемент списка и нажата клавиша `<Enter>`, выбранное свойство или метод будут вставлены в текст программы.

Система подсказок существенно облегчает процесс подготовки текста программы, избавляет от рутины. Кроме того, если во время набора программы подсказка не появилась, то это значит, что программист допустил ошибку, скорее всего, неверно набрал имя процедуры или функции.

```
const Msg: String; DlgType: TMsgDlgType; Buttons: TMsgDlgButtons;
HelpCtx: Integer
```

MessageDlg(|

Рис. 23. Пример подсказки

```
dist := StrToInt(Edit1.)
```

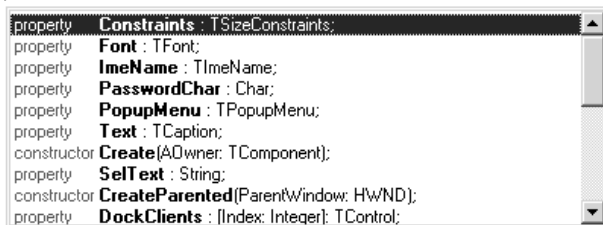


Рис. 24. Редактор кода автоматически выводит список свойств и методов объекта (компонента)

Справочная система

В процессе набора программы можно получить справку, например, о конструкции языка или функции. Для этого нужно в окне редактора кода набрать слово, о котором надо получить справку, и нажать клавишу <F1>.

Справочную информацию можно получить, выбрав из меню **Help** команду **Delphi Help**. В этом случае на экране появится стандартное окно справочной системы (рис. 25). В данном окне на вкладке **Предметный указатель** введите ключевое слово, определяющее тему, по которой нужна справка. Как правило, в качестве ключевого слова используют первые несколько букв имени функции, процедуры, свойства или метода.

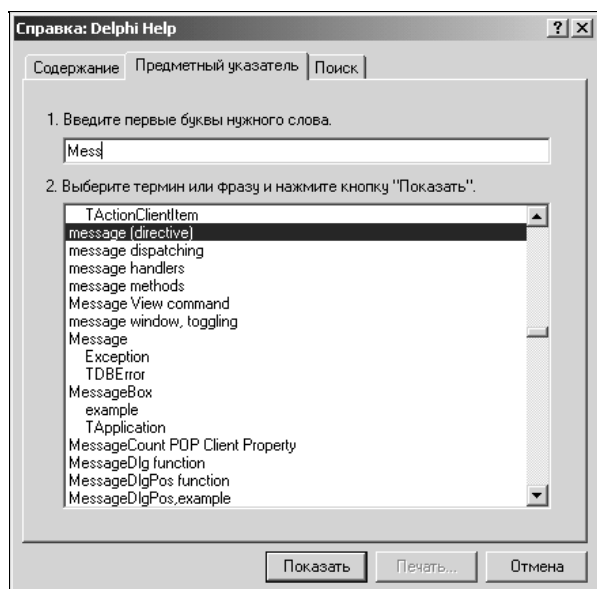


Рис. 25. Поиск справочной информации по ключевому слову

Структура проекта

Проект Delphi представляет собой набор программных единиц — модулей. Один из модулей — главный, содержит инструкции, с которых начинается выполнение программы. Главный модуль приложения полностью формируется Delphi.

Главный модуль представляет собой файл с расширением `dpr`. Для того чтобы увидеть текст главного модуля приложения, нужно из меню **Project** выбрать команду **View Source**

В листинге 3 приведен текст главного модуля программы вычисления скорости бега.

Листинг 3. Главный модуль приложения Скорость бега

```
program vrun;  
  
uses  
    Forms,  
    vrun1 in 'vrun1.pas' {Form1};  
  
{$R *.res}  
  
begin  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    Application.Run;  
end.
```

Начинается главный модуль словом `program`, за которым следует имя программы, совпадающее с именем проекта. Имя проекта задается в момент сохранения проекта, и оно определяет имя создаваемого компилятором исполняемого файла программы. Далее за словом `uses` следуют имена используемых модулей: библиотечного модуля `Forms` и модуля формы `vrun1.pas`.

Строка `{R*.RES}`, которая похожа на комментарий, — это директива компилятору подключить файл ресурсов. Файл ресурсов содержит *ресурсы* приложения: пиктограммы, курсоры, битовые образы и др. Звездочка показывает, что имя файла ресурсов такое же, как и у файла проекта, но с расширением `res`.

Файл ресурсов не является текстовым файлом, поэтому просмотреть его с помощью редактора текста нельзя. Для работы с файлами ресурсов используют специальные программы, например, `Resource Workshop`. Можно также применять входящую в состав Delphi утилиту `Image Editor`, доступ к которой можно получить выбором из меню **Tools** команды **Image Editor**.

Исполняемая часть главного модуля находится между инструкциями `begin` и `end`. Инструкции исполняемой части обеспечивают инициализацию приложения и вывод на экран стартового окна.

Помимо главного модуля каждая программа включает в себя еще как минимум один модуль формы, который содержит описание стартовой формы приложения и поддерживающих ее работу процедур. В Delphi каждой форме соответствует свой модуль.

В листинге 4 приведен текст модуля программы вычисления скорости бега.

Листинг 4. Модуль программы Скорость бега

```
unit vrun1;

interface

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

type
    TForm1 = class(TForm)
        Edit1: TEdit;
        Edit2: TEdit;
        Label1: TLabel;
        Label2: TLabel;
        Label3: TLabel;
        Label4: TLabel;
        Button1: TButton;
        Button2: TButton;
        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);

    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}
```

```
// нажатие кнопки Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    dist : integer; // дистанция, метров
    t:     real;     // время, как дробное число

    min : integer; // время, минуты
    sek : integer; // время, секунды

    v: real;        // скорость
begin
    // получить исходные данные из полей ввода
    dist := StrToInt(Edit1.Text);
    t := StrToFloat(Edit2.Text);

    // предварительные преобразования
    min := Trunc(t); // кол-во минут – это целая часть числа t
    sek := Trunc(t*100) mod 100; // кол-во секунд – это дробная часть
                                // числа t

    // вычисление
    v := (dist/1000) / ((min*60 + sek)/3600);

    // вывод результата
    label4.Caption := 'Дистанция: ' + Edit1.Text + ' м' + #13 +
        'Время: ' + IntToStr(min) + ' мин ' +
        IntToStr(sek) + ' сек ' + #13 +
        'Скорость: ' + FloatToStrF(v,ffFixed,4,2) +
        ' км/час';
end;

// нажатие кнопки Завершить
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.Close;
end;

end.
```

Начинается модуль словом `unit`, за которым следует имя модуля. Именно это имя упоминается в списке используемых модулей в инструкции `uses` главного модуля приложения, текст которого приведен в листинге 3.

Модуль состоит из следующих разделов:

- ☐ интерфейса;
- ☐ реализации;
- ☐ инициализации.

Раздел интерфейса (начинается словом `interface`) сообщает компилятору, какая часть модуля является доступной для других модулей программы. В этом разделе перечислены (после слова `uses`) библиотечные модули, используемые данным модулем. Также здесь находится сформированное Delphi описание формы, которое следует за словом `type`.

Раздел реализации открывается словом `implementation` и содержит объявления локальных переменных, процедур и функций, поддерживающих работу формы.

Начинается раздел реализации директивой `{ $R *.DFM }`, указывающей компилятору, что в процессе генерации выполняемого файла надо использовать описание формы. Описание формы находится в файле с расширением `dfm`, имя которого совпадает с именем модуля. Файл описания формы генерируется средой Delphi на основе внешнего вида формы.

За директивой `{ $R *.DFM }` следуют процедуры обработки событий для формы и ее компонентов. Сюда же программист может поместить другие процедуры и функции.

Раздел инициализации позволяет выполнить инициализацию переменных модуля. Инструкции раздела инициализации располагаются после раздела реализации (описания всех процедур и функций) между `begin` и `end`. Если раздел инициализации не содержит инструкций (как в приведенном примере), то слово `begin` не указывается.

Следует отметить, что значительное количество инструкций модуля формирует Delphi. Например, Delphi, анализируя действия программиста по созданию формы, генерирует описание класса формы (после слова `type`). В приведенном примере инструкции, набранные программистом, выделены фоном. Очевидно, что Delphi выполнила значительную часть работы по составлению текста программы.

Сохранение проекта

Проект — это набор файлов, используя которые компилятор создает исполняемый файл программы (exe-файл). В простейшем случае проект состоит из файла описания проекта (dpr-файл), файла главного модуля (pas-файл), файла ресурсов (res-файл), файла описания формы (dfm-файл), файла описания функций (cpp-файл) и некоторых других файлов.

Чтобы сохранить проект, нужно из меню **File** выбрать команду **Save Project As**. Если проект еще ни разу не был сохранен, то Delphi сначала предложит сохранить модуль (содержимое окна редактора кода), поэтому на экране появится окно **Save Unit1 As**. В этом окне (рис. 26) надо выбрать папку, предназначенную для файлов проекта, и ввести имя модуля. После нажатия кнопки **Сохранить**, появляется следующее окно (рис. 27), в котором необходимо ввести имя файла проекта.

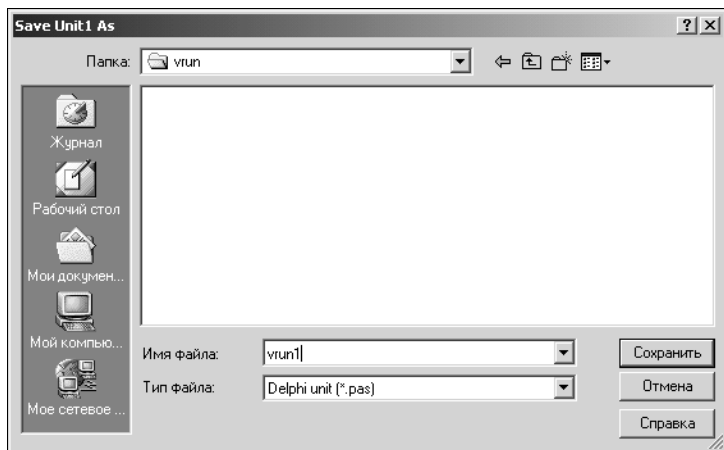


Рис. 26. Сохранение модуля

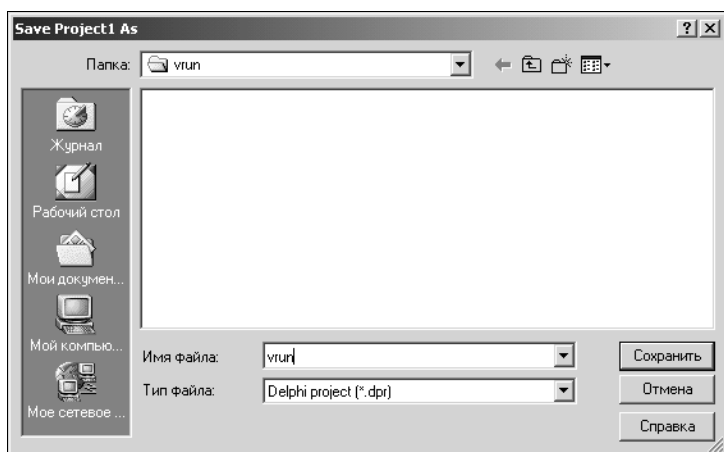


Рис. 27. Сохранение проекта

Обратите внимание на то, имена файлов модуля (pas-файл) и проекта (dpr-файл) должны быть разными. Имя генерируемого компилятором исполняемого файла совпадает с именем проекта. Поэтому файлу проекта следует присвоить такое имя, которое, по вашему мнению, должен иметь исполняемый файл программы, а файлу модуля — какое-либо другое имя, например, полученное путем добавления к имени файла проекта порядкового номера модуля.

Примечание

Так как проект представляет собой набор файлов, то рекомендуется для каждого проекта создавать отдельную папку.

Компиляция

Компиляция — это процесс преобразования исходной программы в исполняемую. Процесс компиляции состоит из двух этапов. На первом этапе выполняется проверка текста программы на отсутствие ошибок, на втором — генерируется исполняемая программа — ехе-файл.

После ввода текста функции обработки события и сохранения проекта можно из меню **Project** выбрать команду **Compile** и выполнить компиляцию. Процесс и результат компиляции отражаются в диалоговом окне **Compiling** (рис. 28). В это окно компилятор выводит количество ошибок (Errors), предупреждений (Warnings) и подсказок (Hints). Сами сообщения об ошибках, предупреждения и подсказки находятся в нижней части окна редактора кода (рис. 29).

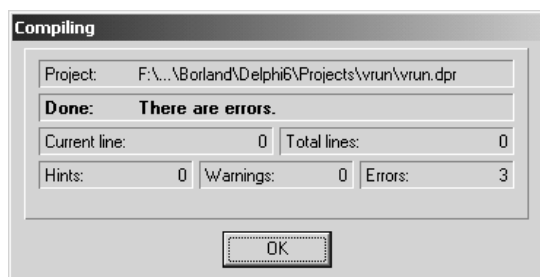


Рис. 28. Результат компиляции

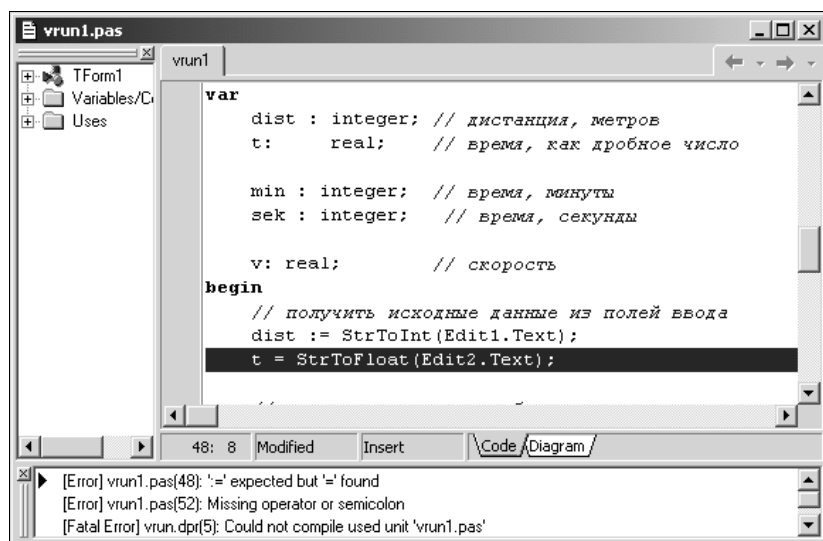


Рис. 29. Сообщения компилятора об обнаруженных ошибках

Примечание

Если во время компиляции окна **Compiling** на экране нет, то выберите из меню **Tools** команду **Environment options** и на вкладке **Preferences** установите во включенное состояние переключатель **Show compiler progress**.

Ошибки

Компилятор генерирует исполняемую программу лишь в том случае, если исходный текст не содержит синтаксических ошибок. В большинстве случаев в только что набранной программе есть ошибки. Программист должен их устранить.

Чтобы перейти к фрагменту кода, который содержит ошибку, надо установить курсор в строку с сообщением об ошибке и из контекстного меню (рис. 30) выбрать команду **Edit source**.

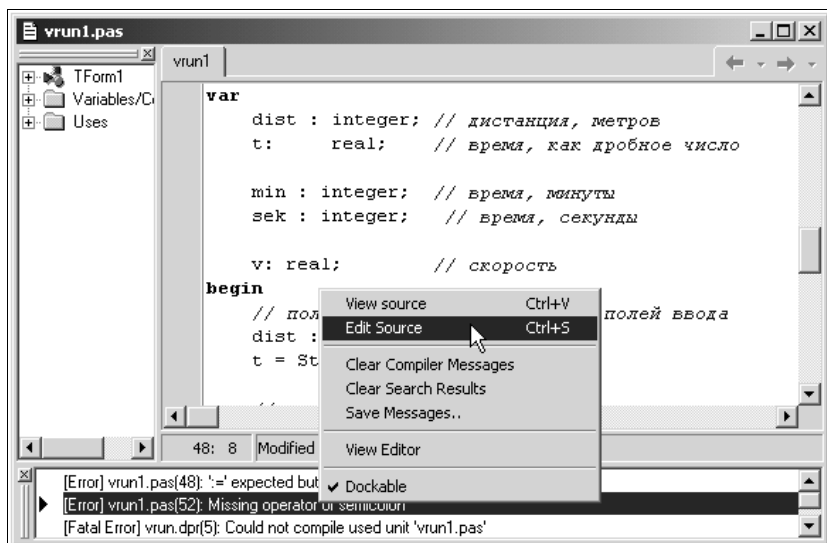


Рис. 30. Переход к фрагменту программы, содержащему ошибку

Процесс устранения ошибок носит итерационный характер. Обычно сначала устраняются наиболее очевидные ошибки, например, декларируются не объявленные переменные. После очередного внесения изменений в текст программы выполняется повторная компиляция. Следует учитывать тот факт, что компилятор не всегда может точно локализовать ошибку. Поэтому, анализируя фрагмент программы, который, по мнению компилятора, содержит ошибку, нужно обращать внимание не только на тот фрагмент кода, на который компилятор установил курсор, но и на тот, который находится в предыдущей строке.

В табл. 10 перечислены наиболее типичные ошибки и соответствующие им сообщения компилятора.

Таблица 10

Сообщение компилятора	Вероятная причина
Undeclared identifier (Необъявленный идентификатор)	а) Используется переменная, не объявленная в разделе <code>var</code> . б) Ошибка при написании имени объявленной переменной. Например, объявлена переменная <code>Summa</code> , а в тексте программы написано: <code>Suma := 0</code> . в) Ошибка при написании имени инструкции. Например, вместо <code>repeat</code> написано <code>repet</code>
Unterminated string (Незавершенная строка)	При записи строковой константы, например сообщения, не поставлена завершающая кавычка
Incompatible types ...and... (Несовместимые типы)	В инструкции присваивания тип выражения не соответствует или не может быть приведен к типу переменной, получающей значение выражения
Missing operator or semicolon (Отсутствует оператор или точка с запятой)	После инструкции не поставлена точка с запятой

Если компилятор обнаружил достаточно много ошибок, то просмотрите все сообщения, устраните сначала наиболее очевидные ошибки и выполните повторную компиляцию. Вполне вероятно, что после этого количество ошибок значительно уменьшится. Это объясняется особенностями синтаксиса языка, когда одна незначительная ошибка, может "тащить" за собой довольно большое количество других.

Если в программе нет синтаксических ошибок, компилятор создает исполняемый файл программы. Имя исполняемого файла такое же, как и у файла проекта, а расширение — `.exe`. Delphi помещает исполняемый файл в тот же каталог, где находится файл проекта.

Предупреждения и подсказки

При обнаружении в программе неточностей, которые не являются ошибками, компилятор выводит подсказки (`Hints`) и предупреждения (`Warnings`).

Например, наиболее часто выводимой подсказкой является сообщение об объявленной, но не используемой переменной:

Variable ... is declared but never used in ...

Действительно, зачем объявлять переменную и не использовать ее?

В табл. 11 приведены предупреждения, наиболее часто выводимые компилятором.

Таблица 11

Предупреждение	Вероятная причина
Variable... is declared but never used in ...	Переменная не используется
Variable ... might not have been initialized. (вероятно, используется не инициализированная переменная)	В программе нет инструкции, которая присваивает переменной начальное значение

Запуск программы

Пробный запуск программы можно выполнить непосредственно из Delphi, не завершая работу со средой разработки. Для этого нужно из меню **Run** выбрать команду **Run** или щелкнуть на соответствующей кнопке панели инструментов.

Ошибки времени выполнения

Во время работы приложения могут возникать ошибки, которые называются ошибками времени выполнения (run-time errors) или исключениями (exceptions). В большинстве случаев причинами исключений являются неверные исходные данные. Например, если во время работы программы вычисления скорости бега в поле **Время** ввести 3.20, т. е. для отделения дробной части числа от целой использовать точку, то в результате нажатия кнопки **Вычислить** на экране появится окно с сообщением об ошибке (рис. 31).

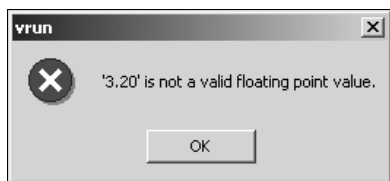


Рис. 31. Пример ошибки времени выполнения (программа запущена из Windows)

Причина возникновения ошибки заключается в следующем. В тексте программы дробная часть числа от целой отделяется точкой. При вводе исходных данных в поле редактирования пользователь может (если не предпринять никаких дополнительных усилий) отделить дробную часть числа от целой точкой или запятой. Какой из этих двух символов является допустимым, зависит от настройки Windows.

Если в настройке Windows указано, что разделитель целой и дробной частей числа — запятая (для России — это стандартная установка), а пользователь во время работы программы введет в поле редактирования, например, строку 3.20, то при выполнении инструкции

```
t = StrToFloat(Edit2.Text)
```

возникнет исключение, т. к. при стандартной для России настройке Windows содержимое поля Edit2 и, следовательно, аргумент функции StrToFloat не является изображением дробного числа.

Если программа запущена из среды разработки, то при возникновении исключения выполнение программы приостанавливается и на экране появляется окно с сообщением об ошибке и ее типе. В качестве примера на рис. 32 приведено окно с сообщением о том, что введенная пользователем строка не является дробным числом.

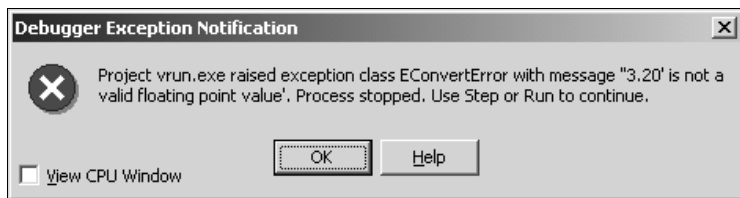


Рис. 32. Пример сообщения о возникновении исключения (программа запущена из Delphi)

После нажатия кнопки **ОК** в окне редактора кода будет выделена строка программы, содержащая инструкцию, выполнение которой вызвало исключение. Программист может продолжить выполнение программы (для этого надо из меню **Run** выбрать команду **Step Over**) или прервать ее выполнение. В последнем случае нужно из меню **Run** выбрать команду **Program Reset**.

При разработке программы программист должен постараться предусмотреть все возможные варианты некорректных действий пользователя, которые могут привести к возникновению ошибок времени выполнения (исключения), и обеспечить способы защиты от них.

Далее будет приведена версия программы **Скорость бега**, в которой реализована защита от некорректных действий пользователя.

Внесение изменений

После нескольких запусков программы **Скорость бега** возникает желание внести изменения в программу. Например, было бы неплохо, чтобы после ввода дистанции и нажатия клавиши <Enter> курсор переходил в поле **Время**. Чтобы в поля **Дистанция** и **Время** пользователь мог ввести только цифры.

Чтобы внести изменения в программу, нужно запустить Delphi и открыть соответствующий проект. Сделать это можно обычным способом, выбрав из меню **File** команду **Open Project**. Можно также воспользоваться командой **Reopen** из меню **File**. При выборе команды **Reopen** открывается список проектов, над которыми работал программист в последнее время.

В листинге 5 приведена программа **Скорость бега**, в которую добавлены процедуры обработки событий `OnKeyPress` для компонентов `Edit1` и `Edit2`. Следует обратить внимание на то, что для добавления в программу процедуры обработки события нужно в окне **Object Inspector** выбрать компонент, для которого создается процедура, затем на вкладке **Events** выбрать событие и сделать двойной щелчок в поле имени процедуры. Delphi сформирует шаблон процедуры обработки события. После этого можно вводить инструкции, реализующие процедуру обработки.

Листинг 5

```
unit vrun1;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  end;

end.
```



```
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

// нажатие кнопки Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    dist : integer; // дистанция, метров
    t:      real;    // время, как дробное число

    min : integer; // время, минуты
    sek : integer; // время, секунды

    v: real;        // скорость
begin
    // получить исходные данные из полей ввода
    dist := StrToInt(Edit1.Text);
    t := StrToFloat(Edit2.Text);

    // предварительные преобразования
    min := Trunc(t); // кол-во минут – это целая часть числа t
    sek := Trunc(t*100) mod 100; // кол-во секунд – это дробная часть
                                // числа t

    // вычисление
    v := (dist/1000) / ((min*60 + sek)/3600);

    // вывод результата
    label4.Caption := 'Дистанция: ' + Edit1.Text + ' м' + #13 +
        'Время: ' + IntToStr(min) + ' мин ' +
        IntToStr(sek) + ' сек ' + #13 +
        'Скорость: ' + FloatToStrF(v,ffFixed,4,2) +
        ' км/час';
end;

// нажатие кнопки Завершить
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.Close;
end;
```

// нажатие клавиши в поле Дистанция

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
```

```
begin
```

```
    // Key — символ, соответствующий нажатой клавише.
```

```
    // Если символ недопустимый, то процедура заменяет его
```

```
    // на символ с кодом 0. В результате этого символ в поле
```

```
    // редактирования не появляется и у пользователя создается
```

```
    // впечатление, что программа не реагирует на нажатие некоторых
```

```
    // клавиш.
```

```
    case Key of
```

```
        '0'..'9':                ; // цифра
```

```
        #8      :                ; // клавиша <Back Space>
```

```
        #13     : Edit2.SetFocus ; // клавиша <Enter>
```

```
    // остальные символы — запрещены
```

```
    else Key :=Chr(0); // символ не отображать
```

```
end;
```

```
end;
```

```
end.
```

После внесения изменений проект следует сохранить. Для этого нужно из меню **File** выбрать команду **Save all**.

Окончательная настройка приложения

После того как программа отлажена, необходимо выполнить ее окончательную настройку, т. е. задать название программы и выбрать значок, который будет изображать исполняемый файл приложения в папке или на рабочем столе, а также на панели задач во время работы программы.

Настройка приложения выполняется на вкладке **Application** диалогового окна **Project Options** (рис. 33), которое появляется в результате выбора из меню **Project** команды **Options**.

В поле **Title** надо ввести название приложения.

Чтобы назначить приложению значок, отличный от стандартного, нужно щелкнуть мышью на кнопке **Load Icon**. Затем, используя стандартное окно просмотра папок, найти подходящий значок (значки хранятся в файлах с расширением **ico**).

Создание уникального значка для приложения

В состав Delphi входит программа **Image Editor** (Редактор изображений), при помощи которой программист может создать для своего приложения уни-

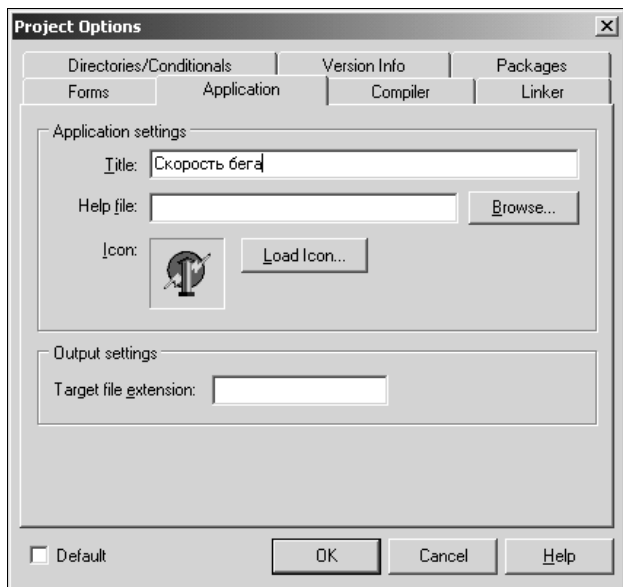


Рис. 33. Используя вкладку **Application**, можно задать значок и название программы

кальный значок. Запускается **Image Editor** выбором соответствующей команды из меню **Tools**.

Чтобы начать работу по созданию нового значка, нужно из меню **File** выбрать команду **New**, а из появившегося списка — опцию **Icon File** (рис. 34).

После выбора типа создаваемого файла открывается окно **Icon Properties** (рис. 35), в котором необходимо выбрать характеристики создаваемого значка: **Size** (Размер) — 32×32 (стандартный размер значков Windows) и **Colors** (Палитра) — 16 цветов. В результате нажатия кнопки **OK** открывается окно **Icon1.ico** (рис. 36), в котором можно, используя стандартные инструменты и палитру, нарисовать нужный значок.

Процесс рисования в Image Editor практически ничем не отличается от процесса создания картинки в обычном графическом редакторе, например в Microsoft Paint. Однако есть одна тонкость. Первоначально поле изображения закрашено "прозрачным" (transparent) цветом. Если значок нарисовать на этом фоне, то при его выводе части поля изображения, закрашенные "прозрачным" цветом, примут цвет фона, на котором будет находиться значок.

В процессе создания картинки можно удалить (стереть) ошибочно нарисованные элементы, закрасив их прозрачным цветом, которому на палитре соответствует левый квадрат в нижнем ряду (рис. 37).

Кроме "прозрачного" цвета в палитре есть "инверсный" цвет. Нарисованные этим цветом части рисунка при выводе на экран окрашиваются "инверсным", относительно цвета фона, цветом.

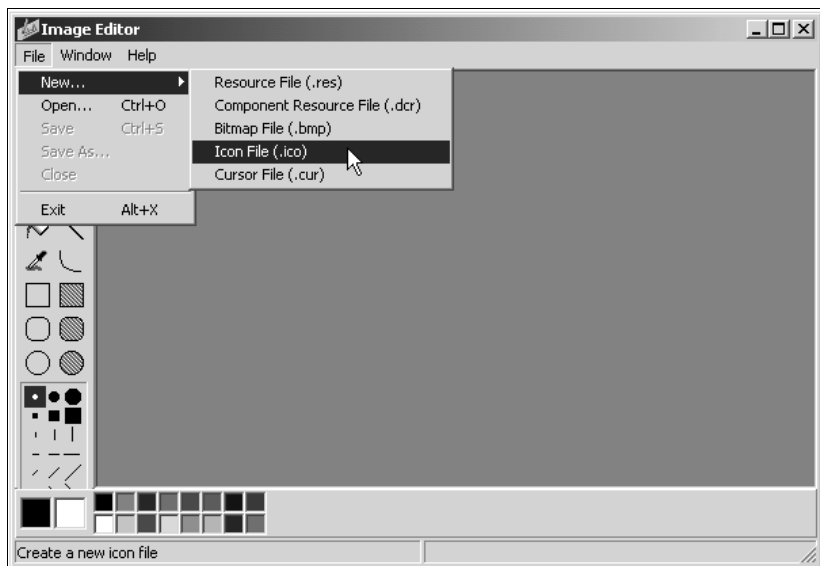


Рис. 34. Начало работы над новым значком

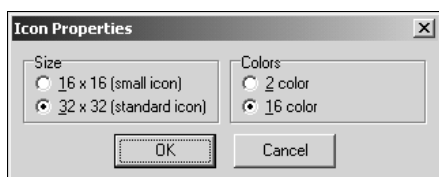


Рис. 35. Выбор параметров значка

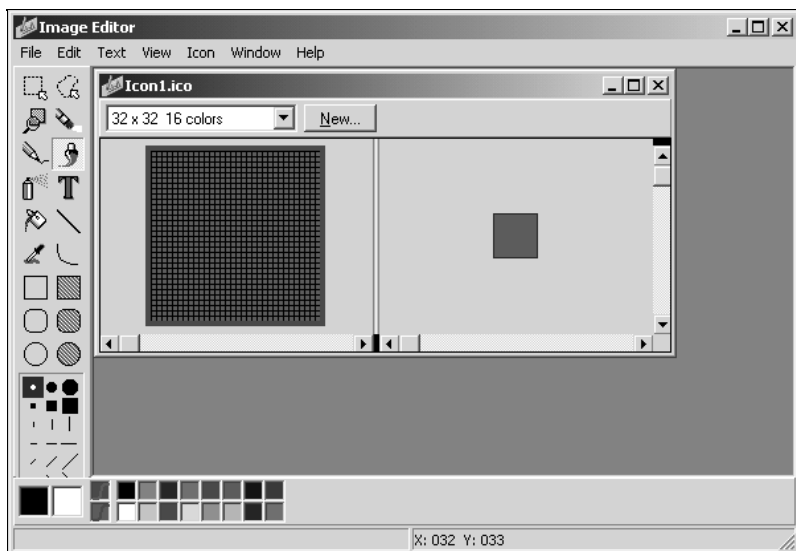


Рис. 36. Окно рисования значка

"Инверсный"



"Прозрачный"

Основные цвета

Рис. 37. Палитра

Сохраняется созданный значок обычным образом, т. е. выбором из меню **File** команды **Save**.

Перенос приложения на другой компьютер

Небольшую программу, которая использует только стандартные компоненты и представляет собой один-единственный `exe`-файл, можно перенести на другой компьютер вручную, например, при помощи дискеты. Как правило, при запуске таких программ на другом компьютере проблем не возникает.

Программы, которые используют библиотеки, драйверы и другие программные компоненты, например элементы контроля типа `ActiveX`, перенести на другой компьютер вручную проблематично. Для таких программ лучше создать установочную дискету. Сделать это можно, например, при помощи пакета `InstallShield Express`, который входит в комплект поставки `Delphi`.

ГЛАВА 1



Основы программирования

Программа

Программа, работающая на компьютере, нередко отождествляется с самим компьютером, т. к. человек, использующий программу, "вводит в компьютер" исходные данные, как правило, при помощи клавиатуры, а компьютер "выдает результат" на экран, на принтер или в файл. На самом деле преобразование исходных данных в результат выполняет процессор компьютера. Процессор преобразует исходные данные в результат по определенному алгоритму, который, будучи записан на специальном языке, называют программой. Таким образом, чтобы компьютер выполнил некоторую работу, необходимо разработать последовательность команд, обеспечивающую выполнение этой работы, или, как говорят, написать программу.

Этапы разработки программы

Выражение "написать программу" отражает только один из этапов создания компьютерной программы, когда разработчик программы (программист) действительно пишет команды (инструкции) на бумаге или при помощи текстового редактора.

Программирование — это процесс создания (разработки) программы, который может быть представлен последовательностью следующих шагов:

1. Определение требований к программе
2. Разработка алгоритма
3. Написание команд (кодирование)
4. Отладка
5. Тестирование

Определение требований к программе

Определение требований к программе — один из важнейших этапов, на котором подробно описывается исходная информация и формулируются требования к результату. Кроме того, описывается поведение программы в особых случаях.

Если программа будет работать в Windows, то нужно разработать диалоговые окна, обеспечивающие взаимодействие пользователя и программы.

Разработка алгоритма

На этапе разработки алгоритма необходимо определить последовательность действий, которые надо выполнить для получения результата. Если задача может быть решена несколькими способами и, следовательно, возможны различные варианты алгоритма решения, то программист, используя некоторый критерий, например скорость решения алгоритма, выбирает наиболее подходящее решение. Результатом этапа разработки алгоритма является подробное словесное описание алгоритма или его блок-схема.

Кодирование

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

Отладка

Отладка — это процесс поиска и устранения ошибок. Ошибки в программе разделяют на две группы: синтаксические (ошибки в тексте) и алгоритмические. Синтаксические ошибки — это наиболее легко устранимые ошибки. Алгоритмические ошибки обнаружить труднее. Этап отладки можно считать законченным, если программа правильно работает на одном-двух наборах входных данных.

Тестирование

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

Алгоритм и программа

На первом этапе создания программы программист должен определить последовательность действий, которые необходимо выполнить, чтобы решить поставленную задачу, т. е. разработать алгоритм. Алгоритм — это точное предписание, определяющее процесс перехода от исходных данных к результату.

Нужно понимать, что предписание считается алгоритмом, если оно обладает тремя следующими свойствами:

- ☐ однозначностью;
- ☐ массовостью;
- ☐ результативностью.

Под *однозначностью* алгоритма понимается единственность толкования правил и порядка выполнения действий. Таким образом, алгоритм не должен оставлять место произволу при его выполнении.

Массовость алгоритма означает возможность применения алгоритма для решения класса задач, предполагает его правильную работу при меняющихся в заданных пределах значениях исходных данных.

Результативность алгоритма предполагает, что выполнение алгоритма должно приводить к получению определенного результата.

Пусть, например, нужно решить квадратное уравнение, которое в общем виде записывают так: $ax^2+bx+c=0$. Шаги, которые должны быть выполнены для этого, хорошо известны:

1. Вычислить по формуле b^2-4ac дискриминант.
2. Если значение дискриминанта больше или равно нулю, то вычислить значения корней уравнения: $x_1 = \frac{-b-\sqrt{d}}{2a}$; $x_2 = \frac{-b+\sqrt{d}}{2a}$, где d — вычисленное на первом шаге значение дискриминанта.
3. Если значение дискриминанта меньше нуля, то это значит, что уравнение не имеет решения.

Приведенное выше предписание обладает всеми свойствами алгоритма:

- ☐ однозначностью (в предписании указано, как обозначаются коэффициенты уравнения, и приведены формулы для вычисления значения дискриминанта и корней уравнения);
- ☐ массовостью (в предписании указаны не конкретные значения коэффициентов, а приведены формулы, в которых использованы обозначения коэффициентов);

□ результативностью (при выполнении предписания получается результат — значения корней уравнения или вывод о том, что уравнение не имеет решения).

При описании алгоритма используются обобщенные понятия, например "коэффициент" и "корень уравнения". При решении задачи эти понятия конкретизируются. Нельзя найти значения корней абстрактного квадратного уравнения, можно решить только конкретное уравнение, т. е. необходимо задать коэффициенты уравнения.

Алгоритм решения задачи может быть представлен в виде словесного описания или графически — в виде блок-схемы. При изображении алгоритма в виде блок-схемы используются специальные символы (рис. 1.1).

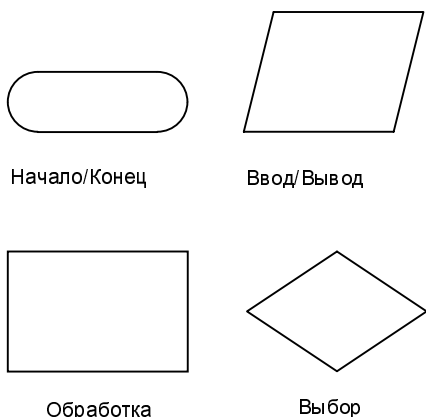


Рис. 1.1. Основные символы, используемые для представления алгоритма в виде блок-схемы

Представление алгоритма в виде блок-схемы позволяет программисту уяснить последовательность действий, которые должны быть выполнены для решения задачи, убедиться в правильности понимания поставленной задачи.

При программировании в Delphi алгоритм решения задачи представляет собой совокупность алгоритмов процедур обработки событий.

В качестве примера на рис. 1.2 приведена совокупность алгоритмов программы **Стоимость покупки**, а на рис. 1.3 — ее диалоговое окно. После разработки диалогового окна и алгоритмов обработки событий можно приступить к написанию программы. Ее текст приведен в листинге 1.1.

Листинг 1.1. Программа Стоимость покупки

```
unit покупка_1;  
  
interface
```

uses

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

type

```
TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Label3: TLabel;
procedure Button1Click(Sender: TObject);
procedure Edit2KeyPress(Sender: TObject; var Key: Char);
procedure Edit1KeyPress(Sender: TObject; var Key: Char);
```

private

```
{ Private declarations }
```

public

```
{ Public declarations }
```

```
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.dfm }
```

```
// подпрограмма
```

procedure Summa;**var**

```
cena: real;           // цена
kol: integer;         // количество
s: real;              // сумма
mes: string[255];     // сообщение
```

begin

```
cena := StrToFloat(Form1.Edit1.Text);
kol := StrToInt(Form1.Edit2.Text);
s := cena * kol;
if s > 500 then
    begin
        s := s * 0.9;
        mes := 'Предоставляется скидка 10%' + #13;
    end;
```

```

mes := mes+ 'Стоимость покупки: '
        + FloatToStrF(s,ffFixed,4,2) +' руб.';
Form1.Label3.Caption := mes;

end;

// щелчок на кнопке Стоимость
procedure TForm1.Button1Click(Sender: TObject);
begin
    Summa; // вычислить сумму покупки
end;

// нажатие клавиши в поле Количество
procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        '0' .. '9', #8: ; // цифры и клавиша <Back Space>
        #13: Summa; // вычислить стоимость покупки
        else Key := Chr(0); // символ не отображать
    end;
end;

// нажатие клавиши в поле Цена
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        '0' .. '9', #8 : ; // цифры и клавиша <Back Space>

        #13: Form1.Edit2.SetFocus; // клавиша <Enter>

        '.', ',', ':
        begin
            if Key = '.'
            then Key := ',';
            if Pos(',', Edit1.Text) <> 0
            then Key := Chr(0);
        end;
        else // все остальные символы запрещены
            Key := Chr(0);
    end;
end;
end.
```

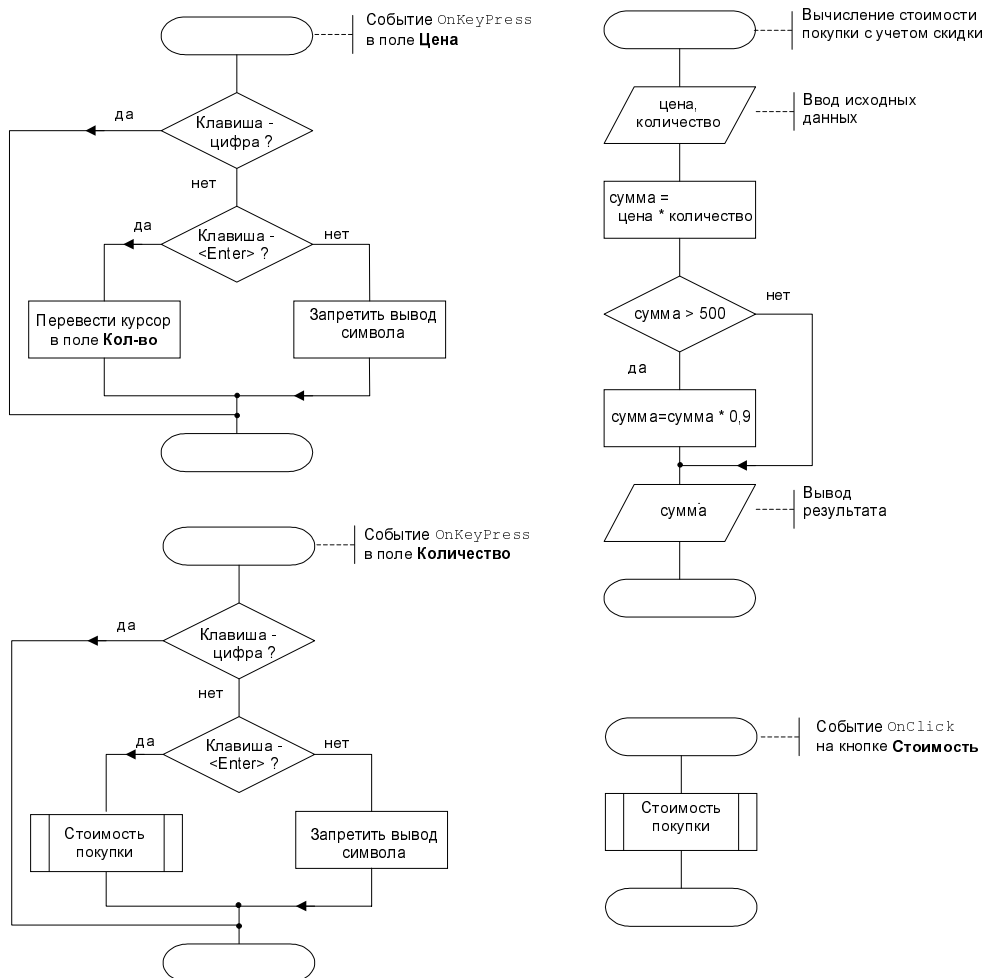


Рис. 1.2. Алгоритм программы вычисления стоимости покупки — совокупность алгоритмов обработки событий на компонентах формы

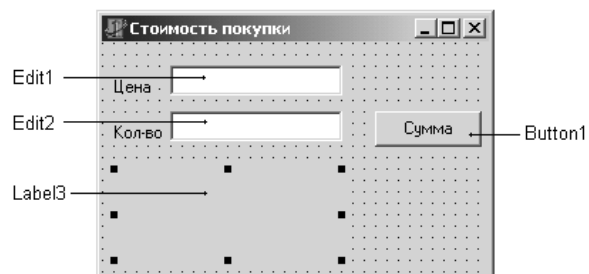


Рис. 1.3. Окно (форма) программы **Стоимость покупки**

Компиляция

Программа, представленная в виде инструкций языка программирования, называется исходной программой. Она состоит из инструкций, понятных человеку, но не понятных процессору компьютера. Чтобы процессор смог выполнить работу в соответствии с инструкциями исходной программы, исходная программа должна быть переведена на машинный язык — язык команд процессора. Задачу преобразования исходной программы в машинный код выполняет специальная программа — компилятор.

Компилятор, схема работы которого приведена на рис. 1.4, выполняет последовательно две задачи:

1. Проверяет текст исходной программы на отсутствие синтаксических ошибок.
2. Создает (генерирует) исполняемую программу — машинный код.

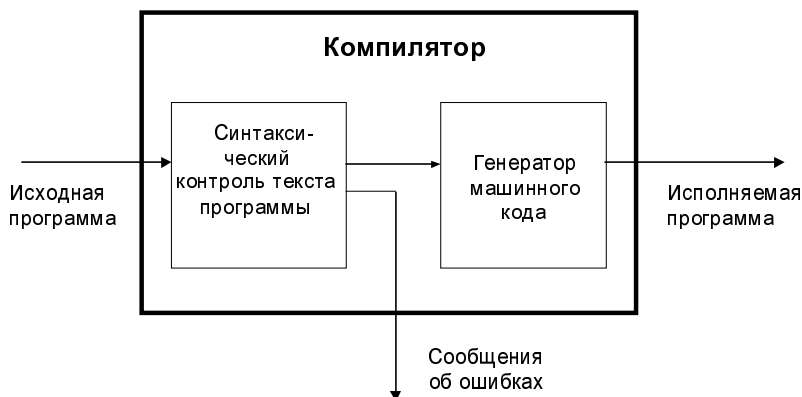


Рис. 1.4. Схема работы компилятора

Следует отметить, что генерация исполняемой программы происходит только в том случае, если в тексте исходной программы нет синтаксических ошибок.

Генерация машинного кода компилятором свидетельствует только об отсутствии в тексте программы синтаксических ошибок. Убедиться в правильности работы программы можно только во время ее тестирования — пробных запусках программы и при анализе полученных результатов. Например, если в программе нахождения корней квадратного уравнения допущена ошибка в записи выражения вычисления дискриминанта, то, даже если это выражение будет синтаксически верно, программа выдаст неверные значения корней.

Язык программирования Object Pascal

Программа, записанная на языке Object Pascal, представляет собой последовательность инструкций, которые довольно часто называют *операторами*. Одна инструкция от другой отделяется точкой с запятой. Каждая инструкция состоит из слов (идентификаторов). Идентификатор может обозначать:

- ☐ инструкцию языка (`:=`, `if`, `while`, `procedure`);
- ☐ подпрограмму (процедуру или функцию);
- ☐ переменную (ячейку памяти);
- ☐ константу;
- ☐ арифметическую (`+`, `-`, `*`, `/`) или логическую (`and`, `or`, `not`) операцию;
- ☐ отмечать начало или конец раздела программы и т. д.

Тип данных

Программа может оперировать с данными различных типов: целыми и дробными числами, символами, строками символов, логическими величинами.

Целый тип

Object Pascal поддерживает семь целых типов данных: `Shortint`, `Smallint`, `Longint`, `Int64`, `Byte`, `Word` и `Longword`, описание которых приведено в табл. 1.1.

Таблица 1.1

Тип	Диапазон	Формат
Shortint	−128..127	8-бит
Smallint	−32768..32767	16-бит
Longint	−2147483648..2147483647	32-бит
Int64	−2 ⁶³ ..2 ⁶³ −1	64-бит
Byte	0..255	8-бит, беззнаковый
Word	0..65535	8-бит, беззнаковый
Longword	0..4294967295	8-бит, беззнаковый

Object Pascal поддерживает и наиболее универсальный целый тип — `Integer`, который эквивалентен `Longint`.

Вещественный тип

Object Pascal поддерживает шесть вещественных типов: Real48, Single, Double, Extended, Comp, Currency. Типы различаются между собой диапазоном допустимых значений, количеством значащих цифр и количеством байт, необходимых для хранения данных в памяти компьютера (табл. 1.2).

Таблица 1.2

Тип	Диапазон	Значащих цифр	Байт
Real48	$2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$	11–12	6
Single	$1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$	7–8	4
Double	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15–16	8
Extended	$3.6 \times 10^{-4951} \dots 1.1 \times 10^{4932}$	19–20	10
Comp	$-2^{63}+1 \dots 2^{63}-1$	19–20	8
Currency	$-922337203685477.5808 \dots 922337203685477.5807$	19–20	8

Object Pascal поддерживает и наиболее универсальный вещественный тип — Real, который эквивалентен Real48.

Символьный тип

Object Pascal поддерживает два символьных типа: AnsiChar и WideChar

AnsiChar — это символы в кодировке ANSI, которым соответствуют числа в диапазоне от 0 до 255. Тип WideChar — это символы в кодировке Unicode, им соответствуют числа от 0 до 65 535.

Object Pascal поддерживает и наиболее универсальный символьный тип — Char, который эквивалентен AnsiChar.

Строковый тип

Object Pascal поддерживает три строковых типа: ShortString, LongString и WideString.

Тип ShortString представляет собой статически размещаемые в памяти компьютера строки длиной от 0 до 255 символов.

Тип LongString представляет собой динамически размещаемые в памяти строки, длина которых ограничена только объемом свободной памяти.

Тип WideString представляет собой динамически размещаемые в памяти строки, длина которых ограничена только объемом свободной памяти. Каждый символ строки типа WideString является Unicode-символом.

В Object Pascal для обозначения строкового типа допускается использование идентификатора `String`. Тип `String` эквивалентен типу `ShortString`.

Логический тип

Логическая величина может принимать одно из двух значений `True` (истина) или `False` (ложь). В Object Pascal логические величины относят к типу `Boolean`.

Переменная

Переменная — это область памяти, в которой находятся данные, с которыми оперирует программа. Когда программа манипулирует с данными, она, фактически, оперирует с содержимым ячеек памяти, т. е. с переменными.

Чтобы программа могла обратиться к переменной (области памяти), например для того, чтобы получить исходные данные для расчета по формуле или сохранить результат, переменная должна иметь имя. Имя переменной придумывает программист.

В качестве имени переменной можно использовать последовательность из букв латинского алфавита, цифр и некоторых специальных символов. Первым символом в имени переменной должна быть буква. Пробел в имени переменной использовать нельзя.

Следует обратить внимание на то, что компилятор Object Pascal не различает прописные и строчные буквы в именах переменных, поэтому имена `SUMMA`, `Summa` и `summa` обозначают одну и ту же переменную.

Желательно, чтобы имя переменной было логически связано с ее назначением. Например, переменным, предназначенным для хранения коэффициентов и корней квадратного уравнения, которое в общем виде традиционно записывают

$$ax^2 + bx + c = 0$$

вполне логично присвоить имена `a`, `b`, `c`, `x1` и `x2`. Другой пример. Если в программе есть переменные, предназначенные для хранения суммы покупки и величины скидки, то этим переменным можно присвоить имена `TotalSumm` и `Discount` или `ObSumma` и `Skidka`.

В Object Pascal каждая переменная перед использованием должна быть объявлена. С помощью объявления устанавливается не только факт существования переменной, но и задается ее тип, тем самым указывается диапазон допустимых значений.

В общем виде инструкция объявления переменной выглядит так:

Имя : тип;

где:

Имя — имя переменной;

тип — тип данных, для хранения которых предназначена переменная.

Пример:

```
a : Real;  
b : Real;  
i : Integer;
```

В приведенных примерах объявлены две переменные типа `real` и одна переменная типа `integer`.

В тексте программы объявление каждой переменной, как правило, помещают на отдельной строке.

Если в программе имеется несколько переменных, относящихся к одному типу, то имена этих переменных можно перечислить в одной строке через запятую, а тип переменных указать после имени последней переменной через двоеточие, например

```
a,b,c : Real;  
x1,x2 : Real;
```

Константы

В Object Pascal существует два вида констант: *обычные* и *именованные*.

Обычная константа — это целое или дробное число, строка символов или отдельный символ, логическое значение.

Числовые константы

В тексте программы числовые константы записываются обычным образом, т. е. так же, как числа, например, при решении математических задач. При записи дробных чисел для разделения целой и дробных частей используется точка. Если константа отрицательная, то непосредственно перед первой цифрой ставится знак "минус".

Ниже приведены примеры числовых констант:

```
123  
0.0  
-524.03  
0
```

Дробные константы могут изображаться в виде числа с плавающей точкой. Представление в виде числа с плавающей точкой основано на том, что любое число может быть записано в алгебраической форме как произведение

числа, меньшего 10, которое называется мантиссой, и степени десятки, именуемой порядком.

В табл. 1.3 приведены примеры чисел, записанных в обычной форме, в алгебраической форме и форме с плавающей точкой.

Таблица 1.3

Число	Алгебраическая форма	Плавающая точка
1 000 000	1×10^6	1,0000000000E+06
−123.452	$-1,23452 \times 10^2$	−1,2345200000E+02
0,0056712	$5,6712 \times 10^{-3}$	5,6712000000E-03

Строковые и символьные константы

Строковые и символьные константы заключаются в кавычки. Ниже приведены примеры строковых констант:

```
'Язык программирования Object Pascal'
'Delphi 6'
'2.4'
'д'
```

Здесь следует обратить внимание на константу '2.4'. Это именно символьная константа, т. е. строка символов, которая изображает число "две целые четыре десятых", а не число 2,4.

Логические константы

Логическое высказывание (выражение) может быть либо истинно, либо ложно. Истине соответствует константа True, значению "ложь" — константа False.

Именованная константа

Именованная константа — это имя (идентификатор), которое в программе используется вместо самой константы.

Именованная константа, как и переменная, перед использованием должна быть объявлена. В общем виде инструкция объявления именованной константы выглядит следующим образом:

```
константа = значение;

где:

константа — имя константы;
значение — значение константы.
```

Ниже приведены примеры объявления именованных констант:

```
Bound = 10;  
Title = 'Скорость бега';  
pi = 3.1415926;
```

После объявления именованной константы в программе вместо самой константы можно использовать ее имя.

Тип константы определяется ее видом, например:

- ☐ 125 — константа целого типа
- ☐ 0.0 — константа вещественного типа
- ☐ 'Выполнить' — строковая константа
- ☐ '\ ' — символьная константа

Инструкция присваивания

Инструкция присваивания является основной вычислительной инструкцией. Если в программе надо выполнить вычисление, то нужно использовать инструкцию присваивания.

В результате выполнения инструкции присваивания значение переменной меняется, ей присваивается значение.

В общем виде инструкция присваивания выглядит так:

Имя := *Выражение*;

где:

Имя — переменная, значение которой изменяется в результате выполнения инструкции присваивания;

:= — символ инструкции присваивания.

Выражение — выражение, значение которого присваивается переменной, имя которой указано слева от символа инструкции присваивания.

Пример:

```
Summa := Cena * Kol;  
Skidka := 10;  
Found := False;
```

Выражение

Выражение состоит из операндов и операторов. Операторы находятся между операндами и обозначают действия, которые выполняются над операндами. В качестве операндов выражения можно использовать: переменную, кон-

станту, функцию или другое выражение. Основные алгебраические операторы приведены в табл. 1.4.

Таблица 1.4

Оператор	Действие
+	Сложение
–	Вычитание
*	Умножение
/	Деление
DIV	Деление нацело
MOD	Вычисление остатка от деления

При записи выражений между операндом и оператором, за исключением операторов DIV и MOD, пробел можно не ставить.

Результат применения операторов +, –, * и / очевиден.

Оператор DIV позволяет получить целую часть результата деления одного числа на другое. Например, значение выражения 5 DIV 2 равно 2.

Оператор MOD, деление по модулю, позволяет получить остаток от деления одного числа на другое. Например, значение выражения 5 MOD 2 равно 1.

В простейшем случае выражение может представлять собой константу или переменную.

Примеры выражений:

```
123
0.001
i
A+B/C
Summa*0.75
(B1+B3+B3) /3
Cena MOD 100
```

При вычислении значений выражений следует учитывать, что операторы имеют разный приоритет. Так у операторов *, /, DIV, MOD более высокий приоритет, чем у операторов + и –.

Приоритет операторов влияет на порядок их выполнения. При вычислении значения выражения в первую очередь выполняются операторы с более высоким приоритетом. Если приоритет операторов в выражении одинаковый, то сначала выполняется тот оператор, который находится левее.

Для задания нужного порядка выполнения операций в выражении можно использовать скобки, например

```
(r1+r2+r3) / (r1*r2*r3)
```

Выражение, заключенное в скобки, трактуется как один операнд. Это означает, что операции над операндами в скобках будут выполняться в обычном порядке, но раньше, чем операции над операндами, находящимися за скобками. При записи выражений, содержащих скобки, должна соблюдаться парность скобок, т. е. число открывающих скобок должно быть равно числу закрывающих скобок. Нарушение парности скобок — наиболее распространенная ошибка при записи выражений.

Тип выражения

Тип выражения определяется типом операндов, входящих в выражение, и зависит от операций, выполняемых над ними. Например, если оба операнда, над которыми выполняется операция сложения, целые, то, очевидно, что результат тоже является целым. А если хотя бы один из операндов дробный, то тип результата дробный, даже в том случае, если дробная часть значения выражения равна нулю.

Важно уметь определять тип выражения. При определении типа выражения следует иметь в виду, что тип константы определяется ее видом, а тип переменной задается в инструкции объявления. Например, константы 0, 1 и -512 — целого типа (*integer*), а константы 1.0, 0.0 и 3.2E-05 — вещественного типа (*real*).

В табл. 1.5 приведены правила определения типа выражения в зависимости от типа операндов и вида оператора.

Таблица 1.5

Оператор	Тип операндов	Тип выражения
<i>*</i> , <i>+</i> , <i>-</i>	Хотя бы один операнд <i>real</i>	<i>real</i>
<i>*</i> , <i>+</i> , <i>-</i>	Оба операнда <i>integer</i>	<i>integer</i>
<i>/</i>	<i>real</i> или <i>integer</i>	Всегда <i>real</i>
<i>DIV</i> , <i>MOD</i>	Всегда <i>integer</i>	Всегда <i>integer</i>

Выполнение инструкции присваивания

Инструкция присваивания выполняется следующим образом:

- сначала вычисляется значение выражения, которое находится справа от символа инструкции присваивания;

- затем вычисленное значение записывается в переменную, имя которой стоит слева от символа инструкции присваивания.

Например, в результате выполнения инструкций:

- `i:=0;` — значение переменной `i` становится равным нулю;
- `a:=b+c;` — значением переменной `a` будет число, равное сумме значений переменных `b` и `c`;
- `j:=j+1;` — значение переменной `j` увеличивается на единицу.

Инструкция присваивания считается верной, если тип выражения соответствует или может быть приведен к типу переменной, получающей значение. Например, переменной типа `real` можно присвоить значение выражения, тип которого `real` или `integer`, а переменной типа `integer` можно присвоить значение выражения только типа `integer`.

Так, например, если переменные `i` и `n` имеют тип `integer`, а переменная `d` — тип `real`, то инструкции

```
i:=n/10;  
i:=1.0;
```

неправильные, а инструкция

```
d:=i+1;
```

правильная.

Во время компиляции выполняется проверка соответствия типа выражения типу переменной. Если тип выражения не соответствует типу переменной, то компилятор выводит сообщение об ошибке:

```
Incompatible types ... and ...,
```

где вместо многоточий указывается тип выражения и переменной. Например, если переменная `n` целого типа, то инструкция `n:=m/2` неверная, поэтому во время компиляции будет выведено сообщение :

```
Incompatible types 'Integer' and 'Extended'.
```

Стандартные функции

Для выполнения часто встречающихся вычислений и преобразований Object Pascal предоставляет программисту ряд стандартных функций.

Значение функции связано с ее именем. Поэтому функцию можно использовать в качестве операнда инструкции присваивания. Например, чтобы вычислить квадратный корень, достаточно записать `k:=Sqrt(n)`, где `Sqrt` — функция вычисления квадратного корня.

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

Арифметические функции

Арифметические функции (табл. 1.6) позволяют выполнять различные вычисления.

Таблица 1.6

Функция	Значение
Abs (n)	Абсолютное значение n
Sqrt (n)	Квадратный корень из n
Sqr (n)	Квадрат n
Sin (n)	Синус n
Cos (n)	Косинус n
Arctan (n)	Арктангенс n
Exp (n)	Экспонента n
Ln (n)	Натуральный логарифм n
Pardom (n)	Случайное целое число в диапазоне от 0 до n-1

Примечание

Величина угла для тригонометрических функций должна быть выражена в радианах.

Функции преобразования типов

Функции преобразования типов (табл. 1.7) наиболее часто используются в инструкциях, обеспечивающих ввод и вывод информации. Например, для того чтобы вывести в поле вывода диалогового окна значение переменной типа real, необходимо сначала получить изображение числа, являющегося значением данной переменной. Это можно сделать при помощи функции FloatToStr, которая возвращает строковое представление значения выражения действительного типа, указанного при вызове функции.

Таблица 1.7

Функция	Значение функции
Chr (n)	Символ, код которого равен n
IntToStr (n)	Изображение целого числа n
FloatToStr (n)	Изображение вещественного n
FloatToStrF (n, f, k, m)	Изображение вещественного n. При вызове функции указывают: f — формат (способ изображения); k — точность (нужное общее количество цифр); m — количество цифр после десятичной точки
StrToInt (s)	Целое, изображением которого является строка s
StrToFloat (s)	Вещественное, изображением которого является строка s
Round (n)	Целое, ближайшее к n
Trunc (n)	Целая часть от n
Frac (n)	Дробная часть вещественного n
Int (n)	Целая часть вещественного n

Использование функций

Обычно функции применяют в качестве операндов выражений. Параметром функции может быть константа, переменная или выражение соответствующего типа. Ниже приведены примеры использования стандартных функций и функций преобразования.

```
n := Round ( (x2-x1) /dx);
x1:=  (-b + Sqrt(d)) /  (2*a);
m := Random(10);
cena := StrToInt(Edit1.Text);
Edit2.Text := IntToStr ( 100);
mes := 'x1=' + FloatToStr ( x1);
```

Ввод данных

Наиболее просто программа может получить исходные данные из окна ввода или из поля редактирования (компонент Edit).

Ввод из окна ввода

Окно ввода — это стандартное диалоговое окно, которое появляется на экране в результате вызова функции InputBox. Значение функции InputBox — строка, которую ввел пользователь.

В общем виде инструкция ввода данных с использованием функции `InputBox` выглядит так:

```
Переменная := InputBox(Заголовок, Подсказка, Значение);
```

где:

Переменная — переменная строкового типа, значение которой должно быть получено от пользователя;

Заголовок — текст заголовка окна ввода;

Подсказка — текст поясняющего сообщения;

Значение — текст, который будет находиться в поле ввода, когда окно ввода появится на экране.

Ниже в качестве примера приведена инструкция, используя которую можно получить исходные данные для программы пересчета веса из фунтов в килограммы. Окно ввода, соответствующее этой инструкции, приведено на рис. 1.5.

```
s:=InputBox('Фунты-килограммы','Введите вес в фунтах','0');
```

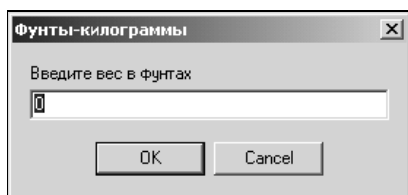


Рис. 1.5. Пример окна ввода

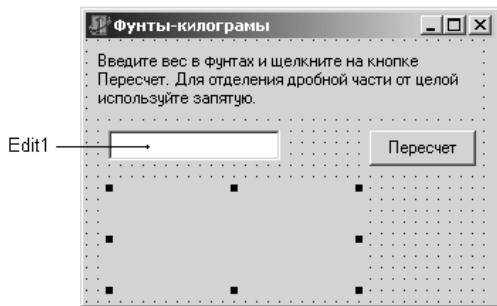
Если во время работы программы пользователь введет строку и щелкнет на кнопке **ОК**, то значением функции `InputBox` будет введенная строка. Если будет сделан щелчок на кнопке **Cancel**, то значением функции будет строка, переданная функции в качестве параметра *Значение*.

Следует еще раз обратить внимание на то, что значение функции `InputBox` строкового (`string`) типа. Поэтому если программе надо получить число, то введенная строка должна быть преобразована в число при помощи соответствующей функции преобразования. Например, фрагмент программы пересчета веса из фунтов в килограммы, обеспечивающий ввод исходных данных из окна ввода, может выглядеть так:

```
s := InputBox('Фунты-килограммы','Введите вес в фунтах','');  
funt := StrToFloat(s);
```

Ввод из поля редактирования

Поле редактирования — это компонент `Edit`. Ввод данных из поля редактирования осуществляется обращением к свойству `Text`.

**Рис. 1.6.** Окно программы

На рис. 1.6 приведен вид диалогового окна программы пересчета веса из фунтов в килограммы. Если `Edit1` — имя поля, предназначенного для ввода веса, то инструкция ввода исходных данных будет иметь вид:

```
Funt := StrToFloat(Edit1.Text);
```

Вывод результатов

Наиболее просто программа может вывести результат своей работы в окно сообщения или в поле вывода диалогового окна.

Вывод в окно сообщения

Окна сообщений используются для привлечения внимания пользователя. При помощи окна сообщения программа может, к примеру, проинформировать об ошибке в исходных данных или запросить подтверждение выполнения необратимой операции, например, удаление файла.

Вывести на экран окно с сообщением можно при помощи процедуры `ShowMessage` или функции `MessageDlg`.

Процедура `ShowMessage` выводит на экран окно с текстом и командной кнопкой **ОК**.

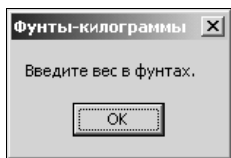
В общем виде инструкция вызова процедуры `ShowMessage` выглядит так:

```
ShowMessage (Сообщение) ;
```

где `Сообщение` — текст, который будет выведен в окне.

На рис. 1.7 приведен вид окна сообщения, полученного в результате выполнения инструкции:

```
ShowMessage ('Введите вес в фунтах.');
```

**Рис. 1.7.** Пример окна сообщения

Следует обратить внимание на то, что в заголовке окна сообщения, выводимого процедурой `ShowMessage`, указано название приложения, которое задается на вкладке **Application** окна **Project Options**. Если название приложения не задано, то в заголовке будет имя исполняемого файла.

Функция `MessageDlg` более универсальная. Она позволяет поместить в окно с сообщением один из стандартных значков, например *"Внимание"*, задать количество и тип командных кнопок и определить, какую из кнопок нажал пользователь. На рис. 1.8 приведено окно, выведенное в результате выполнения инструкции

```
r:=MessageDlg('Файл ' + FName + ' будет удален.',
              mtWarning, [mbOk, mbCancel], 0);
```

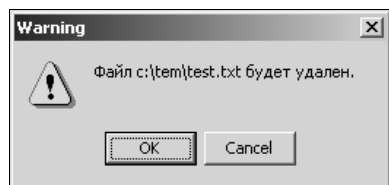


Рис. 1.8. Пример окна сообщения

Значение функции `MessageDlg` — число, проверив значение которого можно определить, выбором какой командной кнопки был завершен диалог.

В общем виде обращение к функции `MessageDlg` выглядит так:

```
Выбор:= MessageDlg (Сообщение, Тип, Кнопки, КонтекстСправки)
```

где:

Сообщение — текст сообщения;

Тип — тип сообщения. Сообщение может быть информационным, предупреждающим или сообщением о критической ошибке. Каждому типу сообщения соответствует определенный значок. Тип сообщения задается именной константой (табл. 1.8);

Таблица 1.8

Константа	Тип сообщения	Значок
<code>MtWarning</code>	Внимание	
<code>MtError</code>	Ошибка	
<code>MtInformation</code>	Информация	
<code>mtConfirmation</code>	Подтверждение	
<code>mtCustom</code>	Обычное	Без значка

Кнопки — список кнопок, отображаемых в окне сообщения. Список может состоять из нескольких разделенных запятыми именованных констант (табл. 1.9). Весь список заключается в квадратные скобки.

Таблица 1.9

Константа	Кнопка	Константа	Кнопка
mbYes	Yes	mbAbort	Abort
mbNo	No	mbRetry	Retry
mbOK	OK	mbIgnore	Ignore
mbCancel	Cancel	mbAll	All
mbHelp	Help		

Например, для того чтобы в окне сообщения появились кнопки **ОК** и **Cancel**, список *Кнопки* должен быть таким:

```
[mbOK,mbCancel]
```

Кроме приведенных констант можно использовать константы: `mbOkCancel`, `mbYesNoCancel` и `mbAbortRetryIgnore`. Эти константы определяют наиболее часто используемые в диалоговых окнах комбинации командных кнопок.

КонтекстСправки — параметр, определяющий раздел справочной системы, который появится на экране, если пользователь нажмет клавишу <F1>. Если вывод справки не предусмотрен, то значение параметра *КонтекстСправки* должно быть равно нулю.

Значение, возвращаемое функцией `MessageDlg` (табл. 1.10), позволяет определить, какая из командных кнопок была нажата пользователем.

Таблица 1.10

Значение функции MessageDlg	Диалог завершен нажатием кнопки
MrAbort	Abort
MrYes	Yes
MrOk	Ok
MrRetry	Retry
MrNo	No
MrCancel	Cancel
MrIgnore	Ignore
MrAll	All

Вывод в поле диалогового окна

Часть диалогового окна, предназначенная для вывода информации, называется полем вывода или полем метки. Поле вывода — это компонент `Label`.

Содержимое поле вывода определяется значением свойства `Caption`. Изменить значение свойства `Caption`, как и большинства свойств других компонентов, можно как во время разработки формы приложения, так и во время работы программы.

Для того чтобы во время работы программы изменить содержимое поле вывода, например, вывести в поле результат работы программы, нужно присвоить свойству новое значение.

На рис. 1.9 изображено диалоговое окно программы пересчета веса из фунтов в килограммы. Окно содержит два компонента `Label`. Компонент `Label1` обеспечивает вывод информационного сообщения, компонент `Label2` — вывод результата работы программы.

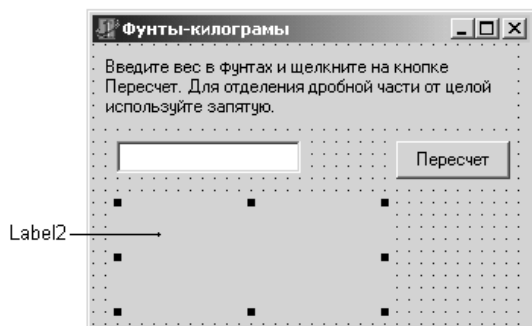


Рис. 1.9. Поле `Label2` предназначено для вывода результата работы программы

Свойство `Caption` символьного типа. Поэтому, для того чтобы во время работы программы вывести в поле метки числовое значение, нужно преобразовать число в строку, например, при помощи функции `FloatToStr` или `IntToStr`.

Ниже в качестве примера приведена инструкция из программы пересчета веса из фунтов в килограммы, которая используется для вывода результата расчета.

```
Label2.Caption:= FloatToStr(kg)+' кг';
```

Процедуры и функции

При программировании в Delphi работа программиста заключается в основном в разработке процедур (подпрограмм) обработки событий.

При возникновении события автоматически запускается процедура обработки события, которую и должен написать программист. Задачу вызова про-

цедуры обработки, при возникновении соответствующего события, берет на себя Delphi.

В языке Object Pascal основной программной единицей является *подпрограмма*. Различают два вида подпрограмм: *процедуры* и *функции*. Как процедура, так и функция, представляют собой последовательность инструкций, предназначенную для выполнения некоторой работы. Чтобы выполнить инструкции подпрограммы, надо вызвать эту подпрограмму. Отличие функции от процедуры заключается в том, что с именем функции связано значение, поэтому имя функции можно использовать в выражениях.

Структура процедуры

Процедура начинается заголовком, за которым следуют:

- раздел объявления констант;
- раздел объявления типов;
- раздел объявления переменных;
- раздел инструкций.

Объявление процедуры в общем виде выглядит следующим образом:

```
procedure Имя (СписокПараметров) ;  
    const    // начало раздела объявления констант  
  
    type     // начало раздела объявления типов  
  
    var      // начало раздела объявления переменных  
  
    begin // начало раздела инструкций  
  
    end;
```

Заголовок процедуры состоит из слова `procedure`, за которым следует имя процедуры. Если у процедуры есть параметры, то они указываются после имени процедуры в скобках. Завершается заголовок процедуры символом *"точка с запятой"*.

Если в процедуре используются именованные константы, то они объявляются в разделе объявления констант, который начинается словом `const`.

За разделом констант следует раздел объявления типов, начинающийся словом `type`.

После раздела объявления типов идет раздел объявления переменных, в котором объявляются (перечисляются) все переменные, используемые в программе. Раздел объявления переменных начинается словом `var`.

За разделом объявления переменных расположен раздел инструкций. Раздел инструкций начинается словом `begin` и заканчивается словом `end`, за кото-

рым следует символ *"точка с запятой"*. В разделе инструкций находятся исполняемые инструкции процедуры.

Ниже, в качестве примера, приведен фрагмент программы вычисления стоимости покупки — процедура Summa.

```

procedure Summa;
var
    cena: real;           // цена
    kol: integer;         // количество
    s: real;              // сумма
    mes: string[255];     // сообщение
begin
    cena := StrToFloat(Form1.Edit1.Text);
    kol := StrToInt(Form1.Edit2.Text);
    s := cena * kol;
    if s > 500 then
        begin
            s := s * 0.9;
            mes := 'Предоставляется скидка 10%' + #13;
        end;
    mes := mes + 'Стоимость покупки: '
        + FloatToStrF(s,ffFixed,4,2) + ' руб.';
    Form1.Label3.Caption := mes;
end;

```

Структура функции

Функция начинается с заголовка, за которым следуют разделы объявления констант, типов и переменных, а также раздел инструкций.

Объявление функции в общем виде выглядит следующим образом:

```

function Имя (СписокПараметров) : Тип;
    const      // начало раздела объявления констант

    type       // начало раздела объявления типов

    var        // начало раздела объявления переменных

    begin // начало раздела инструкций

        result := Значение; // связать с именем функции значение
    end;

```

Заголовок функции начинается словом `function`, за которым следует имя функции. После имени функции в скобках приводится список параметров, за которым через двоеточие указывается тип значения, возвращаемого

функцией (тип функции). Завершается заголовок функции символом *"точка с запятой"*.

За заголовком функции следуют разделы объявления констант, типов и переменных.

В разделе инструкций помимо переменных, перечисленных в разделе описания переменных, можно использовать переменную `result`. По завершении выполнения инструкций функции значение этой переменной становится значением функции. Поэтому среди инструкций функции обязательно должна быть инструкция, присваивающая переменной `result` значение. Как правило, эта инструкция является последней исполняемой инструкцией функции.

Ниже, в качестве примера, приведена функция `FuntToKg`, которая пересчитывает вес из фунтов в килограммы:

```
// Пересчет веса из фунтов в килограммы
function FuntToKg(f:real):real;
const
    // в России 1 фунт равен 409,5 гр.
    K=0.4095; // коэф. Пересчета
begin
    result:=f*K;
end;
```

Запись инструкций программы

Одну инструкцию от другой отделяют точкой с запятой или, другими словами, в конце каждой инструкции ставят точку с запятой.

Хотя в одной строке программы можно записать несколько инструкций, однако, как правило, каждую инструкцию программы записывают в отдельной строке.

Некоторые инструкции (`if`, `case`, `repeat`, `while` и др.) принято записывать в несколько строк, используя для выделения структуры инструкции отступы. Ниже приведен пример инструкции, которая записана в несколько строк и с использованием отступов:

```
if d >= 0
then
    begin
        x1:=(-b+Sqrt(d))/(2*a);
        x2:=(-b-Sqrt(d))/(2*a);
        ShowMessage('x1='+FloatToStr(x1)+
                    'x2='+FloatToStr(x2));
    end
else
    ShowMessage('Уравнение не имеет корней.');
```


Следует обратить внимание на то, что слова `then` и `else` записаны одно под другим (с одинаковым отступом) и с отступом относительно слова `if`. Слово `end` располагается под словом `begin`, а инструкции между `begin` и `end` размещаются одна под другой, но с отступом относительно `begin`.

Приведенную выше инструкцию можно записать и так:

```
if d >= 0 then begin
x1:=(-b+Sqrt(d))/(2*a);
x2:=(-b-Sqrt(d))/(2*a);
ShowMessage('x1='+FloatToStr(x1)+'x2='+FloatToStr(x2));
end
else ShowMessage('Уравнение не имеет корней.');
```

Однако первый вариант лучше, т. к. он отражает структуру алгоритма, реализуемого инструкцией. С первого взгляда видна группа инструкций, которая будет выполнена, если условие `d >= 0` выполняется (в этом случае будут вычислены значения переменных `x1` и `x2`), и инструкция, которая будет выполнена, если условие `d >= 0` не выполняется.

Длинные выражения тоже могут быть записаны в несколько строк. Разорвать выражение и перенести оставшуюся часть на следующую строку можно практически в любом месте. Нельзя разрывать имена переменных, числовые и строковые константы, а также составные операторы, например оператор присваивания.

Ниже приведен пример записи выражения в несколько строк:

```
st:= 'Корни уравнения'+ #13
    +'x1=',FloatToStr(x1)+ #13
    +'x2=',FloatToStr(x2));
```

Еще один момент, на который следует обратить внимание. Компилятор игнорирует "лишние" пробелы и пустые строки. Например, компилятор игнорирует все пробелы в начале строки. Кстати, это и позволяет записывать инструкции с отступами. Не требуются пробелы при записи арифметических и логических выражений (условий), списков параметров. Однако при их использовании программа легче воспринимается. Сравните два варианта записи инструкции присваивания:

```
x1:=(-b+Sqrt(d))/(2*a);
```

и

```
x1 := (-b + Sqrt(d)) / (2 * a);
```

Очевидно, что второй вариант воспринимается лучше.

Для облегчения понимания логики работы программы в текст программы нужно включать поясняющий текст — *комментарии*. В общем случае комментарии заключают в фигурные скобки. Открывающая скобка помечает

начало комментария, закрывающая конец. Если комментарий однострочный или находится после инструкции, то перед комментарием ставят две наклонные черты.

Ниже приведен пример раздела объявления переменных, в котором использованы оба способа записи комментариев:

```
var
  { коэффициенты уравнения }
  a:real; // при второй степени неизвестного
  b:real; // при первой степени неизвестного
  c:real; // при нулевой степени неизвестного

  { корни уравнения }
  x1,x2:real;
```

Стиль программирования

Работая над программой, программист, особенно начинающий, должен хорошо представлять, что программа, которую он разрабатывает, предназначена, с одной стороны, для пользователя, с другой — для самого программиста. Текст программы нужен прежде всего самому программисту, а также другим людям, с которыми он совместно работает над проектом. Поэтому, для того чтобы работа была эффективной, программа должна быть легко читаемой, ее структура должна соответствовать структуре и алгоритму решаемой задачи. Как этого добиться? Надо следовать правилам хорошего стиля программирования. Стиль программирования — это набор правил, которым следует программист (осознано или по тому, что "так делают другие") в процессе своей работы. Очевидно, что хороший программист должен следовать правилам хорошего стиля.

Хороший стиль программирования предполагает:

- использование комментариев;
- использование несущих смысловую нагрузку имен переменных, процедур и функций;
- использование отступов;
- использование пустых строк.

Следование правилам хорошего стиля программирования значительно уменьшает вероятность появления ошибок на этапе набора текста, делает программу легко читаемой, что в свою очередь облегчает процессы отладки и внесения изменений.

Четкого критерия оценки степени соответствия программы хорошему стилю программирования не существует. Вместе с тем, достаточно одного взгляда, чтобы оценить, соответствует ли программа хорошему стилю или нет.

Сводить понятие стиля программирования только к правилам записи текста программы было бы неверно. Стил, которого придерживается программист, проявляется во время работы программы. Хорошая программа должна быть прежде всего надежной и дружественной к пользователю.

Надежность подразумевает, что программа, не полагаясь на "разумное" поведение пользователя, контролирует исходные данные, проверяет результат выполнения операций, которые по какой-либо причине могут быть не выполнены, например операций с файлами.

Дружественность предполагает хорошо спроектированные диалоговые окна, наличие справочной системы, разумное и предсказуемое, с точки зрения пользователя, поведение программы.

Примечание

Приведенные в книге программы могут служить примером следования правилам хорошего стиля программирования.

ГЛАВА 2



Управляющие структуры Object Pascal

На практике редко встречаются задачи, алгоритм решения которых является линейным. Часто оказывается, что алгоритм решения даже элементарной задачи не является линейным. Например, пусть надо вычислить по формуле ток, в электрической цепи. Если предположить, что пользователь всегда будет вводить верные данные, то алгоритм решения этой задачи действительно является линейным. Однако полагаться на то, что пользователь будет себя вести так, как надо программе, не следует. Формула расчета предполагает, что величина сопротивления не равна нулю. А что будет, если пользователь введет 0? Ответ простой: возникнет ошибка "Деление на ноль" и программа аварийно завершит работу. Можно, конечно, возложить ответственность за это на пользователя, но лучше внести изменения в алгоритм решения (рис. 2.1), чтобы расчет выполнялся только в том случае, если введены верные данные.

Точки алгоритма, в которых выполняется выбор дальнейшего хода программы, называются точками выбора. Выбор очередного шага решения задачи осуществляется в зависимости от выполнения некоторого условия.

Условие

В повседневной жизни условие обычно формулируется в виде вопроса, на который можно ответить **Да** или **Нет**. Например:

Величина сопротивления равна нулю?

Ответ правильный?

Сумма покупки больше 300 рублей?

В программе *условие* — это выражение логического типа (Boolean), которое может принимать одно из двух значений: True (истина) или False (ложь).

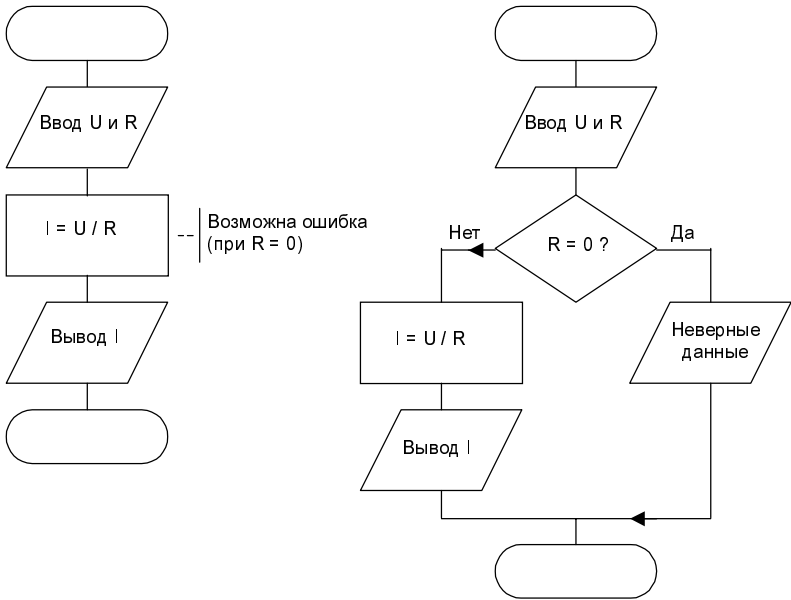


Рис. 2.1. Два варианта алгоритма решения одной задачи

Простое условие состоит из двух операндов и оператора сравнения. В общем виде условие записывается следующим образом:

Оп1 *Оператор* Оп2

где:

Оп1 и Оп2 — операнды условия, в качестве которых может выступать переменная, константа, функция или выражение;

Оператор — оператор сравнения.

В Turbo Pascal есть шесть операторов сравнения, которые приведены в табл. 2.1.

Таблица 2.1. Операторы сравнения

Оператор	Описание	Результат сравнения
>	Больше	True, если первый операнд больше второго, иначе False
<	Меньше	True, если первый операнд меньше второго, иначе False
=	Равно	True, если первый операнд равен второму, иначе False

Таблица 2.1 (окончание)

Оператор	Описание	Результат сравнения
<>	Не равно	True, если первый операнд не равен второму, иначе False
>=	Больше или равно	True, если первый операнд больше или равен второму, иначе False
<=	Меньше или равно	True, если первый операнд меньше или равен второму, иначе False

Ниже приведены примеры условий:

```
Summa < 1000
Score >= HBound
Sim = Chr(13)
```

В первом примере операндами условия является переменная и константа. Значение этого условия зависит от значения переменной `Summa`. Условие будет верным и, следовательно, иметь значение `True`, если значение переменной `Summa` меньше, чем 1000. Если значение переменной `Summa` больше или равно 1000, то значение этого условия будет `False`.

Во втором примере в качестве операндов используются переменные. Значение этого условия будет `True`, если значение переменной `Score` больше или равно значению переменной `HBound`.

В третьем примере в качестве второго операнда используется функция. Значение этого условия будет `True`, если в переменной `Sim` находится символьный код клавиши `<Enter>`, равный 13.

При записи условий следует обратить особое внимание на то, что операнды условия должны быть одного типа или, если тип операндов разный, то тип одного из операндов может быть приведен к типу другого операнда. Например, если в программе переменная `Key` объявлена как `integer`, то условие

```
Key = Chr(13)
```

неверное (`False`), т. к. возвращаемое функцией `Chr` значение имеет тип `char` (символьный).

Во время трансляции программы при обнаружении неверного условия компилятор выводит сообщение: `Incompatible types` (несовместимые типы).

Из простых условий при помощи логических операторов: `and` — "логическое И", `or` — "логическое ИЛИ" и `not` — "отрицание" можно строить сложные условия.

В общем виде сложное условие записывается следующим образом:

условие1 оператор *условие2*

где:

условие1 и *условие2* — простые условия (выражения логического типа);
оператор — оператор AND или OR.

Результат применения логических операторов `and`, `or` и `not` к операндам логического типа представлен в табл. 2.2.

Таблица 2.2. Применение логических операторов к операндам логического типа

Op1	Op2	Op1 and Op2	Op1 or Op2	not Op1
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

При записи сложных условий важно учитывать то, что логические операторы имеют более высокий приоритет, чем операторы сравнения, и поэтому простые условия следует заключать в скобки.

Например, пусть условие предоставления скидки сформулировано следующим образом: "Скидка предоставляется, если сумма покупки превышает 100 руб. и день покупки — воскресенье". Если день недели обозначен как переменная `Day` целого типа, и равенство ее значения семи соответствует воскресенью, то условие предоставления скидки можно записать:

`(Summa > 100) and (Day = 7)`

Если условие предоставления скидки дополнить тем, что скидка предоставляется в любой день, если сумма покупки превышает 500 руб., то условие можно записать:

`((Summa > 100) and (Day = 7)) or (Summa > 500)`

Выбор

Выбор в точке разветвления алгоритма очередного шага программы может быть реализован при помощи инструкций `if` и `case`. Инструкция `if` позволяет выбрать один из двух возможных вариантов, инструкция `case` — один из нескольких.

Инструкция IF

Инструкция `if` позволяет выбрать один из двух возможных вариантов развития программы. Выбор осуществляется в зависимости от *выполнения* условия.

В общем виде инструкция `if` записывается так:

```
if    условие
then
    begin
        //  здесь инструкции, которые надо выполнить,
        //  если условие истинно.
    end
else
    begin
        //  здесь инструкции, которые надо выполнить,
        //  если условие ложно.
    end;
end;
```

Обратите внимание, что после первого `end` точка с запятой не ставится.

Выполняется инструкция `if` следующим образом:

1. Вычисляется значение условия (условие — выражение логического типа, значение которого может быть равно `True` или `False`).
2. Если условие истинно (значение выражения *условие* равно `True`), то выполняются инструкции, следующие за словом `then` (между `begin` и `end`).
3. Если условие ложно (значение выражения *условие* равно `False`), то выполняются инструкции, следующие за словом `else` (между `begin` и `end`).

На рис. 2.2 представлен алгоритм, соответствующий инструкции `if-then-else`.

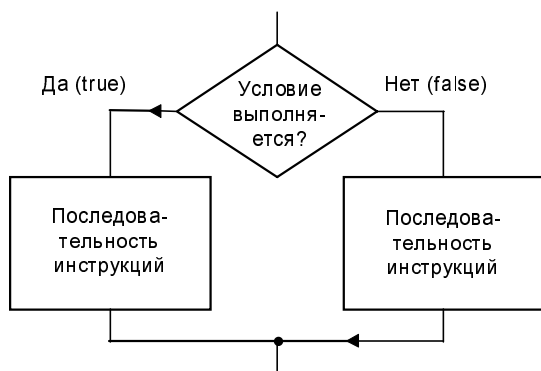


Рис. 2.2. Алгоритм, реализуемый инструкцией `if-then-else`

Например, если переменная `t` обозначает тип соединения сопротивлений в электрической цепи (`t=1` соответствует последовательному соединению, `t=2` — параллельному), а `r1` и `r2` величины сопротивлений, то приведенная ниже инструкция `if` осуществляет выбор формулы, по которой будет выполнен расчет.

```
if t=1
then
begin
    z:=r1+r2;
end
else
begin
    z:=(r1+r2)/(r1*r2);
end;
```

Примечание

Если в инструкции `if` между словами `begin` и `end` находится только одна инструкция программы, то слова `begin` и `end` можно не писать.

Например, инструкцию

```
if otv=3
then
begin
    prav:=prav+1;
end
else
begin
    ShowMessage('Ошибка!');
end;
```

можно переписать так:

```
if otv=3
then
    prav:=prav+1
else
    ShowMessage('Ошибка!');
```

Если какое-либо действие должно быть выполнено только при выполнении определенного условия и пропущено, если это условие не выполняется, то инструкция `if` может быть записана так:

```
if    УСЛОВИЕ
then
begin
```

```
{ инструкции, которые надо выполнить,  
  если условие выполняется, истинно }  
  
end
```

На рис. 2.3 представлен алгоритм, соответствующий инструкции `if-then`.

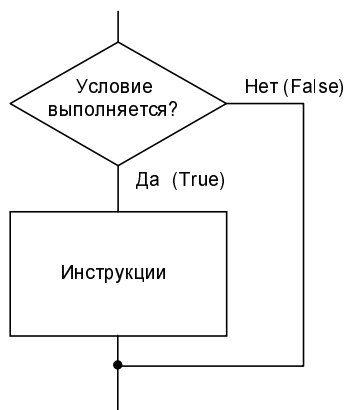


Рис. 2.3. Алгоритм, реализуемый инструкцией `if-then`

Например, инструкция

```
if n=m  
  then   c:=c+1;
```

увеличивает значение переменной `c` только в том случае, если значения переменных `n` и `m` равны.

В качестве примера использования инструкции `if` рассмотрим программу вычисления стоимости телефонного разговора.

Как известно, стоимость разговора по телефону в воскресные дни ниже, чем в обычные. Программа, текст которой приведен в листинге 2.1, запрашивает длительность разговора и день недели, а затем вычисляет стоимость разговора. Если день недели — суббота или воскресенье, то стоимость уменьшается на величину скидки. Цена минуты разговора и величина скидки задаются в тексте программы как константы. Вид диалогового окна программы приведен на рис. 2.4

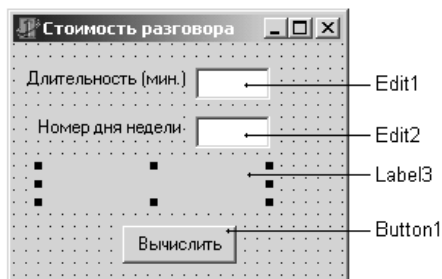


Рис. 2.4. Диалоговое окно программы **Стоимость разговора**

Для ввода исходных данных (длительность разговора, номер дня недели) используются поля редактирования, для вывода результата и пояснительного текста — поля меток. В табл. 2.3 перечислены компоненты и указано их назначение, а в табл. 2.4 — приведены значения свойств этих компонентов.

Примечание

Здесь и далее при описании формы приложения приводятся значения только тех свойств компонентов, которые используются в программе. Значения остальных свойств, в частности определяющих размер и положение компонентов, могут быть оставлены без изменения или изменены произвольным образом, естественно, в разумных пределах (очевидно, что положение командной кнопки или поля редактирования может быть выбрано в пределах формы произвольным образом).

Таблица 2.3. Компоненты формы приложения

Компонент	Назначение
Edit1	Для ввода длительности разговора в минутах
Edit2	Для ввода номера дня недели
Label1, Label2	Для вывода пояснительного текста о назначении полей ввода
Label3	Для вывода результата вычисления — стоимости разговора
Button1	Для активизации процедуры вычисления стоимости разговора

Примечание

В таблицах, содержащих описание значений свойств компонентов формы, указывается имя компонента и через точку имя свойства. Например, строка таблицы

Form1.Caption Стоимость разговора

обозначает, что во время создания формы приложению свойству Caption формы приложения надо присвоить указанное значение — текст "Стоимость разговора".

Таблица 2.4. Значения свойств компонентов формы

Свойство	Значение
Form1.Caption	Стоимость разговора
Edit1.Text	
Edit2.Text	
Label1.Caption	Длительность (мин.)

Таблица 2.4 (окончание)

Свойство	Значение
Label2.Caption	Номер дня недели
Label3. Caption	
Button1.Caption	Вычислить

Программа производит вычисления при щелчке командной кнопки **Вычислить**. В этом случае возникает событие `OnClick`, которое обрабатывается процедурой `TForm1.Button1Click`.

Листинг 2.1. Вычисление стоимости телефонного разговора (Phone_u.pas)

```
unit Phone_u;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // поле ввода длительности разговора
    Edit2: TEdit;           // поле ввода номера дня недели
    Button1: TButton;       // кнопка Вычислить
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```

procedure TForm1.Button1Click(Sender: TObject);
const
    PAY = 0.15;      // цена одной минуты разговора 0.15 рубля
    DISCOUNT = 0.2; // скидка 20 процентов
var
    Time:Real;      // длительность разговора
    Day:integer;    // день недели
    Summa:real;     // стоимость разговора
begin
    // получить исходные данные
    Time:=StrToFloat(Edit1.Text);
    Day:=StrToInt(Edit2.Text);

    // Вычислить стоимость разговора
    Summa:= PAY*Time;
    // Если день суббота или воскресенье, то уменьшить
    // стоимость на величину скидки
    if (Day = 6) OR (Day = 7)
        then Summa:=Summa* (1 -DISCOUNT);

    // вывод результата вычисления
    label3.caption:='К оплате '+FloatToStr(Summa)+'руб.';
end;

end.

```

Часто в программе необходимо реализовать выбор более чем из двух вариантов. Например, известно, что для каждого человека существует оптимальное значение веса, которое может быть вычислено по формуле:

$$\text{Рост (см)} - 100.$$

Реальный вес может отличаться от оптимального: вес может быть меньше оптимального, равняться ему или превышать оптимальное значение.

Следующая программа, диалоговое окно которой приведено на рис. 2.5, запрашивает вес и рост, вычисляет оптимальное значение, сравнивает его с реальным весом и выводит соответствующее сообщение.



Рис. 2.5. Окно программы **Контроль веса**

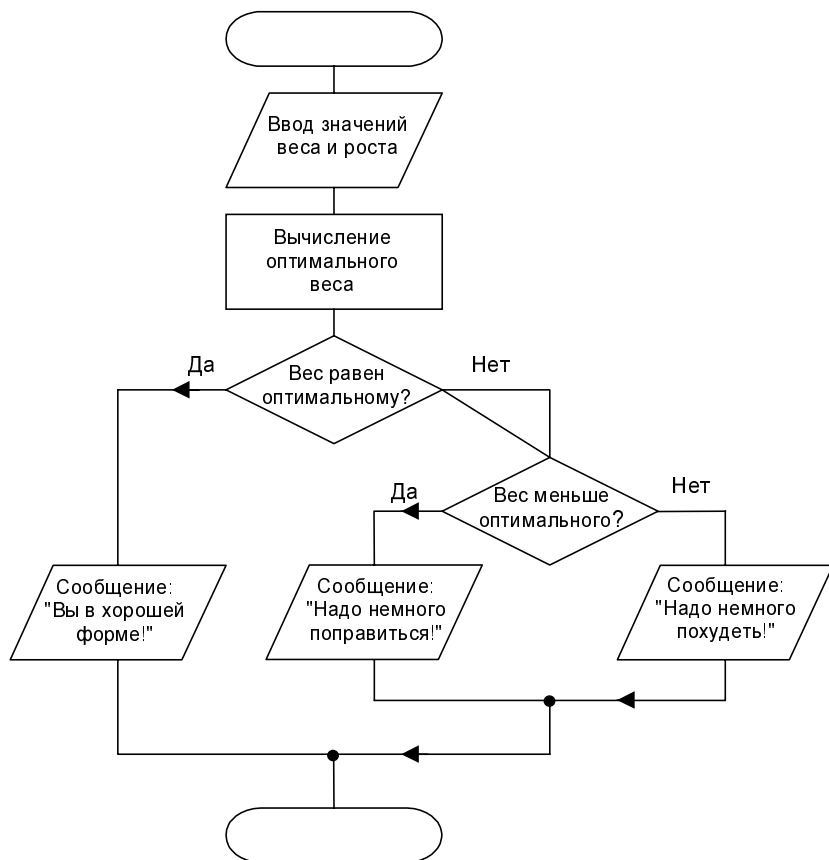


Рис. 2.6. Алгоритм программы **Контроль веса**

Блок-схема программы **Контроль веса** изображена на рис. 2.6.

Как и в предыдущей программе, вычисления выполняются при щелчке кнопки **Вычислить** (ее имя `Button1`). В листинге 2.2 приведен текст программы.

Листинг 2.2. Контроль веса

```

unit wtest_;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)

```

```

Label1: TLabel;
Label2: TLabel;
Edit1: TEdit;      // поле ввода веса
Edit2: TEdit;      // поле ввода роста
Button1: TButton;  // кнопка Вычислить
Label3: TLabel;    // поле вывода сообщения — результата работы
                  // программы

```

```

procedure Button1Click(Sender: TObject);

```

```

private

```

```

    { Private declarations }

```

```

public

```

```

    { Public declarations }

```

```

end;

```

```

var

```

```

    Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

procedure TForm1.Button1Click(Sender: TObject);

```

```

var

```

```

    w:real;    { вес }

```

```

    h:real;    { рост }

```

```

    opt:real;  { ОПТИМАЛЬНЫЙ вес }

```

```

    d:real;    { ОТКЛОНЕНИЕ ОТ ОПТИМАЛЬНОГО веса }

```

```

begin

```

```

    w:=StrToFloat(Edit1.text);

```

```

    h:=StrToInt(Edit2.Text);

```

```

    opt:=h-100;

```

```

    if w=opt

```

```

    then

```

```

        Label3.caption:='Вы в хорошей форме!'

```

```

    else

```

```

        if w < opt

```

```

            then

```

```

                begin

```

```

                    d:=opt-w;

```

```

                    Label3.caption:='Вам надо поправиться, на '
                        + FloatToStr(d)+ 'кг.';

```

```

                end

```

```

            else

```

```

                begin

```

```

                    d:=w-opt;

```

```
Label3.caption:='Надо немного похудеть, на '
                + FloatToStr(d)+ ' кг.';

end;

end;

end.
```

В приведенном примере множественный выбор реализован при помощи двух инструкций `if`, одна из которых "вложена" в другую.

Инструкция **CASE**

В предыдущем примере, в программе контроля веса, множественный выбор был реализован при помощи вложенных одна в другую инструкций `if`. Такой подход не всегда удобен, особенно в том случае, если количество вариантов хода программы велико.

В Object Pascal есть инструкция `case`, которая позволяет эффективно реализовать множественный выбор. В общем виде она записывается следующим образом:

```
case Селектор of
    список1: begin
        { инструкции 1 }
    end;
    список2: begin
        { инструкции 2 }
    end;
    списокN: begin
        { инструкции N }
    end;
    else
        begin
            { инструкции }
        end;
end;
```

где:

Селектор — выражение, значение которого определяет дальнейший ход выполнения программы (т. е. последовательность инструкций, которая будет выполнена);

СписокN — список констант. Если константы представляют диапазон чисел, то вместо списка можно указать первую и последнюю константу диапазона, разделив их двумя точками. Например, список 1, 2, 3, 4, 5, 6 может быть заменен диапазоном 1..6.

Выполняется инструкция `case` следующим образом:

1. Сначала вычисляется значение выражения-селектора.
2. Значение выражения-селектора последовательно сравнивается с константами из списков констант.
3. Если значение выражения совпадает с константой из списка, то выполняется соответствующая этому списку группа инструкций. На этом выполнение инструкции `case` завершается.
4. Если значение выражения-селектора не совпадает ни с одной константой из всех списков, то выполняется последовательность инструкций, следующая за `else`.

Синтаксис инструкции `case` позволяет не писать `else` и соответствующую последовательность инструкций. В этом случае, если значение выражения не совпадает ни с одной константой из всех списков, то выполняется следующая за `case` инструкция программы.

На рис. 2.7 приведен алгоритм, реализуемый инструкцией `case`.

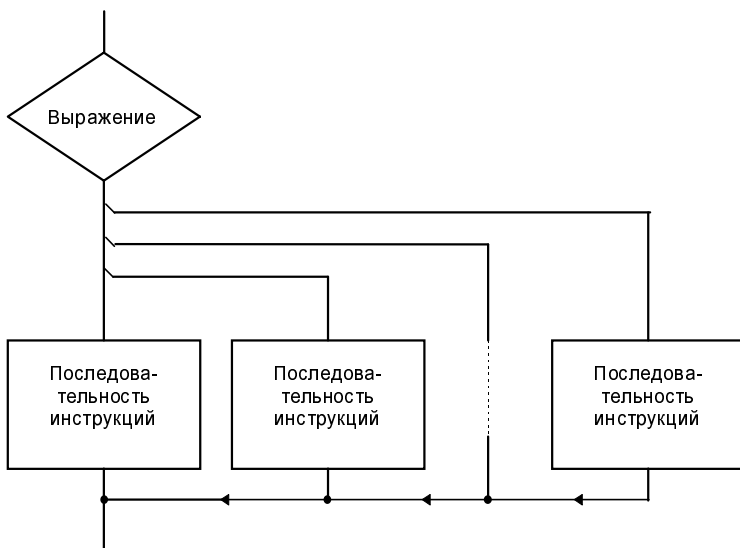


Рис. 2.7. Алгоритм, реализуемый инструкцией `case`

Ниже приведены примеры инструкции `case`.

```
case n_day of
  1,2,3,4,5: day:='Рабочий день.';
  6:       day:='Суббота!';
  7:       day:='Воскресенье!';
end;
```

```
case n_day of
  1..5: day:='Рабочий день.';
  6:    day:='Суббота!';
  7:    day:='Воскресенье!';
end;

case n_day of
  6:    day:='Суббота!';
  7:    day:='Воскресенье!';
  else day:='Рабочий день.';
end;
```

В качестве примера использования инструкции `case` рассмотрим программу, которая пересчитывает вес из фунтов в килограммы. Программа учитывает, что в разных странах фунт "весит" по-разному. Например, в России фунт равен 409,5 граммов, в Англии — 453,592 грамма, а в Германии, Дании и Исландии фунт весит 500 граммов.

В диалоговом окне программы, изображенном на рис. 2.8, для выбора страны используется список **Страна**.

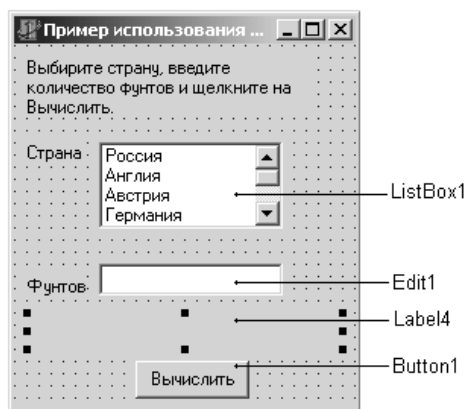


Рис. 2.8. Диалоговое окно программы **Пример использования case**

Для выбора названия страны используется список — компонент `ListBox`. Значок компонента `ListBox` находится на вкладке **Standard** (рис. 2.9). Добавляется список к форме приложения точно так же, как и другие компоненты, например, командная кнопка или поле редактирования. В табл. 2.5 приведены свойства компонента `ListBox`.



Рис. 2.9. Компонент `ListBox`

Таблица 2.5. Свойства компонента `Listbox`

Свойство	Определяет
Name	Имя компонента. В программе используется для доступа к свойствам компонента
Items	Элементы списка
ItemIndex	Номер выбранного элемента списка. Номер первого элемента списка равен нулю
Left	Расстояние от левой границы списка до левой границы формы
Top	Расстояние от верхней границы списка до верхней границы формы
Height	Высоту поля списка
Width	Ширину поля списка
Font	Шрифт, используемый для отображения элементов списка
ParentFont	Признак наследования свойств шрифта родительской формы

Наибольший интерес представляют свойства `Items` и `ItemIndex`.

Свойство `Items` содержит элементы списка.

Свойство `ItemIndex` задает номер выбранного элемента списка. Если ни один из элементов не выбран, то значение свойства равно минус единице.

Список может быть сформирован во время создания формы или во время работы программы.

Для формирования списка во время создания формы надо в окне **Object Inspector** выбрать свойство `Items` и щелкнуть на кнопке запуска редактора списка (рис. 2.10).

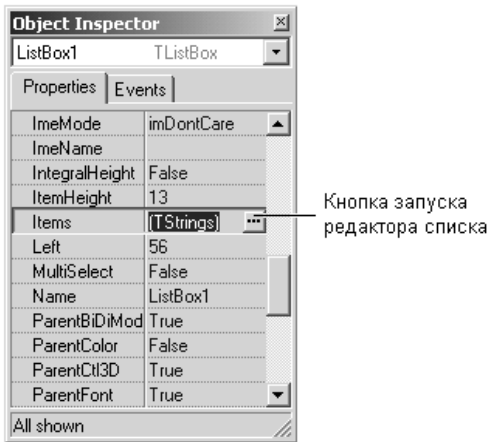


Рис. 2.10. Кнопка запуска редактора списка

В открывшемся диалоговом окне **String List Editor** (рис. 2.11) нужно набрать список, каждый элемент списка в отдельной строке. После ввода очередного элемента списка, для перехода к новой строке, необходимо нажать клавишу <Enter>. После ввода последнего элемента списка клавишу <Enter> нажимать не надо. Завершив ввод списка, следует щелкнуть на кнопке **OK**.

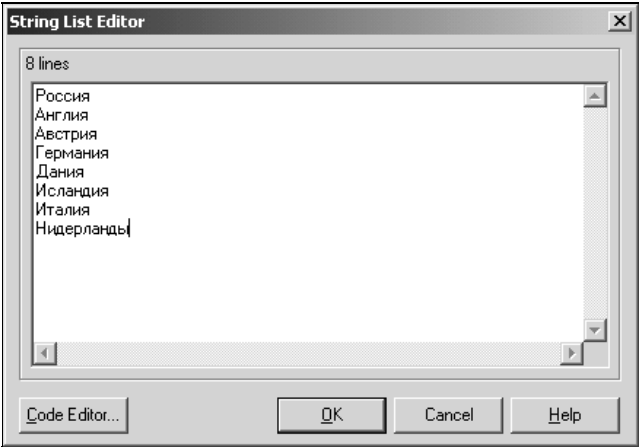


Рис. 2.11. Редактор списка

В табл. 2.6 перечислены компоненты формы приложения, а в табл. 2.7 приведены значения свойств компонентов.

***Таблица 2.6.** Компоненты формы приложения*

Компонент	Назначение
ListBox1	Для выбора страны, для которой надо выполнить пересчет
Edit1	Для ввода веса в фунтах
Label1, Label2, Label3	Для вывода пояснительного текста о назначении полей ввода
Label4	Для вывода результата пересчета
Button1	Для активизации процедуры пересчета веса из фунтов в килограммы

***Таблица 2.7.** Значения свойств компонентов формы*

Свойство	Значение
Form1.Caption	Пример использования case
Edit1.Text	

Таблица 2.7 (окончание)

Свойство	Значение
Label1.Caption	Выберите страну, введите количество фунтов и щелкните на кнопке Вычислить
Label2.Caption	Страна
Label3. Caption	Фунтов
Button1.Caption	Вычислить

Процедура пересчета, которая выполняется в результате щелчка на командной кнопке **Вычислить**, умножает вес в фунтах на коэффициент, равный количеству килограммов в одном фунте. Значение коэффициента определяется по номеру выбранного из списка элемента.

В листинге 2.3 приведен текст программы пересчета веса из фунтов в килограммы.

Листинг 2.3. Пересчет веса из фунтов в килограммы

```

unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label2: TLabel;
    Edit1: TEdit;      // поле ввода веса в фунтах
    Button1: TButton;  // кнопка Вычислить
    Label1: TLabel;
    Label3: TLabel;
    ListBox1: TListBox; // список стран
    Label4: TLabel;    // поле вывода рез-та — веса в килограммах
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

```

```
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
{
    ListBox1.items.add('Россия');
    ListBox1.items.add('Австрия');
    ListBox1.items.add('Англия');
    ListBox1.items.add('Германия');
    ListBox1.items.add('Дания');
    ListBox1.items.add('Исландия');
    ListBox1.items.add('Италия');
    ListBox1.items.add('Нидерланды');
}
ListBox1.itemindex:=0;

end;

procedure TForm1.Button1Click(Sender: TObject);
var
    funt:real; // вес в фунтах
    kg:real;   // вес в килограммах
    k:real;    // коэффициент пересчета

begin
    case ListBox1.ItemIndex of
        0: k:=0.4095;    // Россия
        1: k:=0.453592;  // Англия
        2: k:=0.56001;   // Австрия
        3..5,7:k:=0.5;    // Германия, Дания, Исландия, Нидерланды
        6: k:=0.31762;   // Италия
    end;

    funt:=StrToFloat(Edit1.Text);
    kg:=k*funt;
    label4.caption:=Edit1.Text
        + ' фунт. — это '
        + FloatToStrF(kg,ffFixed,6,3)
        + ' кг.';

end;

end.
```

Следует обратить внимание на процедуру обработки события `FormCreate`, которое происходит в момент создания формы (форма создается автомати-

чески при запуске программы). Эту процедуру можно использовать для инициализации переменных программы, в том числе и для добавления элементов в список. В приведенном тексте программы инструкции создания списка закомментированы, т. к. список был создан при помощи редактора строк во время создания формы.

Рассмотрим еще один пример использования инструкции `case`. При выводе числовой информации с поясняющим текстом возникает проблема согласования выводимого значения и окончания поясняющего текста.

Например, в зависимости от числового значения поясняющий текст к денежной величине может быть: "рубль", "рублей" или "рубля" (123 рубля, 120 рублей, 121 рубль). Очевидно, что окончание поясняющего слова определяется последней цифрой числа, что отражено в табл. 2.8.

Таблица 2.8. Зависимость окончания текста от последней цифры числа

Цифра	Поясняющий текст
0, 5, 6, 7, 8, 9	Рублей
1	Рубль
2, 3, 4	Рубля

Приведенное в таблице правило имеет исключение для чисел, оканчивающихся на 11, 12, 13, 14. Для них поясняющий текст должен быть "рублей".

Диалоговое окно программы приведено на рис. 2.12, а текст — в листинге 2.4.

Поясняющий текст формирует процедура обработки события `OnKeyPress`.

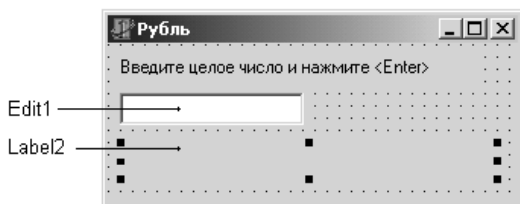


Рис. 2.12. Диалоговое окно программы

Листинг 2.4

```
unit rub_1;

interface

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

```

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// нажатие клавиши
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  n : integer;      // число
  r : integer;      // остаток от деления n на 10
  text: string[10]; // формируемый поясняющий текст
begin
  if Key = chr(VK_RETURN) then
    begin
      n := StrToInt(Edit1.Text);
      if n > 100
        then n:=n mod 100;
      if ( n >= 11) and ( n <= 14)
        then
          text:=' рублей'
        else
          begin
            r:= n mod 10;
            case r of
              1:      text:=' рубль';
              2 .. 4: text:=' рубля';
              else    text:=' рублей';
            end;
          end;
      Label2.Caption := IntToStr(n)+ text;
    end;
end;

end.

```


Рассмотрим фрагмент программы, которая вычисляет дату следующего дня, используя сегодняшнюю дату, представленную тремя переменными day (день), month (месяц) и year (год).

Сначала с помощью инструкции case проверяется, является ли текущий день последним днем месяца. Если текущий месяц — февраль и если текущее число — 28, то дополнительно выполняется проверка: является ли год високосным. Для этого вычисляется остаток от деления года на 4. Если остаток равен нулю, то год високосный, и число 28 не является последним днем месяца.

Если выясняется, что текущий день — последний день месяца, то следующее число — первое. Затем проверяется, не является ли текущий месяц декабрем. Если — нет, то увеличивается номер месяца, а если — да, то увеличивается номер года, а номеру месяца присваивается значение 1.

```
// вычисление даты следующего дня
```

```
var
```

```
  day: integer;    // день
  month: integer;  // месяц
  year: integer;   // год
  last: boolean;   // если день — последний день месяца,
                  // то last = True
  r: integer;      // если год не високосный, то остаток
                  // от деления year на 4 не равен нулю
```

```
begin
```

```
  { переменные day, month и year
    содержат сегодняшнюю дату }
```

```
  last := False; // пусть день — не последний день месяца
```

```
  case month of
```

```
    4,6,9,11: if day = 30 then last:= True;
```

```
    2:      if day = 28 then
              begin
                r:= year mod 4;
                if r <> 0 then last:= True;
              end;
```

```
    else:    if day=31 then last:= True;
```

```
  end;
```

```
  if last then
```

```
    begin // последний день месяца
```

```
      day:= 1;
```

```
      if month = 12 then
```

```
begin // последний месяц
    month:= 1;
    year:= year + 1;
end
else month:= month + 1;
end
else day:= day + 1;

// переменные day, month и year
// содержат завтрашнюю дату
end;
```

Циклы

Алгоритмы решения многих задач являются циклическими, т. е. для достижения результата определенная последовательность действий должна быть выполнена несколько раз.

Например, программа контроля знаний выводит вопрос, принимает ответ, добавляет оценку за ответ к сумме баллов, затем повторяет это действие еще и еще раз, и так до тех пор, пока испытуемый не ответит на все вопросы.

Другой пример. Для того чтобы найти фамилию человека в списке, надо проверить первую фамилию списка, затем вторую, третью и т. д. до тех пор, пока не будет найдена нужная фамилия или не будет достигнут конец списка.

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

В программе цикл может быть реализован при помощи инструкций `for`, `while` и `repeat`.

Инструкция **FOR**

Рассмотрим следующую задачу. Пусть нужно написать программу, которая вычисляет значение функции $y = 5 \times 2^{-7}$ в точках -1, -0.5, 0, 0.5 и 1 (таблица должна быть выведена в поле метки формы приложения). Процедура, обеспечивающая решение поставленной задачи, может выглядеть так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    y: real; // значение функции
    x: real; // аргумент функции
    dx: real; // приращение аргумента
```

```

st: string; // изображение таблицы

begin
  st:='';

  x := -1;
  dx := 0.5;

  y := 5*x*x -7;
  st := st+ FloatToStr(x)+'  ' + FloatToStr(y)+chr(13);
  x :=x + dx;

  y := 5*x*x -7;
  st := st+ FloatToStr(x)+'  ' + FloatToStr(y)+chr(13);
  x :=x + dx;

  y := 5*x*x -7;
  st := st+ FloatToStr(x)+'  ' + FloatToStr(y)+chr(13);
  x :=x + dx;

  y := 5*x*x -7;
  st := st+ FloatToStr(x)+'  ' + FloatToStr(y)+chr(13);
  x :=x + dx;

  y := 5*x*x -7;
  st := st+ FloatToStr(x)+'  ' + FloatToStr(y)+chr(13);
  x :=x + dx;

  Label1.Caption := st;

end;
```

Из текста процедуры видно, что группа инструкций

```

y := 5*x*x -7;
st := st+ FloatToStr(x)+'  ' + FloatToStr(y)+chr(13);
x :=x + dx;
```

обеспечивающая вычисление значения функции, формирование строки таблицы и увеличение аргумента, выполняется 5 раз.

Воспользовавшись инструкцией `for`, приведенную процедуру можно переписать следующим образом:

```

procedure TForm1.Button1Click(Sender: TObject);
var
  y: real;    // значение функции
  x: real;    // аргумент функции
  dx: real;   // приращение аргумента
```

```

st: string; // изображение таблицы

i : integer; // счетчик циклов

begin
  st:='';
  x := -1;
  dx := 0.5;
  for i:=1 to 5 do
    begin
      y := 5*x*x -7;
      st := st+ FloatToStr(x)+' ' + FloatToStr(y)+chr(13);
      x :=x + dx;
    end;
  Label1.Caption := st;
end;

```

Второй вариант процедуры, во-первых, требует меньше усилий при наборе, во-вторых, процедура более гибкая: для того чтобы увеличить количество строк в выводимой таблице, например до десяти, достаточно в строке `for i:=1 to 5 do` число 5 заменить на 10.

Инструкция `for` используется в том случае, если некоторую последовательность действий (инструкций программы) надо выполнить несколько раз, причем число повторений заранее известно.

В общем виде инструкция `for` записывается следующим образом:

```

for счетчик := нач_знач to кон_знач do
begin
    // здесь инструкции, которые надо выполнить несколько раз
end

```

где:

счетчик — переменная-счетчик числа повторений инструкций цикла;

нач_знач — выражение, определяющее начальное значение счетчика циклов;

кон_знач — выражение, определяющее конечное значение счетчика циклов.

Переменная *счетчик*, выражения *нач_знач* и *кон_знач* должны быть целого типа.

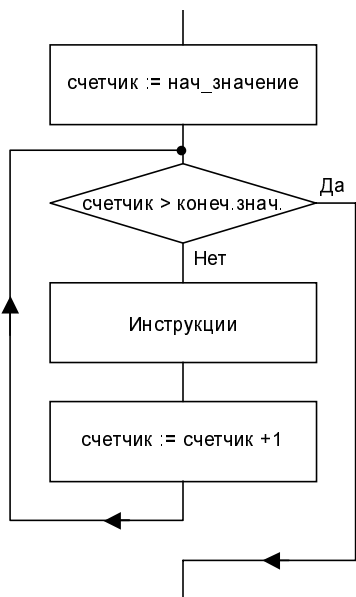
Количество повторений инструкций цикла можно вычислить по формуле $(кон_знач - нач_знач + 1)$.

Примеры:

```
for i:=1 to 10 do
begin
    label1.caption:=label1.caption + '*';
end;
for i:=1 to n do
    s := s+i;
```

Примечание

Если между словами `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать.



Алгоритм, соответствующий инструкции `for`, представлен на рис. 2.13. Обратите внимание, что если начальное значение счетчика больше конечного значения, то последовательность операторов между `begin` и `end` не будет выполнена ни разу.

Кроме того, после каждого выполнения инструкций тела цикла счетчик циклов увеличивается автоматически.

Рис. 2.13. Алгоритм инструкции `for`

Переменную-счетчик можно использовать внутри цикла (но ни в коем случае не изменять). Например, в результате выполнения следующих инструкций:

```
tabl:='';
for i:=1 to 5 do
begin
    tabl:=tabl+IntToStr(i)+' '+IntToStr(i*i)+chr(13);
end;
```

переменная `tabl` будет содержать изображения таблицы квадратов чисел.

Рассмотрим программу, которая вычисляет сумму первых 10 элементов ряда: $1+1/2+1/3+\dots$ (значение i -го элемента ряда связано с его номером форму-

лой 1/i). Диалоговое окно программы должно содержать, по крайней мере, два компонента: поле метки (Label1) и командную кнопку (Button1).

Вычисление суммы ряда и вывод результата выполняет процедура обработки события OnClick, текст которой приведен ниже. После вычисления очередного элемента ряда процедура выводит в поле Label1 номер элемента и его значение в поле метки формы, предназначенное для вывода результата.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;    { номер элемента ряда }
  elem:real;    { значение элемента ряда }
  summ:real;    { сумма элементов ряда }

begin
  summ:=0;
  label1.caption:='';
  for i:=1 to 10 do
    begin
      elem:=1/i;
      label1.caption:=label1.caption+
        IntToStr(i)+' '+FloatToStr(elem)+'13;
      summ:=summ+elem;
    end;
  label1.caption:=label1.caption+
    'Сумма ряда:'+FloatToStr(summ);
end;
```

Если в инструкции for вместо слова to записать downto, то после очередного выполнения инструкций тела цикла значение счетчика будет не увеличиваться, а уменьшаться.

Инструкция WHILE

Инструкция (цикл) while используется в том случае, если некоторую последовательность действий (инструкций программы) надо выполнить несколько раз, причем необходимое число повторений во время разработки программы неизвестно и может быть определено только во время работы программы.

Типичными примерами использования цикла while являются вычисления с заданной точностью, поиск в массиве или в файле.

В общем виде инструкция while записывается следующим образом:

```
while условие do
  begin
    // здесь инструкции, которые надо выполнить несколько раз
  end
```

где *условие* — выражение логического типа, определяющее условие выполнения инструкций цикла.

Инструкция `while` выполняется следующим образом:

1. Сначала вычисляется значение выражения *условие*.
2. Если значение выражения *условие* равно `False` (условие не выполняется), то на этом выполнение инструкции `while` завершается.
3. Если значение выражения *условие* равно `True` (условие выполняется), то выполняются расположенные между `begin` и `end` инструкции тела цикла. После этого снова проверяется выполнение условия. Если условие выполняется, то инструкции цикла выполняются еще раз. И так до тех пор, пока условие не станет ложным (`False`).

Алгоритм, соответствующий инструкции `while`, представлен на рис. 2.14.

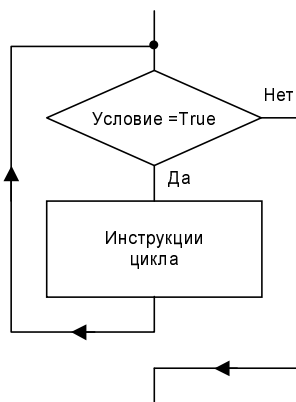


Рис. 2.14. Алгоритм инструкции `while`

Внимание!

Для того чтобы инструкции цикла `while`, находящиеся между `begin` и `end`, были выполнены хотя бы один раз, необходимо, чтобы перед выполнением инструкции `while` значение выражения *условие* было истинно.

Для того чтобы цикл завершился, нужно, чтобы последовательность инструкций между `begin` и `end` влияла на значение выражения *условие* (изменяла значения переменных, входящих в выражение *условие*).

Рассмотрим программу, которая вычисляет значение числа π с точностью, задаваемой пользователем во время работы программы. В основе алгоритма вычисления лежит тот факт, что сумма ряда $1-1/3+1/5-1/7+1/9+\dots$ приближается к значению $\pi/4$ при достаточно большом количестве членов ряда.

Каждый член ряда с номером n вычисляется по формуле: $1/(2 \times n - 1)$ и умножается на минус один, если n четное (определить, является ли n четным, можно проверкой остатка от деления n на 2). Вычисление заканчивается

тогда, когда значение очередного члена ряда становится меньше, чем заданная точность вычисления.

Вид диалогового окна программы во время ее работы приведен на рис. 2.15. Пользователь вводит точность вычисления в поле ввода (Edit1). После щелчка командной кнопки **Вычислить** (Button1) программа вычисляет значение числа π и выводит результат в поле метки (Label1).

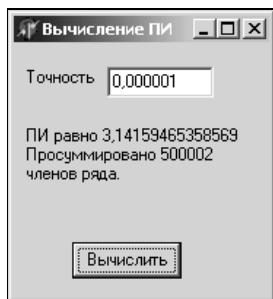


Рис. 2.15. Диалоговое окно программы **Вычисление ПИ**

Текст программы приведен в листинге 2.5. Как и в предыдущих примерах, основную работу выполняет процедура обработки события `OnClick`.

Листинг 2.5. Вычисление числа π

```
unit pi_;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // точность вычисления
    Button1: TButton;       // кнопка ВЫЧИСЛИТЬ
    Label1: TLabel;
    Label2: TLabel;        // поле вывода результата
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```


implementation

```
{$R *.DFM}
```

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

```
var
```

```
pi:real;    // вычисляемое значение ПИ
```

```
t:real;     // точность вычисления
```

```
n:integer;  // номер члена ряда
```

```
elem:real;  // значение члена ряда
```

```
begin
```

```
pi := 0;
```

```
n := 1;
```

```
t := StrToFloat(edit1.text);
```

```
elem := 1; // чтобы начать цикл
```

```
while elem >= t do
```

```
begin
```

```
elem := 1 / (2*n - 1);
```

```
if n MOD 2 = 0
```

```
then pi := pi - elem
```

```
else pi := pi + elem;
```

```
n := n + 1;
```

```
end;
```

```
pi = pi * 4;
```

```
labell.caption:= 'ПИ равно ' + FloatToStr(pi) + #13
```

```
+ 'Просуммировано ' + IntToStr(n) + ' членов ряда.';
```

```
end;
```

```
end.
```

Инструкция *REPEAT*

Инструкция `repeat`, как и инструкция `while`, используется в программе в том случае, если необходимо выполнить повторные вычисления (организовать цикл), но число повторений во время разработки программы неизвестно и может быть определено только во время работы программы, т. е. определяется ходом вычислений.

В общем виде инструкция `repeat` записывается следующим образом:

```
repeat
```

```
    // инструкции
```

```
until условие
```

где *условие* — выражение логического типа, определяющее условие завершения цикла.

Инструкция `repeat` выполняется следующим образом:

1. Сначала выполняются находящиеся между `repeat` и `until` инструкции тела цикла.
2. Затем вычисляется значение выражения *условие*. Если условие ложно (значение выражения *условие* равно `False`), то инструкции тела цикла выполняются еще раз.
3. Если условие истинно (значение выражения *условие* равно `True`), то выполнение цикла прекращается.

Таким образом, инструкции цикла, находящиеся между `repeat` и `until`, выполняются до тех пор, пока *условие* ложно (значение выражения *условие* равно `False`).

Алгоритм, соответствующий инструкции `repeat`, представлен на рис. 2.16.

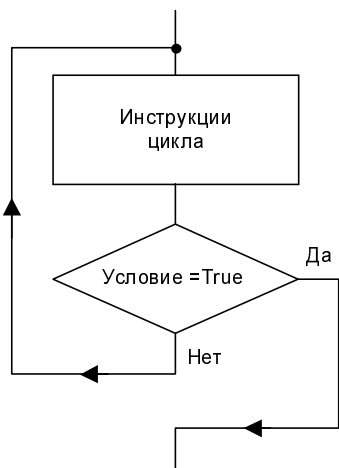


Рис. 2.16. Алгоритм, соответствующий инструкции `repeat`

Внимание!

Инструкции цикла, находящиеся между `repeat` и `until`, выполняются, как минимум, один раз. Для того чтобы цикл завершился, необходимо, чтобы инструкции цикла, располагающиеся между `repeat` и `until`, изменяли значения переменных, входящих в выражение *условие*.

В качестве примера использования инструкции `repeat` рассмотрим программу, которая проверяет, является ли введенное пользователем число простым (как известно, число называется простым, если оно делится только на единицу и само на себя). Например, число 21 — обычное (делится на 3), а число 17 — простое (делится только на 1 и на 17).

Проверить, является ли число n простым, можно делением числа n на два, на три и т. д. до n и проверкой остатка после каждого деления. Если после

очередного деления остаток равен нулю, то это означает, что найдено число, на которое n делится без остатка. Сравнив n и число, на которое n разделилось без остатка, можно определить, является ли n простым числом.

Форма приложения **Простое число** изображена на рис. 2.17, программа приведена в листинге 2.6.

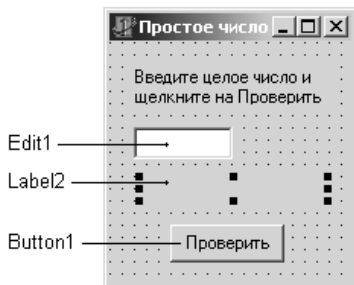


Рис. 2.17. Форма приложения Простое число

Листинг 2.6. Простое число

```
unit simple_;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton; // кнопка Проверить
    Label1: TLabel;
    Edit1: TEdit;     // поле ввода числа
    Label2: TLabel;   // поле вывода результата
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
    n: integer; // проверяемое число
    d: integer; // делитель
    r: integer; // остаток от деления n на d
begin
    n:=StrToInt(Edit1.text);
    d := 2;           // сначала будем делить на два
    repeat
        r := n mod d;
        if r <> 0 // n не разделилось нацело на d
            then d := d + 1;
    until r = 0; // найдено число, на которое n разделилось без остатка
    label2.caption:=Edit1.text;
    if d = n
        then label2.caption:=label2.caption + ' – простое число.'
        else label2.caption:=label2.caption + ' – обычное число.';
end;

end.
```

Инструкция *GOTO*

Инструкции *if* и *case* используются для перехода к последовательности инструкций программы в зависимости от некоторого условия. Поэтому их иногда называют инструкциями *условного перехода*. Помимо этих инструкций управления ходом выполнения программы существует еще одна — инструкция *безусловного перехода* *goto*.

В общем виде инструкция *goto* записывается следующим образом:

```
goto Метка
```

где *Метка* — это идентификатор, находящийся перед инструкцией, которая должна быть выполнена после инструкции *goto*.

Метка, используемая в инструкции *goto*, должна быть описана в разделе меток программы, который начинается словом *label* и располагается перед разделом описания переменных.

В тексте программы метка ставится перед инструкцией, которая должна быть выполнена после инструкции *goto*. Сразу после метки ставится двоеточие.

В листинге 2.7 приведен вариант процедуры проверки простого числа, в которой инструкция *goto* используется для перехода к последней инструкции процедуры в том случае, если пользователь введет неверные данные.

Листинг 2.7. Простое число (использование инструкции goto)

```

procedure TForm1.Button1Click(Sender: TObject);
  label // раздел объявления меток
  bye;

var
  n: integer; // проверяемое число
  d: integer; // делитель
  r: integer; // остаток от деления n на d

begin
  n:=StrToInt(Edit1.text);
  if n <= 0 then
    begin
      MessageDlg('Число должно быть больше нуля.',
        mtError, [mbOk], 0);
      Edit1.text:= '';
      goto bye;
    end;

  // введено положительное число
  d:= 2; // сначала будем делить на два
  repeat
    r:= n mod d;
    if r <> 0 // n не разделилось нацело на d
      then d:= d + 1;
  until r = 0;

  label2.caption:=Edit1.text;
  if d = n
    then label2.caption:=label2.caption
      + ' - простое число.'
    else label2.caption:=label2.caption
      + ' - обычное число.';

  bye:
end;

```

В литературе по программированию можно встретить суждения о недопустимости применения инструкции `goto`, поскольку она приводит к запутанности программ. Однако с категоричностью таких утверждений согласиться нельзя. В некоторых случаях применение инструкции `goto` вполне оправдано. Приведенный пример, где инструкция `goto` используется для аварийного завершения процедуры, относится именно к таким случаям.

ГЛАВА 3



Символы и строки

Компьютер может обрабатывать не только числовую информацию, но и символьную. Object Pascal оперирует с символьной информацией, которая может быть представлена как отдельными символами, так и строками (последовательностью символов).

Символы

Для хранения и обработки символов используются переменные типа `AnsiChar` и `WideChar`. Тип `AnsiChar` представляет собой набор ANSI-символов, в котором каждый символ кодируется восьмиразрядным двоичным числом (байтом). Тип `WideChar` представляет собой набор символов в кодировке Unicode, в которой каждый символ кодируется двумя байтами.

Значением переменной символьного типа может быть любой отображаемый символ:

- ☐ буква русского или латинского алфавитов;
- ☐ цифра;
- ☐ знак препинания;
- ☐ специальный символ, например, "новая строка".

Примечание

Для обеспечения совместимости с предыдущими версиями поддерживается тип `Char`, который эквивалентен `AnsiChar`.

Переменная символьного типа должна быть объявлена в разделе объявления переменных. Инструкция объявления символьной переменной в общем виде выглядит так:

Имя: `char;`

где:

Имя — имя переменной символьного типа;

`char` — ключевое слово обозначения символьного типа.

Примеры:

```
otv: char;
```

```
ch: char;
```

Как и любая переменная программы, переменная типа `char` может получить значение в результате выполнения инструкции присваивания. Если переменная типа `char` получает значение в результате выполнения операции присваивания, то справа от знака `:=` должно стоять выражение типа `char`, например, переменная типа `char` или символьная константа — символ, заключенный в кавычки.

В результате выполнения инструкций

```
c1 := '*';
```

```
c2 := c1;
```

переменная `c1` получает значение присваиванием значения константы, а переменная `c2` — присваиванием значения переменной `c1` (предполагается, что переменные `c1` и `c2` являются переменными символьного типа).

Переменную типа `char` можно сравнить с другой переменной типа `char` или с символьной константой. Сравнение основано на том, что каждому символу поставлено в соответствие число (см. приложение 2 *"Кодировка символов в Windows"*), причем символу '0' соответствует число меньшее, чем символу '9', символу 'A' — меньшее, чем 'B', символу 'Z' — меньшее, чем 'a'. Таким образом, можно записать:

```
'0' < '1' < ... < '9' < ... < 'A' < 'B' < ... < 'Z' < 'a' < 'b' < ... < 'z'
```

Символам русского алфавита соответствуют числа большие, чем символам латинского алфавита, при этом справедливо следующее:

```
'А' < 'Б' < 'В' < ... < 'Ю' < 'Я' < 'а' < 'б' < 'в' < ... < 'э' < 'ю' < 'я'.
```

В тексте программы вместо символа можно указать его код, поставив перед числом оператор `#`. Например, вместо константы 'Б' можно записать `#193`. Такой способ записи, как правило, используют для записи служебных символов или символов, которые во время набора программы нельзя ввести с клавиатуры. К примеру, часто используемый при записи сообщений символ "новая строка" записывается так: `#13`.

В программах обработки символьной информации часто используют функции `Chr` и `Ord`. Значением функции `Chr` является символ, код которого указан в качестве параметра. Например, в результате выполнения инструкции

`c:=Chr(32)` переменной `c` будет присвоено значение пробел. Функция `Ord` позволяет определить код символа, который передается ей в качестве параметра. Например, в результате выполнения инструкции `k:=Ord('*')` переменная `k` будет содержать число 42 — код символа `*`.

Программа, текст которой приведен в листинге 3.1, выводит таблицу кодировки букв русского алфавита. Вид окна программы представлен на рис. 3.1. Основную работу выполняет процедура обработки события `OnActivate`, которая формирует и выводит в поле метки (`Label1`) таблицу. Событие `OnActivate` происходит при активизации формы приложения и, поэтому, процедура `TForm1.FormActivate` выполняется автоматически, сразу после появления формы на экране.

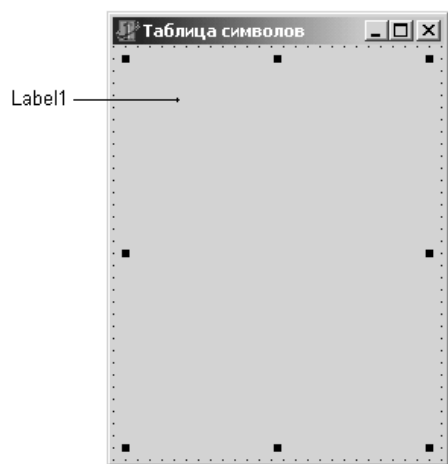


Рис. 3.1. Форма приложения во время разработки

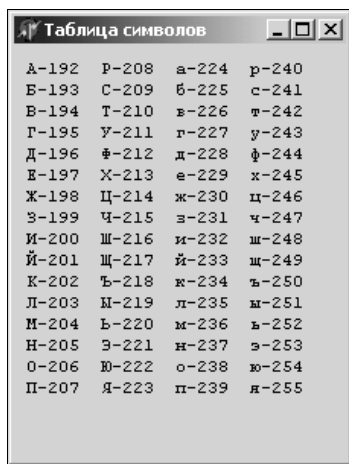


Рис. 3.2. Форма приложения во время работы

Листинг 3.1. Таблица символов

```
unit tablsim_;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

```
StdCtrls;
```

```
type
```

```
TForm1 = class(TForm)
```

```
Label1: TLabel;
```

```
procedure FormActivate(Sender: TObject);
```



```

private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormActivate(Sender: TObject);
var
    st:string;    // таблица формируется как строка символов
    dec: byte;    // код символа
    i,j:integer;  // номер строки и колонки таблицы
begin
    st:='';
    dec:=192;
    for i:=0 to 15 do // шестнадцать строк
    begin
        dec:=i + 192;
        for j:=1 to 4 do // четыре колонки
        begin
            st:=st+chr(dec)+'-'+IntToStr(dec)+'  ';
            dec:=dec + 16;
        end;
        st:=st + #13;    // переход к новой строке экрана
    end;
    Label1.caption:=st;
end;

end.

```

Форма рассматриваемого приложения содержит только один компонент — поле метки (Label1). Для того чтобы колонки таблицы имели одинаковую ширину, свойству Label1.Font.Name следует присвоить имя шрифта, у которого все символы имеют одинаковую ширину, например, Courier New Cyr.

Вид окна приложения во время работы приведен на рис. 3.2.

Строки

Строки могут быть представлены следующими типами: `ShortString`, `LongString` и `WideString`. Различаются эти типы предельно допустимой длиной строки, способом выделения памяти для переменных этого типа и кодировкой символов.

Переменной типа `ShortString` память выделяется статически, т. е. до начала выполнения программы, и количество символов такой строки не может превышать 255. Переменным типа `LongString` и `WideString` память выделяется динамически — во время работы программы, и длина таких строк практически не ограничена.

Помимо перечисленных выше типов можно применять универсальный строковый тип `String`. Тип `String` эквивалентен типу `ShortString`.

Переменная строкового типа должна быть объявлена в разделе объявления переменных с использованием утверждения:

```
Имя: String;
```

или

```
Имя: String [длина]
```

где:

Имя — имя переменной строкового типа;

`String` — ключевое слово обозначения строкового типа;

длина — константа целого типа, определяющая максимальную длину строки.

Пример объявления переменных строкового типа:

```
name: string[30];  
buff: string;
```

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

```
stroka: string [255];  
stroka: string;
```

эквивалентны.

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

```
parol:= 'Большой секрет';
```

или

```
parol:= '2001';
```

Следует обратить внимание, что инструкция `parol:=2001;` неверная, т. к. тип константы не соответствует типу переменной. Во время компиляции этой инструкции будет выведено сообщение: `Incompatible types: 'Char' and 'Integer'` (типы `Char` и `Integer` несовместимы).

Используя операции `=`, `<`, `>`, `<=`, `>=` и `<>`, переменную типа `String` можно сравнить с другой переменной типа `String` или со строковой константой. Строки сравниваются посимвольно, начиная с первого символа. Если все символы сравниваемых строк одинаковые, то такие строки считаются равными. Если в одинаковых позициях строк находятся разные символы, большей считается та строка, у которой в этой позиции находится символ с большим кодом. В табл. 3.1 приведены примеры сравнения строк.

Таблица 3.1

Строка 1	Строка 2	Результат сравнения
Иванов	Иванов	Строки равны
васильев	Васильев	Строка 1 больше строки 2
Алексеев	Петров	Строка 1 меньше строки 2
Иванова	Иванов	Строка 1 больше строки 2

Кроме операции сравнения к строковым переменным и константам можно применить операцию сложения, в результате выполнения которой получается новая строка. Например, в результате выполнения инструкций

```
first_name:='Иван';
last_name:='Иванов';
ful_name:=first_name+last_name;
```

переменная `ful_name` получит значение `'Иван Иванов'`.

Операции со строками

В Object Pascal есть несколько полезных при работе со строками функций и процедур. Ниже приведено их краткое описание и примеры использования.

Функция **LENGTH**

Функция `length` возвращает длину строки. У этой функции один параметр — выражение строкового типа. Значением функции `length` (целое число) является количество символов, из которых состоит строка.

Например, в результате выполнения инструкций

```
n:=length('Иванов');  
m:=length('  Невский проспект ');
```

значение переменных *n* и *m* будет равно 6 и 20.

Процедура **DELETE**

Процедура `delete` позволяет удалить часть строки. В общем виде обращение к этой процедуре выглядит так:

```
delete(Строка, р, n)
```

где:

Строка — переменная или константа строкового типа;

р — номер символа, с которого начинается удаляемая подстрока;

n — длина удаляемой подстроки.

Например, в результате выполнения инструкций

```
s:='Город Санкт-Петербург';  
delete(s,7,6);
```

значением переменной *s* будет строка 'Город Петербург'.

Функция **POS**

Функция `pos` позволяет определить положение подстроки в строке. В общем виде обращение к функции выглядит так:

```
pos(Подстрока, Строка);
```

где *Подстрока* — строковая константа или переменная, которую надо найти в строковой константе или переменной *Строка*.

Например, в результате выполнения инструкции

```
р := pos('Пе', 'Санкт-Петербург');
```

значение переменной *р* будет равно 7. Если в строке нет искомой подстроки, то значение функции `pos` будет равно нулю.

Ниже приведена инструкция `while`, в результате выполнения которой удаляются начальные пробелы из строки *st*.

```
while(pos(' ',st) = 1) and(length(st) > 0) do  
  delete(st,1,1);
```

Пробелы удаляет инструкция `delete(st,1,1)`, которая выполняется в цикле до тех пор, пока первым символом строки является пробел (в этом случае

значение `pos(' ', st)` равно единице). Необходимость проверки условия `length(st) > 0` объясняется возможностью того, что введенная строка состоит только из пробелов.

Функция *COPY*

Функция `copy` позволяет выделить фрагмент строки. В общем виде обращение к функции `copy` выглядит так:

`copy(Строка, p, n)`

где:

Строка — выражение строкового типа, содержащее строку, фрагмент которой надо получить;

p — номер первого символа, с которого начинается выделяемая подстрока;

n — длина выделяемой подстроки.

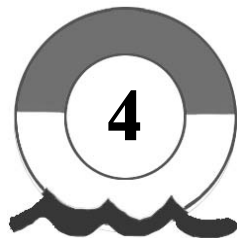
Например, в результате выполнения инструкций

```
st:= 'Инженер Иванов';
```

```
fam:=copy(st, 9, 6);
```

значением переменной `fam` будет строка `'Иванов'`.

ГЛАВА 4



Консольное приложение

Хотя данная книга посвящена программированию в Windows, нельзя обойти вниманием так называемые консольные приложения. Консоль — это монитор и клавиатура, рассматриваемые как единое устройство. Консольное приложение — это программа, предназначенная для работы в операционной системе MS DOS (или в окне DOS), для которой устройством ввода является клавиатура, а устройством вывода — монитор, работающий в режиме отображения символьной информации (буквы, цифры и специальные знаки).

Консольные приложения удобны как иллюстрации при рассмотрении общих вопросов программирования, когда надо сосредоточиться на сути проблемы, а также как небольшие утилиты "для внутреннего потребления".

Перед тем как приступить к созданию консольного приложения, рассмотрим инструкции, обеспечивающие вывод информации на экран и ввод данных с клавиатуры.

Инструкции *WRITE* и *WRITELN*

Инструкция `write` предназначена для вывода на экран монитора сообщений и значений переменных. После слова `write` в скобках задается список переменных, значения которых должно быть выведено. Кроме имен переменных в список можно включить сообщение — текст, заключенный в одиночные кавычки.

Например

```
write(Summa);  
write('Результат вычислений');  
write('Корни уравнения. x1=', x1, ' x2=', x2);
```

После имени переменной через двоеточие можно поместить описание (формат) поля вывода значения переменной.

Для переменной типа `Integer` формат — это целое число, которое задает ширину поля вывода (количество позиций на экране).

Например, инструкция

```
write(d:5);
```

показывает, что для вывода значения переменной `d` используется 5 позиций.

Если значение переменной такое, что его изображение занимает меньше позиций, чем указано в формате, то перед первой цифрой числа будут выведены пробелы так, чтобы общее количество выведенных символов было равно указанному в формате.

Например, если значение переменной `Kol` типа `integer` равно 15, то в результате выполнения инструкции

```
write('Всего изделий:', Kol:5);
```

на экран будет выведено:

```
Всего изделий:   15
```

Для переменных типа `Real` формат представляет собой два целых числа, разделенных двоеточием. Первое число определяет ширину поля вывода, второе — количество цифр, дробной части числа. Если задать только ширину поля, то на экране появится число, представленное в формате с плавающей точкой.

Например, пусть переменные `x1` и `x2` типа `real` имеют значения 13.25 и -0.3401, тогда в результате выполнения инструкции

```
write('x1=',x1:5:2,'   x2=',x2:12)
```

на экран будет выведено:

```
x1=13.25   x2=-3.40100E-01
```

Если ширины поля, указанной в формате, недостаточно для вывода значения переменной, то выводится число в формате с плавающей точкой и десятью цифрами после запятой (все поле вывода в этом случае занимает 17 позиций).

После выполнения инструкции `write` курсор остается в той позиции экрана, в которую он переместился после вывода последнего символа, выведенного этой инструкцией. Следующая инструкция `write` начинает вывод именно с этой позиции. Например, в результате выполнения инструкций

```
x:=-2.73;  
write('Значение перем');  
write('енной:');  
write('x=');  
write(x:8:5);
```

на экран будет выведено:

```
Значение переменной: x=-2.73000
```

Инструкция `writeln` отличается от инструкции `write` только тем, что после вывода сообщения или значений переменных курсор переводится в начало следующей строки. Например, если значением переменной `x1` является число `-3.561`, а значением переменной `x2` — число `10.345`, то результатом выполнения следующих инструкций

```
writeln('Значения корней уравнения:');  
writeln('x1=', x:7:3);  
writeln('x2=', x:7:3);
```

на экран будет выведено:

```
Значения корней уравнения:  
x1=-3.5610  
x2= 10.345
```

Инструкции **READ** и **READLN**

Инструкция `read` предназначена для ввода с клавиатуры значений переменных (исходных данных). В общем виде инструкция выглядит следующим образом:

```
read(Переменная1, Переменная2, ... ПеременнаяN)
```

где *ПеременнаяN* — имя переменной, значение которой должно быть введено с клавиатуры во время выполнения программы.

Приведем примеры записи инструкции `read`:

```
read(a);  
read(Cena, Kol);
```

При выполнении инструкции `read` происходит следующее:

1. Программа приостанавливает свою работу и ждет, пока на клавиатуре будут набраны нужные данные и нажата клавиша `<Enter>`.
2. После нажатия клавиши `<Enter>` введенное значение присваивается переменной, имя которой указано в инструкции.

Например, в результате выполнения инструкции

```
read(Temperat);
```

и ввода с клавиатуры строки `21`, значением переменной `Temperat` будет число `21`.

Одна инструкция `read` позволяет получить значения нескольких переменных. При этом вводимые числа должны быть набраны в одной строке и разделены пробелами. Например, если тип переменных `a`, `b` и `c` — `real`, то в результате выполнения инструкции `read(a,b,c);` и ввода с клавиатуры строки:

```
4.5 23 0.17
```

переменные будут иметь следующие значения: `a = 4,5`; `b = 23,0`; `c = 0,17`.

Если в строке набрано больше чисел, чем задано переменных в инструкции `read`, то оставшаяся часть строки будет обработана следующей инструкцией `read`. Например, в результате выполнения инструкций:

```
read(A,B);  
read(C);
```

и ввода с клавиатуры строки

```
10 25 18
```

переменные получают следующие значения: `A = 10`, `B = 25`. Инструкция `read(C);` присвоит переменной `C` значение 18.

Инструкция `readln` отличается от инструкции `read` тем, что после выделения очередного числа из введенной с клавиатуры строки и присваивания его последней переменной из списка инструкции `readln`, оставшаяся часть строки теряется, и следующая инструкция `read` или `readln` будет требовать нового ввода.

Например, в результате выполнения инструкции:

```
readln(A,B);  
read(C);
```

и вводе с клавиатуры строки

```
10 25 18
```

переменные получают следующие значения: `A = 10`, `B = 25`. После чего программа будет ожидать ввода нового числа, чтобы присвоить его переменной `C`.

Перед каждой инструкцией `read` или `readln` следует располагать инструкцию `write`, для того чтобы подсказать пользователю, какие данные ожидает от него программа. Например, фрагмент программы вычисления стоимости покупки может иметь вид:

```
writeln('Введите исходные данные.');
```

```
write('Цена изделия:');
```

```
readln(Cena);
```

```
write('Количество в партии:');
```

```
readln(Kol);  
write('Скидка:');  
readln(Skidka);
```

Если тип данных, вводимых с клавиатуры, не соответствует или не может быть приведен к типу переменных, имена которых указаны в инструкции `read` (`readln`), то программа аварийно завершает работу (инструкции, следующие за `read` не выполняются), и на экран выводится сообщение об ошибке.

Создание консольного приложения

Создается консольное приложение следующим образом. После запуска Delphi или выбора из меню **File** команды **New Application** необходимо закрыть ненужные окна: окно стартовой формы (**Form1**) и окно модуля приложения (**Unit1.pas**). При закрытии окна модуля приложения Delphi выводит запрос: *Save changes to Unit1.pas?* (Сохранить изменения в Unit1.pas?), на который надо ответить "Нет" (нажать кнопку **No**). В результате на экране остается только главное окно Delphi и окно **Object Inspector**, которое тоже можно закрыть.

После этого нужно из меню **Project** выбрать команду **View Source**. В результате открывается окно **Project1.dpr**, в котором находится сформированный Delphi шаблон главной процедуры приложения.

В окне главной процедуры приложения (рис. 4.1) можно набирать инструкции программы.

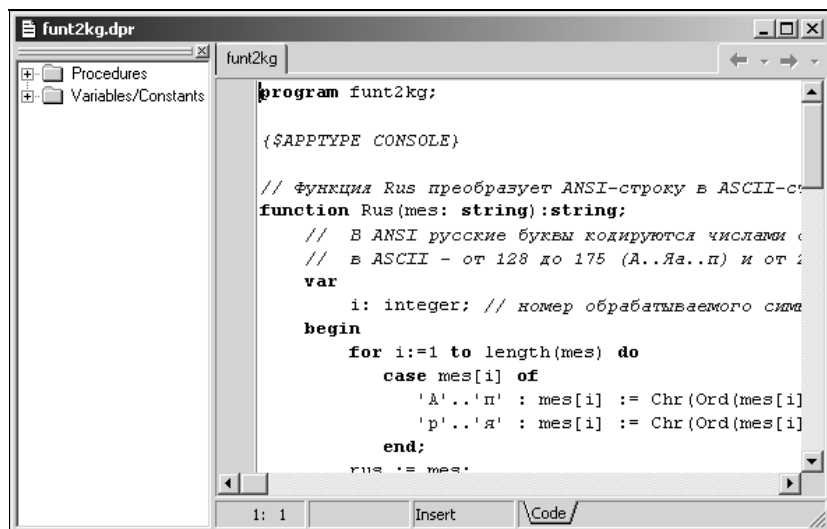


Рис. 4.1. Окно главной процедуры приложения

Примечание

Следует обратить внимание на то, что консольное приложение создается в Windows, а выполняется как программа DOS. В DOS используется кодировка ASCII, а в Windows — ANSI, буквы русского алфавита в которых имеют разные коды. Это приводит к тому, что вместо сообщений на русском языке консольное приложение выводит "абракадабру". Поэтому консольные приложения должны выводить сообщения на английском, что не всегда удобно.

Проблему вывода сообщений на русском языке консольными приложениями можно решить, разработав функцию перекодировки ANSI строки в строку ASCII. Если эту функцию назвать `Rus`, то инструкция вывода сообщения на русском языке может выглядеть, например, так: `writeln( Rus('У лукоморья дуб зеленый'))`.

В листинге 4.1 приведен пример программы, которая запрашивает у пользователя вес в фунтах, пересчитывает его в килограммы и выводит результат на экран. Для вывода сообщений используется функция `Rus`.

Листинг 4.1. Пересчет веса из фунтов в килограммы (консольное приложение)

```
program funt2kg;

{$APPTYPE CONSOLE}

// Функция Rus преобразует ANSI-строку в ASCII-строку
function Rus(mes: string):string;
    // В ANSI русские буквы кодируются числами от 192 до 255,
    // в ASCII — от 128 до 175 (А..Я..п) и от 224 239 (р..я).
    var
        i: integer; // номер обрабатываемого символа
    begin
        for i:=1 to length(mes) do
            case mes[i] of
                'А'..'п' : mes[i] := Chr(Ord(mes[i]) - 64);
                'р'..'я' : mes[i] := Chr(Ord(mes[i]) - 16);
            end;
        rus := mes;
    end;

// основная программа
var
    f:real;    // вес в фунтах
    w:real;    // вес в граммах
    k:integer; // кол-во килограмм
    g:integer; // кол-во грамм
```

```
// w = f*0,4095 = k*1000 + g
```

```
begin
  writeln( Rus('Фунты-килограммы'));
  writeln( Rus('Введите вес в фунтах и нажмите <Enter>'));
  write('-> ');
  readln(f);

  w := f * 409.5; // один фунт — это 409,5 гр.
  if w > 1000 then
    begin
      k:=Trunc(w/1000);
      g:=Round(w - k*1000);
    end
  else
    begin
      k:=0;
      g:=Round(w);
    end;

  write(f:4:2, Rus(' ф. — это '));
  if k >= 1 then write(k, Rus(' кг. '));
  writeln(g, Rus(' гр. '));
  write( Rus('Для завершения нажмите <Enter>'));
  readln;
end.
```

Начинается текст программы строкой `{$APPTYPE CONSOLE}`, которая, хотя и похожа на комментарий, таковым не является, т. к. сразу за открывающей скобкой следует знак денежной единицы. Эта директива предназначена для компилятора. Следуя ее указаниям, компилятор генерирует исполняемую программу как консольное приложение.

Компиляция консольного приложения выполняется обычным образом, т. е. выбором из меню **Project** команды **Compile**.

После успешной компиляции программа может быть запущена выбором из меню **Run** команды **Run**. При запуске консольного приложения на экране появляется стандартное окно DOS-программы. На рис. 4.2 приведен вид DOS-окна, в котором работает консольное приложение, созданное в Delphi.

Следует обратить внимание на то, что если в текст программы консольного приложения не поместить директиву `{$APPTYPE CONSOLE}`, то при запуске программы возникнет ошибка с кодом 105 (Run time error 105).

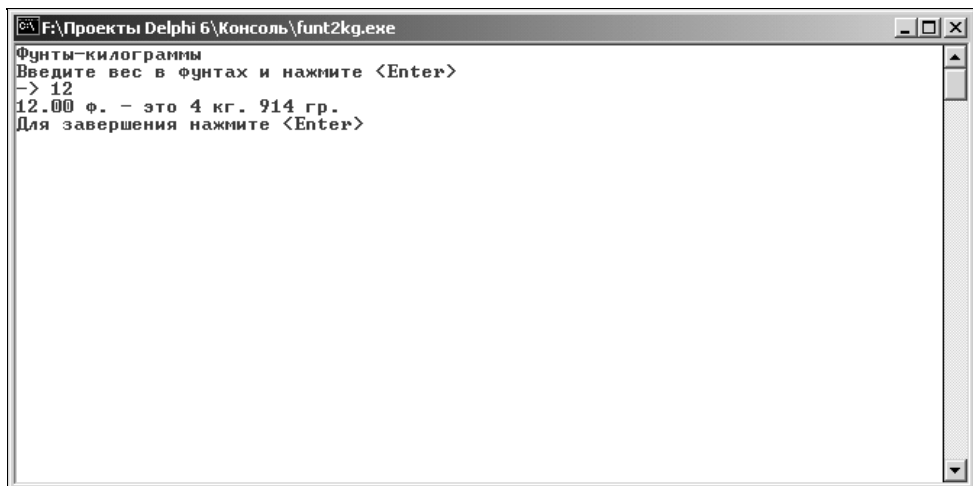


Рис. 4.2. Пример DOS-окна, в котором работает консольное приложение

ГЛАВА 5



Массивы

Массив — это структура данных, представляющая собой набор переменных одинакового типа, имеющих общее имя. Массивы удобно использовать для хранения однородной по своей природе информации, например, таблиц и списков.

Объявление массива

Массив, как и любая переменная программы, перед использованием должен быть объявлен в разделе объявления переменных. В общем виде инструкция объявления массива выглядит следующим образом:

Имя: **array** [*нижний_индекс*..*верхний_индекс*] **of** *тип*

где:

Имя — имя массива;

array — зарезервированное слово языка Object Pascal, обозначающее, что объявляемое имя является именем массивом;

нижний_индекс и *верхний_индекс* — целые константы, определяющие диапазон изменения индекса элементов массива и, неявно, количество элементов (размер) массива;

тип — тип элементов массива.

Примеры объявления массивов:

```
temper:array[1..31] of real;  
koef:array[0..2] of integer;  
name:array[1..30] of string[25];
```

При объявлении массива удобно использовать именованные константы. Именованная константа объявляется в разделе объявления констант, кото-

рый обычно располагают перед разделом объявления переменных. Начинается раздел объявления констант словом `const`. В инструкции объявления именованной константы указывают имя константы и ее значение, которое отделяется от имени символом "равно". Например, чтобы объявить именованную константу `НВ`, значение которой равно 10, в раздел `const` надо записать инструкцию: `НВ=10`. После объявления именованной константы ее можно использовать в программе как обычную числовую или символьную константу. Ниже, в качестве примера, приведено объявление массива названий команд-участниц чемпионата по футболу, в котором используются именованные константы.

```
const
    NT = 18;    // число команд
    SN = 25;    // предельная длина названия команды

var
    team: array[1..NT] of string[SN];
```

Для того чтобы в программе использовать элемент массива, надо указать имя массива и номер элемента (индекс), заключив индекс в квадратные скобки. В качестве индекса можно использовать константу или выражение целого типа, например:

```
team[1] := 'Зенит';
d := koef[1]*koef[1]-4*koef[2]*koef[1];
ShowMessage(name[n+1]);
temper[i] := StrToFloat(Edit1.text);
```

Если массив не является локальным, т. е. объявлен не в процедуре обработки события, а в разделе переменных модуля, то одновременно с объявлением массива можно выполнить его инициализацию, т. е. присвоить начальные значения элементам массива. Инструкция объявления массива с одновременной его инициализацией в общем виде выглядит так:

Имя: **array**[нижний_индекс..верхний_индекс] **of** тип = (список);

где *список* — разделенные запятыми значения элементов массива.

Например:

```
a: array[10] of integer = (0,0,0,0,0,0,0,0,0,0);
Team: array[1..5] of String[10]=
    ('Зенит', 'Динамо', 'Спартак', 'Ротор', 'СКА');
```

Обратите внимание, что количество элементов списка инициализации должно соответствовать размерности массива. Если это будет не так, то компилятор выведет сообщения об ошибке: `Number of elements differs from declaration` (количество элементов не соответствует указанному в объявлении).

При попытке инициализировать локальный массив, компилятор выводит сообщение об ошибке: `Cannot initialize local variables` (локальная переменная не может быть инициализирована). Локальный массив можно инициализировать только во время работы программы, например, так:

```
for i := 1 to 10 do  
    a[i] := 0;
```

Операции с массивами

Типичными операциями при работе с массивами являются:

- ☐ Вывод массива
- ☐ Ввод массива
- ☐ Поиск максимального или минимального элемента массива
- ☐ Поиск заданного элемента массива
- ☐ Сортировка массива

Вывод массива

Под выводом массива понимается вывод на экран монитора (в диалоговое окно) значений элементов массива.

Если в программе необходимо вывести значения всех элементов массива, то для этого удобно использовать инструкцию `for`, при этом переменная-счетчик инструкции `for` может быть использована в качестве индекса элемента массива.

В листинге 5.1 приведен модуль простого приложения, которое выводит в поле метки диалогового окна пронумерованный список названий команд.

Листинг 5.1. Инициализация и вывод массива

```
unit outar_;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
    Dialogs, StdCtrls;  
  
type  
    TForm1 = class(TForm)  
        Button1: TButton;  
        Label1: TLabel;  
        procedure Button1Click(Sender: TObject);  
    end;  
end;
```



```

private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation
{$R *.dfm}

const
    NT = 5;

var
    team: array[1..NT] of string[10] =
        ('Зенит', 'Динамо', 'Ротор', 'Спартак', 'СКА');

procedure TForm1.Button1Click(Sender: TObject);

var
    st:string; // СПИСОК КОМАНД
    i:integer; // ИНДЕКС, НОМЕР ЭЛЕМЕНТА МАССИВА
begin
    // ФОРМИРОВАНИЕ СПИСКА ДЛЯ ОТОБРАЖЕНИЯ В ФОРМЕ
    for i:=1 to NT do      st := st + IntToStr(i)+ ' ' + team[i] + #13;

    // ВЫВОД СПИСКА
    Label1.Caption := st;
end;

end.

```

Ввод массива

Под вводом массива понимается процесс получения от пользователя (или из файла) во время работы программы значений элементов массива.

"Лобовое" решение задачи ввода элементов массива — для каждого элемента массива создать поле ввода. Однако если требуется ввести достаточно большой массив, то такое решение неприемлемо. Представьте форму, например, с десятью полями редактирования!

Очевидно, что последовательность чисел удобно вводить в строку таблицы, где каждое число находится в отдельной ячейке. Ниже рассматриваются два варианта организации ввода массива с использованием компонентов `StringGrid` и `Мемо`.

Использование компонента *StringGrid*

Для ввода массива удобно использовать компонент `StringGrid`. Значок компонента `StringGrid` находится на вкладке **Additional** (рис. 5.1).



Рис. 5.1. Компонент `StringGrid`

Компонент `StringGrid` представляет собой таблицу, ячейки которой содержат строки символов. В табл. 5.1 перечислены некоторые свойства компонента `StringGrid`.

Таблица 5.1. Свойства компонента `StringGrid`

Свойство	Определяет
Name	Имя компонента. Используется в программе для доступа к свойствам компонента
ColCount	Количество колонок таблицы
RowCount	Количество строк таблицы
Cells	Соответствующий таблице двумерный массив. Ячейка таблицы, находящаяся на пересечении столбца номер <code>col</code> и строки номер <code>row</code> определяется элементом <code>cells[col, row]</code>
FixedCols	Количество зафиксированных слева колонок таблицы. Зафиксированные колонки выделяются цветом и при горизонтальной прокрутке таблицы остаются на месте
FixedRows	Количество зафиксированных сверху строк таблицы. Зафиксированные строки выделяются цветом и при вертикальной прокрутке таблицы остаются на месте
Options.goEditing	Признак допустимости редактирования содержимого ячеек таблицы. <code>True</code> — редактирование разрешено, <code>False</code> — запрещено
Options.goTab	Разрешает (<code>True</code>) или запрещает (<code>False</code>) использование клавиши <code><Tab></code> для перемещения курсора в следующую ячейку таблицы

Таблица 5.1 (окончание)

Свойство	Определяет
<code>Options.GoAlwaysShowEditor</code>	Признак нахождения компонента в режиме редактирования. Если значение свойства <code>False</code> , то для того, чтобы в ячейке появился курсор, надо начать набирать текст, нажать клавишу <code><F2></code> или сделать щелчок мышью
<code>DefaultColWidth</code>	Ширину колонок таблицы
<code>DefaultRowHeight</code>	Высоту строк таблицы
<code>GridLineWidth</code>	Ширину линий, ограничивающих ячейки таблицы
<code>Left</code>	Расстояние от левой границы поля таблицы до левой границы формы
<code>Top</code>	Расстояние от верхней границы поля таблицы до верхней границы формы
<code>Height</code>	Высоту поля таблицы
<code>Width</code>	Ширину поля таблицы
<code>Font</code>	Шрифт, используемый для отображения содержимого ячеек таблицы
<code>ParentFont</code>	Признак наследования характеристик шрифта формы

В качестве примера использования компонента `StringGrid` для ввода массива рассмотрим программу, которая вычисляет среднее арифметическое значение элементов массива. Диалоговое окно программы содержит компоненты: `StringGrid` — для ввода массива, `Label1` и `Label2` — для вывода пояснительного текста и результата расчета, `Button1` — для запуска процесса расчета.

Добавляется компонент `StringGrid` в форму точно так же, как и другие компоненты.

После добавления компонента к форме нужно выполнить его настройку в соответствии с табл. 5.2. Значения свойств `Height` и `Width` нужно при помощи мыши установить такими, чтобы размер компонента был равен размеру строки.

Таблица 5.2. Параметры настройки компонента `StringGrid`

Свойство	Значение
<code>Name</code>	<code>StringGrid1</code>
<code>ColCount</code>	5

Таблица 5.2 (окончание)

Свойство	Значение
FixedCols	0
RowCount	1
DefaultRowHeight	24
Height	24
DefaultColWidth	64
Width	328
Options.goEditing	True
Options.AlwaysShowEditing	True
Options.goTabs	True

Диалоговое окно программы после установки свойств компонента `StringGrid1` приведено на рис. 5.2, а текст программы — в листинге 5.2.



Рис. 5.2. Диалоговое окно программы **Ввод и обработка массива**

Листинг 5.2. Ввод и обработка массива целых чисел

```
unit getar_;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
```

```
  Dialogs, Grids, StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    Label1: TLabel;
```

```

StringGrid1: TStringGrid;
Button1: TButton;
Label2: TLabel;
procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
    a : array[1..5] of integer; // массив
    summ: integer;              // сумма элементов
    sr: real;                   // среднее арифметическое
    i: integer;                 // индекс
begin
    // ввод массива
    // считаем, что если ячейка пустая, то соответствующий
    // ей элемент массива равен нулю
    for i:= 1 to 5 do
        if Length(StringGrid1.Cells[i-1,0]) <> 0
            then a[i] := StrToInt(StringGrid1.Cells[i-1,0])
            else a[i] := 0;

    // обработка массива
    summ := 0;
    for i :=1 to 5 do
        summ := summ + a[i];
    sr := summ / 5;

    // вывод результата
    Label2.Caption :=
        'Сумма элементов: ' + IntToStr(summ) + #13+
        'Среднее арифметическое: ' + FloatToStr(sr);
end;

end.

```

После пробных запусков программы возникает желание внести изменения в процесс ввода массива. Например, было бы неплохо, чтобы курсор автома-

тически переходил в следующую ячейку таблицы, например, в результате нажатия клавиши <Enter>. Сделать это можно при помощи процедуры обработки события `OnKeyPress`. На эту же процедуру можно возложить задачу фильтрации вводимых в ячейку таблицы данных. В нашем случае надо разрешить ввод в ячейку только цифр.

Текст процедуры обработки события `OnKeyPress` приведен в листинге 5.3. Следует обратить внимание на свойство `Col`, которое во время работы программы содержит номер колонки таблицы, в которой находится курсор. Это свойство можно также использовать для перемещения курсора в нужную ячейку таблицы. Однако нужно учитывать, что колонки таблицы, впрочем, как и строки, нумеруются с нуля.

Листинг 5.3. Процедура обработки события `OnKeyPress`

```
procedure TForm1.StringGrid1KeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9' : ; // цифры и клавиша <Back Space>
        #13:        // клавиша <Enter>
            if StringGrid1.Col < StringGrid1.ColCount - 1
                then StringGrid1.Col := StringGrid1.Col + 1;
            else    key := Chr(0); // остальные символы запрещены
    end;
end;
```

Если нужно ввести массив дробных чисел (`a: array[1..5] of real`), то процедура обработки события `OnKeyPress` несколько усложнится, т. к. помимо цифр допустимым символом является символ-разделитель (запятая или точка, какой — зависит от настройки Windows) и минус. С целью обеспечения некоторой дружелюбности программы к пользователю можно применить трюк: подменить вводимый пользователем неверный разделитель верным. Определить, какой символ-разделитель допустим в текущей настройке Windows, можно обратившись к глобальной переменной `DecimalSeparator`.

В листинге 5.4 приведен текст модуля приложения ввода и обработки массива дробных чисел. Процедура обработки события `OnKeyPress` обеспечивает ввод в ячейку таблицы только допустимых при записи дробного числа символов.

Листинг 5.4. Ввод и обработка массива дробных чисел

```
unit getar_1;

interface
```

uses

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

type

```
TForm1 = class(TForm)
    Label1: TLabel;
    StringGrid1: TStringGrid;
    Button1: TButton;
    Label2: TLabel;
procedure Button1Click(Sender: TObject);
procedure StringGrid1KeyPress(Sender: TObject; var Key: Char);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

Form1: TForm1;

implementation

{ \$R *.dfm }

procedure TForm1.Button1Click(Sender: TObject);

var

```
  a : array[1..5] of real;      // массив
  summ: real;                  // сумма элементов
  sr:  real;                    // среднее арифметическое
  i: integer;                   // индекс
```

begin

```
  // ввод массива
  // считаем, что если ячейка пустая, то соответствующий
  // ей элемент массива равен нулю
```

for i:= 1 **to** 5 **do**

```
  if Length(StringGrid1.Cells[i-1,0]) <> 0
    then a[i] := StrToFloat(StringGrid1.Cells[i-1,0])
    else a[i] := 0;
```

```
  // обработка массива
```

summ := 0;

for i :=1 **to** 5 **do**

summ := summ + a[i];

sr := summ / 5;

```

// вывод результата
Label2.Caption :=
    'Сумма элементов: ' + FloatToStr(summ) + #13+
    'Среднее арифметическое: ' + FloatToStr(sr);
end;

// Функция обеспечивает ввод в ячейку только допустимых символов
procedure TForm1.StringGrid1KeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9' : ; // цифры и <Back Space>

        #13: // клавиша <Enter>
            if StringGrid1.Col < StringGrid1.ColCount - 1
                then StringGrid1.Col := StringGrid1.Col + 1;

        '.', ',', ': // разделитель целой и дробной частей числа
            begin
                if Key <> DecimalSeparator then
                    Key := DecimalSeparator; // заменим разделитель
                                                // на допустимый
                if Pos(StringGrid1.Cells[StringGrid1.Col, 0],
                    DecimalSeparator) <> 0
                    then Key := Chr(0); // запрет ввода второго
                                                // разделителя
            end;

        '-': // минус можно ввести только первым символом,
            // т. е. когда ячейка пустая
            if Length(StringGrid1.Cells[StringGrid1.Col, 0]) <> 0
                then Key := Chr(0);

        else // остальные символы запрещены
            key := Chr(0);
    end;
end;
end.

```

Использование компонента *Мемо*

В некоторых случаях для ввода массива можно использовать компонент Мемо. Компонент Мемо позволяет вводить текст, состоящий из достаточно большого количества строк, поэтому его удобно использовать для ввода символьного массива. Компонент Мемо добавляется в форму обычным образом. Значок компонента находится на вкладке **Standard** (рис. 5.3).



Рис. 5.3. Компонент Memo

В табл. 5.3 перечислены некоторые свойства компонента Memo.

Таблица 5.3. Свойства компонента Memo

Свойство	Определяет
Name	Имя компонента. Используется в программе для доступа к свойствам компонента
Text	Текст, находящийся в поле Memo. Рассматривается как единое целое
Lines	Текст, находящийся в поле Memo. Рассматривается как совокупность строк. Доступ к строке осуществляется по номеру
Lines.Count	Количество строк текста в поле Memo
Left	Расстояние от левой границы поля до левой границы формы
Top	Расстояние от верхней границы поля до верхней границы формы
Height	Высоту поля
Width	Ширину поля
Font	Шрифт, используемый для отображения вводимого текста
ParentFont	Признак наследования свойств шрифта родительской формы

При использовании компонента Memo для ввода массива значение каждого элемента массива следует вводить в отдельной строке и после ввода каждого элемента массива нажимать клавишу <Enter>.

Получить доступ к находящейся в поле Memo строке текста можно при помощи свойства Lines, указав в квадратных скобках номер нужной строки (строки нумеруются с 0).

Следующая программа, текст которой приведен в листинге 5.5, демонстрирует использование компонента Memo для ввода символьного массива.

Основной цикл процедуры ввода символьного массива из компонента Memo может выглядеть так:

```
for i:=1 to SIZE do
  a[i] := Memo1.Lines[i];
```

где:

SIZE — именованная константа, определяющая размер массива;

a — массив;

Memo1 — имя Мемо-компонента;

Lines — свойство компонента Мемо, представляющее собой массив, каждый элемент которого содержит одну строку находящегося в поле Мемо текста.

Форма программы приведена на рис. 5.4. Помимо поля Мемо она содержит командную кнопку (Button1), при щелчке на которой выполняется ввод значений элементов массива из поля Мемо.

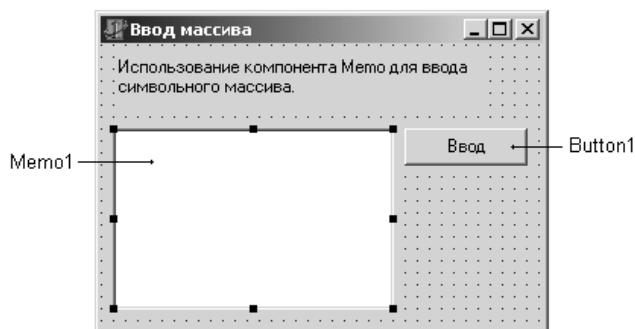


Рис. 5.4. Диалоговое окно приложения **Ввод массива**

Листинг 5.5. Ввод массива строк из компонента Мемо

```
unit fr_memo_;

interface

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

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);

const
    SIZE=5;  // размер массива

var
    a:array[1..SIZE]of string[30]; // массив
    n: integer; // количество строк, введенных в поле Мемо
    i:integer;  // индекс элемента массива
    st:string;

begin
    n:=Memo1.Lines.Count;
    if n = 0 then
        begin
            ShowMessage('Исходные данные не введены!');
            Exit;  // выход из процедуры обработки события
        end;

    // в поле Мемо есть текст
    if n > SIZE then
        begin
            ShowMessage('Количество строк превышает размер массива.');
            n:=SIZE; // будем вводить только первые SIZE строк
        end;

    for i:=1 to n do
        a[i]:=Form1.Memo1.Lines[i-1]; // строки Мемо пронумерованы с нуля

    // вывод массива в окно сообщения
    if n > 0 then
        begin
            st:='Введенный массив:'+#13;
            for i:=1 to n do
                st:=st+IntToStr(i)+'  '+ a[i]+#13;
            ShowMessage(st);
        end;

end;
end.

```

Основную работу выполняет процедура TForm1.Button1Click, которая сначала проверяет, есть ли в поле Memo1 текст. Если текст есть (в этом случае зна-

чение свойства `Lines.Count` больше нуля), то процедура сравнивает количество введенных строк и размер массива. Если это количество превышает размер массива, то программа изменяет значение `n`, тем самым подготавливает ввод только первых `SIZE` строк.

На рис. 5.5 приведен вид диалогового окна приложения **Ввод массива**. После щелчка командной кнопки **Ввод** появляется окно (рис. 5.6), которое содержит значения элементов массива, полученные из Мемо-поля.

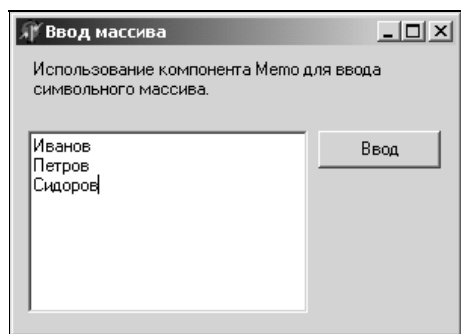


Рис. 5.5. Окно приложения **Ввод массива**

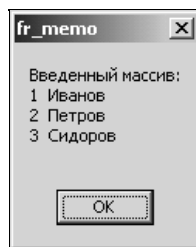


Рис. 5.6. Массив, введенный из Мемо-поля

Поиск минимального (максимального) элемента массива

Задачу поиска минимального элемента массива рассмотрим на примере массива целых чисел.

Алгоритм поиска минимального (максимального) элемента массива довольно очевиден: сначала делается предположение, что первый элемент массива является минимальным (максимальным), затем остальные элементы массива последовательно сравниваются с этим элементом. Если во время очередной проверки обнаруживается, что проверяемый элемент меньше (больше) принятого за минимальный (максимальный), то этот элемент становится минимальным (максимальным) и продолжается проверка оставшихся элементов.

Диалоговое окно приложения поиска минимального элемента массива содержит соответствующим образом настроенный компонент `StringGrid1`, который применяется для ввода элементов массива, два поля меток (`Label1` и `Label2`), использующиеся для вывода информационного сообщения и результата работы программы, и командную кнопку (`Button1`), при щелчке на которой выполняется поиск минимального элемента массива. В табл. 5.4 приведены значения свойств компонента `StringGrid1`.

Таблица 5.4. Свойства компонента *StringGrid1*

Свойство	Значение
Name	StringGrid1
ColCount	5
FixedCols	0
RowCount	1
DefaultRowHeight	24
Height	24
DefaultColWidth	64
Width	328
Options.goEditing	True
Options.AlwaysShowEditing	True
Options.goTabs	True

В листинге 5.6 приведена процедура обработки события `OnClick` для командной кнопки `Button1`, которая вводит массив, выполняет поиск минимального элемента и выводит результат — номер и значение минимального элемента массива.

Листинг 5.6. Поиск минимального элемента массива

```
unit lookmin_;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

```
  StdCtrls, Grids;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    Label1: TLabel;
```

```
    Button1: TButton;
```

```
    Label2: TLabel;
```

```
    StringGrid1: TStringGrid;
```

```
    procedure Button1Click(Sender: TObject);
```

```
  private
```

```
    { Private declarations }
```

```

public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
const
    SIZE=5;
var
    a:array[1..SIZE]of integer; // массив целых
    min:integer; // номер минимального эл-та массива
    i:integer;    // номер эл-та сравниваемого с минимальным
begin
    // ввод массива
    for i:=1 to SIZE do
        a[i]:=StrToInt(StringGrid1.Cells[i-1,0]);

    // поиск минимального элемента
    min:=1; // пусть первый элемент минимальный
    for i:=2 to SIZE do
        if a[i]< a[min] then min:=i;

    // вывод результата
    label2.caption:='Минимальный элемент массива:'+IntToStr(a[min])
        +#13+'Номер элемента:'+ IntToStr(min);
end;
end.

```

На рис. 5.7 приведен вид диалогового окна приложения после щелчка на кнопке **Поиск**.

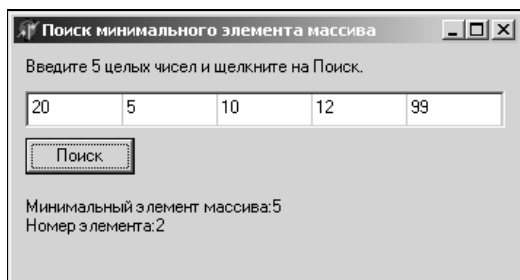


Рис. 5.7. Окно приложения **Поиск минимального элемента массива**

Поиск в массиве заданного элемента

При решении многих задач возникает необходимость определить, содержит ли массив определенную информацию или нет. Например, проверить, есть ли в списке студентов фамилия Петров. Задачи такого типа называются *поиском в массиве*.

Для организации поиска в массиве могут быть использованы различные алгоритмы. Наиболее простой — это алгоритм *простого перебора*. Поиск осуществляется последовательным сравнением элементов массива с образцом до тех пор, пока не будет найден элемент, равный образцу, или не будут проверены все элементы. Алгоритм простого перебора применяется, если элементы массива не упорядочены.

Алгоритм простого перебора

Ниже приведен текст программы поиска в массиве целых чисел. Перебор элементов массива осуществляется инструкцией `repeat`, в теле которой инструкция `if` сравнивает текущий элемент массива с образцом и присваивает переменной `found` значение `true`, если текущий элемент и образец равны.

Цикл завершается, если в массиве обнаружен элемент, равный образцу (в этом случае значение переменной `found` равно `true`), или если проверены все элементы массива. По завершении цикла по значению переменной `found` можно определить, успешен поиск или нет.

Вид диалогового окна программы **Поиск в массиве** приведен на рис. 5.8.

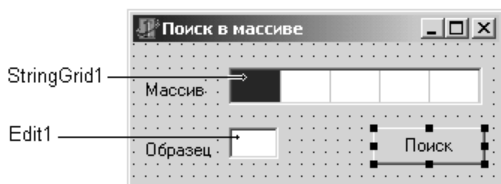


Рис. 5.8. Диалоговое окно программы **Поиск в массиве**

Щелчок командной кнопки **Поиск** (`Button1`) запускает процедуру `TForm1.Button1Click` (ее текст приведен в листинге 5.7), которая из компонента `StringGrid1` вводит массив, а из поля редактирования `Edit2` — число (образец). Затем выполняется проверка: содержит ли массив введенное число. После завершения проверки процедура `ShowMessage` выводит сообщение о результате поиска.

Листинг 5.7. Поиск в массиве

```
unit s_found_;
```

```
interface
```

uses

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Grids;
```

type

```
TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Edit2: TEdit;
    StringGrid1: TStringGrid;
    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
{ поиск в массиве перебором }
```

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

const

```
SIZE=5;
```

var

```
a: array[1..SIZE] of integer; // массив
obr: integer;                // образец для поиска
found: boolean;               // TRUE – совпадение образца с элементом
                               // массива
i: integer;                   // индекс элемента массива
```

begin

```
// ввод массива
```

```
for i:=1 to SIZE do
```

```
    a[i] := StrToInt(StringGrid1.Cells[i-1,0]);
```

```
// ввод образца для поиска
```

```
obr := StrToInt(edit2.text);
```

```
// поиск
```

```
found := FALSE; // пусть нужного эл-та в массиве нет
```

```
i := 1;
```



```
repeat
    if a[i] = obr
    then found := TRUE
    else i := i+1;
until (i > SIZE) or (found = TRUE);

if found
then ShowMessage('Совпадение с элементом номер '
                  +IntToStr(i)+#13+'Поиск успешен.')
else ShowMessage('Совпадений с образцом нет.');
```

end;

end.

Очевидно, что чем больше элементов в массиве и чем дальше расположен нужный элемент от начала массива, тем дольше программа будет искать необходимый элемент.

Так как операции сравнения применимы как к числам, так и к строкам, то данный алгоритм может использоваться для поиска как в числовых, так и в строковых массивах.

Метод бинарного поиска

На практике довольно часто производится поиск в массиве, элементы которого упорядочены по некоторому критерию (такие массивы называются упорядоченными). Например, массив фамилий, как правило, упорядочен по алфавиту, массив данных о погоде — по датам наблюдений. В случае, если массив упорядочен, то применяют другие, более эффективные по сравнению с методом простого перебора алгоритмы, один из которых — *метод бинарного поиска*.

Пусть есть упорядоченный по возрастанию массив целых чисел. Нужно определить, содержит ли этот массив некоторое число (образец).

Метод (алгоритм) бинарного поиска реализуется следующим образом:

1. Сначала образец сравнивается со средним (по номеру) элементом массива (рис. 5.9, а).
 - Если образец равен среднему элементу, то задача решена.
 - Если образец больше среднего элемента, то это значит, что искомый элемент расположен ниже среднего элемента (между элементами с номерами `sred+1` и `niz`) и за новое значение `verh` принимается `sred+1`, а значение `niz` не меняется (рис. 5.9, б).
 - Если образец меньше среднего элемента, то это значит, что искомый элемент расположен выше среднего элемента (между элементами с номерами `verh` и `sred-1`) и за новое значение `niz` принимается `sred-1`, а значение `verh` не меняется (рис. 5.9, в).

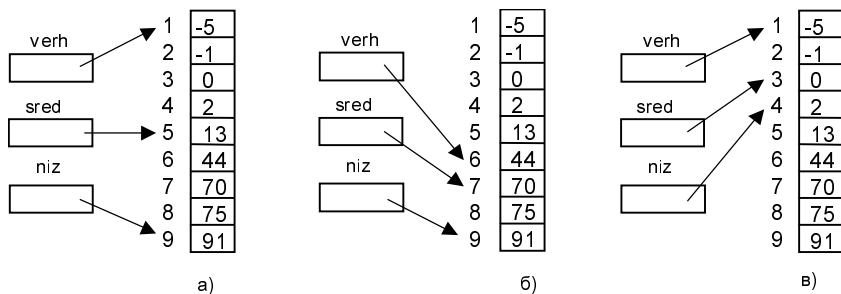


Рис. 5.9. Выбор среднего элемента массива при бинарном поиске

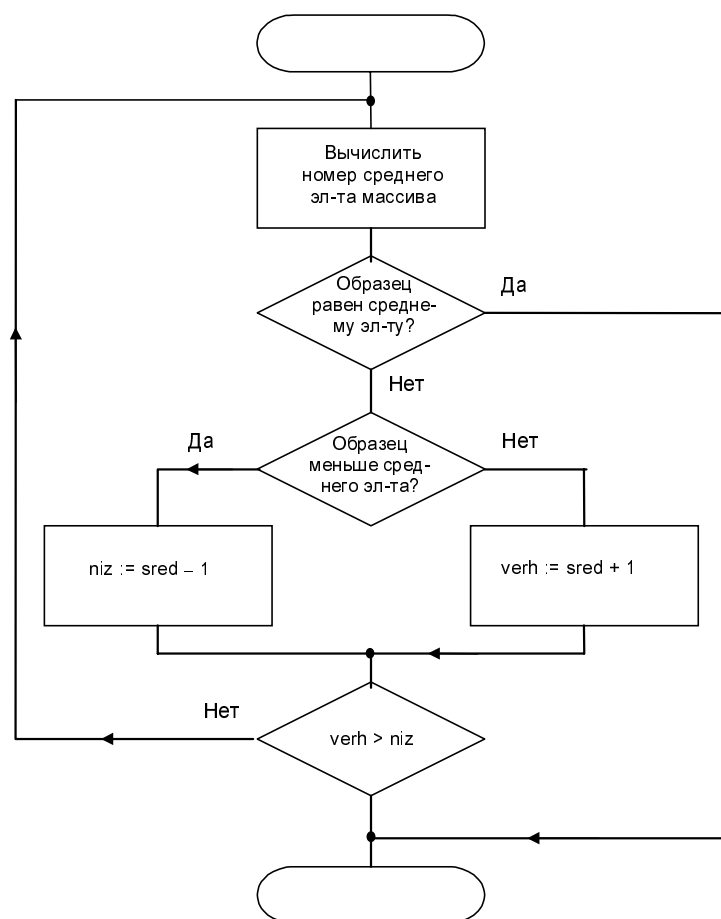


Рис. 5.10. Алгоритм бинарного поиска в упорядоченном по возрастанию массиве

- После того как определена часть массива, в которой может находиться искомый элемент, по формуле $(\text{niz} - \text{verh}) / 2 + \text{verh}$ вычисляется новое значение `sred` и поиск продолжается.

Алгоритм бинарного поиска, блок-схема которого представлена на рис. 5.10, заканчивает свою работу, если искомый элемент найден, или если перед выполнением очередного цикла поиска обнаруживается, что значение `verh` больше, чем `niz`.

Вид диалогового окна программы **Бинарный поиск в массиве** приведен на рис. 5.11. Поле метки `Label3` используется для вывода результатов поиска и протокола поиска. Протокол поиска выводится, если установлен флажок **выводить протокол**. Протокол содержит значения переменных `verh`, `niz`, `sred`. Эта информация, выводимая во время поиска, полезна для понимания сути алгоритма.

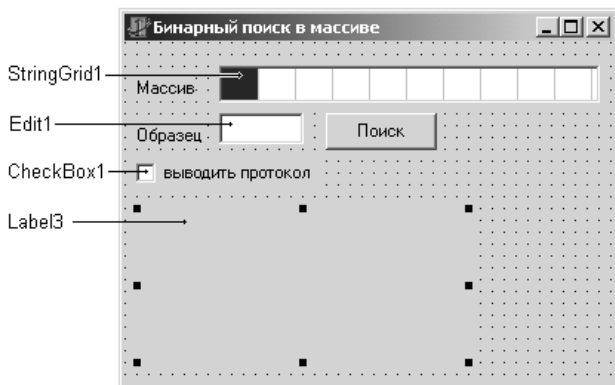


Рис. 5.11. Диалоговое окно программы **Бинарный поиск в массиве**

В форме приложения появился новый компонент, который до этого момента в программах не использовался, — *флажок* (компонент `CheckBox`). Значок компонента `CheckBox` находится на вкладке **Standard** (рис. 5.12). Добавляется к форме он точно так же, как и другие компоненты. Свойства компонента `CheckBox` перечислены в табл. 5.5.

Таблица 5.5. Свойства компонента `CheckBox`

Свойство	Определяет
Name	Имя компонента. Используется в программе для доступа к свойствам компонента
Caption	Текст, поясняющий назначение флажка
Checked	Состояние, внешний вид флажка: если флажок установлен (в квадратике есть "галочка"), то <code>Checked = TRUE</code> ; если флажок сброшен (нет "галочки"), то <code>Checked=FALSE</code>

Таблица 5.5 (окончание)

Свойство	Определяет
State	Состояние флажка. В отличие от свойства Checked, позволяет различать установленное, сброшенное и промежуточное состояния. Состояние флажка определяют константы: cbChecked (установлен); cbGrayed (серый, неопределенное состояние); cbUnChecked (сброшен)
AllowGrayed	Может ли флажок быть в промежуточном состоянии: если AllowGrayed = FALSE, то флажок может быть только установленным или сброшенным; если AllowGrayed = TRUE, то допустимо промежуточное состояние
Left	Расстояние от левой границы флажка до левой границы формы
Top	Расстояние от верхней границы флажка до верхней границы формы
Height	Высоту поля вывода поясняющего текста
Width	Ширину поля вывода поясняющего текста
Font	Шрифт, используемый для отображения поясняющего текста
ParentFont	Признак наследования характеристик шрифта родительской формы

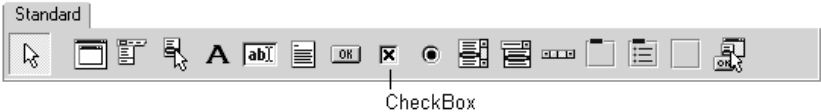


Рис. 5.12. Компонент CheckBox

После того как компонент `CheckBox` будет добавлен к форме, а добавляется он обычным образом, нужно установить значения его свойств в соответствии с табл. 5.6.

Таблица 5.6. Свойства компонента `CheckBox`

Свойство	Значение
Name	CheckBox1
Caption	Выводить протокол
Checked	True

В листинге 5.8 приведен текст процедуры обработки события `OnClick` для командной кнопки **Поиск** (`Button1`). Процедура вводит значения элементов

массива и образец, затем, используя алгоритм бинарного поиска, проверяет, содержит ли массив элемент, равный образцу. Кроме того, переменная *n* (число сравнений с образцом) позволяет оценить эффективность алгоритма бинарного поиска по сравнению с поиском методом простого перебора.

При вычислении номера среднего элемента используется функция *Trunc*, которая округляет к ближайшему целому и преобразует к типу *Integer* выражение, полученное в качестве аргумента. Необходимость использования *Trunc* объясняется тем, что выражение $(\text{низ} - \text{верх}) / 2$ — дробного типа, переменная *sred* — целого, а переменной целого типа присвоить дробное значение нельзя (компилятор выдаст сообщение об ошибке).

Обратите внимание на процедуры обработки события *OnKeyPress* для компонентов *StringGrid1* и *Edit1*. Первая из них обеспечивает перемещение курсора в следующую ячейку таблицы или в поле *Edit1* (из последней ячейки) в результате нажатия клавиши <Enter>, вторая — активизирует командную кнопку **Поиск** так же, в результате нажатия <Enter>.

Листинг 5.8. Бинарный поиск в массиве

```
unit b_found_;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Label3: TLabel;
    CheckBox1: TCheckBox;
    StringGrid1: TStringGrid;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);

    procedure StringGrid1KeyPress(Sender: TObject; var Key: Char);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```

var
    Form1: TForm1;

implementation

{$R *.DFM}

{ Бинарный поиск в массиве }
procedure TForm1.Button1Click(Sender: TObject);
const
    SIZE=10;
var
    a:array[1..SIZE] of integer; { массив }
    obr:integer;                { образец для поиска}

    verh:integer; { верхняя граница поиска }
    niz: integer; { нижняя граница поиска }
    sred:integer; { номер среднего элемента }

    found:boolean;{ TRUE — совпадение образца с элементом массива }
    n:integer;    { число сравнений с образцом }
    i:integer;

begin
    // ввод массива и образца
    for i:=1 to SIZE do
        a[i]:=StrToInt(StringGrid1.Cells[i-1,0]);
    obr := StrToInt(Edit1.text);

    // поиск
    verh:=1;
    niz:=SIZE;
    n:=0;
    found:=FALSE;
    label3.caption:='';

    if CheckBox1.State = cbChecked
        then Label3.caption:='verh'+#9+'niz'+#9'sred'#13;

    // бинарный поиск в массиве
    repeat
        sred:=Trunc((niz-verh)/2)+verh;
        if CheckBox1.Checked
            then Label3.caption:=label3.caption
                +IntToStr(verh) + #9
                +IntToStr(niz) + #9
                +IntToStr(sred) + #13;

```

```

n:=n+1;
if a[sred] = obr
then found:=TRUE
else
  if obr < a[sred]
  then niz:=sred-1
  else verh:=sred+1;
until (verh > niz) or found;

if found
then label3.caption:=label3.caption
  + 'Совпадение с элементом номер '
  + IntToStr(sred) + #13
  + 'Сравнений ' + IntToStr(n)
else label3.caption:=label3.caption
  + 'Образец в массиве не найден.';
end;

// нажатие клавиши в ячейке StringGrid
procedure TForm1.StringGrid1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then // нажата клавиша <Enter>
  if StringGrid1.Col < StringGrid1.ColCount - 1
  then // курсор в следующую ячейку таблицы
    StringGrid1.Col := StringGrid1.Col + 1
  else // курсор в поле Edit1, в поле ввода образца
    Edit1.SetFocus;
end;

// нажатие клавиши в поле Edit1
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 // нажата клавиша <Enter>
  then // сделать активной командную кнопку
    Button1.SetFocus;
end;

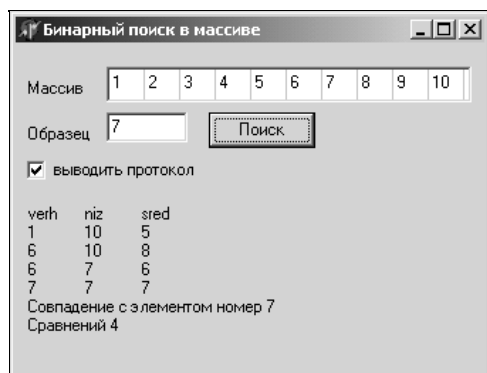
end.

```

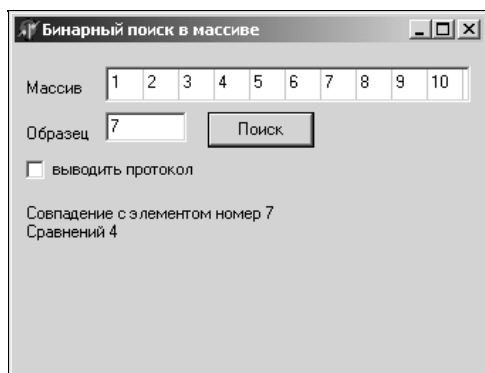
Ниже приведены примеры диалоговых окон программы **Бинарный поиск в массиве** после выполнения поиска (с выводом протокола (рис. 5.13, а) и без протокола (рис. 5.13, б)).

Здесь следует обратить внимание на то, что элемент массива, находящийся на седьмом месте, программа бинарного поиска находит всего за четыре

шага, в то время как программе, реализующей алгоритм простого перебора, потребовалось бы семь шагов.



а)



б)

Рис. 5.13. Два примера работы программы бинарного поиска в массиве

Сортировка массива

Под сортировкой массива подразумевается процесс перестановки элементов массива, целью которого является размещение элементов массива в определенном порядке. Например, если имеется массив целых чисел a , то после выполнения сортировки по возрастанию должно выполняться условие:

$$a[1] \leq a[2] \leq \dots \leq a[\text{SIZE}]$$

где SIZE — верхняя граница индекса массива.

Примечание

Задача сортировки распространена в информационных системах и используется как предварительный этап задачи поиска, т. к. поиск в упорядоченном (отсортированном) массиве проводится намного быстрее, чем в неупорядоченном (см. рассмотренный ранее метод бинарного поиска).

Существует много методов (алгоритмов) сортировки массивов. Рассмотрим два из них:

- ☐ Метод прямого выбора
- ☐ Метод прямого обмена

Сортировка методом прямого выбора

Алгоритм сортировки массива по возрастанию методом прямого выбора может быть представлен так:

1. Просматривая массив от первого элемента, найти минимальный элемент и поместить его на место первого элемента, а первый — на место минимального.
2. Просматривая массив от второго элемента, найти минимальный элемент и поместить его на место второго элемента, а второй — на место минимального.
3. И так далее до предпоследнего элемента.

Ниже представлена программа сортировки массива целых чисел по возрастанию, диалоговое окно которой изображено на рис. 5.14.

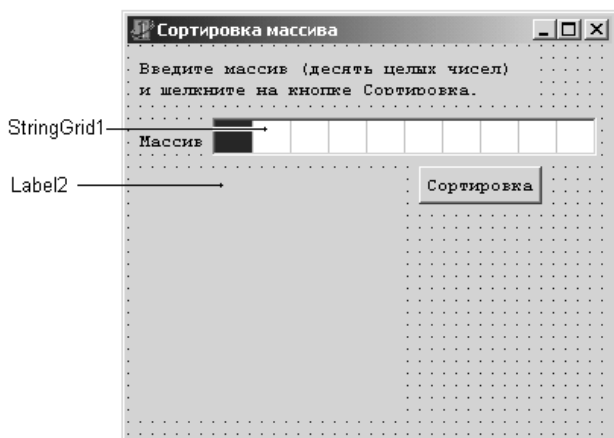


Рис. 5.14. Диалоговое окно программы сортировки массива простым выбором

Процедура сортировки, текст которой приведен в листинге 5.9, запускается нажатием кнопки **Сортировка** (Button1). Значения элементов массива вводятся из ячеек компонента StringGrid1. После выполнения очередного цикла поиска минимального элемента в части массива процедура выводит массив в поле метки (Label2).

Листинг 5.9. Сортировка массива простым выбором

```
procedure TForm1.Button1Click(Sender: TObject);
const
    SIZE=10;
var
    a:array[1..SIZE] of integer;
    min:integer; { номер минимального элемента в части
                  массива от i до верхней границы массива }
    j:integer; { номер эл-та, сравниваемого с минимальным }
    buf:integer; { буфер, используемый при обмене эл-тов массива }
    i,k:integer;
```

```

begin
    // ВВОД МАССИВА
    for i:=1 to SIZE do
        a[i]:=StrToInt(StringGrid1.Cells[i-1,0]);

    label2.caption:='';
    for i:=1 to SIZE-1 do
        begin
            { ПОИСК МИНИМАЛЬНОГО ЭЛ-ТА
              В ЧАСТИ МАССИВА ОТ a[1] ДО a[SIZE]}
            min:=i;
            for j:=i+1 to SIZE do
                if a[j] < a[min]
                    then min:=j;

            { ПОМЕНЯЕМ МЕСТАМИ a[min] И a[i] }
            buf:=a[i];
            a[i]:=a[min];
            a[min]:=buf;

            { ВЫВОД МАССИВА }
            for k:=1 to SIZE do
                Label2.caption:=label2.caption+' '+IntToStr(a[k]);
                Label2.caption:=label2.caption+#13;

        end;
        Label2.caption:=label2.caption+#13+'Массив отсортирован.';
    end;
end;

```

На рис. 5.15 приведено диалоговое окно программы после завершения процесса сортировки.

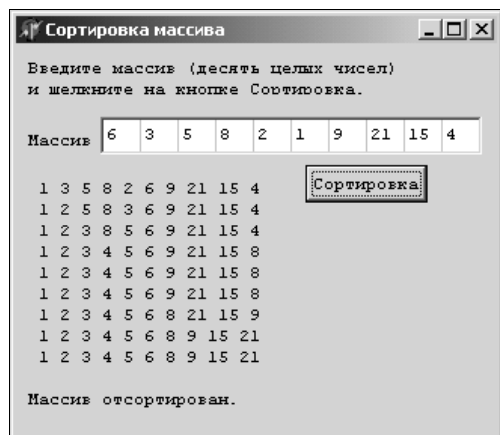


Рис. 5.15. Диалоговое окно программы **Сортировка массива**

Сортировка методом обмена

В основе алгоритма лежит обмен соседних элементов массива. Каждый элемент массива, начиная с первого, сравнивается со следующим, и если он больше следующего, то элементы меняются местами. Таким образом, элементы с меньшим значением продвигаются к началу массива (всплывают), а элементы с большим значением — к концу массива (тонут). Поэтому данный метод сортировки обменом иногда называют методом "пузырька". Этот процесс повторяется столько раз, сколько элементов в массиве минус единица.

На рис. 5.16 цифрой **1** обозначено исходное состояние массива и перестановки на первом проходе, цифрой **2** — состояние после перестановки на первом проходе и перестановки на втором проходе и т. д.

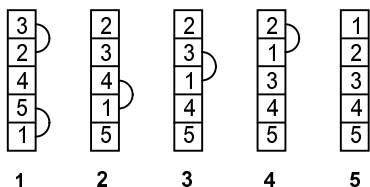


Рис. 5.16. Процесс сортировки массива



Рис. 5.17. Диалоговое окно программы Сортировка методом обмена

На рис. 5.17 приведено диалоговое окно программы сортировки массива методом обмена.

Процедура сортировки, текст которой приведен в листинге 5.10, запускается нажатием кнопки **Сортировка** (Button1). Значения элементов массива вводятся из ячеек компонента StringGrid1. Во время сортировки, после выполнения очередного цикла обменов элементов массива, программа выводит массив в поле метки Label2.

Листинг 5.10. Сортировка массива методом обмена

```
procedure TForm1.Button1Click(Sender: TObject);
const
    SIZE=5;
var
    a:array[1..SIZE] of integer;
```

```
k:integer;           // текущий элемент массива
i:integer;           // индекс для ввода и вывода массива
changed:boolean;     // TRUE, если в текущем цикле были обмены

buf:integer;         // буфер для обмена элементами массива

begin
    // ввод массива
    for i:=1 to SIZE do
        a[i] := StrToInt(StringGrid1.Cells[i-1,0]);

    label2.caption:='';

    // сортировка массива
    repeat
        changed:=FALSE;           // пусть в текущем цикле нет обменов
        for k:=1 to SIZE-1 do
            if a[k] > a[k+1] then
                begin // обменяем k-й и k+1-й элементы
                    buf := a[k];
                    a[k] := a[k+1];
                    a[k+1] := buf;
                    changed := TRUE;
                end;

            // вывод массива
        for i:=1 to SIZE do
            Label2.caption:=label2.caption+' '+IntToStr(a[i]);
            Label2.caption:=label2.caption+#13;

    until not changed; // если не было обменов, значит
                        // массив отсортирован

    Label2.caption:=label2.caption+#13+'Массив отсортирован.';
end;
```

Следует отметить, что максимальное необходимое количество циклов проверки соседних элементов массива равно количеству элементов массива минус один. Вместе с тем возможно, что массив реально будет упорядочен за меньшее число циклов. Например, последовательность чисел 5 1 2 3 4, если ее рассматривать как представление массива, будет упорядочена за один цикл, и выполнение оставшихся трех циклов не будет иметь смысла.

Поэтому в программу введена логическая переменная `changed`, которой перед выполнением очередного цикла присваивается значение `FALSE`. Процесс сортировки (цикл `repeat`) завершается, если после выполнения очередного цикла проверки соседних элементов массива (цикл `for`) ни один элемент массива не был обменен с соседним, и, следовательно, массив уже упорядочен.

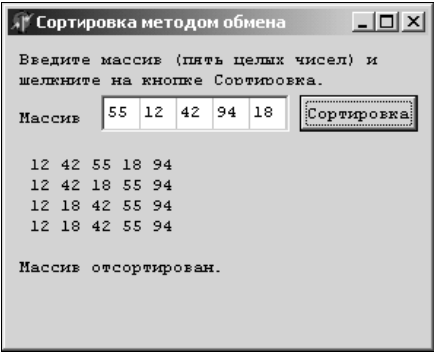


Рис. 5.18. Пример работы программы сортировки массива методом обмена

На рис. 5.18 приведено диалоговое окно программы сортировки массива методом обмена после завершения процесса сортировки.

Многомерные массивы

В повседневной жизни довольно часто приходится иметь дело с информацией, которая представлена в табличной форме. Например, результат деятельности некоторой фирмы, торгующей автомобилями, может быть представлен в виде табл. 5.7.

Таблица 5.7

	Январь	Февраль	Март	...	Ноябрь	Декабрь
BA3 2106						
BA3 2107						
BA3 2108						
BA3 2109						
BA3 2110						
BA3 2111						

Колонки и (или) строки таблицы, как правило, состоят из однородной информации. Поэтому в программе, обрабатывающей табличные данные, имеет смысл использовать массивы для хранения и обработки таблиц. Так, приведенная выше таблица может быть представлена как совокупность одномерных массивов:

```
vaz2106: array [1..12] of integer;  
vaz2107: array [1..12] of integer;  
vaz2108: array [1..12] of integer;  
vaz2109: array [1..12] of integer;
```

```
vaz2110: array [1..12] of integer;  
vaz2111: array [1..12] of integer;
```

Каждый из приведенных массивов может хранить информацию о количестве проданных автомобилей одной марки, причем значение элемента массива отражает количество проданных машин в соответствующем месяце.

Возможно и такое представление таблицы:

```
jan: array [1..6] of integer;  
feb: array [1..6] of integer;  
mar: array [1..6] of integer;  
...  
dec: array [1..6] of integer;
```

В этом случае каждый массив предназначен для хранения информации о количестве проданных за месяц автомобилей, причем значение элемента массива отражает проданное количество автомобилей одной марки.

Если вся таблица содержит однородную информацию, например, только целые числа, то такая таблица может быть представлена как двумерный массив.

В общем виде инструкция объявления двумерного массива выглядит так:

```
Имя: array [НижняяГраница1..ВерхняяГраница1,  
            НижняяГраница2..ВерхняяГраница2] of Тип
```

где:

Имя — имя массива;

array — слово языка Pascal, указывающее, что объявляемый элемент данных является массивом;

НижняяГраница1, *ВерхняяГраница1*, *НижняяГраница2*, *ВерхняяГраница2* — целые константы, определяющие диапазон изменения индексов и, следовательно, число элементов массива;

Тип — тип элементов массива.

Табл. 5.7 может быть представлена в виде двумерного массива следующим образом:

```
itog: array [1..12, 1..6] of integer
```

Количество элементов двумерного массива можно вычислить по формуле:

$(\text{ВГ1}-\text{НГ1}+1) \times (\text{ВГ2}-\text{НГ2}+1)$:

где:

ВГ1 и ВГ2 — верхняя граница первого и второго индексов;

НГ1 и НГ2 — нижняя граница первого и второго индексов.

Таким образом, массив *itog* состоит из 60 элементов типа *integer*.

Для того чтобы использовать элемент массива, нужно указать имя массива и индексы элемента. Первый индекс обычно соответствует номеру строки таблицы, второй — номеру колонки. Так, элемент `itog[2,3]` содержит число проданных в марте (третий месяц) автомобилей марки ВАЗ 2107 (данные о продаже ВАЗ 2107 находятся во второй строке таблицы).

При работе с таблицами (массивами) удобно использовать инструкцию `for`. Например, фрагмент программы, вычисляющий количество проданных за год автомобилей одного наименования, выглядит так:

```
s := 0;
for j := 1 to 12 do
    s := s + itog[2,j];
```

Следующий фрагмент программы вычисляет сумму элементов массива (общее количество автомобилей, проданных за год).

```
s:=0;
for i := 1 to 6 do    // шесть моделей автомобилей
    for j := 1 to 12 do // 12 месяцев
        s := s + itog[i,j];
```

В приведенном фрагменте программы каждый раз, когда внутренний цикл (цикл по `j`) завершается, во внешнем цикле значение `i` увеличивается на единицу, и внутренний цикл выполняется вновь. Таким образом, к текущему значению переменной `s` последовательно прибавляются значения элементов массива `itog`: `itog[1,1]`, `itog[1,2]`, ..., `itog[1,12]`, `itog[2,1]`, `itog[2,2]`, ..., `itog[2,12]` и т. д.

В качестве примера рассмотрим программу, которая обрабатывает результаты спортивных соревнований (летней олимпиады в Сиднее, 2000 г.). Исходные данные представлены в табл. 5.8.

Таблица 5.8. Результаты олимпиады 2000 г. в Сиднее

Страна	Золотых	Серебряных	Бронзовых
Австралия	16	25	17
Беларусь	3	3	11
Великобритания	11	10	7
Германия	14	17	26
Италия	13	8	13
Китай	28	16	15
Корея	8	9	11
Куба	11	11	7

Таблица 5.8 (окончание)

Страна	Золотых	Серебряных	Бронзовых
Нидерланды	12	9	4
Россия	32	28	28
Румыния	11	6	9
США	39	25	33
Франция	13	14	11
Япония	5	8	5

Программа должна вычислить общее количество медалей, завоеванных представителями каждой страны, и соответствующее количество очков (баллов), которое вычисляется по следующему правилу: за каждую золотую медаль команда получает семь очков, за серебряную — шесть очков, за бронзовую — пять очков.

Вид диалогового окна программы приведен на рис. 5.19.

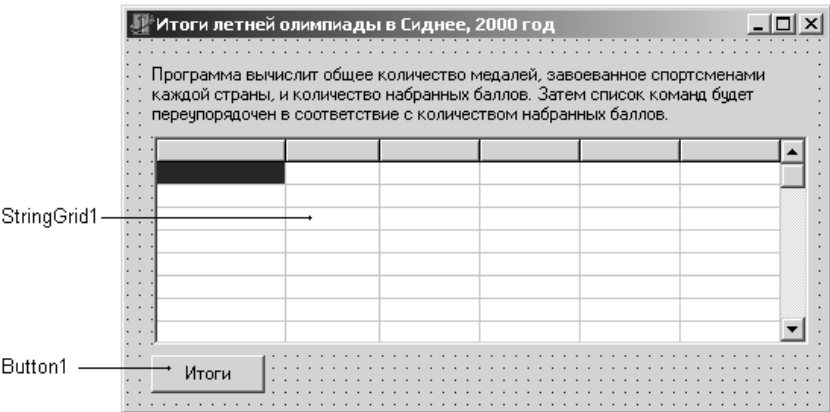


Рис. 5.19. Диалоговое окно программы **Итоги олимпиады**

Для ввода исходных данных и отображения результата используется компонент `StringGrid`, свойства которого приведены в табл. 5.9.

Таблица 5.9. Свойства компонента `StringGrid`

Свойство	Значение
Name	Tabl
ColCount	6

Таблица 5.9 (окончание)

Свойство	Значение
RowCount	14
FixedCols	0
FixedRows	1
Options.goEditing	TRUE
DefaultColWidth	65
DefaultRowHeight	14
GridLineWidth	1

Ячейки первой зафиксированной строки таблицы используются в качестве заголовков колонок таблицы. Во время создания формы приложения нельзя установить значения элементов массива `Cells`, т. к. элементы массива доступны только во время работы программы. Поэтому значения элементов массива `Cells`, соответствующих первой строке таблицы, устанавливает процедура обработки события `OnActivate` (ее текст приведен в листинге 5.11), которое происходит во время активизации формы приложения. Кроме того, эта процедура вписывает в первую колонку таблицы названия стран-участниц соревнований.

Листинг 5.11. Инициализация таблицы

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

```
begin
```

```
    tabl.Cells[0,0] := 'Страна';
    tabl.Cells[1,0] := 'Золотых';
    tabl.Cells[2,0] := 'Серебряных';
    tabl.Cells[3,0] := 'Бронзовых';
    tabl.Cells[4,0] := 'Всего';
    tabl.Cells[5,0] := 'Баллов';
    tabl.Cells[0,1] := 'Австралия';
    tabl.Cells[0,2] := 'Белоруссия';
    tabl.Cells[0,3] := 'Великобритания';
    tabl.Cells[0,4] := 'Германия';
    tabl.Cells[0,5] := 'Италия';
    tabl.Cells[0,6] := 'Китай';
    tabl.Cells[0,7] := 'Корея';
    tabl.Cells[0,8] := 'Куба';
    tabl.Cells[0,9] := 'Нидерланды';
    tabl.Cells[0,10] := 'Россия';
```

```
tabl.Cells[0,11]:='США';  
tabl.Cells[0,12]:='Франция';  
tabl.Cells[0,13]:='Япония';
```

```
end;
```

Программа обработки исходной таблицы (листинг 5.12) запускается щелчком мыши на командной кнопке **Итоги** (Button1).

Листинг 5.12. Обработка двумерного массива

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
  c,r:integer; // номер колонки и строки таблицы  
  s:integer;   // всего медалей у команды  
  p:integer;   // очков у команды  
  
  m:integer;   // номер строки с максимальным количеством очков  
  buf:array[0..5] of string; // буфер для обмена строк  
  i:integer;   // номер строки. Используется во время сортировки  
  
begin  
  for r:=1 to tabl.rowcount do // обработать все строки  
  begin  
    s:=0;  
    // вычисляем общее кол-во медалей  
    for c:=1 to 3 do  
      if tabl.cells[c,r] <> ''  
      then s:=s+StrToInt(tabl.cells[c,r])  
      else tabl.cells[c,r]:='0';  
  
    // вычисляем количество очков  
    p:=7*StrToInt(tabl.cells[1,r])+  
      6*StrToInt(tabl.cells[2,r])+  
      5*StrToInt(tabl.cells[3,r]);  
  
    // вывод результата  
    tabl.cells[4,r]:=IntToStr(s); // всего медалей  
    tabl.cells[5,r]:=IntToStr(p); // очков  
  end;  
  
  // сортировка таблицы по убыванию в соответствии  
  // с количеством баллов (по содержимому 5-го столбца)  
  // сортировка методом выбора  
  for r:=1 to tabl.rowcount-1 do  
  begin  
    m:=r; // максимальный элемент — в r-й строке
```

```

for i:=r to tabl.rowcount-1 do
  if StrToInt (tabl.cells[5,i])>StrToInt (tabl.cells[5,m] )
    then m:=i;

if r <> m then
begin // обменяем r-ю и m-ю строки таблицы
  for c:=0 to 5 do
    begin
      buf[c]:=tabl.Cells[c,r] ;
      tabl.Cells[c,r]:=tabl.Cells[c,m] ;
      tabl.Cells[c,m]:=buf[c] ;
    end;
  end;
end;
end;

```

Сначала для каждой страны программа вычисляет общее количество медалей и соответствующее количество очков. Затем, используя метод простого выбора, программа выполняет сортировку таблицы по убыванию количества набранных очков. Во время сортировки для обмена строк таблицы используется строковый массив `buf`, индекс которого, как и индекс столбца таблицы, меняется от нуля до пяти (см. инструкцию объявления массива в тексте программы). Такой прием позволяет наиболее просто выполнить копирование замещаемой строки в буфер и замещение строки содержимым буфера.

На рис. 5.20 приведено диалоговое окно программы после завершения процесса обработки массива.



Рис. 5.20. Диалоговое окно программы **Итоги олимпиады**

Ошибки при использовании массивов

При использовании массивов наиболее распространенной ошибкой является выход значения индексного выражения за допустимые границы, указанные при объявлении массива.

Если в качестве индекса используется константа и ее значение выходит за допустимые границы, то такая ошибка обнаруживается на этапе компиляции программы. Например, если в программе объявлен массив

```
day : array[0..6] of string[11],
```

то во время компиляции программы инструкция

```
day[7] := 'Воскресенье';
```

будет помечена как ошибочная.

Если для доступа к элементу массива в качестве индекса используется переменная или выражение, то возможно возникновение ошибки (исключения) времени выполнения программы. Например, если в программе объявлен массив

```
tabl: array [1..N] of integer;
```

то инструкция

```
for i:=0 to N do  
    tabl[i] := 5;
```

формально является верной, и ее компиляция будет успешно выполнена.

Однако во время выполнения программы, при попытке присвоить значение несуществующему нулевому элементу массива `tab`, на экран будет выведено сообщение об ошибке. Вид окна и текст сообщения зависит от того, откуда запущена программа.

При запуске рассматриваемой программы из Delphi возникает исключение и сообщение имеет вид, приведенный на рис. 5.21.

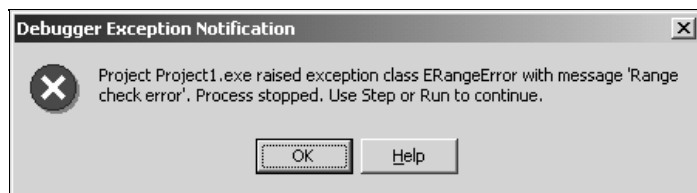


Рис. 5.21. Сообщение об ошибке при обращении к несуществующему элементу массива (программа запущена из Delphi)

Если программа запущена из Windows, то при попытке присвоить значение несуществующему элементу массива на экран будет выведено сообщение

Range check error (ошибка контроля диапазона). В заголовке окна будет указано имя приложения, в процессе выполнения которого произошла ошибка (рис. 5.22).

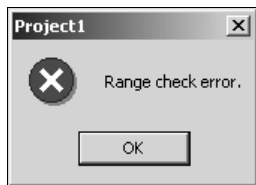


Рис. 5.22. Сообщение об ошибке при обращении к несуществующему элементу массива (программа запущена из Windows)

Примечание

Поведение программы при выходе индексного выражения за границы диапазона допустимых значений определяется настройкой компилятора. Для того чтобы программа контролировала значения индексных выражений (в этом случае Delphi добавляет в выполняемую программу инструкции, обеспечивающие этот контроль), необходимо на вкладке **Compiler** диалогового окна **Project Options**, которое открывается выбором из меню **Project** команды **Options**, установить флажок **Range checking** (Контроль диапазона), находящийся в группе **Runtime errors** (Ошибки времени выполнения) (рис. 5.23).

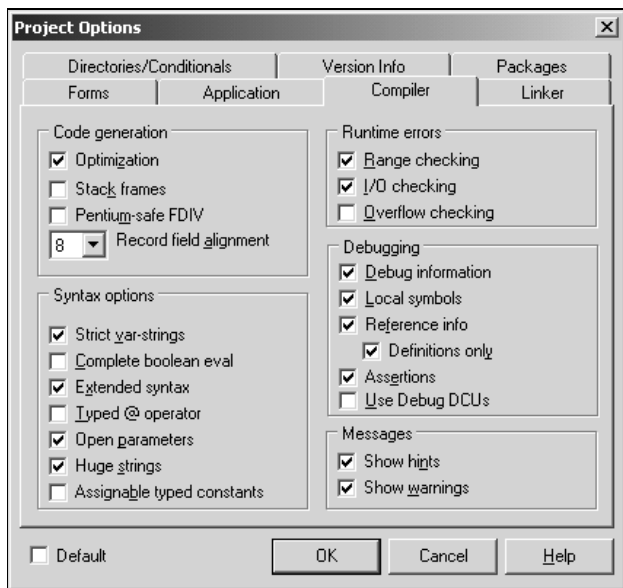
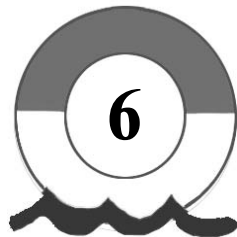


Рис. 5.23. Вкладка **Compiler** диалогового окна **Project Options**

ГЛАВА 6



Процедуры и функции

Часто, работая над программой, программист замечает, что некоторая последовательность инструкций встречается в разных частях программы несколько раз. В качестве примера в листинге 6.1 приведена программа пересчета веса из фунтов в килограммы. Обратите внимание, что в листинге инструкции, обеспечивающие ввод исходных данных из полей редактирования, расчет и вывод результата, которые выделены фоном, есть как в процедуре обработки события на кнопке **Вычислить**, так и в процедуре обработки события OnKeyPress в поле Edit1.

Листинг 6.1. Пересчет веса из фунтов в килограммы

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;           // пояснительный текст
    Edit1: TEdit;             // поле ввода веса в фунтах
    Button1: TButton;         // кнопка Вычислить
    Label2: TLabel;           // поле вывода результата

    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
```

```

public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation
{$R *.dfm}

// щелчок на кнопке Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    f : real;    // вес в фунтах
    kg : real;   // вес в килограммах
begin
    f := StrToFloat(Edit1.Text);
    kg := f * 0.4059;
    Label2.Caption := Edit1.Text + ' ф. — это ' +
        FloatToStrF(kg, ffGeneral, 4, 2) + ' кг.';
end;

// нажатие клавиши в поле ввода исходных данных
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
    f : real;    // вес в фунтах
    kg : real;   // вес в килограммах
begin
    if Key = Char(VK_RETURN) then
        begin
            f := StrToFloat(Edit1.Text);
            kg := f * 0.4059;
            Label2.Caption := Edit1.Text + ' ф. — это ' +
                FloatToStrF(kg, ffGeneral, 4, 2) + ' кг.';
        end;
end;

end.

```

Можно избежать дублирования кода в программе. Для этого надо оформить инструкции, которые встречаются в программе несколько раз, как подпрограмму и заменить инструкции, оформленные в виде подпрограммы, инструкцией вызова подпрограммы.

В листинге 6.2 приведена программа пересчета веса из фунтов в килограммы, в которой ввод исходных данных, вычисления и вывод результата объединены в подпрограмму, которая реализована как функция.

**Листинг 6.2. Пересчет веса из фунтов в килограммы
(использование процедуры)**

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;           // пояснительный текст
    Edit1: TEdit;             // поле ввода веса в фунтах
    Button1: TButton;         // кнопка Вычислить
    Label2: TLabel;           // поле вывода результата

    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// процедура программиста
procedure FuntToKg;
var
  f  : real;    // вес в фунтах
  kg : real;    // вес в килограммах
begin
  f := StrToFloat(Form1.Edit1.Text);
  kg := f * 0.4059;
  Form1.Label2.Caption := Form1.Edit1.Text + ' ф. — это ' +
    FloatToStrF(kg, ffGeneral, 4, 2) + ' кг.';
end;

// щелчок на кнопке Вычислить
procedure TForm1.Button1Click(Sender: TObject);
```



```
begin
    FuntToKg;    // вызов процедуры FuntToKg
end;

// нажатие клавиши в поле ввода исходных данных
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = Char(VK_RETURN)
    then FuntToKg;    // вызов процедуры FuntToKg
end;

end.
```

Преимущества использования подпрограмм очевидны. Во-первых, в программе нет дублирования кода, что сокращает трудоемкость создания программы, делает более удобным процесс отладки и внесения изменений. Представьте, что нужно изменить пояснительный текст, выводимый программой пересчета веса из фунтов в килограммы. В программе, не использующей подпрограмму, нужно просмотреть весь текст и сделать необходимые изменения. Если программа использует подпрограмму, то изменения надо внести только в текст подпрограммы. Во-вторых, значительно повышается надежность программы. Следует обратить внимание, что подпрограммы используют не только тогда, когда нужно избежать дублирования кода. Удобно большую задачу разделить на несколько подзадач и оформить каждую задачу как подпрограмму. В этом случае значительно улучшается "читаемость" программы и, как следствие, существенно облегчается процесс отладки.

Подпрограмма — это модуль программы, который решает часть общей задачи. В Object Pascal есть два вида подпрограмм — процедура и функция. У каждой подпрограммы есть имя, которое используется в программе для вызова подпрограммы (процедуры).

Отличие функции от процедуры состоит в том, что с именем функции связано значение, поэтому функцию можно использовать в качестве операнда, например, в выражении.

Как правило, подпрограмма имеет параметры. Различают формальные и фактические параметры.

Параметры, которые указываются в объявлении функции, называются *формальными*. Параметры, которые указываются в инструкции вызова процедуры, называются *фактическими*.

Параметры используются:

- ☐ для передачи данных в подпрограмму;
- ☐ для получения из подпрограммы результата.

В общем случае в качестве фактического параметра процедуры можно использовать выражение, тип которого должен совпадать с типом соответствующего формального параметра.

Функция

Функция — это подпрограмма, т. е. последовательность инструкций, имеющая имя.

Процесс перехода к инструкциям функции называется вызовом функции или обращением к функции. Процесс перехода от инструкций функции к инструкциям программы, вызвавшей функцию, называется возвратом из функции.

В общем виде инструкция обращения к функции выглядит так:

```
Переменная := Функция(Параметры);
```

где:

Переменная — имя переменной, которой надо присвоить значение, вычисляемое функцией;

Функция — имя функции, значение которой надо присвоить переменной;

Параметры — список формальных параметров, которые применяются для вычисления значения функции. В качестве параметров обычно используют переменные или константы.

Следует обратить внимание на то, что:

- каждая функция возвращает значение определенного типа, поэтому тип переменной, которой присваивается значение функции, должен соответствовать типу функции;
- тип и количество параметров для каждой конкретной функции строго определены.

Объявление функции

Объявление функции в общем виде выглядит так:

```
function Имя(параметр1 : тип1, ..., параметрK : типK) : Тип;  
var  
    // здесь объявления внутренних (локальных) переменных  
  
begin  
    // здесь инструкции функции  
    Имя := Выражение;  
end;
```

где:

function — зарезервированное слово языка Object Pascal, обозначающее, что далее следуют инструкции, реализующие функцию программиста;

Имя — имя функции. Используется для перехода из программы к инструкциям функции;

параметр — это переменная, значение которой используется для вычисления значения функции. Отличие параметра от обычной переменной состоит в том, что он объявляется не в разделе объявления переменных, который начинается словом **var**, а в заголовке функции. Конкретное значение параметр получает во время работы программы в результате вызова функции из основной программы;

Тип — тип значения, которое функция возвращает в вызвавшую ее программу.

Следует обратить внимание, что последовательность инструкций, реализующих функцию, завершается инструкцией, которая присваивает значение имени функции. Тип выражения, определяющего значение функции, должен совпадать с типом функции, указанным в ее объявлении.

В качестве примера в листинге 6.3 приведены функции `IsInt` и `IsFloat`. Функция `IsInt` проверяет, является ли символ, соответствующий нажатой во время ввода целого числа в поле редактирования, допустимым. Предполагается, что допустимыми являются цифры, клавиши `<Enter>` и `<Back Space>`. Функция `IsFloat` решает аналогичную задачу, но для дробного числа. У функции `IsFloat` два параметра: код нажатой клавиши и строка символов, которая уже введена в поле редактирования.

Листинг 6.3. Примеры функций

```
// проверяет, является ли символ допустимым
// во время ввода целого числа
function IsInt(ch : char) : Boolean;
begin
    if      (ch >= '0') and (ch <= '9') // цифры
    or      (ch = #13)              // клавиша <Enter>
    or      (ch = #8)                // клавиша <Back Space>
    then IsInt := True              // символ допустим
    else IsInt := False;           // недопустимый символ
end;

// проверяет, является ли символ допустимым
// во время ввода дробного числа
function IsFloat(ch : char; st: string) : Boolean;
begin
```

```

if      (ch >= '0') and (ch <= '9') // цифры
or      (ch = #13)           // клавиша <Enter>
or      (ch = #8)            // клавиша <Back Space>
then
begin
    IsFloat := True; // символ верный
    Exit;           // выход из функции
end;

case ch of
    '-': if Length(st) = 0 then IsFloat := True;
    ',': if      ( Pos(',',st) = 0)
           and (st[Length(st)] >= '0')
           and (st[Length(st)] <= '9')
        then // разделитель можно ввести только после цифры
              // и если он еще не введен
              IsFloat := True;
    else // остальные символы запрещены
        IsFloat := False;
end;
end;
end;

```

Использование функции

Если вы собираетесь использовать в программе свою функцию, то в простейшем случае ее объявление следует поместить в текст программы, перед подпрограммой, которая применяет эту функцию.



Рис. 6.1. Окно программы **Поездка на дачу**

Следующая программа (ее текст приведен в листинге 6.4, а вид диалогового окна на рис. 6.1) вычисляет стоимость поездки на дачу. Исходными данными для программы являются: расстояние, цена одного литра бензина и по-

требление бензина на 100 км пути. Для ввода исходных данных применяются поля Edit1, Edit2 и Edit3. Функции обработки события OnKeyPress используют функцию IsFloat для фильтрации вводимых в эти поля символов, во время работы программы в полях ввода отображаются только допустимые символы.

Листинг 6.4. Пример использования функций программиста

```
unit fazenda_;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // расстояние
    Edit2: TEdit;           // цена литра бензина
    Edit3: TEdit;           // потребление бензина на 100 км
    CheckBox1: TCheckBox;   // True – поезда туда и обратно
    Button1: TButton;       // кнопка Вычислить
    Label4: TLabel;         // поле вывода результата расчета
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
    procedure Edit2KeyPress(Sender: TObject; var Key: Char);
    procedure Edit3KeyPress(Sender: TObject; var Key: Char);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

// проверяет, является ли символ допустимым
// во время ввода дробного числа
function IsFloat(ch : char; st: string) : Boolean;
```

```

begin
  if      (ch >= '0') and (ch <= '9') // цифры
    or    (ch = #13)           // клавиша <Enter>
    or    (ch = #8)            // клавиша <Back Space>
  then
    begin
      IsFloat := True; // СИМВОЛ верный
      Exit;           // ВЫХОД ИЗ функции
    end;
  case ch of
    '-': if Length(st) = 0 then IsFloat := True;
    ',': if      ( Pos(',',st) = 0)
          and (st[Length(st)] >= '0')
          and (st[Length(st)] <= '9')
        then // разделитель можно ввести только после цифры
              // и если он еще не введен
              IsFloat := True;
    else // остальные символы запрещены
          IsFloat := False;
  end;
end;

{$R *.dfm}

// нажатие клавиши в поле Расстояние
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = Char(VK_RETURN)
  then Edit2.SetFocus // переместить курсор в поле Цена
  else If not IsFloat(Key,Edit2.Text) then Key := Chr(0);
end;

// нажатие клавиши в поле Цена
procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = Char(VK_RETURN)
  then Edit3.SetFocus // переместить курсор в поле Потребление
  else If not IsFloat(Key,Edit2.Text) then Key := Chr(0);
end;

// нажатие клавиши в поле Потребление
procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);

```

```

begin
    if Key = Char(VK_RETURN)
    then Button1.SetFocus // // сделать активной кнопку Вычислить
    else If not IsFloat(Key, Edit2.Text) then Key := Chr(0);
end;

// щелчок на кнопке Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    rast : real; // расстояние
    cena : real; // цена
    potr : real; // потребление на 100 км
    summ : real; // сумма
    mes: string;
begin
    rast := StrToFloat(Edit1.Text);
    cena := StrToFloat(Edit2.Text);
    potr := StrToFloat(Edit3.Text);
    summ := rast / 100 * potr * cena;
    if CheckBox1.Checked then
        summ := summ * 2;
    mes := 'Поездка на дачу';
    if CheckBox1.Checked then
        mes := mes + ' и обратно';
    mes := mes + 'обойдется в ' + FloatToStrF(summ, ffGeneral, 4, 2)
        + ' руб.';
    Label4.Caption := mes;
end;

end.

```

Процедура

Процедура — это разновидность подпрограммы. Обычно подпрограмма реализуется как процедура в двух случаях:

- ☐ когда подпрограмма не возвращает в основную программу никаких данных. Например, вычерчивает график в диалоговом окне;
- ☐ когда подпрограмма возвращает в вызвавшую ее программу больше, чем одно значение. Например, подпрограмма, которая решает квадратное уравнение, должна вернуть в вызвавшую ее программу два дробных числа — корни уравнения.

Объявление процедуры

В общем виде объявление процедуры выглядит так:

```
procedure Имя(var параметр1:тип1; var параметр2:тип2; ...
    var параметрK:типK) ;
var
    // здесь объявление внутренних (локальных) переменных

begin

    // здесь инструкции процедуры

end;
```

где:

procedure — зарезервированное слово языка Object Pascal, обозначающее, что далее следует объявление процедуры;

Имя — имя процедуры, которое используется для вызова процедуры;

ПараметрK — формальный параметр, переменная, которая используется в инструкциях процедуры. Слово **var**, перед именем параметра, не является обязательным. Однако если оно стоит, то это означает, что в инструкции вызова процедуры фактическим параметром обязательно должна быть переменная.

Параметры процедуры используются для передачи данных в процедуру, а также для возврата данных из процедуры в вызвавшую ее программу.

В качестве примера в листинге 6.5 приведена процедура решения квадратного уравнения (которое в общем виде записывается так: $ax^2+bx+c=0$). У процедуры шесть параметров: первые три предназначены для передачи в процедуру исходных данных — коэффициентов уравнения; параметры *x1* и *x2* используются для возврата результата — корней уравнения; параметр *ok* служит для передачи информации о том, что решение существует.

Листинг 6.5. Процедура SqRoot

```
// решает квадратное уравнение
procedure SqRoot( a,b,c : real; var x1,x2 : real; var ok : boolean);
    { a,b,c — коэффициенты уравнения
      x1,x2 — корни уравнения
      ok = True — решение есть
      ok = False — решения нет }
var
    d : real; // дискриминант
```



```

begin
  d:= Sqr( b) - 4*a*c;
  if d < 0
    then
      ok := False  // уравнение не имеет решения
    else
      begin
        ok := True;
        x1 := (-b + Sqrt(d)) / (2*a);
        x2 := ( b + Sqrt(d)) / (2*a);
      end;
end;

```

Использование процедуры

Разработанную процедуру нужно поместить в раздел `implementation`, перед подпрограммой, которая использует эту процедуру.

Инструкция вызова процедуры в общем виде выглядит так:

Имя (СписокПараметров);

где:

Имя — имя вызываемой процедуры;

СписокПараметров — разделенные запятыми фактические параметры.

Фактическим параметром, в зависимости от описания формального параметра в объявлении процедуры, может быть переменная, выражение или константа соответствующего типа.

Например, инструкция вызова приведенной выше процедуры решения квадратного уравнения может выглядеть следующим образом:

```

SqRoot(StrToFloat(Edit1.Text), StrToFloat(Edit2.Text),
  StrToFloat(Edit3.Text), k1,k2,rez);

```

Если в описании процедуры перед именем параметра стоит слово `var`, то при вызове процедуры на месте соответствующего параметра должна стоять переменная основной программы. Использование константы или выражения считается ошибкой, и компилятор в этом случае выведет сообщение: `Types of actual and formal var parameters must be identical` (тип фактического параметра должен соответствовать типу формального параметра).

В листинге 6.6 приведена программа решения квадратного уравнения, в которой используется процедура `SqRoot`. Окно программы представлено на рис. 6.2.

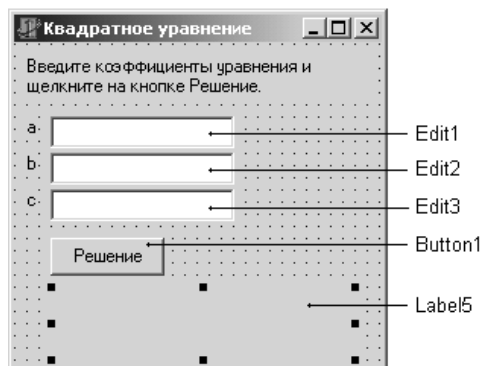


Рис. 6.2. Окно программы
Квадратное уравнение

Листинг 6.6

```
unit SqRoot_;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Button1: TButton;
    Label5: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation*// решает квадратное уравнение***procedure** SqRoot(a,b,c : real; **var** x1, x2 : real; **var** ok : boolean);*{ a,b,c – коэффициенты уравнения**x1,x2 – корни уравнения**ok = True – решение есть**ok = False – решения нет }***var**d : real; *// дискриминант***begin**

d:= Sqr(b) – 4*a*c;

if d < 0**then**ok := False *// уравнение не имеет решения***else****begin**

ok := True;

x1 := (-b + Sqrt(d)) / (2*a);

x2 := (b + Sqrt(d)) / (2*a);

end;**end;**

{ \$R *.dfm }

procedure TForm1.Button1Click(Sender: TObject);**var**k1,k2: real; *// корни уравнения*rez: boolean; *// True – решение есть, False – решения нет*mes: **string**; *// сообщение***begin**

SqRoot(StrToFloat(Edit1.Text), StrToFloat(Edit2.Text),

StrToFloat(Edit3.Text), k1,k2, rez);

if rez **then**

mes := 'Корни уравнения' + #13 +

'x1=' + FloatToStrF(k1, ffGeneral, 4, 2) + #13 +

'x2=' + FloatToStrF(k2, ffGeneral, 4, 2) + #13

else

mes := 'Уравнение не имеет решения';

label5.Caption := mes;

end;**end.**

Повторное использование функций и процедур

Разработав некоторую функцию, программист может использовать ее в другой программе, поместив текст этой функции в раздел `implementation`. Однако этот способ неудобен, т. к. приходится набирать текст функции заново или копировать его из текста другой программы.

Создание модуля

Delphi позволяет программисту поместить свои функции и процедуры в отдельный модуль, а затем использовать процедуры и функции модуля в своих программах, указав имя модуля в списке модулей, необходимых программе (инструкция `uses`).

Чтобы приступить к созданию модуля, нужно сначала закрыть окно формы и окно модуля формы (в ответ на вопрос о необходимости сохранения модуля следует выбрать **No**, т. е. модуль, соответствующий закрытой форме, сохранять не надо). Затем из меню **File** нужно выбрать команду **New/Unit**. В результате открывается окно редактора кода, в котором находится сформированный Delphi шаблон модуля. Его текст приведен в листинге 6.7.

Листинг 6.7. Шаблон модуля

```
unit Unit1;  
  
interface  
  
implementation  
  
end.
```

Начинается модуль заголовком — инструкцией `unit`, в которой указано имя модуля. Во время сохранения модуля это имя будет автоматически заменено на имя, указанное программистом.

Слово `interface` отмечает раздел интерфейса модуля. В этот раздел программист должен поместить объявления находящихся в модуле процедур и функций, которые могут быть вызваны из других модулей, использующих данный.

В раздел `implementation` (реализация) нужно поместить процедуры и функции, объявленные в разделе `interface`.

В качестве примера в листинге 6.8 приведен модуль программиста, который содержит рассмотренные ранее функции `IsInt` и `IsFloat`.

Листинг 6.8. Модуль программиста

```

unit my_unit;

interface // объявления процедур и функций,
           // доступных программам,
           // использующим этот модуль

function IsInt(ch : char) : Boolean;
    // функция IsInt проверяет, является ли символ
    // допустимым во время ввода целого числа

function IsFloat(ch : char; st: string) : Boolean;
    // Функция IsFloat проверяет, является ли символ допустимым
    // во время ввода дробного числа
    // ch – очередной символ
    // st – уже введенные символы

implementation // реализация

    // проверяет, является ли символ допустимым
    // во время ввода целого числа
    function IsInt(ch : char) : Boolean;
    begin
        if      (ch >= '0') and (ch <= '9') // цифры
        or      (ch = #13)                // клавиша <Enter>
        or      (ch = #8)                  // клавиша <Back Space>
        then IsInt := True    // символ допустим
        else IsInt := False;  // недопустимый символ
    end;

    // проверяет, является ли символ допустимым
    // во время ввода дробного числа
    function IsFloat(ch : char; st: string) : Boolean;
        // ch – очередной символ
        // st – уже введенные символы
    begin
        if      (ch >= '0') and (ch <= '9') // цифры
        or      (ch = #13)                // клавиша <Enter>
        or      (ch = #8)                  // клавиша <Back Space>
        then
            begin
                IsFloat := True; // символ верный
                Exit;           // выход из функции
            end;
        end;
    end;

```

```

case ch of
  '-': if Length(st) = 0 then IsFloat := True;
  ',': if      ( Pos(',',st) = 0)
        and (st[Length(st)] >= '0')
        and (st[Length(st)] <= '9')
      then // разделитель можно ввести только после цифры
            // и если он еще не введен
            IsFloat := True;
        else // остальные символы запрещены
            IsFloat := False;
end;
end;
end.

```

Сохраняется модуль обычным образом, т. е. выбором из меню **File** команды **Save**. Вместе с тем, для модулей повторно используемых процедур и функций лучше создать отдельную папку, назвав ее, например, **Units**.

Использование модуля

Для того чтобы в программе могли применяться функции и процедуры модуля, программист должен добавить этот модуль к проекту и указать имя модуля в списке используемых модулей (обычно имя модуля программиста помещают в конец сформированного Delphi списка используемых модулей).

В листинге 6.9 приведен вариант программы **Поездка на дачу**. Процедура обработки события `OnKeyPress` в полях ввода исходных данных обращается к функции `IsFloat`, которая находится в модуле `my_unit.pas`, поэтому в списке используемых модулей указано имя модуля `my_unit`.

Листинг 6.9. Использование функции из модуля программиста

```

unit fazenda_;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls,
  my_unit; // модуль программиста

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // расстояние
    Edit2: TEdit;           // цена литра бензина

```

```

Edit3: TEdit;           // потребление бензина на 100 км
CheckBox1: TCheckBox; // True — поезда туда и обратно
Button1: TButton;      // кнопка Вычислить
Label4: TLabel;        // поле вывода результата расчета
Label1: TLabel;
Label2: TLabel;
Label3: TLabel;
procedure Edit1KeyPress(Sender: TObject; var Key: Char);
procedure Edit2KeyPress(Sender: TObject; var Key: Char);
procedure Edit3KeyPress(Sender: TObject; var Key: Char);
procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

// нажатие клавиши в поле Расстояние
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = Char(VK_RETURN)
    then Edit2.SetFocus // переместить курсор в поле Цена
    else If not IsFloat(Key,Edit2.Text) then Key := Chr(0);
end;

// нажатие клавиши в поле Цена
procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = Char(VK_RETURN)
    then Edit3.SetFocus // переместить курсор в поле Потребление
    else If not IsFloat(Key,Edit2.Text) then Key := Chr(0);
end;

// нажатие клавиши в поле Потребление
procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = Char(VK_RETURN)
    then Button1.SetFocus // // сделать активной кнопку Вычислить

```

```

    else If not IsFloat (Key,Edit2.Text) then Key := Chr(0);
end;

// щелчок на кнопке Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    rast : real; // расстояние
    cena : real; // цена
    potr : real; // потребление на 100 км
    summ : real; // сумма
    mes: string;
begin
    rast := StrToFloat(Edit1.Text);
    cena := StrToFloat(Edit2.Text);
    potr := StrToFloat(Edit3.Text);
    summ := rast / 100 * potr * cena;
    if CheckBox1.Checked then
        summ := summ * 2;
    mes := 'Поездка на дачу';
    if CheckBox1.Checked then
        mes := mes + ' и обратно';
    mes := mes + ' обойдется в ' + FloatToStrF(summ,ffGeneral,4,2)
        + ' руб.';
    Label4.Caption := mes;
end;
end.

```

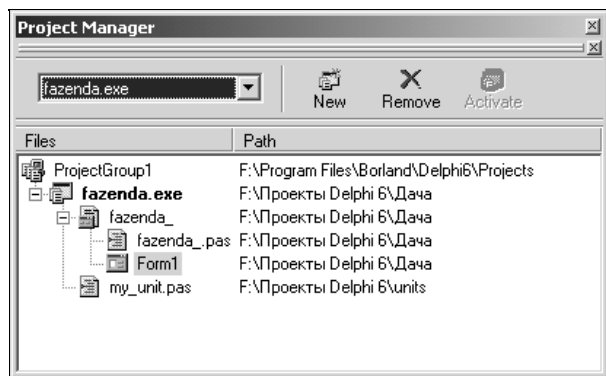


Рис. 6.3. Структура проекта отражается в окне **Project Manager**

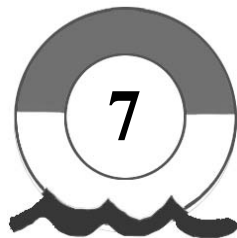
После добавления имени модуля в список модулей, используемых приложением, сам модуль нужно добавить в проект. Для этого из меню **Project** надо выбрать команду **Add to Project** и в открывшемся диалоговом окне — имя

файла модуля. В результате добавления модуля к проекту в окне редактора появится вкладка с текстом добавленного к проекту модуля.

Увидеть структуру проекта можно в окне **Project Manager**, которое появляется в результате выбора соответствующей команды из меню **View**. В качестве примера на рис. 6.3 приведена структура проекта **Поездка на дачу**.

После добавления модуля к проекту и включения его имени в список используемых модулей (инструкция `uses`) можно выполнить компиляцию программы.

ГЛАВА 7



Файлы

Программы, которые до настоящего момента рассматривались в книге, выводили результат своей работы на экран. Вместе с тем, Pascal позволяет сохранить результаты работы программы на диске компьютера, в файле, что дает возможность использовать эти данные для дальнейшей обработки, минуя процесс их ввода с клавиатуры.

Объявление файла

Файл — это именованная структура данных, представляющая собой последовательность элементов данных одного типа, причем количество элементов последовательности практически не ограничено. В первом приближении файл можно рассматривать как массив переменной длины неограниченного размера.

Как и любая структура данных (переменная, массив) программы, файл должен быть объявлен в разделе описания переменных. При объявлении файла указывается тип элементов файла.

В общем виде объявление файла выглядит так:

```
Имя:file of ТипЭлементов;
```

Примеры:

```
res: file of char;      // файл символов
coef: file of real;     // файл вещественных чисел
f: file of integer;     // файл целых чисел
```

Файл, компонентами которого являются данные символьного типа, называется *символьным* или *текстовым*. Описание текстового файла в общем виде выглядит так:

```
Имя:TextFile;
```

где:

Имя — имя файловой переменной;

TextFile — обозначение типа, показывающее, что *Имя* — это файловая переменная, представляющая текстовый файл.

Назначение файла

Объявление файловой переменной задает только тип компонентов файла. Для того чтобы программа могла выводить данные в файл или считывать данные из файла, необходимо указать конкретный файл, т. е. связать файловую переменную с конкретным файлом (задать имя файла).

Имя файла задается вызовом процедуры AssignFile, связывающей файловую переменную с конкретным файлом.

Описание процедуры AssignFile выглядит следующим образом:

```
AssignFile(var f, ИмяФайла: string)
```

Имя файла задается согласно принятым в Windows правилам. Оно может быть полным, т. е. состоять не только непосредственно из имени файла, но и включать путь к файлу (имя диска, каталогов и подкаталогов).

Ниже приведены примеры вызова процедуры AssignFile:

```
AssignFile(f, 'a:\result.txt');  
AssignFile(f, '\students\ivanov\korni.txt');  
fname=('otchet.txt');  
AssignFile(f, fname);
```

Вывод в файл

Непосредственно вывод в текстовый файл осуществляется при помощи инструкции write или writeln. В общем виде эти инструкции записываются следующим образом:

```
write(ФайловаяПеременная, СписокВывода);  
writeln(ФайловаяПеременная, СписокВывода);
```

где:

ФайловаяПеременная — переменная, идентифицирующая файл, в который выполняется вывод;

СписокВывода — разделенные запятыми имена переменных, значения которых надо вывести в файл. Помимо имен переменных в список вывода можно включать строковые константы.

Например, если переменная `f` является переменной типа `TextFile`, то инструкция вывода значений переменных `x1` и `x2` в файл может быть такой:

```
write(f, 'Корни уравнения', x1, x2);
```

Различие между инструкциями `write` и `writeln` состоит в том, что инструкция `writeln` после вывода всех значений, указанных в списке вывода, записывает в файл символ "новая строка".

Открытие файла для вывода

Перед выводом в файл его необходимо открыть. Если программа, формирующая выходной файл, уже использовалась, то, возможно, что файл с результатами работы программы уже есть на диске. Поэтому программист должен решить, как поступить со старым файлом: заменить старые данные новыми или новые данные добавить к старым. Способ использования старого варианта определяется во время открытия файла.

Возможны следующие режимы открытия файла для записи в него данных:

- ❑ Перезапись (запись нового файла поверх существующего или создание нового файла).

- ❑ Добавление в существующий файл.

Для того чтобы открыть файл в режиме создания нового файла или замены существующего, необходимо вызвать процедуру `Rewrite(f)`, где `f` — файловая переменная типа `TextFile`.

Для того чтобы открыть файл в режиме добавления к уже существующим данным, находящимся в этом файле, нужно вызвать процедуру `Append(f)`, где `f` — файловая переменная типа `TextFile`.

На рис. 7.1 приведено диалоговое окно программы, которая выполняет запись или добавление в текстовый файл.

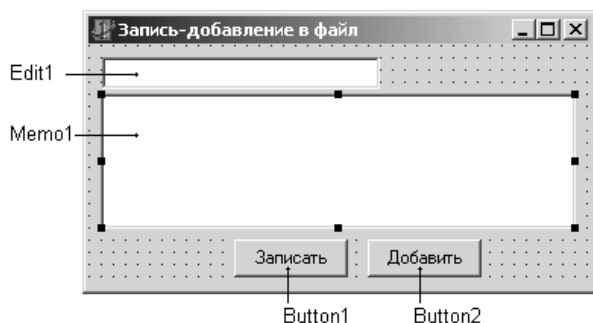


Рис. 7.1. Диалоговое окно программы записи-добавления в файл

В листинге 7.1 приведена процедура, которая запускается нажатием командной кнопки **Записать**. Она открывает файл в режиме создания нового или

замещения существующего файла и записывает текст, находящийся в поле компонента `Memol`.

Имя файла нужно ввести во время работы в поле `Edit1`. Можно задать predetermined имя файла во время разработки формы приложения. Для этого надо присвоить значение, например `test.txt`, свойству `Edit1.Text`.

Листинг 7.1. Создание нового или замещение существующего файла

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f: TextFile;           // файл
  fName: String[80];     // имя файла
  i: integer;
begin
  fName := Edit1.Text;
  AssignFile(f, fName);

  Rewrite(f); // открыть для перезаписи

  // запись в файл
  for i:=0 to Memol.Lines.Count do // строки нумеруются с нуля
    writeln(f, Memol.Lines[i]);

  CloseFile(f); // закрыть файл

  MessageDlg('Данные ЗАПИСАНЫ в файл ', mtInformation, [mbOk], 0);
end;
```

В листинге 7.2 приведена процедура, которая запускается нажатием командной кнопки **Добавить**. Она открывает файл, имя которого указано в поле `Edit1`, и добавляет в него содержимое поля `Memol`.

Листинг 7.2. Добавление в существующий файл

```
procedure TForm1.Button2Click(Sender: TObject);
var
  f: TextFile;           // файл
  fName: String[80];     // имя файла
  i: integer;
begin
  fName := Edit1.Text;
  AssignFile(f, fName);

  Append(f); // открыть для добавления
```

```
// запись в файл
for i:=0 to Memo1.Lines.Count do // строки нумеруются с нуля
    writeln(f, Memo1.Lines[i]);

CloseFile(f); // закрыть файл

MessageDlg('Данные ДОБАВЛЕНЫ в файл ',mtInformation,[mbOk],0);
end;
```

Ошибки открытия файла

Попытка открыть файл может завершиться неудачей и вызвать ошибку времени выполнения программы. Причин неудачи открытия файлов может быть несколько. Например, программа попытается открыть файл на гибком диске, который не готов к работе (не закрыта шторка накопителя или диск не вставлен в накопитель). Другая причина — отсутствие открываемого в режиме добавления файла (файла нет — добавлять некуда).

При запуске программы из Delphi в случае ошибки во время открытия файла возникает исключение, и на экране появляется диалоговое окно с сообщением (рис. 7.2).

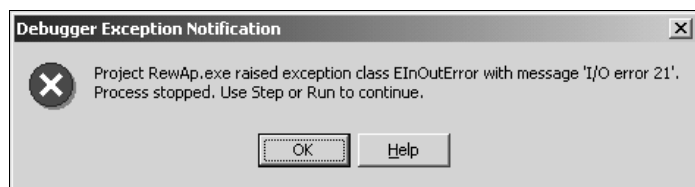


Рис. 7.2. Пример сообщения об ошибке открытия файла (запуск из Delphi)

Если программа запускается из Windows, то окно с сообщением об ошибке выглядит иначе (рис. 7.3).

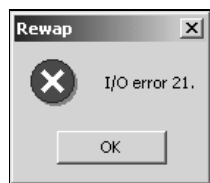


Рис. 7.3. Пример сообщения об ошибке открытия файла (запуск из Windows)

Программа может взять на себя задачу контроля за результатом выполнения инструкции открытия файла. Сделать это можно, проверив значение функции `IOResult` (Input-Output Result — результат ввода-вывода). Функция `IOResult` возвращает 0, если операция ввода-вывода завершилась успешно; в противном случае — код ошибки (не ноль).

Для того чтобы программа смогла проверить результат выполнения операции ввода-вывода, нужно разрешить ей это делать. Для этого надо перед инструкцией вызова процедуры открытия файла поместить директиву компилятору — строку `{SI-}`, которая запрещает автоматическую обработку ошибок ввода-вывода. Эта директива сообщает компилятору, что программа берет на себя контроль ошибок. После инструкции открытия файла следует поместить директиву `{SI+}`, восстанавливающую режим автоматической обработки ошибок ввода-вывода.

На рис. 7.4 приведена блок-схема алгоритма открытия файла для добавления, обеспечивающего создание файла (и тем самым устраняющего ошибку, возникающую при попытке открыть несуществующий файл) в случае, если открываемого для добавления файла на диске еще нет.

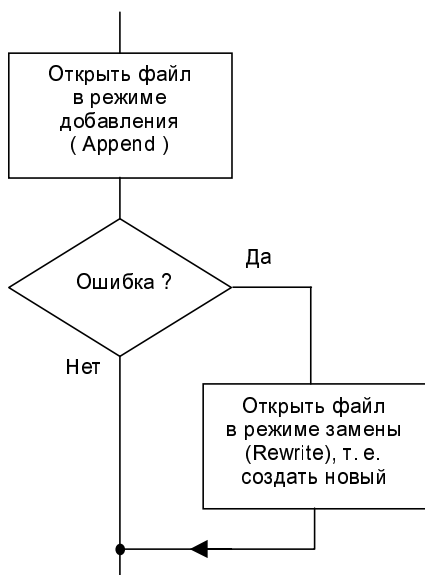


Рис. 7.4. Алгоритм открытия файла с обработкой возможной ошибки

Ниже приведен фрагмент программы, реализующий приведенный выше алгоритм открытия файла.

```

AssignFile(f, filename);
{SI-}
Append(f) // открыть для добавления
{SI+}
if IOResult <> 0      // ошибка открытия
  then Rewrite(f);    // открыть для записи

// здесь открыт существующий или новый файл
  
```

Закрытие файла

Перед завершением работы программа должна закрыть все открытые файлы. Это делается вызовом процедуры `Close`. Процедура `Close` имеет один параметр — имя файловой переменной. Пример использования процедуры:

```
Close(f).
```

Пример программы

Следующая программа ведет простую базу данных. При каждом ее запуске на экране появляется диалоговое окно (рис. 7.5), в поля которого пользователь может ввести дату и температуру воздуха.

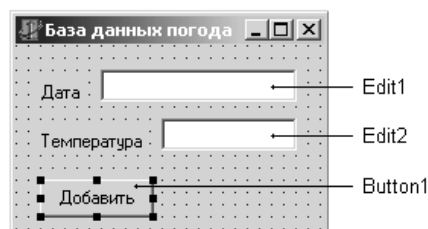


Рис. 7.5. Диалоговое окно программы База данных "Погода"

Дата вводится в поле `Edit1`, температура — в поле `Edit2`. Текст программы приведен в листинге 7.3.

Листинг 7.3. Простая база данных

```
unit pogoda_;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // дата
    Edit2: TEdit;           // температура
    Button1: TButton;       // кнопка Добавить
    Label1: TLabel;
    Label2: TLabel;

  procedure FormActivate(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
```



```

private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

const
    DBNAME = 'a:\pogoda.db';

var
    db: TextFile; // файл — база данных

procedure TForm1.FormActivate(Sender: TObject);
begin
    AssignFile(db, DBNAME);
    {$I-}
    Append(db);
    if IOResult = 0
    then
        begin
            Edit1.Text := DateToStr( Date); // получить текущую дату
            Edit2.SetFocus;                 // курсор в поле Edit2
        end
    else
        begin
            Rewrite(db);
            if IOResult <> 0 then
                begin
                    // сделать недоступными поля ввода
                    // и командную кнопку
                    Edit1.Enabled := False;
                    Edit2.Enabled := False;
                    Button1.Enabled := False;

                    ShowMessage('Ошибка создания '+DBNAME);
                end;
            end;
        end;
    end;

end;

// щелчок на кнопке Добавить
procedure TForm1.Button1Click(Sender: TObject);

```

```
begin
    if (Length(edit1.text)=0) or (Length(edit2.text)=0)
    then ShowMessage('Ошибка ввода данных.'
        + #13 + 'Все поля должны быть заполнены.')
    else writeln(db, edit1.text, ' ', edit2.text);
end;

// Событие OnClose возникает при закрытии формы
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    CloseFile(db); // закрыть файл БД
end;

end.
```

Файл базы данных открывает процедура `FormActivate`, которая обрабатывает событие `OnActivate`. Событие `OnActivate` возникает в момент активизации формы, поэтому процедура запускается автоматически, при активизации формы приложения. Если операция открытия файла завершается успешно, то в поле `Edit1` записывается текущая дата. Информация о текущей дате возвращает функция `Date`. Для преобразования возвращаемого функцией `Date` значения (числа типа `Double`) в удобную для восприятия форму используется функция `DateToStr`. После записи даты в поле `Edit1` процедура обработки события `OnActivate` с применением метода `SetFocus` устанавливает курсор в поле ввода температуры. Если в процессе открытия или создания нового файла возникает ошибка, то процедура делает недоступной кнопку **Добавить** и выводит информационное сообщение.

Процедура `TForm1.Button1Click` (процедура обработки события `OnClick`) запускается нажатием кнопки **Добавить** (`Button1`). В результате введенная информация записывается в базу данных — файл `rogoda.db`. Перед выполнением записи программа проверяет, все ли поля формы заполнены, и, если не все, то выводит информационное сообщение.

В результате работы процедуры в конец файла `rogoda.db` будет добавлена строка, содержащая дату (число, месяц, год) и температуру.

В данной программе используется инструкция `writeln`, а не `write`, для того чтобы данные за каждый день располагались в базе данных на отдельной строке.

Примечание

Обратите внимание, что список вывода инструкции `writeln` состоит из трех элементов. После вывода в файл даты (`edit1.text`) в файл записывается пробел, а затем — температура (`edit2.txt`). Если температуру записать в файл сразу после даты, то числа, соответствующие году и температуре, сольются в одну последовательность цифр.

Закрывает базу данных процедура `TForm1.FormClose`, которая обрабатывает событие `OnClose`, возникающее при закрытии формы приложения.

После нескольких запусков программы файл `pogoda.db` может быть, например, таким:

```
9.05.2001 10
10.05.2001 12
11.05.2001 10
12.05.2001 7
```

Ввод из файла

Программа может вводить исходные данные не только с клавиатуры, но и из текстового файла. Для того чтобы воспользоваться этой возможностью, нужно объявить файловую переменную типа `TextFile`, назначить ей при помощи инструкции `AssignFile` имя файла, из которого будут считываться данные, открыть файл для чтения (ввода) и прочитать (ввести) данные, используя инструкцию `read` или `readln`.

Открытие файла

Открытие файла для ввода (чтения) выполняется вызовом процедуры `Reset`, имеющей один параметр — файловую переменную. Перед вызовом процедуры `Reset` с помощью функции `AssignFile` файловая переменная должна быть связана с конкретным файлом.

Например, следующие инструкции открывают файл для ввода:

```
AssignFile(f, 'c:\data.txt');
Reset(f);
```

Если имя файла указано неверно, например файла с указанным именем на диске нет, то возникает ошибка времени выполнения программы. Следует отметить, что другой причиной возникновения ошибки при открытии файла, находящегося на гибком диске, может быть отсутствие готовности дисковода, проще говоря, отсутствие диска в накопителе.

Поэтому в программе следует предусмотреть возможность повторной попытки открытия файла после подтверждения повторения операции.

Как и при открытии файла для записи, программа может взять на себя задачу обработки возможной ошибки при открытии файла, проверяя значение функции `IOResult`.

Фрагмент программы, текст которого приведен в листинге 7.4, использует значение функции `IOResult` для проверки результата открытия файла. Если

попытка открыть файл вызывает ошибку, то программа выводит диалоговое окно с сообщением об ошибке и запросом на подтверждение повторного открытия файла.

Листинг 7.4. Обработка ошибки открытия файла (фрагмент программы)

```
var
  fname : string[80];
  f : TextFile;           // текстовый файл
  res : integer;           // значение IOResult
  answ : word;             // ответ пользователя
begin
  fname := 'a:\test.txt';
  AssignFile(f, fname);
  repeat
    {$I-}
    Reset(f);             // открыть файл для чтения
    {$I+}
    res:=IOResult;
    if res <> 0
      then answ:=MessageDlg('Ошибка открытия ' + fname+#13
                           + 'Повторить попытку?', mtWarning,
                           [mbYes, mbNo], 0);
  until (res= 0) OR (answ = mrNo);

  if res <> 0
    then exit;            // завершение процедуры

  // здесь инструкции, которые выполняются
  // в случае успешного открытия файла
```

Чтение данных из файла

Чтение из файла выполняется при помощи инструкций `read` и `readln`, которые в общем виде записываются следующим образом:

```
read(ФайловаяПеременная, СписокПеременных);
readln(ФайловаяПеременная, СписокПеременных);
```

где:

ФайловаяПеременная — переменная типа `TextFile`;

СписокПеременных — имена переменных, разделенные запятыми.

Чтение чисел

Следует понимать, что в текстовом файле находятся не числа, а их изображения. Действие, выполняемое инструкциями `read` или `readln`, фактически состоит из двух: сначала из файла читаются символы до появления разделителя (пробела или конца строки), затем прочитанные символы, являющиеся изображением числа, преобразуются в число, и полученное значение присваивается переменной, имя которой указано в качестве параметра инструкции `read` или `readln`.

Например, если текстовый файл `a:\data.txt` содержит следующие строки:

```
23 15
45 28
56 71
```

то в результате выполнения инструкций:

```
AssignFile(f, 'a:\data.txt');
Reset(f);           // открыть для чтения
read(f, a);
read(f, b, c);
read(f, d);
```

значения переменных будут следующими: `a = 23`, `b = 15`, `c = 45`, `d = 28`.

Отличие инструкции `readln` от `read` состоит в том, что после считывания из файла очередного числа и присвоения полученного значения переменной, имя которой стоит последним в списке параметров инструкции `readln`, указатель чтения из файла автоматически перемещается в начало следующей строки файла, даже в том случае, если за прочитанным числом есть еще числа.

Поэтому в результате выполнения инструкций

```
AssignFile(f, 'a:\data.txt');
Reset(f);
readln(f, a);
readln(f, b, c);
readln(f, d);
```

значения переменных будут следующими: `a = 23`, `b = 45`, `c = 28`, `d = 56`.

Если при чтении значения численной переменной в файле вместо изображения числа будет какая-то другая последовательность символов, то произойдет ошибка.

Чтение строк

В программе строковая переменная может быть объявлена с указанием длины или без нее.

Например:

```
stroka1:string[10];
stroka2:string;
```

При чтении из файла значения строковой переменной, длина которой явно задана в ее объявлении, считывается столько символов, сколько указано в объявлении, но не больше, чем в текущей строке.

При чтении из файла значения строковой переменной, длина которой явно не задана в объявлении переменной, значением переменной становится оставшаяся после последнего чтения часть текущей строки. Другими словами, если надо прочитать из файла всю строку, то объявите строковую переменную, длина которой заведомо больше самой длинной строки файла, и считывайте строки в эту переменную.

Если одной инструкцией `readln` осуществляется ввод нескольких, например двух, переменных, то первая переменная будет содержать столько символов, сколько указано в ее объявлении или, если длина не указана, всю строку файла. Вторая переменная будет содержать оставшиеся символы текущей строки или, если таких символов нет, не будет содержать ни одного символа (длина строки равна нулю).

Пусть, например, текстовый файл `freinds.txt` содержит строки:

```
Косичкина      Маша
Васильев       Антон
Цой            Лариса
```

В табл. 7.1 приведено несколько вариантов объявления переменных, инструкции чтения из файла `freinds.txt` и значения переменных после выполнения инструкций чтения.

Таблица 7.1. Примеры чтения данных из файла

Объявления переменных	Инструкция чтения из файла	Значение переменных после чтения из файла
fam: string[15] name: string[10]	Readln(f, fam, name)	fam='Косичкина' name='Маша'
fam, name: string;	Readln(f, fam, name)	fam='Косичкина Маша' name=''
drug: string[80]	Readln(f, drug)	drug ='Косичкина Маша'

Определение конца файла

Пусть на диске есть некоторый текстовый файл. Нужно в диалоговое окно вывести содержимое этого файла. Решение задачи довольно очевидно: надо открыть файл, прочитать первую строку, затем вторую, третью и т. д. до тех

пор, пока не будет достигнут конец файла. Но как определить, что прочитана последняя строка, достигнут конец файла?

Для определения конца файла можно воспользоваться функцией `EOF` (End Of File — конец файла). У функции `EOF` один параметр — файловая переменная. Значение функции `EOF` равно `False`, если прочитанный элемент данных не является последним в файле, т. е. возможно дальнейшее чтение. Если прочитанный элемент данных является последним, то значение `EOF` равно `True`.

Примечание

Значение функции `EOF` можно проверить сразу после открытия файла. Если при этом оно окажется равным `True`, то это значит, что файл не содержит ни одного элемента данных, т. е. является пустым (размер такого файла равен нулю).

В листинге 7.5 приведена процедура, которая выполняет поставленную задачу. Она читает строки из файла, имя которого ввел пользователь во время работы программы, и выводит эти строки в поле `Мемо`. Окно программы приведено на рис. 7.6.

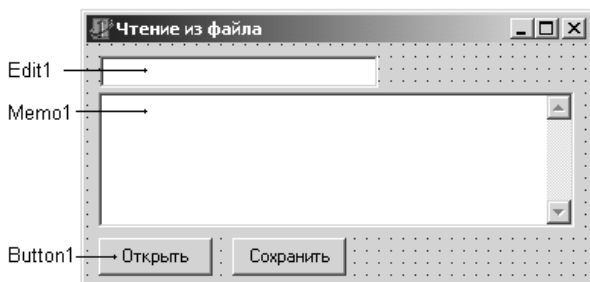


Рис. 7.6. Окно программы **Чтение из файла**

Листинг 7.5. Чтение из файла

```
unit rd_;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
```

```
  Dialogs, StdCtrls, Buttons;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    Button2: TButton;
```

```
    Edit1: TEdit;
```

```
    Memo1: TMemo;
```

```
    Button1: TButton;
```

```
procedure Button2Click(Sender: TObject);
procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
    f: TextFile;           // файл
    fName: String[80];     // имя файла
    buf: String[80];       // буфер для чтения из файла
begin
    fName := Edit1.Text;
    AssignFile(f, fName);

    {$I-}
    Reset(f); // открыть для чтения
    {$I+}
    if IOResult <> 0 then
        begin
            MessageDlg('Ошибка доступа к файлу ' + fName,
                mtError, [mbOk], 0);
            exit;
        end;

    // чтение из файла
    while not EOF(f) do
        begin
            readln(f, buf); // прочитать строку из файла
            Memo1.Lines.Add(buf); // добавить строку в поле Memo1
        end;

    CloseFile(f); // закрыть файл
end;

// запись в файл
procedure TForm1.Button2Click(Sender: TObject);
```



```
var
  f: TextFile;           // файл
  fName: String[80];     // имя файла
  i: integer;

begin
  fName := Edit1.Text;
  AssignFile(f, fName);

  Rewrite(f); // открыть для перезаписи

  // запись в файл
  for i:=0 to Memo1.Lines.Count do // строки нумеруются с нуля
    writeln(f, Memo1.Lines[i]);

  CloseFile(f); // закрыть файл

  MessageDlg('Данные записаны в файл ', mtInformation, [mbOk], 0);
end;

end.
```

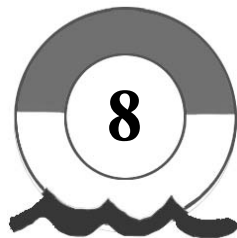
Примечание

Для организации обработки файла использована инструкция цикла `while`, которая обеспечивает проверку значения функции `EOF` перед каждым чтением, в том числе и перед первым.

Наличие кнопки **Сохранить** и соответствующей ей процедуры позволяет сохранить содержимое поля `Мемо` в файле, т. е. программа чтение из файла представляет собой примитивный редактор текста.

Добавление очередной прочитанной из файла строки в поле `Мемо` выполняется применением метода `Add` к свойству `Lines`.

ГЛАВА 8



Типы данных, определяемые программистом

До этого момента в программах использовались стандартные типы данных: *Integer*, *Real*, *Char*, *String* и *Boolean*. Вместе с тем, язык *Pascal* позволяет программисту определить свой собственный тип данных, а затем данные этого типа использовать в программе.

Объявляемый программистом новый тип данных базируется на стандартных типах или на типах, созданных программистом ранее. Тип, определенный программистом, может быть отнесен к:

- ☐ перечисляемому;
- ☐ интервальному;
- ☐ составному типу данных (записи).

Перечисляемый тип

Определить перечисляемый тип — это значит перечислить все значения, которые может принимать переменная, относящаяся к данному типу.

В общем виде объявление перечисляемого типа выглядит так:

Тип = (*Значение1*, *Значение2*, ... *Значениеi*)

где:

Тип — имя перечисляемого типа данных;

Значениеi — символьная константа, определяющая одно из значений, которое может принимать переменная типа *Тип*.

Примеры:

```
TDayOfWeek = (MON, TUE, WED, THU, FRI, SAT, SUN);
```

```
TColor = (Red, Yellow, Green);
```

Примечание

Согласно принятому в Delphi соглашению, имена типов должны начинаться с буквы T (от слова *Type* — тип).

После объявления типа можно объявить переменную, относящуюся к этому типу, например

type

```
TDayOfWeek = (MON, TUE, WED, THU, FRI, SAT, SUN);
```

var

```
ThisDay, LastDay: TDayOfWeek;
```

Помимо указания значений, которые может принимать переменная, описание типа задает, как значения соотносятся друг с другом. Считается, что самый левый элемент списка значений является минимальным, а самый правый — максимальным. Для элементов типа *DayOfWeek* справедливо:

MON < TUE < WED < THU < FRI < SAT < SUN

Свойство упорядоченности элементов перечисляемого типа позволяет использовать переменные перечисляемого типа в управляющих инструкциях, например, так:

```
if (Day = SAT) OR (Day = SUN) then
```

```
  begin
```

```
    { действия, если день — суббота или воскресенье }
```

```
  end;
```

Приведенную инструкцию можно записать и так:

```
if Day > FRI then
```

```
  begin
```

```
    { действия, если день — суббота или воскресенье }
```

```
  end;
```

Очевидно, что программа, написанная с использованием объявленного программистом типа, более наглядна, легче читается и, следовательно, уменьшается вероятность появления ошибки.

Во время компиляции Delphi проверяет соответствие типа переменной типу выражения, которое присваивается переменной. Если тип выражения не может быть приведен к типу переменной, то выводится сообщение об ошибке.

Например, в фрагменте программы:

type

```
TDayOfWeek = (MON, TUE, WED, THU, FRI, SAT, SUN);
```

var

```
ThisDay: TDayOfWeek;
```

```
begin
```

```
    ThisDay:=1;
```

```
    if ThisDay = 6 then
```

```
        begin
```

```
            { блок инструкций }
```

```
        end;
```

инструкция `ThisDay:= 1;` ошибочна, т. к. переменная `ThisDay` принадлежит к определенному программистом перечисляемому типу `TDayOfWeek`, а константа, значение которой ей присваивается, принадлежит к целому типу (`integer`). В условии инструкции `if` тоже ошибка.

Можно утверждать, что объявление перечисляемого типа — это сокращенная форма записи объявления именованных констант. Например, приведенное выше объявление типа `TDayOfWeek` равносильно следующему объявлению:

```
const
```

```
    MON=0;
```

```
    TUE=1;
```

```
    WED=2;
```

```
    THU=3;
```

```
    FRI=4;
```

```
    SAT=5;
```

```
    SUN=6;
```

Интервальный тип

Интервальный тип является отрезком или частью другого типа, называемого *базовым*. В качестве базового обычно используют целый тип данных (`integer`).

При объявлении интервального типа указываются нижняя и верхняя границы интервала, т. е. наименьшее и наибольшее значение, которое может принимать переменная объявляемого типа. В общем виде объявление интервального типа выглядит так:

```
Тип = НижняяГраница .. ВерхняяГраница;
```

где:

Тип — имя объявляемого интервального типа данных;

НижняяГраница — наименьшее значение, которое может принимать переменная объявляемого типа;

ВерхняяГраница — наибольшее значение, которое может принимать переменная объявляемого типа.

Примеры:

```
TIndex = 0 .. 100;  
TRusChar = 'А' .. 'я';
```

В объявлении интервального типа можно использовать именованные константы. В следующем примере в объявлении интервального типа `TIndex` использована именованная константа `HBOUND`:

```
const  
    HBOUND=100;  
type  
    TIndex=1..HBOUND;
```

Интервальный тип удобно использовать при объявлении массивов, например, так:

```
type  
    TIndex = 1 .. 100;  
var  
    tabl : array[TIndex] of integer;  
    i : TIndex;
```

Помимо целого типа в качестве базового можно использовать перечисляемый тип, созданный программистом. В следующем фрагменте на основе типа `TMonth` объявлен интервальный тип `TSammer`:

```
type  
    TMonth = (Jan, Feb, Mar, Apr, May, Jun,  
              Jul, Aug, Sep, Oct, Nov, Dec);  
    TSammer = Jun .. Aug;
```

Запись

В практике программирования довольно часто приходится иметь дело с данными, которые естественным образом состоят из других данных. Например, сведения об учащемся содержат фамилию, имя, отчество, число, месяц и год рождения, домашний адрес и другие данные. Для представления подобной информации в языке Pascal используется структура, которая носит название *запись* (record).

С одной стороны, запись можно рассматривать как единую структуру, а с другой — как набор отдельных элементов, компонентов. Характерной особенностью записи является то, что составляющие ее компоненты могут быть разного типа. Другая особенность записи состоит в том, что каждый компонент записи имеет имя.

Итак, запись — это структура данных, состоящая из отдельных именованных компонентов разного типа, называемых *полями*.

Объявление записи

Как любой тип, создаваемый программистом, тип "запись" должен быть объявлен в разделе `type`. В общем виде объявление типа "запись" выглядит так:

```
Имя = record  
    Поле_1 : Тип_1;  
    Поле_2 : Тип_2;  
    Поле_K : Тип_K;  
end;
```

где:

Имя — имя типа "запись";

`record` — зарезервированное слово языка Pascal, означающее, что далее следует объявление компонентов (полей) записи;

Поле_i и *Тип_i* — имя и тип *i*-го компонента (поля) записи, где $i = 1, \dots, k$;

`end` — зарезервированное слово языка Pascal, означающее, что список полей закончен.

Примеры объявлений:

```
type  
    TPerson = record  
        f_name: string[20];  
        l_name: string[20];  
        day: integer;  
        month: integer;  
        year: integer;  
        address: string[50];  
end;
```

```
TDate = record  
    day: integer;  
    month: integer;  
    year: integer;  
end;
```

После объявления типа записи можно объявить переменную-запись (или просто запись), например:

```
var  
    student : TPerson;  
    birthday : TDate;
```

Для того чтобы получить доступ к элементу (полю) переменной-записи (записи), нужно указать имя записи и имя поля, разделив их точкой. Например, инструкция

```
ShowMessage('Имя: ', student.f_name + #13 +  
            'Адрес: ', student.address);
```

выводит на экран содержимое полей `f_name` (имя) и `address` (адрес) переменной-записи `student`.

Иногда тип переменной-записи объявляют непосредственно в разделе объявления переменных. В этом случае тип записи указывается сразу за именем переменной, через двоеточие. Например, запись `student` может быть объявлена в разделе `var` следующим образом:

```
student: record  
    f_name:string[20];  
    l_name:string[20];  
    day:integer;  
    month:integer;  
    year:integer;  
    address:string[50];  
end;
```

Инструкция *WITH*

Инструкция `with` позволяет использовать в тексте программы имена полей без указания имени переменной-записи. В общем виде инструкция `with` выглядит следующим образом:

```
with Имя do  
begin  
    { инструкции программы }  
end;
```

где:

Имя — имя переменной-записи;

`with` — зарезервированное слово языка Pascal, означающее, что далее, до слова `end`, при обращении к полям записи *Имя*, имя записи можно не указывать.

Например, если в программе объявлена запись

```
student:record // информация о студенте  
    f_name:  string[30]; // фамилия  
    l_name:  string[20]; // имя  
    address: string[50]; // адрес  
end;
```

и данные о студенте находятся в полях Edit1, Edit2 и Edit3 диалогового окна, то вместо инструкций

```
student.f_name := Edit1.text;  
student.l_name := Edit2.text;  
student.address := Edit3.text;
```

можно записать

```
with student do  
begin  
    f_name := Edit1.text;  
    l_name := Edit2.text;  
    address := Edit3.text;  
end;
```

Ввод и вывод записей в файл

Записи можно хранить в файле. Для того чтобы программа могла сохранить значение переменной-записи в файле или ввести его из файла, необходимо объявить файл, указав в качестве типа его компонентов тип "запись". Например, инструкции

```
type  
    TPerson = record  
        f_name: string[20];  
        l_name: string[20];  
        address: string[50];  
    end;  
var  
    f: file of TPerson;
```

объявляют файл, компонентами которого являются записи типа TPerson.

Процесс работы с файлом записей практически ничем не отличается от процесса работы с обычным файлом. Сначала надо объявить файловую переменную и с помощью процедуры Assign связать эту переменную с конкретным файлом. Затем нужно открыть файл (для чтения, записи или обновления). После этого можно прочитать запись из файла или записать запись в файл.

Вывод записи в файл

Рассмотрим программу, которая записывает в файл введенные пользователем данные о результатах соревнований, формируя, таким образом, простую базу данных. Исходные данные вводятся в поля диалогового окна (рис. 8.1) и сохраняются в файле, компонентами которого являются записи типа TMedal.

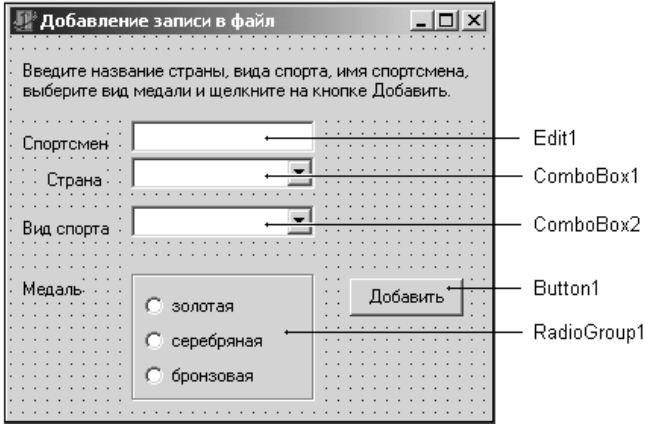


Рис. 8.1. Окно программы **Добавление записи в файл**

Для ввода фамилии спортсмена применяется поле редактирования (компонент `Edit`). Для ввода названия вида спорта и страны используются компоненты `ComboBox` (комбинированный список).



Рис. 8.2. Значок компонента `ComboBox`

Компонент `ComboBox`, значок которого находится на вкладке **Standard** (рис. 8.2), дает возможность ввести данные либо непосредственно в поле ввода-редактирования, либо путем выбора из списка, который появляется в результате щелчка на кнопке раскрывающегося списка.

В табл. 8.1 перечислены свойства компонента `ComboBox`.

Таблица 8.1. Свойства компонента `ComboBox`

Свойство	Определяет
Name	Имя компонента. Используется для доступа к свойствам компонента
Text	Текст, находящийся в поле ввода-редактирования
Items	Элементы раскрывающегося списка
DropDownCount	Количество отображаемых элементов в раскрытом списке
Left	Расстояние от левой границы компонента до левой границы формы

Таблица 8.1 (окончание)

Свойство	Определяет
Top	Расстояние от верхней границы компонента до верхней границы формы
Height	Высоту компонента (поля ввода-редактирования)
Width	Ширину компонента
Font	Шрифт, используемый для отображения элементов списка
ParentFont	Признак наследования свойств шрифта родительской формы

Список, который появляется в результате щелчка на кнопке раскрытия списка, может быть сформирован как в процессе разработки формы приложения, так и во время работы программы. Чтобы сформировать список во время разработки программы, нужно в окне **Object Inspector** выбрать свойство **Items**, щелкнуть на кнопке активизации редактора списка строк и ввести элементы списка.

Текст программы приведен в листинге 8.1.

Листинг 8.1. Добавление записей в файл

```
unit apprec_;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Edit1: TEdit;           // спортсмен
    ComboBox1: TComboBox;  // страна
    ComboBox2: TComboBox;  // вид спорта
    RadioGroup1: TRadioGroup; // медаль
    Button1: TButton;       // кнопка Добавить
    Label5: TLabel;
    Label4: TLabel;
  procedure FormActivate(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure Button1Click(Sender: TObject);
```

```

private
    { Private declarations }
public
    { Public declarations }
end;

// тип медали
TKind = (GOLD, SILVER, BRONZE);

// запись файла
TMedal=record
    country: string[20]; // страна
    sport:   string[20]; // вид спорта
    person:  string[40]; // спортсмен
    kind:    TKind;      // медаль
end;

var
    Form1: TForm1;
    f: file of TMedal; // файл записей – база данных

implementation

{$R *.DFM}

// активизация формы
procedure TForm1.FormActivate(Sender: TObject);
var
    resp : word; // ответ пользователя
begin
    AssignFile(f, 'a:\medals.db');
    {$I-}
    Reset(f); // открыть файл
    Seek( f, FileSize(f)); // указатель записи в конец файла
    {$I+}
    if IOResult = 0
    then button1.enabled:=TRUE // теперь кнопка Добавить доступна
    else
        begin
            resp:=MessageDlg('Файл базы данных не найден.'+
                'Создать новую БД?',
                mtInformation, [mbYes,mbNo], 0);
            if resp = mrYes then
                begin
                    {$I-}
                    rewrite(f);
                    {$I+}

```

```
        if IOResult = 0
            then button1.enabled:=TRUE
            else ShowMessage('Ошибка создания файла БД.');
```

end;

end;

end;

// щелчок на кнопке Добавить

```
procedure TForm1.Button1Click(Sender: TObject);
var
    medal: TMedal;
begin
    with medal do
    begin
        country := ComboBox1.Text;
        sport    := ComboBox2.Text;
        person   := Edit1.Text;
        case RadioGroup1.ItemIndex of
            0: kind := GOLD;
            1: kind := SILVER;
            2: kind := BRONZE;
        end;
    end;
    write(f, medal); // записать содержимое полей записи в файл
end;
```

// завершение работы программы

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    CloseFile( f); // закрыть файл
end;
```

end.

В представленной программе процедура TForm1.FormActivate открывает файл базы данных для добавления. Здесь следует обратить внимание на то, как это реализовано. Процедуру AppendFile, которая открывает файл для добавления в конец, использовать нельзя, т. к. файл не является текстовым. Поэтому файл сначала открывается процедурой Rewrite в режиме перезаписи, а затем процедура Seek устанавливает указатель чтения-записи в конец файла. Параметром процедуры Seek является функция FileSize, значение которой равно размеру файла (в байтах).

Процедура TForm1.Button1Click, которая запускается нажатием кнопки **Добавить** (Button1), выполняет непосредственное добавление записи в файл.

Поля `country` и `sport` добавляемой записи заполняются из свойства `Text` комбинированных списков **Страна** (`ComboBox1`) и **Вид спорта** (`ComboBox2`).

Примечание

Во время создания формы приложения нужно установить значение свойства `Items` компонентов `ComboBox1` и `ComboBox2` — названия стран и видов спорта.

Поле `person` заполняется из поля **Спортсмен** (`Edit1`). Значение поля `medal` определяется выбранной кнопкой компонента `RadioGroup1`.

Процедура `TForm1.FormClose` закрывает файл базы данных.

Поскольку тип `TMedal` используется двумя процедурами (`TForm1.FormActivate` и `TForm1.Button1Click`), то его описание помещено в раздел `type` модуля формы. Объявление файловой переменной `f` по этой же причине помещено в раздел объявления переменных модуля формы.

В приведенном варианте программы предполагается, что списки стран и названий видов спорта формируются при помощи редактора списка строк во время разработки формы. Вместе с тем, список можно сформировать во время разработки программы. Для этого надо к свойству `Items` применить метод `Add`. Например, список стран может быть сформирован при помощи следующих инструкций (их нужно поместить в процедуру `Tform1.FormActivate`):

```
Form1.ComboBox1.Item.Add('Россия');  
Form1.ComboBox1.Item.Add('Австрия');  
Form1.ComboBox1.Item.Add('Германия');  
Form1.ComboBox1.Item.Add('Франция');
```

Ввод записи из файла

Рассмотрим программу, демонстрирующую процесс чтения и обработки записей файла. Программа **Чтение записей из файла**, ее диалоговое окно представлено на рис. 8.3, а текст в листинге 8.2, открывает файл, сформированный программой **Добавление записи в файл**, и, в зависимости от того, какой из переключателей **все** или **выбрать** установлен, выводит список медалей, выигранных представителями всех стран или страны, название которой введено в поле **Страна**. Для вывода результата чтения из файла используется компонент `Memo1`.

В табл. 8.2 приведены значения свойств компонентов формы.

Так как компонент `Memo1` предназначен только для просмотра информации, то свойству `ReadOnly` (только чтение, просмотр) присвоено значение `True`. Свойство `ScrollBars` (полосы прокрутки) компонента `Memo` позволяет задавать отображаемые полосы прокрутки. По умолчанию свойству `ScrollBars` присвоено значение `ssNone`, т. е. полосы прокрутки не отображаются. В рас-

смаатриваемом примере выводится вертикальная полоса, поэтому свойству `ScrollBars` присвоено значение `ssVertical`.

Таблица 8.2

Компонент. свойство	Значение
RadioButton1.Checked	True
Label1.Enabled	False
ComboBox1.Enabled	False
Memo1.ReadOnly	True
Memo1.ScrollBars	SsVertical

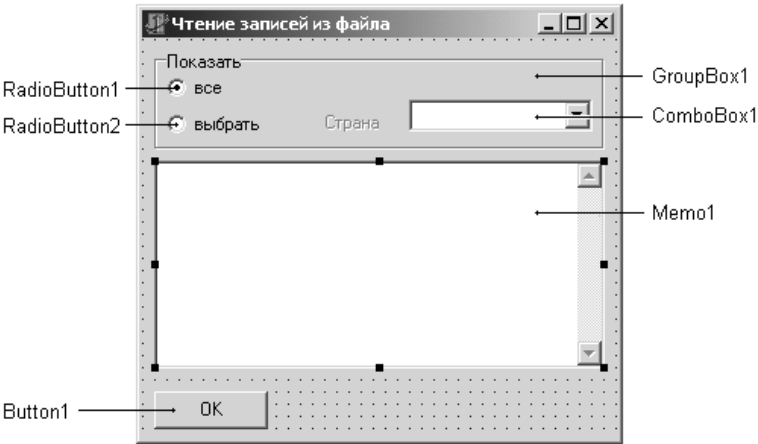


Рис. 8.3. Окно программы Чтение записей из файла

Для ввода названия страны используется компонент `ComboBox1`, что позволяет задавать имя не только прямым вводом названия, но и выбором из списка. Список стран нужно сформировать во время создания формы путем присвоения значения свойству `Items`.

Чтобы сразу после запуска программы список выбора страны был недоступен (т. к. выбран переключатель **все** группы **Показать**), свойству `Enabled` компонентов `ComboBox1` и `Label1` во время создания формы нужно присвоить значение `False`.

Список ввода-выбора названия страны (`ComboBox1`) становится доступным в результате выбора во время работы программы переключателя **выбрать**. Процедура обработки события `OnClick` на переключателе `RadioButton2` делает доступным поле `ComboBox1`.

Листинг 8.2. Чтение записей из файла

```

unit rdrec_;

interface

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

type
    TForm1 = class(TForm)
        RadioButton1: TRadioButton; // переключатель Все
        RadioButton2: TRadioButton; // переключатель Выбрать
        ComboBox1: TComboBox;       // комбинированный список
                                     // для ввода названия страны
        Memo1: TMemo;                // поле вывода записей, удовлетворяющих
                                     // условию запроса
        Button1: TButton;            // кнопка ОК
        GroupBox1: TGroupBox;
        Label1: TLabel;              // текст Страна
        procedure Button1Click(Sender: TObject);
        procedure RadioButton2Click(Sender: TObject);
        procedure RadioButton1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
type
    // тип медали
    TKind = (GOLD, SILVER, BRONZE);

    // запись файла
    TMedal = record
        country: string[20];
        sport: string[20];

```

```
    person:string[40];
    kind:TKind;
end;

var
  f: file of TMedal; // файл записей
  rec: TMedal;        // запись, прочитанная из файла
  n: integer;         // кол-во записей, удовлетворяющих запросу
  st: string[80];

begin
  AssignFile(f, 'a:\medals.db');
  {$I-}
  Reset(f); // открыть файл для чтения
  {$I+}
  if IOResult <> 0 then
    begin
      ShowMessage('Ошибка открытия файла БД.');
```

Exit;

end;

// обработка БД

if RadioButton2.Checked then

Memol.Lines.Add('*** ' + ComboBox1.Text + ' ***');

n := 0;

Memol.Clear; // очистить список поля Мемо

while not EOF(f) do

begin

read(f, rec); // прочитать запись

if RadioButton1.Checked or

(rec.country = ComboBox1.Text) then

begin

n := n + 1;

st := rec.person + ', ' + rec.sport;

if RadioButton1.Checked then

st := st + ', ' + rec.country;

case rec.kind of

GOLD: st := st + ', золото';

SILVER: st := st + ', серебро';

BRONZE: st := st + ', бронза';

end;

Memol.Lines.Add(st);

end;

end;


```
CloseFile(f);  
if n = 0 then  
    ShowMessage('В БД нет запрашиваемой информации.');
```

end;

// переключатель Выбрать

```
procedure TForm1.RadioButton2Click(Sender: TObject);  
begin  
    Label1.Enabled := True;  
    ComboBox1.Enabled := True;    // теперь поле Страна доступно  
    ComboBox1.SetFocus;           // курсор в поле Страна  
end;
```

// переключатель Все

```
procedure TForm1.RadioButton1Click(Sender: TObject);  
begin  
    Label1.Enabled := False;  
    ComboBox1.Enabled := False;    // теперь поле Страна не доступно  
end;
```

end.

Процедура TForm1.Button1Click открывает файл и последовательно считывает находящиеся в нем записи. Содержимое записи добавляется в поле Memo1, если прочитанная запись удовлетворяет условию запроса, т. е. содержимое поля country совпадает с названием страны, введенным пользователем в поле редактирования компонента ComboBox1, или если выбран переключатель RadioButton1.

Информация в поле Мемо добавляется инструкцией Memo1.Lines.Add(st), которая является инструкцией применения метода **Add** (Добавить) к компоненту Memo1.

Примечание

Понятие "метод" будет подробно рассмотрено позже, в разделе, посвященном объектно-ориентированному программированию. Сейчас только скажем, что метод — это процедура, инструкция вызова которой записывается особым образом с целью показать, что одним из ее параметров является *объект*.

Динамические структуры данных

До этого момента мы работали только с данными, имеющими статическую, неизменяемую во время исполнения программы структуру. Во время работы программы могли изменяться только значения переменных, в то время как

количество переменных всегда оставалось постоянным (отсюда и название — статические структуры). Это не всегда удобно.

Например, в программе, предназначенной для ввода и обработки данных об учениках класса, для хранения данных используются массивы. При определении размера массива программисту приходится ориентироваться на некоторое среднее или предельное количество учеников в классе. При этом, если реально учеников в классе меньше предполагаемого количества, то неэффективно используется память компьютера, а если это число больше, то программу использовать уже нельзя (надо внести изменения в исходный текст и выполнить компиляцию).

Задачи, обрабатывающие данные, которые по своей природе являются динамическими, удобно решать с помощью *динамических структур*.

Указатели

Обычно переменная хранит некоторые данные. Однако помимо обычных существуют переменные, которые ссылаются на другие переменные. Такие переменные называются *указателями*. Указатель — это переменная, значением которой является адрес другой переменной или структуры данных. Графически указатель может быть изображен так, как на рис. 8.4.

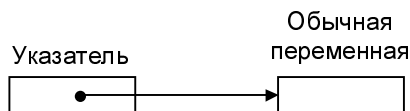


Рис. 8.4. Переменная-указатель

Указатель, как и любая другая переменная программы, должен быть объявлен в разделе объявления переменных. В общем виде объявление указателя выглядит следующим образом:

Имя: ^ *Тип*;

где:

Имя — имя переменной-указателя;

Тип — тип переменной, на которую указывает переменная-указатель;

значок ^ показывает, что объявляемая переменная является указателем.

Приведем примеры объявления указателей:

```
p1: ^integer;  
p2: ^real;
```

В приведенном примере переменная `p1` — это указатель на переменную типа `integer`, а `p2` — указатель на переменную типа `real`.

Тип переменной, на которую указывает указатель, называют типом указателя. Например, если в программе объявлен указатель `p: ^Integer`, то говорят: "p — указатель целого типа" или "p — это указатель на целое".

В начале работы программы переменная-указатель "ни на что не указывает". В этом случае говорят, что значение указателя равно `NIL`. Зарезервированное слово `NIL` соответствует значению указателя, который ни на что не указывает.

Идентификатор `NIL` можно использовать в инструкциях присваивания и в условиях. Например, если переменные `p1` и `p2` объявлены как указатели, то инструкция

```
p1 := NIL;
```

устанавливает значение переменной, а инструкция

```
if p2 = NIL
then ShowMessage('Указатель p2 не инициализирован!');
```

проверяет, инициализирован ли указатель `p2`.

Указателю можно присвоить значение — адрес переменной соответствующего типа (в тексте программы адрес переменной — это имя переменной, перед которым стоит оператор `@`). Ниже приведена инструкция, после выполнения которой переменная `p` будет содержать адрес переменной `n`.

```
p := @n;
```

Помимо адреса переменной указателю можно присвоить значение другого указателя при условии, что они являются указателями на переменную одного типа. Например, если переменные `p1` и `p2` являются указателями типа `integer`, то в результате выполнения инструкции

```
p2 := p1;
```

переменные `p1` и `p2` указывают на одну и ту же переменную.

Указатель можно использовать для доступа к переменной, адрес которой содержит указатель. Например, если `p` указывает на переменную `i`, то в результате выполнения инструкции

```
p^ := 5;
```

значение переменной `i` будет равно пяти. В приведенном примере значок "`^`" показывает, что значение пять присваивается переменной, на которую указывает переменная-указатель.

Динамические переменные

Динамической переменной называется переменная, память для которой выделяется во время работы программы.

Выделение памяти для динамической переменной осуществляется вызовом процедуры `new`. У процедуры `new` один параметр — указатель на переменную

того типа, память для которой надо выделить. Например, если `p` является указателем на тип `real`, то в результате выполнения процедуры `new(p)`; будет выделена память для переменной типа `real` (создана переменная типа `real`), и переменная-указатель `p` будет содержать адрес памяти выделенной для этой переменной.

У динамической переменной нет имени, поэтому обратиться к ней можно только при помощи указателя.

Процедура, использующая динамические переменные, перед завершением своей работы должна освободить занимаемую этими переменными память или, как говорят программисты, "уничтожить динамические переменные". Для освобождения памяти, занимаемой динамической переменной, используется процедура `Dispose`, которая имеет один параметр — указатель на динамическую переменную.

Например, если `p` — указатель на динамическую переменную, память для которой выделена инструкцией `new(p)`, то инструкция `dispose(p)` освобождает занимаемую динамической переменной память.

Следующая процедура, ее текст приведен в листинге 8.3, демонстрирует создание, использование и уничтожение динамических переменных.

Листинг 8.3. Создание, использование и уничтожение динамических переменных

```
procedure TForm1.Button1Click(Sender: TObject);
var
  p1,p2,p3: ^integer; // указатели на переменные типа integer
begin
  // создадим динамические переменные типа integer
  // (выделим память для динамических переменных)
  New(p1);
  New(p2);
  New(p3);

  p1^ := 5;
  p2^ := 3;
  p3^ := p1^ + p2^;
  ShowMessage('Сумма чисел равна ' + IntToStr( p3^));

  // уничтожим динамические переменные
  // (освободим память, занимаемую динамическими переменными)
  Dispose(p1);
  Dispose(p2);
  Dispose(p3);
end;
```

В начале работы процедура создает три динамические переменные. Две переменные, на которые указывают `p1` и `p2`, получают значение в результате выполнения инструкции присваивания. Значение третьей переменной вычисляется как сумма первых двух.

Списки

Указатели и динамические переменные позволяют создавать сложные динамические структуры данных, такие как *списки* и *деревья*.

Список можно изобразить графически (рис. 8.5).

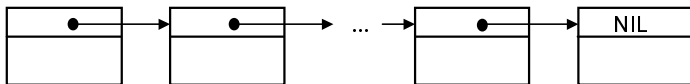


Рис. 8.5. Графическое изображение списка

Каждый элемент списка (узел) представляет собой запись, состоящую из двух частей. Первая часть — информационная. Вторая часть отвечает за связь со следующим и, возможно, с предыдущим элементом списка. Список, в котором обеспечивается связь только со следующим элементом, называется *односвязным*.

Для того чтобы программа могла использовать список, надо определить тип компонентов списка и переменную-указатель на первый элемент списка. Ниже приведен пример объявления компонента списка студентов:

type

```
TPStudent = ^TStudent; // указатель на переменную типа TStudent
```

```
// описание типа элемента списка
```

```
TStudent = record
```

```
  surname: string[20]; // фамилия
```

```
  name: string[20]; // имя
```

```
  group: integer; // номер группы
```

```
  address: string[60]; // домашний адрес
```

```
  next: TPStudent; // указатель на следующий элемент списка
```

```
end;
```

var

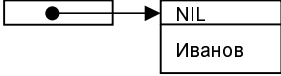
```
head: TPStudent; // указатель на первый элемент списка
```

Добавлять данные можно в начало, в конец или в нужное место списка. Во всех этих случаях необходимо корректировать указатели. На рис. 8.6 изображен процесс добавления элементов в начало списка.

head

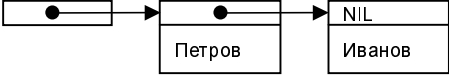
Список пустой. Значение указателя head равно NIL

head



После добавления элемента, head указывает на этот элемент

head



После добавления второго элемента в начало списка, head указывает на новый элемент

Рис. 8.6. Добавление элементов в список

Следующая программа, ее текст приведен в листинге 8.4, формирует список студентов, добавляя фамилии в начало списка. Данные вводятся в поля редактирования диалогового окна программы (рис. 8.7) и добавляются в список нажатием кнопки **Добавить** (Button1).

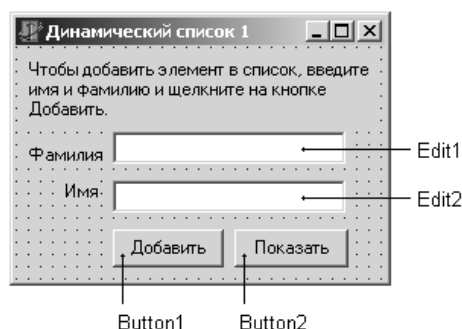


Рис. 8.7. Окно программы **Динамический список 1**

Листинг 8.4. Добавление элемента в начало динамического списка

```
unit dlist1_;

interface

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

type

```
TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Edit1: TEdit;      // фамилия
    Edit2: TEdit;      // имя
    Button1: TButton;  // кнопка Добавить
    Button2: TButton;  // кнопка Показать
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

type

```
TPStudent = ^TStudent; // указатель на тип TStudent

TStudent = record
    f_name: string[20]; // фамилия
    l_name: string[20]; // имя
    next: TPStudent;    // следующий элемент списка
end;
```

var

```
head: TPStudent; // начало (голова) списка
```

```
// добавить элемент в начало списка
```

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

var

```
curr: TPStudent; // новый элемент списка
```

begin

```
new(curr); // выделить память для элемента списка
```

```
curr^.f_name := Edit1.Text;
```

```
curr^.l_name := Edit2.Text;
```

```
// добавление в начало списка
```

```
curr^.next := head;
```

```
head := curr;
```

```
// ОЧИСТИТЬ ПОЛЯ ВВОДА
Edit1.text:='';
Edit2.text:='';

end;

// ВЫВЕСТИ СПИСОК
procedure TForm1.Button2Click(Sender: TObject);
var
  curr: TPStudent; // текущий элемент списка
  n: integer;       // длина (кол-во элементов) списка
  st: string;       // строковое представление списка
begin
  n := 0;
  st := '';
  curr := head; // указатель на первый элемент списка
  while curr <> NIL do
    begin
      n := n + 1;
      st := st + curr^.f_name + ' ' + curr^.l_name + #13;
      curr := curr^.next; // указатель на следующий элемент
    end;
  if n <> 0
    then ShowMessage('Список: ' + #13 + st)
    else ShowMessage('В списке нет элементов. ');
end;

end.
```

Добавление элемента в список выполняет процедура `TForm1.Button1Click`, которая создает динамическую переменную-запись, присваивает ее полям значения, соответствующие содержимому полей ввода диалогового окна, и корректирует значение указателя `head`.

Вывод списка выполняет процедура `TForm1.Button2Click`, которая запускается нажатием кнопки **Показать**. Для доступа к элементам списка используется указатель `curr`. Сначала он содержит адрес первого элемента списка. После того как первый элемент списка будет обработан, указателю `curr` присваивается значение поля `next` той записи, на которую указывает `curr`. В результате этого переменная `curr` содержит адрес второго элемента списка. Таким образом, указатель перемещается по списку. Процесс повторяется до тех пор, пока значение поля `next` текущего элемента списка (элемента, адрес которого содержит переменная `curr`) не равно `NIL`.

Упорядоченный список

Как правило, списки упорядочены. Порядок следования элементов в списке определяется содержимым одного из полей. Например, список с информацией о людях обычно упорядочен по полю, содержащему фамилию.

Добавление элемента в список

Для того чтобы добавить узел в упорядоченный список, необходимо найти узел, за которым должен следовать добавляемый элемент, и установить его указатель на новый элемент, а указатель нового узла установить на узел, перед которым он вставляется (рис. 8.8).

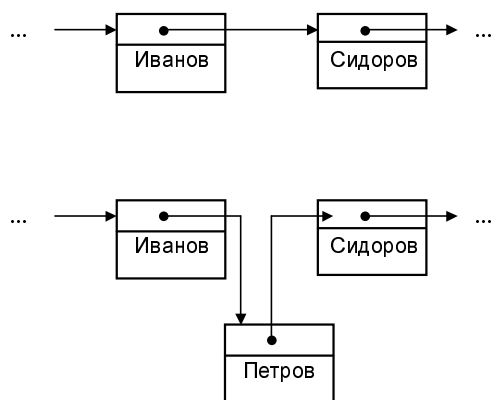


Рис. 8.8. Добавление элемента в упорядоченный список



Рис. 8.9. Диалоговое окно программы **Упорядоченный динамический список 2**

Следующая программа, ее текст приведен в листинге 8.5, а диалоговое окно — на рис. 8.9, формирует список, упорядоченный по полю **Фамилия**. Данные вводятся в поля редактирования (Edit1 и Edit2) и нажатием кнопки **Добавить** (Button1) добавляются в список таким образом, что список всегда упорядочен по полю **Фамилия**.

Листинг 8.5. Добавление элемента в упорядоченный список

```
unit dlist2_;

interface

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

```

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Button2: TButton;
    Label3: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

type
  TPStudent=^TStudent; //указатель на тип TStudent

  TStudent = record
    f_name:string[20]; // фамилия
    l_name:string[20]; // имя
    next: TPStudent; // следующий элемент списка
  end;

var
  head: TPStudent; // начало (голова) списка

procedure TForm1.Button1Click(Sender: TObject);
var
  node: TPStudent; // новый узел списка
  curr: TPStudent; // текущий узел списка
  pre: TPStudent; // предыдущий, относительно curr, узел
begin
  new(node); // создание нового элемента списка
  node^.f_name:=Edit1.Text; // фамилия
  node^.l_name:=Edit2.Text; // имя

```

```
// добавление узла в список
// сначала найдем подходящее место в списке для узла
curr:=head;
pre:=NIL;
{ Внимание!
  Если приведенное ниже условие заменить
  на (node.f_name>curr^.f_name)and(curr<>NIL),
  то при добавлении первого узла возникает ошибка времени
  выполнения, т. к. curr = NIL и, следовательно,
  переменной curr^.name нет!
  В используемом варианте условия ошибка не возникает, т. к.
  сначала проверяется условие (curr <> NIL), значение которого
  FALSE, и второе условие в этом случае не проверяется.
}
```

```
while (curr <> NIL) and (node.f_name > curr^.f_name) do
begin
```

```
    // введенное значение больше текущего
    pre:= curr;
    curr:=curr^.next; // к следующему узлу
```

```
end;
```

```
if pre = NIL
```

```
    then
```

```
        begin
```

```
            // новый узел в начало списка
            node^.next:=head;
            head:=node;
```

```
        end
```

```
    else
```

```
        begin
```

```
            // новый узел после pre, перед curr
            node^.next:=pre^.next;
            pre^.next:=node;
```

```
        end;
```

```
Edit1.text:='';
```

```
Edit2.text:='';
```

```
Edit1.SetFocus;
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
var
```

```
    curr: TPStudent; // текущий элемент списка
    n: integer; // длина (кол-во элементов) списка
    st: string; // строковое представление списка
```

```
begin
  n:=0;
  st:='';
  curr:=head;
  while curr <> NIL do
    begin
      n:=n+1;
      st:=st+curr^.f_name+' '+curr^.l_name+#13;
      curr:=curr^.next;
    end;
  if n <> 0
  then ShowMessage('Список: '+#13+st)
  else ShowMessage('В списке нет элементов.');
```

end;

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  head:=NIL;
```

end;

```
end.
```

Процедура `TForm1.Button1Click` создает динамическую переменную-запись, присваивает ее полям значения, соответствующие содержимому полей ввода диалогового окна (рис. 8.9), находит подходящее место для узла и добавляет этот узел в список, корректируя при этом значение указателя узла `next`, после которого должен быть помещен новый узел.

Вывод списка выполняет процедура `TForm1.Button2Click`, которая запускается нажатием кнопки **Показать**. После запуска программы и ввода нескольких фамилий, например, в такой последовательности: Иванов, Яковлев, Алексеев, Петров, список выглядит так, как показано на рис. 8.10.

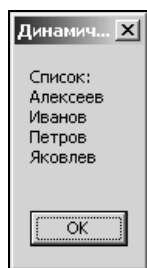


Рис. 8.10. Пример упорядоченного списка, сформированного программой

Удаление элемента из списка

Для того чтобы удалить узел, необходимо скорректировать значение указателя узла, который находится перед удаляемым узлом (рис. 8.11).

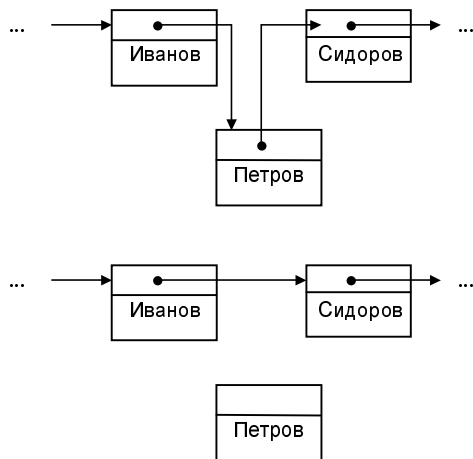


Рис. 8.11. Удаление элемента из списка

Поскольку узел является динамической переменной, то после исключения узла из списка занимаемая им память должна быть освобождена. Освобождение динамической памяти, или, как иногда говорят, "уничтожение переменной", выполняется вызовом процедуры `dispose`. У процедуры `dispose` один параметр — указатель на динамическую переменную. Память, занимаемая этой динамической переменной, должна быть освобождена. Например, в программе

```
var
  p: ^integer;
begin
  new(p);
  { инструкции программы }
  dispose(p);
end
```

создается динамическая переменная `p`, а затем она уничтожается. Освободившуюся память смогут использовать другие переменные. Если этого не сделать, то возможно из-за недостатка свободной памяти в какой-то момент времени программа не сможет создать очередную динамическую переменную.

Следующая программа позволяет добавлять и удалять узлы упорядоченного списка. Диалоговое окно программы приведено на рис. 8.12.

Процедуры добавления узла в список и вывода списка, а также объявление типа узла списка ничем не отличаются от соответствующих процедур рассмотренной ранее программы **Упорядоченный динамический список 2**, поэтому они здесь не приводятся.

Удаление узла из списка выполняет процедура `TForm1.Button3Click`, которая запускается нажатием кнопки **Удалить** (`Button3`). Текст процедуры приведен в листинге 8.6.

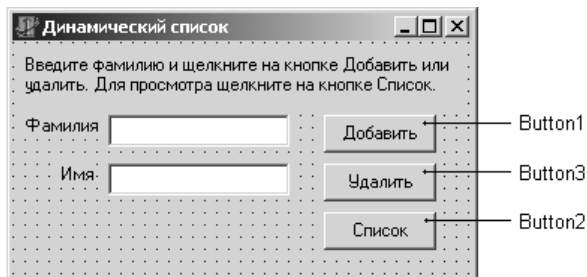


Рис. 8.12. Окно программы **Динамический список**

Листинг 8.6. Удаление узла из динамического списка

```
// щелчок на кнопке Удалить
procedure TForm1.Button3Click(Sender: TObject);
var
    curr:TPStudent; // текущий, проверяемый узел
    pre: TPStudent; // предыдущий узел
    found:boolean; // TRUE – узел, который надо удалить, есть в списке
begin
    if head = NIL then
        begin
            MessageDlg('Список пустой!', mtError, [mbOk], 0);
            Exit;
        end;
    curr := head; // текущий узел – первый узел
    pre := NIL; // предыдущего узла нет
    found := FALSE;

    // найти узел, который надо удалить
    while (curr <> NIL) and (not found) do
        begin
            if (curr^.f_name = Edit1.Text) and (curr^.l_name = Edit2.Text)
                then found:=TRUE // нужный узел найден
            else // к следующему узлу
                begin
                    pre:=curr;
                    curr:=curr^.next;
                end;
        end;
    end;
    if found then
        begin
```

```

// нужный узел найден
if MessageDlg('Узел будет удален из списка!',
              mtWarning, [mbOk, mbCancel], 0) <> mrYes
then Exit;

// удаляем узел
if pre = NIL
then head:=curr^.next      // удаляем первый узел списка
else pre^.next:=curr.next;
Dispose(curr);
MessageDlg('Узел' + #13 +
          'Имя:' + Edit1.Text + #13 +
          'Фамилия:' + Edit2.Text + #13 +
          'удален из списка.',
          mtInformation, [mbOk], 0);

end

else // узла, который надо удалить, в списке нет
MessageDlg('Узел' + #13 +
          'Имя:' + Edit1.Text + #13 +
          'Фамилия:' + Edit2.Text + #13 +
          'в списке не найден.',
          mtError, [mbOk], 0);

Edit1.Text:='';
Edit1.Text:='';
Edit1.SetFocus;

end;

```

Процедура просматривает список от начала, сравнивая содержимое полей текущего узла с содержимым полей ввода диалогового окна. Если их содержимое совпадает, то процедура просит пользователя подтвердить удаление узла. Если пользователь нажатием кнопки **ОК** подтверждает, что узел должен быть удален, то процедура удаляет его. Если узла, который хочет удалить пользователь, в списке нет, то программа выводит сообщение об ошибке.



Введение в объектно-ориентированное программирование

Исторически сложилось так, что программирование возникло и развивалось как *процедурное программирование*, которое предполагает, что основой программы является алгоритм, процедура обработки данных.

Объектно-ориентированное программирование (ООП) — это методика разработки программ, в основе которой лежит понятие объекта, как некоторой структуры, описывающей объект реального мира, его поведение. Задача, решаемая с использованием методики ООП, описывается в терминах объектов и операций над ними, а программа при таком подходе представляет собой набор объектов и связей между ними.

Примечание

Строго говоря, для разработки приложения в Delphi на базе компонентов, предоставляемых средой разработки, знание концепции ООП не является необходимым. Однако материал данной главы будет весьма полезен для более глубокого понимания того, как программа взаимодействует с компонентами, что и почему Delphi добавляет в текст программы.

Класс

Классический язык Pascal позволяет программисту определять свои собственные сложные типы данных — *записи* (records). Язык Object Pascal, поддерживая концепцию объектно-ориентированного программирования, дает возможность определять *классы*. Класс — это сложная структура, включающая, помимо описания данных, описание процедур и функций, которые могут быть выполнены над представителем класса — *объектом*.

Вот пример объявления простого класса:

```
TPerson = class
  private
    fname: string[15];
    faddress: string[35];
  public
    procedure Show;
end;
```

Данные класса называются *полями*, процедуры и функции — *методами*. В приведенном примере TPerson — это имя класса, fname и faddress — имена полей, Show — имя метода.

Примечание

Согласно принятому в Delphi соглашению, имена полей должны начинаться с буквы f (от слова field — поле).

Описание класса помещают в программе в раздел описания типов (type).

Объект

Объекты как представители класса объявляются в программе в разделе var, например:

```
var
  student: TPerson;
  professor: TPerson;
```

Примечание

В Delphi объект — это динамическая структура. Переменная-объект содержит не данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных.

Выделение памяти осуществляется при помощи специального метода класса — *конструктора*, которому обычно присваивают имя Create (создать). Для того чтобы подчеркнуть особую роль и поведение конструктора, в описании класса вместо слова procedure используется слово constructor.

Ниже приведено описание класса TPerson, в состав которого введен конструктор:

```
TPerson = class
  private
    fname: string[15];
    faddress: string[35];
  constructor Create; // конструктор
```

```
public
  procedure show;          // метод
end;
```

Выделение памяти для данных объекта происходит путем присваивания значения результата применения метода-конструктора к типу (классу) объекта. Например, после выполнения инструкции

```
professor := TPerson.Create;
```

выделяется необходимая память для данных объекта `professor`.

Помимо выделения памяти конструктор, как правило, решает задачу присваивания полям объекта начальных значений, т. е. осуществляет инициализацию объекта. Ниже приведен пример реализации конструктора для объекта `TPerson`:

```
constructor TPerson.Create;
begin
  fname := '';
  faddress := '';
end;
```

Реализация конструктора несколько необычна. Во-первых, в теле конструктора нет привычных инструкций `New`, обеспечивающих выделение динамической памяти (всю необходимую работу по выделению памяти выполняет компилятор). Во-вторых, формально, конструктор не возвращает значения, хотя в программе обращение к конструктору осуществляется как к методу-функции.

После объявления и инициализации объект можно использовать, например, установить значение поля объекта. Доступ к полю объекта осуществляется указанием имени объекта и имени поля, которые отделяются друг от друга точкой. Хотя объект является ссылкой, однако, правило доступа к данным с помощью ссылки, согласно которому после имени переменной, являющейся ссылкой, надо ставить значок `^`, на объекты не распространяется.

Например, для доступа к полю `fname` объекта `professor` вместо

```
professor.fname
```

надо писать

```
professor.fname
```

Очевидно, что такой способ доступа к полям объекта более естественен.

Если в программе какой-либо объект больше не используется, то можно освободить память, занимаемую полями данного объекта. Для выполнения этого действия используют *метод-деструктор* `Free`. Например, для того чтобы освободить память, занимаемую полями объекта `professor`, достаточно записать

```
professor.Free;
```

Метод

Методы класса (процедуры и функции, объявление которых включено в описание класса) выполняют действия над объектами класса. Для того чтобы метод был выполнен, необходимо указать имя объекта и имя метода, отделив одно имя от другого точкой. Например, инструкция

```
professor.Show;
```

вызывает применение метода Show к объекту professor. Фактически инструкция применения метода к объекту — это специфический способ записи инструкции вызова процедуры.

Методы класса определяются в программе точно так же, как и обычные процедуры и функции, за исключением того, что имя процедуры или функции, являющейся методом, состоит из двух частей: имени класса, к которому принадлежит метод, и имени метода. Имя класса от имени метода отделяется точкой.

Ниже приведен пример определения метода Show класса TPerson:

```
// метод Show класса TPerson
procedure TPerson.Show;
begin
    ShowMessage('Имя:' + fname + #13 + 'Адрес:' + faddress);
end;
```

Примечание

В инструкциях метода доступ к полям объекта осуществляется без указания имени объекта.

Инкапсуляция и свойства объекта

Под *инкапсуляцией* понимается скрытие полей объекта с целью обеспечения доступа к ним только посредством методов класса.

В Object Pascal ограничение доступа к полям объекта реализуется при помощи свойств объекта. Свойство объекта характеризуется полем, сохраняющим значение свойства, и двумя методами, обеспечивающими доступ к полю свойства. Метод установки значения свойства называется *методом записи свойства* (write), а метод получения значения свойства — *методом чтения свойства* (read).

В описании класса перед именем свойства записывают слово `property` (свойство). После имени свойства указывается его тип, затем — имена методов, обеспечивающих доступ к значению свойства. После слова `read` указы-

вается имя метода, обеспечивающего чтение свойства, после слова `write` — имя метода, отвечающего за запись свойства.

Ниже приведен пример описания класса `TPerson`, содержащего два свойства: `Name` и `Address`.

```
type
  TName = string[15];
  TAddress = string[35];

  TPerson=class // класс
  private
    FName: TName;           // значение свойства Name
    FAddress: TAddress;      // значение свойства Address
    Constructor Create (Name:TName);
    Procedure Show;
    Function GetName: TName;
    Function GetAddress: TAddress;
    Procedure SetAddress (NewAddress:TAddress);
  public
    Property Name: Tname      // свойство Name
      read GetName;           // доступно только для чтения
    Property Address: TAddress // свойство Address
      read GetAddress         // доступно для чтения
      write SetAddress;        // и записи
end;
```

В программе для установки значения свойства не нужно записывать инструкцию применения к объекту метода установки значения свойства, а надо записать обычную инструкцию присваивания значения свойству. Например, чтобы присвоить значение свойству `Address` объекта `student`, достаточно записать

```
student.Address := 'С.Петербург, ул.Садовая 21, кв.3';
```

Компилятор перетранслирует приведенную инструкцию присваивания значения свойству в инструкцию вызова метода

```
student.SetAddress('С.Петербург, ул.Садовая 21, кв.3');
```

Внешне применение свойств в программе ничем не отличается от использования полей объекта. Однако между свойством и полем объекта существует принципиальное отличие: при присвоении и чтении значения свойства автоматически вызывается процедура, которая выполняет некоторую работу.

В программе на методы свойства можно возложить некоторые дополнительные задачи. Например, с помощью метода можно проверить корректность

присваиваемых свойству значений, установить значения других полей, логически связанных со свойством, вызвать вспомогательную процедуру.

Оформление данных объекта как свойства позволяет ограничить доступ к полям, хранящим значения свойств объекта, например, можно разрешить только чтение. Для того чтобы инструкции программы не могли изменить значение свойства, в описании свойства надо указать лишь имя метода чтения. Попытка присвоить значение свойству, предназначенному только для чтения, вызывает ошибку времени компиляции. В приведенном выше описании класса `TPerson` свойство `Name` доступно только для чтения, а свойство `Address` — для чтения и записи.

Установить значение свойства, защищенного от записи, можно во время инициализации объекта. Ниже приведены методы класса `TPerson`, обеспечивающие создание объекта класса `TPerson` и доступ к его свойствам.

```
// конструктор объекта TPerson
Constructor TPerson.Create (Name:TName);
begin
    FName:=Name;
end;

// метод получения значения свойства Name
Function TPerson.GetName;
begin
    Result:=FName;
end;

// метод получения значения свойства Address
function TPerson.GetAddress;
begin
    Result:=FAddress;
end;

// метод изменения значения свойства Address
Procedure TPerson.SetAddress (NewAddress:TAddress);
begin
    if FAddress = ''
        then FAddress := NewAddress;
end;
```

Приведенный конструктор объекта `TPerson` создает объект и устанавливает значение поля `FName`, определяющего значение свойства `Name`.

Инструкции программы, обеспечивающие создание объекта класса `TPerson` и установку его свойства, могут быть, например, такими:

```
student := TPerson.Create('Иванов');
student.Address := 'ул. Садовая, д.3, кв.25';
```

Наследование

Концепция объектно-ориентированного программирования предполагает возможность определять новые классы посредством добавления полей, свойств и методов к уже существующим классам. Такой механизм получения новых классов называется *порождением*. При этом новый, порожденный класс (потомок) наследует свойства и методы своего базового, родительского класса.

В объявлении класса-потомка указывается класс родителя. Например, класс `TEmployee` (сотрудник) может быть порожден от рассмотренного выше класса `TPerson` путем добавления поля `FDepartment` (отдел). Объявление класса `TEmployee` в этом случае может выглядеть так:

```
TEmployee = class(TPerson)
    FDepartment: integer;    // номер отдела
    constructor Create(Name:TName; Dep:integer);
end;
```

Заключенное в скобки имя класса `TPerson` показывает, что класс `TEmployee` является производным от класса `TPerson`. В свою очередь, класс `TPerson` является базовым для класса `TEmployee`.

Класс `TEmployee` должен иметь свой собственный конструктор, обеспечивающий инициализацию класса-родителя и своих полей. Вот пример реализации конструктора класса `TEmployee`:

```
constructor TEmployee.Create(Name:Tname;Dep:integer);
begin
    inherited Create(Name);
    FDepartment:=Dep;
end;
```

В приведенном примере директивой `inherited` вызывается конструктор родительского класса. После этого присваивается значение полю класса-потомка.

После создания объекта производного класса в программе можно использовать поля и методы родительского класса. Ниже приведен фрагмент программы, демонстрирующий эту возможность:

```
engineer := TEmployee.Create('Сидоров',413);
engineer.address := 'ул.Блохина, д.8, кв.10';
```

Первая инструкция создает объект типа `TEmployee`, вторая — устанавливает значение свойства, которое относится к родительскому классу.

Директивы *Protected* и *Private*

Помимо объявления элементов класса (полей, методов, свойств) описание класса, как правило, содержит директивы *Protected* (защищенный) и *Private* (закрытый), которые устанавливают степень видимости элементов класса в программе.

Элементы класса, объявленные в секции *Protected*, доступны только в порожденных от него классах. Область видимости элементов класса этой секции не ограничивается модулем, в котором находится описание класса. Обычно в секцию *Protected* помещают описание методов класса.

Элементы класса, объявленные в секции *Private*, видимы только внутри модуля. Эти элементы не доступны за пределами модуля, даже в производных классах. Обычно в секцию *Private* помещают описание полей класса, а методы, обеспечивающие доступ к этим полям, помещают в секцию *Protected*.

Ниже приведено описание класса *TPerson*, в которое включены директивы управления доступом.

```
TPerson = class
  private
    FName: TName;           // значение свойства Name
    FAddress: TAddress;      // значение свойства Address
  protected
    Constructor Create (Name:TName);
    Function GetName: TName;
    Function GetAddress: TAddress;
    Procedure SetAddress (NewAddress:TAddress);
    Property Name: TName
      read GetName;
    Property Address: TAddress
      read GetAddress
      write SetAddress;
end;
```

Примечание

Иногда нужно полностью скрыть элементы класса. В этом случае определение класса следует поместить в отдельный модуль, а в программу, которая использует объекты этого класса, поместить ссылку на модуль.

Полиморфизм и виртуальные методы

Полиморфизм — это возможность использовать одинаковые имена для методов, входящих в различные классы. Концепция полиморфизма обеспечивает в случае применения метода к объекту использование именно того метода, который соответствует классу объекта.

Пусть определены три класса, один из которых является базовым для двух других:

type

```
// базовый класс
TPerson = class
    fname: string; // имя
    constructor Create(name:string);
    function info: string; virtual;
end;

// производный от TPerson
TStud = class(TPerson)
    fgr:integer; // номер учебной группы
    constructor Create(name:string;gr:integer);
    function info: string; override;
end;

// производный от TPerson
TProf = class(TPerson)
    fdep:string; // название кафедры
    constructor Create(name:string;dep:string);
    function info: string; override;
end;
```

В каждом из этих классов определен метод `info`. В базовом классе при помощи директивы `virtual` метод `info` объявлен виртуальным. Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим собственным. В каждом дочернем классе определен свой метод `info`, который замещает соответствующий метод родительского класса (метод порожденного класса, замещающий виртуальный метод родительского класса, помечается директивой `override`).

Ниже приведено определение метода `info` для каждого класса.

```
function TPerson.info:string;
begin
    result := '';
end;
```



```

function TStud.info:string;
begin
    result := fname + ' rp.' + IntToStr(fgr);
end;

function TProf.info:string;
begin
    result := fname + ' каф.' + fdep;
end;

```

Так как оба класса порождены от одного и того же базового, объявить список студентов и преподавателей можно так (здесь следует вспомнить, что объект — это указатель):

```
list: array[1..SZL] of TPerson;
```

Объявить подобным образом список можно потому, что Object Pascal позволяет указателю на родительский класс присвоить значение указателя на дочерний класс. Поэтому элементами массива `list` могут быть как объекты класса `TStud`, так и объекты класса `TProf`.

Вывести список студентов и преподавателей можно применением метода `info` к элементам массива. Например, так:

```

st := '';
for i:=1 to SZL do      // SZL — размер массива-списка
    if list[i] <> NIL
        then st := st + list[i].Info + #13;
ShowMessage(st);

```

Во время работы программы каждый элемент массива может содержать как объект типа `TStud`, так и объект типа `TProf`. Концепция полиморфизма обеспечивает применение к объекту именно того метода, который соответствует типу объекта.

Следующая программа, используя рассмотренные выше объявления классов `TPerson`, `TStud` и `TProf`, формирует и выводит список студентов и преподавателей. Текст программы приведен в листинге 9.1, а диалоговое окно — на рис. 9.1.



Рис. 9.1. Диалоговое окно программы **Полиморфизм**

Листинг 9.1. Демонстрация полиморфизма

```
unit polimor_;

interface

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

type
    TForm1 = class(TForm)
        Edit1: TEdit;
        Edit2: TEdit;
        GroupBox1: TGroupBox;
        RadioButton1: TRadioButton;
        RadioButton2: TRadioButton;
        Label1: TLabel;
        Label2: TLabel;
        Button1: TButton;
        Button2: TButton;
        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

type
    // базовый класс
    TPerson = class
        fName: string; // имя
        constructor Create(name:string);
        function info:string; virtual;
    end;

    // класс Студент
    TStud = class(TPerson)
        fGr:integer; // номер группы
        constructor Create(name:string;gr:integer);
        function info:string; override;
    end;

    // класс Преподаватель
    TProf = class(TPerson)
```

```
fdep:string; // название кафедры
constructor Create(name:string;dep:string);
function info:string; override;
end;

const
    SZL = 10; // размер списка
var
    Form1: TForm1;
    List: array[1..SZL] of TPerson; // список
    n:integer = 0;                // кол-во людей в списке

implementation

{$R *.DFM}

constructor TPerson.Create(name:string);
begin
    fName := name;
end;

constructor TStud.Create(name:string;gr:integer);
begin
    inherited create(name); // вызвать конструктор базового класса
    fGr := gr;
end;

constructor TProf.create(name:string; dep:string);
begin
    inherited create(name); // вызвать конструктор базового класса
    fDep := dep;
end;

function TPerson.Info:string;
begin
    result := fName;
end;

function TStud.Info:string;
begin
    result := fName + ' rp.' + IntToStr(fGr);
end;

function TProf.Info:string;
begin
    result := fName + ' каф.' + fDep;
end;
```

```

// щелчок на кнопке Добавить
procedure TForm1.Button1Click(Sender: TObject);
begin
    if n < SZL then
    begin
        // добавить объект в список
        n:=n+1;
        if RadioButton1.Checked
        then // создадим объект TStud
            List[n]:=TStud.Create(Edit1.Text, StrToInt(Edit2.Text))
        else // создать объект TProf
            List[n]:=TProf.Create(Edit1.Text, Edit2.Text);

        // очистить поля ввода
        Edit1.Text := '';
        Edit2.Text := '';
        Edit1.SetFocus; // курсор в поле Фамилия
    end
    else ShowMessage('Список заполнен!');
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    i:integer; // индекс
    st:string; // список
begin
    for i:=1 to SZL do
        if list[i] <> NIL then st:=st + list[i].info + #13;
        ShowMessage('Список'+#13+st);
    end;
end.

```

Процедура `TForm1.Button1Click`, которая запускается нажатием кнопки **Добавить** (`Button1`), создает объект `list[n]` класса `TStud` или `TProf`. Класс создаваемого объекта определяется состоянием переключателя `RadioButton`. Установка переключателя в положение **студент** (`RadioButton1`) определяет класс `TStud`, а в положение **преподаватель** (`RadioButton2`) — класс `TProf`.

Процедура `TForm1.Button2Click`, которая запускается нажатием кнопки **Список** (`Button2`), применяя метод `info` к каждому объекту списка (элементу массива), формирует строку, представляющую собой весь список.

Классы и объекты Delphi

Для реализации интерфейса Delphi использует библиотеку классов, которая содержит большое количество разнообразных классов, поддерживающих форму и различные компоненты формы (командные кнопки, поля редактирования и т. д.).

Во время проектирования формы приложения Delphi автоматически добавляет в текст программы необходимые объекты. Если сразу после запуска Delphi просмотреть содержимое окна редактора кода, то там можно обнаружить следующие строки:

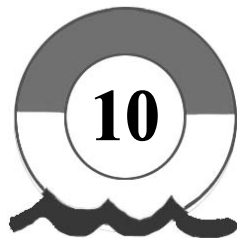
```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1
```

Это описание класса исходной, пустой формы приложения и объявление объекта — формы приложения.

Когда программист, добавляя необходимые компоненты, создает форму, Delphi формирует описание класса формы. Когда программист создает функцию обработки события формы или ее компонента, Delphi добавляет объявление метода в описание класса формы приложения.

Помимо классов визуальных компонентов в библиотеку классов входят и другие классы, например класс обработки исключительных ситуаций (ошибок), класс общего управления приложением и др. Рассмотрение этих классов в задачу данной книги не входит.

ГЛАВА 10



Графические возможности Delphi

Delphi позволяет программисту разрабатывать программы, которые могут выводить графику: схемы, чертежи, иллюстрации.

Программа выводит графику на *поверхность* объекта (формы или компонента Image). Поверхности объекта соответствует свойство Canvas. Для того чтобы вывести на поверхность объекта графический элемент (прямую линию, окружность, прямоугольник и т. д.), необходимо применить к свойству Canvas этого объекта соответствующий метод. Например, инструкция `Form1.Canvas.Rectangle(10,10,100,100)` вычерчивает в окне программы прямоугольник.

Холст

Как было сказано выше, поверхности, на которую программа может вывести графику, соответствует свойство Canvas. В свою очередь, свойство Canvas — это объект типа TCanvas. Методы этого типа обеспечивают вывод графических примитивов (точек, линий, окружностей, прямоугольников и т. д.), а свойства позволяют задать характеристики выводимых графических примитивов: цвет, толщину и стиль линий; цвет и вид заполнения областей; характеристики шрифта при выводе текстовой информации.

Методы вывода графических примитивов рассматривают свойство Canvas как некоторый абстрактный *холст*, на котором они могут *рисовать* (Canvas переводится как "поверхность", "холст для рисования"). Холст состоит из отдельных точек — пикселей. Положение пикселя характеризуется его горизонтальной (X) и вертикальной (Y) координатами. Левый верхний пиксел имеет координаты (0, 0). Координаты возрастают сверху вниз и слева направо (рис. 10.1). Значения координат правой нижней точки холста зависят от размера холста.

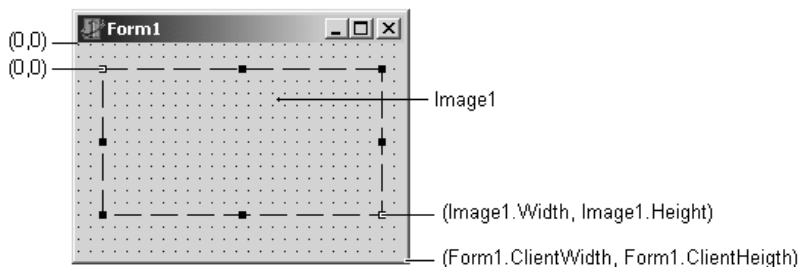


Рис. 10.1. Координаты точек холста

Размер холста можно получить, обратившись к свойствам `Height` и `Width` области иллюстрации (`Image`) или к свойствам формы: `ClientHeight` и `ClientWidth`.

Карандаш и кисть

Художник в своей работе использует карандаши и кисти. Методы, обеспечивающие вычерчивание на поверхности холста графических примитивов, тоже используют *карандаш* и *кисть*. Карандаш применяется для вычерчивания линий и контуров, а кисть — для закрашивания областей, ограниченных контурами.

Карандашу и кисти, используемым для вывода графики на холсте, соответствуют свойства `Pen` (карандаш) и `Brush` (кисть), которые представляют собой объекты типа `TPen` и `TBrush`, соответственно. Значения свойств этих объектов определяют вид выводимых графических элементов.

Карандаш

Карандаш используется для вычерчивания точек, линий, контуров геометрических фигур: прямоугольников, окружностей, эллипсов, дуг и др. Вид линии, которую оставляет карандаш на поверхности холста, определяют свойства объекта `TPen`, которые перечислены в табл. 10.1.

Таблица 10.1. Свойства объекта `TPen`

Свойство	Определяет
<code>Color</code>	Цвет линии
<code>Width</code>	Толщину линии
<code>Style</code>	Вид линии
<code>Mode</code>	Режим отображения

Свойство `Color` задает цвет линии вычерчиваемой карандашом. В табл. 10.2 перечислены именованные константы (тип `TColor`), которые можно использовать в качестве значения свойства `Color`.

Таблица 10.2

Константа	Цвет	Константа	Цвет
<code>clBlack</code>	Черный	<code>ClSilver</code>	Серебристый
<code>clMaroon</code>	Каштановый	<code>ClRed</code>	Красный
<code>clGreen</code>	Зеленый	<code>ClLime</code>	Салатный
<code>clOlive</code>	Оливковый	<code>ClBlue</code>	Синий
<code>clNavy</code>	Темно-синий	<code>ClFuchsia</code>	Ярко-розовый
<code>clPurple</code>	Розовый	<code>ClAqua</code>	Бирюзовый
<code>clTeal</code>	Зелено-голубой	<code>ClWhite</code>	Белый
<code>clGray</code>	Серый		

Свойство `Width` задает толщину линии (в пикселах). Например, инструкция `Canvas.Pen.Width:=2` устанавливает толщину линии в 2 пиксела.

Свойство `Style` определяет вид линии, которая может быть непрерывной или прерывистой, состоящей из штрихов различной длины. В табл. 10.3 перечислены именованные константы, позволяющие задать стиль линии.

Таблица 10.3

Константа	Вид линии
<code>psSolid</code>	Сплошная линия
<code>psDash</code>	Пунктирная линия, длинные штрихи
<code>psDot</code>	Пунктирная линия, короткие штрихи
<code>psDashDot</code>	Пунктирная линия, чередование длинного и короткого штрихов
<code>psDashDotDot</code>	Пунктирная линия, чередование одного длинного и двух коротких штрихов
<code>psClear</code>	Линия не отображается (используется, если не надо изображать границу области, например, прямоугольника)

Примечание

Толщина пунктирной линии не может быть больше 1. Если значение свойства `Pen.Width` больше единицы, то пунктирная линия будет выведена как сплошная.

Свойство `Mode` определяет, как будет формироваться цвет точек линии в зависимости от цвета точек холста, через которые эта линия прочерчивается. По умолчанию вся линия вычерчивается цветом, определяемым значением свойства `Pen.Color`.

Однако программист может задать инверсный цвет линии по отношению к цвету фона. Это гарантирует, что независимо от цвета фона все участки линии будут видны, даже в том случае, если цвет линии и цвет фона совпадают.

В табл. 10.4 перечислены некоторые константы, которые можно использовать в качестве значения свойства `Pen.Mode`.

Таблица 10.4

Константа	Цвет линии
<code>pmBlack</code>	Черный, не зависит от значения свойства <code>Pen.Color</code>
<code>pmWhite</code>	Белый, не зависит от значения свойства <code>Pen.Color</code>
<code>pmCopy</code>	Цвет линии определяется значением свойства <code>Pen.Color</code>
<code>pmNotCopy</code>	Цвет линии является инверсным по отношению к значению свойства <code>Pen.Color</code>
<code>pmNot</code>	Цвет точки линии определяется как инверсный по отношению к цвету точки холста, в которую выводится точка линии

Кисть

Кисть (`Canvas.Brush`) используется методами, обеспечивающими вычерчивание замкнутых областей, например геометрических фигур, для заливки (закраски) этих областей. Кисть, как объект, обладает двумя свойствами, перечисленными в табл. 10.5.

Таблица 10.5

Свойство	Определяет
<code>Color</code>	Цвет закрашивания замкнутой области
<code>Style</code>	Стиль (тип) заполнения области

Область внутри контура может быть закрашена или заштрихована. В первом случае область полностью перекрывает фон, а во втором — сквозь не заштрихованные участки области будет виден фон.

В качестве значения свойства `Color` можно использовать любую из констант типа `TColor` (см. список констант для свойства `Pen.Color` в табл. 10.2).

Константы, позволяющие задать стиль заполнения области, приведены в табл. 10.6.

Таблица 10.6

Константа	Тип заполнения (заливки) области
bsSolid	Сплошная заливка
bsClear	Область не закрашивается
bsHorizontal	Горизонтальная штриховка
bsVertical	Вертикальная штриховка
bsFDDiagonal	Диагональная штриховка с наклоном линий вперед
bsBDDiagonal	Диагональная штриховка с наклоном линий назад
bsCross	Горизонтально-вертикальная штриховка, в клетку
bsDiagCross	Диагональная штриховка, в клетку

В качестве примера в листинге 10.1 приведена программа **Стили заполнения областей**, которая в окно (рис. 10.2) выводит восемь прямоугольников, закрашенных черным цветом с использованием разных стилей.

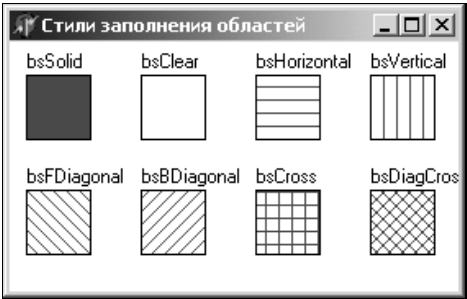


Рис. 10.2. Окно программы Стили заполнения областей

Листинг 10.1. Стили заполнения областей

```
unit brustyle_;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
  end;
```

```
private
    { Private declarations}
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormPaint(Sender: TObject);
const
    bsName: array[1..8] of string =
        ('bsSolid','bsClear','bsHorizontal',
         'bsVertical','bsFDiagonal','bsBDiagonal',
         'bsCross','bsDiagCross');
var
    x,y: integer;    // координаты левого верхнего угла прямоугольника
    w,h: integer;    // ширина и высота прямоугольника
    bs: TBrushStyle; // стиль заполнения области
    k: integer;      // номер стиля заполнения
    i,j: integer;

begin
    w:=40; h:=40; // размер области (прямоугольника)
    y:=20;

    for i:=1 to 2 do
    begin
        x:=10;
        for j:=1 to 4 do
        begin
            k:=j+(i-1)*4; // номер стиля заполнения
            case k of
                1: bs := bsSolid;
                2: bs := bsClear;
                3: bs := bsHorizontal;
                4: bs := bsVertical;
                5: bs := bsFDiagonal;
                6: bs := bsBDiagonal;
                7: bs := bsCross;
                8: bs := bsDiagCross;
            end;
        end;
    end;
```

```

// вывод прямоугольника
Canvas.Brush.Color := clGreen; // цвет закрашки – зеленый
Canvas.Brush.Style := bs;      // стиль закрашки
Canvas.Rectangle(x, y, x+w, y+h);

// вывод названия стиля
Canvas.Brush.Style := bsClear;
Canvas.TextOut(x, y-15, bsName[k]); // вывод названия стиля
x := x+w+30;

end;
y := y+h+30;
end;
end;
end.

```

Вывод текста

Для вывода текста на поверхность графического объекта используется метод `TextOut`. Инструкция вызова метода `TextOut` в общем виде выглядит следующим образом:

`Объект.Canvas.TextOut(x, y, Текст)`

где:

Объект — имя объекта, на поверхность которого выводится текст;

x, *y* — координаты точки графической поверхности, от которой выполняется вывод текста (рис. 10.3);

Текст — переменная или константа символьного типа, значение которой определяет выводимый методом текст.

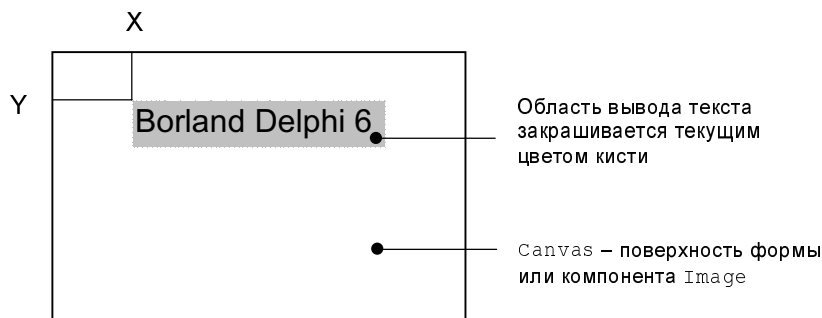


Рис. 10.3. Координаты области вывода текста

Шрифт, который используется для вывода текста, определяется значением свойства `Font` соответствующего объекта `Canvas`. Свойство `Font` представляет собой объект типа `TFont`. В табл. 10.7 перечислены свойства объекта `TFont`, позволяющие задать характеристики шрифта, используемого методами `TextOut` и `TextRect` для вывода текста.

Таблица 10.7. Свойства `TFont`

Свойство	Определяет
Name	Используемый шрифт. В качестве значения следует использовать название шрифта, например <code>Arial</code>
Size	Размер шрифта в пунктах (<code>points</code>). Пункт — это единица измерения размера шрифта, используемая в полиграфии. Один пункт равен 1/72 дюйма
Style	Стиль начертания символов. Может быть: нормальным, полужирным, курсивным, подчеркнутым, перечеркнутым. Стиль задается при помощи следующих констант: <code>fsBold</code> (полужирный), <code>fsItalic</code> (курсив), <code>fsUnderline</code> (подчеркнутый), <code>fsStrikeOut</code> (перечеркнутый). Свойство <code>Style</code> является множеством, что позволяет комбинировать необходимые стили. Например, инструкция программы, устанавливающая стиль "полужирный курсив", выглядит так: <pre>Объект.Canvas.Font:=[fsBold, fsItalic]</pre>
Color	Цвет символов. В качестве значения можно использовать константу типа <code>TColor</code>

Внимание!

Область вывода текста закрашивается текущим цветом кисти. Поэтому перед выводом текста свойству `Brush.Color` нужно присвоить значение `bsClear` или задать цвет кисти, совпадающий с цветом поверхности, на которую выводится текст.

Следующий фрагмент программы демонстрирует использование функции `TextOut` для вывода текста на поверхность формы:

```
with Form1.Canvas do
begin
    // установить характеристики шрифта
    Font.Name := 'Tahoma';
    Font.Size := 20;
    Font.Style := [fsItalic, fsBold];
```

```
Brush.Style := bsClear; // область вывода текста не закрашивается  
  
TextOut(10, 10, 'Borland Delphi 6');  
end;
```

После вывода текста методом `TextOut` указатель вывода (карандаш) перемещается в правый верхний угол области вывода текста.

Иногда требуется вывести какой-либо текст после сообщения, длина которого во время разработки программы неизвестна. Например, это может быть слово "руб." после значения числа, записанного прописью. В этом случае необходимо знать координаты правой границы уже выведенного текста. Координаты правой границы текста, выведенного методом `TextOut`, можно получить, обратившись к свойству `PenPos`.

Следующий фрагмент программы демонстрирует возможность вывода строки текста при помощи двух инструкций `TextOut`.

```
with Form1.Canvas do  
begin  
    TextOut(10, 10, 'Borland ');  
    TextOut(PenPos.X, PenPos.Y, 'Delphi 6');  
end;
```

Методы вычерчивания графических примитивов

Любая картинка, чертеж, схема могут рассматриваться как совокупность графических *примитивов*: точек, линий, окружностей, дуг и др. Таким образом, для того чтобы на экране появилась нужная картинка, программа должна обеспечить вычерчивание (вывод) графических примитивов, составляющих эту картинку.

Вычерчивание графических примитивов на поверхности компонента (формы или области вывода иллюстрации) осуществляется применением соответствующих методов к свойству `Canvas` этого компонента.

Линия

Вычерчивание прямой линии осуществляет метод `LineTo`, инструкция вызова которого в общем виде выглядит следующим образом:

```
Компонент.Canvas.LineTo(x, y)
```

Метод `LineTo` вычерчивает прямую линию от текущей позиции карандаша в точку с координатами, указанными при вызове метода.

Начальную точку линии можно задать, переместив карандаш в нужную точку графической поверхности. Сделать это можно при помощи метода `MoveTo`, указав в качестве параметров координаты нового положения карандаша.

Вид линии (цвет, толщина и стиль) определяется значениями свойств объекта `Pen` графической поверхности, на которой вычерчивается линия.

Довольно часто результаты расчетов удобно представить в виде графика. Для большей информативности и наглядности графики изображают на фоне координатных осей и оцифрованной сетки. В листинге 10.2 приведен текст программы, которая на поверхность формы выводит координатные оси и оцифрованную сетку (рис. 10.4).



Рис. 10.4. Форма приложения Координатная сетка

Листинг 10.2. Оси координат и оцифрованная сетка

```
unit grid_;

interface

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

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormPaint(Sender: TObject);
var
    x0,y0:integer; // координаты начала координатных осей
    dx,dy:integer; // шаг координатной сетки (в пикселах)
    h,w:integer;   // высота и ширина области вывода координатной сетки
    x,y:integer;

    lx,ly:real;    // метки (оцифровка) линий сетки по X и Y
    dlx,dly:real;  // шаг меток (оцифровки) линий сетки по X и Y
    cross:integer; // счетчик неоцифрованных линий сетки
    dcross:integer; // количество неоцифрованных линий между оцифрованными
begin
    x0:=30; y0:=220; // оси начинаются в точке (40,250)
    dx:=40; dy:=40; // шаг координатной сетки 40 пикселей
    dcross:=1;      // помечать линии сетки X: 1 – каждую;
                    //                                     2 – через одну;
                    //                                     3 – через две;

    dlx:=0.5;       // шаг меток оси X
    dly:=1.0;       // шаг меток оси Y, метками будут: 1, 2, 3 и т. д.

    h:=200;
    w:=300;

    with form1.Canvas do
    begin
        cross:=dcross;
        MoveTo(x0,y0); LineTo(x0,y0-h); // ось X
        MoveTo(x0,y0); LineTo(x0+w,y0); // ось Y

        // засечки, сетка и оцифровка по оси X
        x:=x0+dx;
        lx:=dlx;
        repeat
            MoveTo(x,y0-3);LineTo(x,y0+3); // засечка
            cross:=cross-1;
            if cross = 0 then // оцифровка
            begin
                TextOut(x-8,y0+5,FloatToStr(lx));
                cross:=dcross;
            end;
        until x>=x0+w;
    end;

```



```

    Pen.Style:=psDot;
    MoveTo(x,y0-3);LineTo(x,y0-h);  // линия сетки
    Pen.Style:=psSolid;
    lx:=lx+dlx;
    x:=x+dx;
until (x>x0+w);

// засечки, сетка и оцифровка по оси Y
y:=y0-dy;
ly:=dly;
repeat
    MoveTo(x0-3,y);LineTo(x0+3,y);    // засечка
    TextOut(x0-20,y,FloatToStr(ly));  // оцифровка
    Pen.Style:=psDot;
    MoveTo(x0+3,y); LineTo(x0+w,y);    // линия сетки
    Pen.Style:=psSolid;
    y:=y-dy;
    ly:=ly+dly;
until (y<y0-h);
end;
end;
end.
```

Особенность приведенной программы заключается в том, что она позволяет задавать шаг сетки и оцифровку. Кроме того, программа дает возможность оцифровывать не каждую линию сетки оси x , а через одну, две, три и т. д. Сделано это для того, чтобы предотвратить возможные наложения изображений чисел оцифровки друг на друга в случае, если эти числа состоят из нескольких цифр.

Ломаная линия

Метод `Polyline` вычерчивает ломаную линию. В качестве параметра метод получает массив типа `TPoint`. Каждый элемент массива представляет собой запись, поля (x, y) которой содержат координаты точки перегиба ломаной. Метод `Polyline` вычерчивает ломаную линию, последовательно соединяя прямыми точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д.

В качестве примера использования метода `Polyline` в листинге 10.3 приведена процедура, которая выводит график изменения некоторой величины. Предполагается, что исходные данные находятся в доступном процедуре массиве `Data` (тип `Integer`).

Листинг 10.3. Построение графика с использованием метода Polyline

```

procedure TForm1.Button1Click(Sender: TObject);

var
  gr: array[1..50] of TPoint; // график — ломаная линия
  x0,y0: integer; // координаты точки начала координат
  dx,dy: integer; // шаг координатной сетки по осям X и Y

  i: integer;

begin
  x0 := 10; y0 := 200;
  dx :=5; dy := 5;
  // заполним массив gr
  for i:=1 to 50 do
    begin
      gr[i].x := x0 + (i-1)*dx;
      gr[i].y := y0 - Data[i]*dy;
    end;

  // строим график
  with form1.Canvas do
    begin
      MoveTo(x0,y0); LineTo(x0,10); // ось Y
      MoveTo(x0,y0); LineTo(200,y0); // ось X
      Polyline(gr); // график
    end;
end;

```

Метод Polyline можно использовать для вычерчивания замкнутых контуров. Для этого надо, чтобы первый и последний элементы массива содержали координаты одной и той же точки. В качестве примера использования метода Polyline для вычерчивания замкнутого контура в листинге 10.4 приведена программа, которая на поверхности диалогового окна, в точке нажатия левой кнопки мыши, вычерчивает контур пятиконечной звезды (рис. 10.5).

Листинг 10.4

```

unit Stars_;

interface

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

```

type

```

TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
private
    { Private declarations }
public
    { Public declarations }
end;

```

var

```
Form1: TForm1;
```

implementation

```
{$R *.dfm}
```

```
procedure StarLine(x0,y0,r: integer; Canvas: TCanvas);
```

```
    // x0,y0 – координаты центра звезды
```

```
    // r – радиус звезды
```

var

```
p : array[1..11] of TPoint; // массив координат лучей и впадин
```

```
a: integer; // угол между осью OX и прямой, соединяющей
```

```
    // центр звезды и конец луча или впадину
```

```
i: integer;
```

begin

```
a := 18; // строим от правого гор. луча
```

```
for i:=1 to 10 do
```

```
    begin
```

```
        if (i mod 2 = 0) then
```

```
            begin // впадина
```

```
                p[i].x := x0+Round(r/2*cos(a*2*pi/360));
```

```
                p[i].y:=y0-Round(r/2*sin(a*2*pi/360));
```

```
            end
```

```
        else
```

```
            begin // луч
```

```
                p[i].x:=x0+Round(r*cos(a*2*pi/360));
```

```
                p[i].y:=y0-Round(r*sin(a*2*pi/360));
```

```
            end;
```

```
        a := a+36;
```

```
    end;
```

```
p[11].X := p[1].X; // чтобы замкнуть контур звезды
```

```
p[11].Y := p[1].Y;
```

```

Canvas.Polyline(p); // начертить звезду
end;

// нажатие кнопки мыши
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if Button = mbLeft // нажата левая кнопка?
    then Form1.Canvas.Pen.Color := clRed
    else Form1.Canvas.Pen.Color := clGreen;
    StarLine(x, y, 30, Form1.Canvas);
end;
end.

```

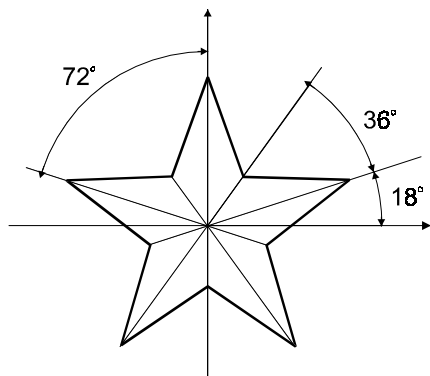


Рис. 10.5. Звезда

Звезду вычерчивает процедура `StarLine`, которая в качестве параметров получает координаты центра звезды и холст, на котором звезда должна быть выведена. Сначала вычисляются координаты концов и впадин звезды, которые записываются в массив `p`. Затем этот массив передается в качестве параметра методу `Polyline`.

Примечание

Обратите внимание, что размер массива `p` на единицу больше, чем количество концов и впадин звезды, и что значения первого и последнего элементов массива совпадают.

Окружность и эллипс

Метод `Ellipse` вычерчивает эллипс или окружность, в зависимости от значений параметров. Инструкция вызова метода в общем виде выглядит следующим образом:

```
Объект.Canvas.Ellipse(x1,y1, x2,y2)
```

где:

Объект — имя объекта (компонента), на поверхности которого выполняется вычерчивание;

$x1, y1, x2, y2$ — координаты прямоугольника, внутри которого вычерчивается эллипс или, если прямоугольник является квадратом, окружность (рис. 10.6).

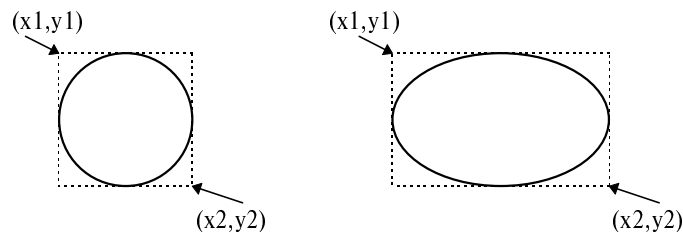


Рис. 10.6. Значения параметров метода `Ellipse` определяют вид геометрической фигуры

Цвет, толщина и стиль линии эллипса определяются значениями свойства `Pen`, а цвет и стиль заливки области внутри эллипса — значениями свойства `Brush` поверхности (`Canvas`), на которую выполняется вывод.

Дуга

Вычерчивание дуги выполняет метод `Arc`, инструкция вызова которого в общем виде выглядит следующим образом:

`Объект.Canvas.Arc(x1, y1, x2, y2, x3, y3, x4, y4)`

где:

$x1, y1, x2, y2$ — параметры, определяющие эллипс (окружность), частью которого является вычерчиваемая дуга;

$x3, y3$ — параметры, определяющие начальную точку дуги;

$x4, y4$ — параметры, определяющие конечную точку дуги.

Начальная (конечная) точка — это точка пересечения границы эллипса и прямой, проведенной из центра эллипса в точку с координатами $x3$ и $y3$ ($x4, y4$). Дуга вычерчивается против часовой стрелки от начальной точки к конечной (рис. 10.7).

Цвет, толщина и стиль линии, которой вычерчивается дуга, определяются значениями свойства `Pen` поверхности (`Canvas`), на которую выполняется вывод.

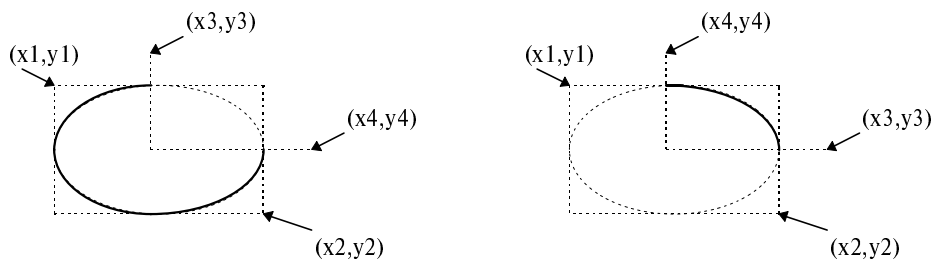


Рис. 10.7. Значения параметров метода Arc определяют дугу как часть эллипса (окружности)

Прямоугольник

Прямоугольник вычерчивается методом `Rectangle`, инструкция вызова которого в общем виде выглядит следующим образом:

```
Объект.Canvas.Rectangle(x1,y1,x2,y2)
```

где:

Объект — имя объекта (компонента), на поверхности которого выполняется вычерчивание;

$x1$, $y1$ и $x2$, $y2$ — координаты левого верхнего и правого нижнего углов прямоугольника.

Метод `RoundRect` тоже вычерчивает прямоугольник, но со скругленными углами. Инструкция вызова метода `RoundRect` выглядит так:

```
Объект.Canvas.RoundRect(x1,y1,x2,y2,x3,y3)
```

где:

$x1$, $y1$, $x2$, $y2$ — параметры, определяющие положение углов прямоугольника, в который вписывается прямоугольник со скругленными углами;

$x3$ и $y3$ — размер эллипса, одна четверть которого используется для вычерчивания скругленного угла (рис. 10.8).

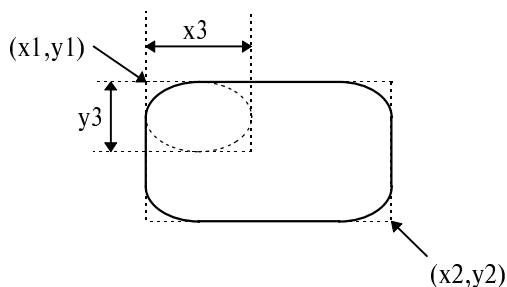


Рис. 10.8. Метод `RoundRect` вычерчивает прямоугольник со скругленными углами

Вид линии контура (цвет, ширина и стиль) определяется значениями свойства `Pen`, а цвет и стиль заливки области внутри прямоугольника — значениями свойства `Brush` поверхности (`Canvas`), на которой прямоугольник вычерчивается.

Есть еще два метода, которые вычерчивают прямоугольник, используя в качестве инструмента только кисть (`Brush`). Метод `FillRect` вычерчивает закрашенный прямоугольник, а метод `FrameRect` — только контур. У каждого из этих методов лишь один параметр — структура типа `TRect`. Поля структуры `TRect` содержат координаты прямоугольной области, они могут быть заполнены при помощи функции `Rect`.

Ниже в качестве примера использования методов `FillRect` и `FrameRect` приведена процедура, которая на поверхности формы вычерчивает прямоугольник с красной заливкой и прямоугольник с зеленым контуром.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    r1, r2: TRect; // координаты углов прямоугольников

begin
    // заполнение полей структуры
    // зададим координаты углов прямоугольников
    r1 := Rect(20,20,60,40);
    r2 := Rect(10,10,40,50);

    with form1.Canvas do
        begin
            Brush.Color := clRed;
            FillRect(r1);           // закрашенный прямоугольник
            Brush.Color := clGreen;
            FrameRect(r2);         // только граница прямоугольника
        end;
end;
```

Многоугольник

Метод `Polygon` вычерчивает многоугольник. В качестве параметра метод получает массив типа `TPoint`. Каждый элемент массива представляет собой запись, поля (`x`, `y`) которой содержат координаты одной вершины многоугольника. Метод `Polygon` вычерчивает многоугольник, последовательно соединяя прямыми линиями точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д. Затем соединяются последняя и первая точки.

Цвет и стиль границы многоугольника определяются значениями свойства `Pen`, а цвет и стиль заливки области, ограниченной линией границы, — зна-

чениями свойства `Brush`, причем область закрашивается с использованием текущего цвета и стиля кисти.

Ниже приведена процедура, которая, используя метод `Polygon`, вычерчивает треугольник:

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    pol: array[1..3] of TPoint; // координаты точек треугольника  
begin  
    pol[1].x := 10;  
    pol[1].y := 50;  
    pol[2].x := 40;  
    pol[2].y := 10;  
    pol[3].x := 70;  
    pol[3].y := 50;  
    Form1.Canvas.Polygon(pol);  
end;
```

Сектор

Метод `Pie` вычерчивает сектор эллипса или круга. Инструкция вызова метода в общем виде выглядит следующим образом:

```
Объект.Canvas.Pie(x1,y1,x2,y2,x3,y3,x4,y4)
```

где:

x_1, y_1, x_2, y_2 — параметры, определяющие эллипс (окружность), частью которого является сектор;

x_3, y_3, x_4, y_4 — параметры, определяющие координаты конечных точек прямых, являющихся границами сектора.

Начальные точки прямых совпадают с центром эллипса (окружности). Сектор вырезается против часовой стрелки от прямой, заданной точкой с координатами (x_3, y_3) , к прямой, заданной точкой с координатами (x_4, y_4) (рис. 10.9).

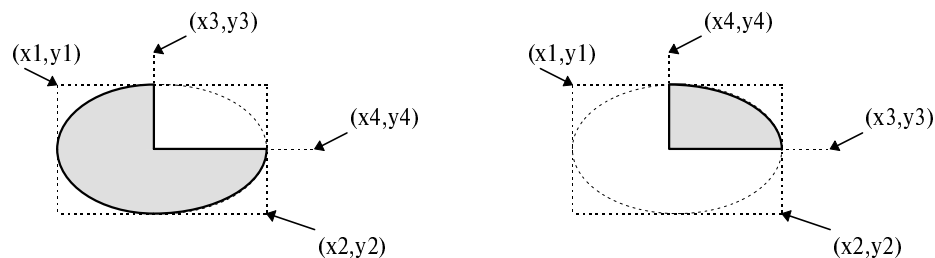


Рис. 10.9. Значения параметров метода `Pie` определяют сектор как часть эллипса (окружности)

Точка

Поверхности, на которую программа может осуществлять вывод графики, соответствует объект `Canvas`. Свойство `Pixels`, представляющее собой двумерный массив типа `TColor`, содержит информацию о цвете каждой точки графической поверхности. Используя свойство `Pixels`, можно задать требуемый цвет для любой точки графической поверхности, т. е. "нарисовать" точку. Например, инструкция

```
Form1.Canvas.Pixels[10,10]:=clRed
```

окрашивает точку поверхности формы в красный цвет.

Размерность массива `Pixels` определяется размером графической поверхности. Размер графической поверхности формы (рабочей области, которую также называют *клиентской*) задается значениями свойств `ClientWidth` и `ClientHeight`, а размер графической поверхности компонента `Image` — значениями свойств `Width` и `Height`.левой верхней точке рабочей области формы соответствует элемент `Pixels[0,0]`, а правой нижней — `Pixels[ClientWidth - 1,ClientHeight - 1]`.

Свойство `Pixels` можно использовать для построения графиков. График строится, как правило, на основе вычислений по формуле. Границы диапазона изменения аргумента функции являются исходными данными. Диапазон изменения значения функции может быть вычислен. На основании этих данных можно вычислить масштаб, позволяющий построить график таким образом, чтобы он занимал всю область формы, предназначенную для вывода графика.

Например, если некоторая функция $f(x)$ может принимать значения от нуля до 1000, и для вывода ее графика используется область формы высотой в 250 пикселей, то масштаб оси Y вычисляется по формуле: $m = 250/1000$. Таким образом, значению $f(x) = 1000$ будет соответствовать точка области с координатой $Y = 0$ ($Y = 250 - f(x) \times m = 250 - 1000 \times (250/1000)$), а значению $f(x) = 70$ — с координатой $Y = 233$ ($Y = 250 - 70 \times (250/1000)$).

Примечание

Обратите внимание на то, что точное значение выражения $250 - 70 \times (250/1000)$ равно 232,5. Но т. к. индексом свойства `Pixels`, которое используется для вывода точки на поверхность `Canvas`, может быть только целое значение, то число 232,5 округляется к ближайшему целому, которым является число 233.

Следующая программа, ее текст приведен в листинге 10.5, используя свойство `Pixels`, выводит график функции $y = 2 \sin(x) e^{x/5}$. Для построения графика используется вся доступная область формы, причем если во время работы программы пользователь изменит размер окна, то график будет выведен заново с учетом реальных размеров окна.

Листинг 10.5. График функции

```
unit grfunc_;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

// Функция, график которой надо построить
function f(x:real):real;
begin
  f:=2*Sin(x)*exp(x/5);
end;

// строит график функции
procedure GrOfFunc;
var
  x1,x2:real;    // границы изменения аргумента функции
  y1,y2:real;    // границы изменения значения функции
  x:real;        // аргумент функции
  y:real;        // значение функции в точке x
  dx:real;       // приращение аргумента
  l,b:integer;   // левый нижний угол области вывода графика
  w,h:integer;   // ширина и высота области вывода графика
  mx,my:real;    // масштаб по осям X и Y
  x0,y0:integer; // точка — начало координат
```

```

begin
    // область вывода графика
    l:=10;                      // X – координата левого верхнего угла
    b:=Form1.ClientHeight-20;   // Y – координата левого верхнего угла
    h:=Form1.ClientHeight-40;   // высота
    w:=Form1.Width-40;          // ширина

    x1:=0;                      // нижняя граница диапазона аргумента
    x2:=25;                     // верхняя граница диапазона аргумента
    dx:=0.01;                   // шаг аргумента

    // найдем максимальное и минимальное значения
    // функции на отрезке [x1,x2]
    y1:=f(x1); // минимум
    y2:=f(x1); // максимум
    x:=x1;
    repeat
        y := f(x);
        if y < y1 then y1:=y;
        if y > y2 then y2:=y;
        x:=x+dx;
    until (x >= x2);

    // вычислим масштаб
    my:=h/abs(y2-y1); // масштаб по оси Y
    mx:=w/abs(x2-x1); // масштаб по оси X

    // оси
    x0:=1;
    y0:=b-Abs(Round(y1*my));

    with form1.Canvas do
    begin
        // оси
        MoveTo(l,b);LineTo(l,b-h);
        MoveTo(x0,y0);LineTo(x0+w,y0);
        TextOut(l+5,b-h,FloatToStrF(y2,ffGeneral,6,3));
        TextOut(l+5,b,FloatToStrF(y1,ffGeneral,6,3));
        // построение графика
        x:=x1;
        repeat
            y:=f(x);
            Pixels[x0+Round(x*mx),y0-Round(y*my)]:=clRed;
            x:=x+dx;
        until (x >= x2);
    end;
end;
end;

```

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    GrOfFunc;  
end;  
  
// изменился размер окна программы  
procedure TForm1.FormResize(Sender: TObject);  
begin  
    // очистить форму  
    form1.Canvas.FillRect(Rect(0,0,ClientWidth,ClientHeight));  
    // построить график  
    GrOfFunc;  
end;  
  
end.
```

Основную работу выполняет процедура `GrOfFunc`, которая сначала вычисляет максимальное (y_2) и минимальное (y_1) значения функции на отрезке $[x_1, x_2]$. Затем, используя информацию о ширине (`Form1.ClientWidth - 40`) и высоте (`Form1.ClientHeight - 40`) области вывода графика, вычисляет масштаб по осям X (mx) и Y (my).

Высота и ширина области вывода графика определяется размерами рабочей (клиентской) области формы, т. е. без учета области заголовка и границ. После вычисления масштаба процедура вычисляет координату Y горизонтальной оси (y_0) и вычерчивает координатные оси графика. Затем выполняется непосредственное построение графика (рис. 10.10).

Вызов процедуры `GrOfFunc` выполняют процедуры обработки событий `OnPaint` и `OnFormResize`. Процедура `TForm1.FormPaint` обеспечивает вычерчивание графика после появления формы на экране в результате запуска программы, а также после появления формы во время работы программы, например, в результате удаления или перемещения других окон, полностью или частично перекрывающих окно программы. Процедура `TForm1.FormResize` обеспечивает вычерчивание графика после изменения размера формы.

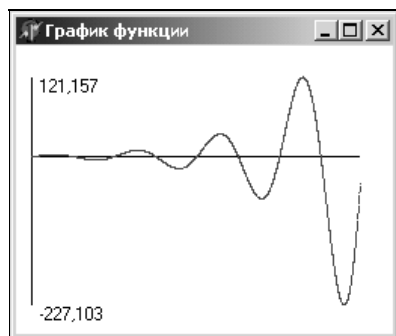


Рис. 10.10. График, построенный процедурой `GrOfFunc`

Приведенная программа довольно универсальна. Заменяв инструкции в теле функции $f(x)$, можно получить график другой функции. Причем независимо от вида функции ее график будет занимать всю область, предназначенную для вывода.

Примечание

Рассмотренная программа работает корректно, если функция, график которой надо построить, принимает как положительные, так и отрицательные значения. Если функция во всем диапазоне только положительная или только отрицательная, то в программу следует внести изменения. Какие — пусть это будет упражнением для читателя.

Вывод иллюстраций

Наиболее просто вывести иллюстрацию, которая находится в файле с расширением bmp, jpg или ico, можно при помощи компонента Image, значок которого находится на вкладке **Additional** палитры (рис. 10.11).



Рис. 10.11. Значок компонента Image

В табл. 10.8 перечислены основные свойства компонента Image.

Таблица 10.8. Свойства компонента Image

Свойство	Определяет
Picture	Иллюстрацию, которая отображается в поле компонента
Width, Height	Размер компонента. Если размер компонента меньше размера иллюстрации и значение свойств AutoSize и Stretch равно False, то отображается часть иллюстрации
AutoSize	Признак автоматического изменения размера компонента в соответствии с реальным размером иллюстрации
Stretch	Признак автоматического масштабирования иллюстрации в соответствии с реальным размером компонента. Чтобы было выполнено масштабирование, значение свойства AutoSize должно быть False
Visible	Отображается ли компонент, и соответственно иллюстрация, на поверхности формы

Иллюстрацию, которая будет выведена в поле компонента `Image`, можно задать как во время разработки формы приложения, так и во время работы программы.

Во время разработки формы иллюстрация задается установкой значения свойства `Picture` путем выбора файла иллюстрации в стандартном диалоговом окне, которое появляется в результате щелчка на командной кнопке **Load** окна **Picture Editor** (рис. 10.12). Чтобы запустить `Image Editor`, нужно в окне **Object Inspector** выбрать свойство `Picture` и щелкнуть на кнопке с тремя точками.

Если размер иллюстрации больше размера компонента, то свойству `Stretch` нужно присвоить значение `True` и установить значения свойств `Width` и `Height` пропорционально реальным размерам иллюстрации.

Чтобы вывести иллюстрацию в поле компонента `Image` во время работы программы, нужно применить метод `LoadFromFile` к свойству `Picture`, указав в качестве параметра имя файла иллюстрации. Например, инструкция

```
Form1.Image1.Picture.LoadFromFile('e:\temp\bart.bmp')
```

загружает иллюстрацию из файла `bart.bmp` и выводит ее в поле вывода иллюстрации (`Image1`).

Метод `LoadFromFile` позволяет отображать иллюстрации различных графических форматов: `BMP`, `WMF`, `JPEG` (файлы с расширением `jpg`).

Следующая программа, ее текст приведен в листинге 10.6, использует компонент `Image` для просмотра иллюстраций, которые находятся в указанном пользователем каталоге. Диалоговое окно программы приведено на рис. 10.13.

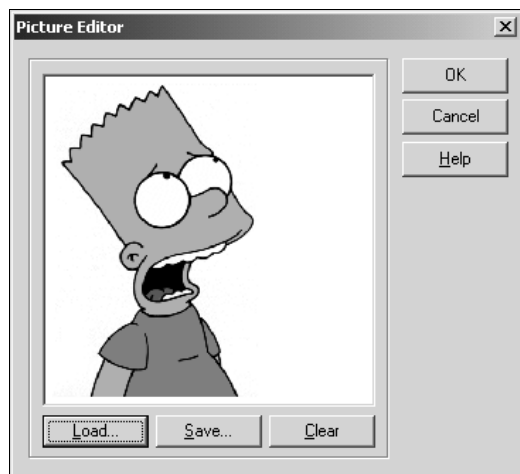


Рис. 10.12. Окно **Picture Editor**



Рис. 10.13. Слайд-проектор

Листинг 10.6

```
unit shpic_;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, Menu;

type
  TForm1 = class(TForm)
    Image1: TImage;
    Button1: TButton;
    procedure FormActivate(Sender: TObject);
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  aSearchRec : TSearchRec;

  aPath : String; // каталог, в котором находятся иллюстрации
  aFile : String; // файл иллюстрации

  iw,ih: integer; // первоначальный размер компонента Image

implementation

{$R *.DFM}

// изменение размера области вывода иллюстрации
// пропорционально размеру иллюстрации
Procedure ScaleImage;

var
  pw, ph : integer;      // размер иллюстрации
  scaleX, scaleY : real; // масштаб по X и Y
  scale : real;          // общий масштаб
begin
  // иллюстрация уже загружена
  // получим ее размеры
  pw := Form1.Image1.Picture.Width;
  ph := Form1.Image1.Picture.Height;
```

```
aFile := aPath + aSearchRec.Name;
```



```
Form1.Image1.Picture.LoadFromFile(aFile);
ScaleImage;

// подготовим вывод следующей иллюстрации
r := FindNext(aSearchRec); // найти следующий файл
if r <> 0
    then // больше нет иллюстраций
        Form1.Button1.Enabled := False;
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
    Image1.AutoSize := False; // запрет автоизменения размера компонента
    Image1.Stretch := True;   // разрешим масштабирование

    // запомним первоначальный размер области вывода иллюстрации
    iw := Image1.Width;
    ih := image1.Height;

    Button1.Enabled := False; // сделаем недоступной кнопку Далее
    FirstPicture; // вывести первую иллюстрацию
end;

// щелчок на кнопке Далее
procedure TForm1.Button1Click(Sender: TObject);
begin
    NextPicture;
end;

end.
```

Программа выполняет масштабирование выводимых иллюстраций без искажения, чего нельзя добиться простым присвоением значения `True` свойству `Stretch`. Загрузку и вывод первой и остальных иллюстраций выполняют соответственно процедуры `FirstPicture` и `NextPicture`. Процедура `FirstPicture` использует функцию `FindFirst` для того, чтобы получить имя первого `bmp`-файла. В качестве параметров функции `FindFirst` передаются:

- ☐ имя каталога, в котором должны находиться иллюстрации;
- ☐ структура `aSearchRec`, поле `Name` которой, в случае успеха, будет содержать имя файла, удовлетворяющего критерию поиска;
- ☐ маска файла иллюстрации.

Если в указанном при вызове функции `FindFirst` каталоге есть хотя бы один `bmp`-файл, значение функции будет равно нулю. В этом случае метод `LoadFromFile` загружает файл иллюстрации, после чего вызывается функ-

ция `ScaleImage`, которая устанавливает размер компонента пропорционально размеру иллюстрации. Размер загруженной иллюстрации можно получить, обратившись к свойствам `Form1.Image1.Picture.Width` и `Form1.Image1.Picture.Height`, значения которых не зависят от размера компонента `Image`.

Битовые образы

При работе с графикой удобно использовать объекты типа `TBitmap` (битовый образ). Битовый образ представляет собой находящуюся в памяти компьютера, и следовательно невидимую, графическую поверхность, на которой программа может сформировать изображение. Содержимое битового образа (картинка) легко и, что особенно важно, быстро может быть выведено на поверхность формы или области вывода иллюстрации (`Image`). Поэтому в программах битовые образы обычно используются для хранения небольших изображений, например картинок командных кнопок.

Загрузить в битовый образ нужную картинку можно при помощи метода `LoadFromFile`, указав в качестве параметра имя bmp-файла, в котором находится нужная иллюстрация.

Например, если в программе объявлена переменная `pic` типа `TBitmap`, то после выполнения инструкции

```
pic.LoadFromFile('e:\images\aplane.bmp')
```

битовый образ `pic` будет содержать изображение самолета.

Вывести содержимое битового образа (картинку) на поверхность формы или области вывода иллюстрации можно путем применения метода `Draw` к соответствующему свойству поверхности (`Canvas`). Например, инструкция

```
Image1.Canvas.Draw(x,y, bm)
```

выводит картинку битового образа `bm` на поверхность компонента `Image1` (параметры `x` и `y` определяют положение левого верхнего угла картинки на поверхности компонента).

Если перед применением метода `Draw` свойству `Transparent` объекта `TBitmap` присвоить значение `True`, то фрагменты рисунка, окрашенные цветом, совпадающим с цветом левого нижнего угла картинки, не будут выведены — через них будет как бы проглядывать фон. Если в качестве "прозрачного" нужно использовать цвет, отличный от цвета левой нижней точки рисунка, то свойству `TransparentColor` следует присвоить значение символьной константы, обозначающей необходимый цвет.

Следующая программа, текст которой приведен в листинге 10.7, демонстрирует использование битовых образов для формирования изображения из нескольких элементов.

Листинг 10.7. Использование битовых образов

```
unit aplanes_;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
    Dialogs;  
  
type  
    TForm1 = class(TForm)  
        procedure FormPaint(Sender: TObject);  
    private  
        { Private declarations }  
    public  
        { Public declarations }  
    end;  
  
var  
    Form1: TForm1;  
    sky, aplane: TBitmap; // битовые образы: небо и самолет  
  
implementation  
  
{$R *.DFM}  
  
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    // создать битовые образы  
    sky := TBitmap.Create;  
    aplane := TBitmap.Create;  
  
    // загрузить картинки  
    sky.LoadFromFile('sky.bmp');  
    aplane.LoadFromFile('aplane.bmp');  
  
    Form1.Canvas.Draw(0,0,sky);      // отрисовка фона  
    Form1.Canvas.Draw(20,20,aplane); // отрисовка левого самолета  
  
    aplane.Transparent:=True;  
    // теперь элементы рисунка, цвет которых совпадает с цветом  
    // левой нижней точки битового образа, не отрисовываются  
    Form1.Canvas.Draw(120,20,aplane); // отрисовка правого самолета  
  
    // освободить память  
    sky.free;  
    aplane.free;  
  
end;  
  
end.
```

После запуска программы в окне приложения (рис. 10.14) появляется изображение летящих на фоне неба самолетов. Фон и изображение самолета — битовые образы, загружаемые из файлов. Белое поле вокруг левого самолета показывает истинный размер картинки битового образа `aplane`. Белое поле вокруг правого самолета отсутствует, т. к. перед его выводом свойству `Transparent` битового образа было присвоено значение `True`.



Рис. 10.14. Окно приложения

Мультипликация

Под мультипликацией обычно понимается движущийся и меняющийся рисунок. В простейшем случае рисунок может только двигаться или только меняться.

Как было показано выше, рисунок может быть сформирован из графических примитивов (линий, окружностей, дуг, многоугольников и т. д.). Обеспечить перемещение рисунка довольно просто: надо сначала вывести рисунок на экран, затем через некоторое время стереть его и снова вывести этот же рисунок, но уже на некотором расстоянии от его первоначального положения. Подбором времени между выводом и удалением рисунка, а также расстояния между старым и новым положением рисунка (шага перемещения), можно добиться того, что у наблюдателя будет складываться впечатление, что рисунок равномерно движется по экрану.

Следующая простая программа, текст которой приведен в листинге 10.8, а вид формы — на рис. 10.15, демонстрирует движение окружности от левой к правой границе окна программы.



Рис. 10.15. Форма программы **Движущаяся окружность**

Листинг 10.8. Двигающаяся окружность

```

unit mcircle_;

interface

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

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

implementation

{$R *.DFM}

var
  Form1: TForm1;
  x, y: byte;      // координаты центра окружности
  dx: byte;        // приращение координаты x при движении окружности

procedure Ris;
begin
  // стереть окружность
  form1.Canvas.Pen.Color:=form1.Color;
  form1.Canvas.Ellipse(x, y, x+10, y+10);

  x:=x+dx;

  // нарисовать окружность на новом месте
  form1.Canvas.Pen.Color:=clBlack;
  form1.Canvas.Ellipse(x, y, x+10, y+10);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Ris;
end;

```

```

procedure TForm1.FormActivate(Sender: TObject);
begin
    x:=0;
    y:=10;
    dx:=5;
    timer1.Interval:=50; // период возникновения события OnTimer – 0.5 сек
    form1.canvas.brush.color:=form1.color;
end;

end.

```

Основную работу выполняет процедура `Ris`, которая стирает окружность и выводит ее на новом месте. Стирание окружности выполняется путем перерисовки окружности поверх нарисованной, но цветом фона.

Для обеспечения периодического вызова процедуры `Ris` в форму программы добавлен невидуальный компонент `Timer` (таймер), значок которого находится в вкладке **System** палитры компонентов (рис. 10.16).



Рис. 10.16. Значок компонента `Timer`

В табл. 10.9 перечислены свойства компонента `Timer`.

Таблица 10.9. Значения свойства `Timer`

Свойство	Назначение
<code>Name</code>	Имя компонента. Используется для доступа к компоненту
<code>Interval</code>	Период генерации события <code>OnTimer</code> . Задается в миллисекундах
<code>Enabled</code>	Разрешение работы. Разрешает (значение <code>True</code>) или запрещает (значение <code>False</code>) генерацию события <code>OnTimer</code>

Добавляется компонент `Timer` к форме обычным образом, однако, т. к. компонент `Timer` является невидуальным, т. е. во время работы программы он не отображается на форме, поэтому его значок можно поместить в любое место формы.

Компонент `Timer` генерирует событие `OnTimer`. Период возникновения события `OnTimer` измеряется в миллисекундах и определяется значением свойства `Interval`. Следует обратить внимание на свойство `Enabled`. Оно дает возможность программе "запустить" или "остановить" таймер. Если значение свойства `Enabled` равно `False`, то событие `OnTimer` не возникает.

Событие `OnTimer` в рассматриваемой программе обрабатывается процедурой `Timer1Timer`, которая, в свою очередь, вызывает процедуру `Ris`. Таким образом, в программе реализован механизм периодического вызова процедуры `Ris`.

Примечание

Переменные x , y (координаты центра окружности) и dx (приращение координаты x при движении окружности) объявлены вне процедуры `Ris`, т. е. они являются глобальными. Поэтому надо не забыть выполнить их инициализацию (в программе инициализацию глобальных переменных реализует процедура `FormActivate`).

Метод базовой точки

При программировании сложных изображений, состоящих из множества элементов, используется метод, который называется *методом базовой точки*. Суть этого метода заключается в следующем:

- ❑ Выбирается некоторая точка изображения, которая принимается за базовую.
- ❑ Координаты остальных точек отсчитываются от базовой точки.
- ❑ Если координаты точек изображения отсчитывать от базовой в относительных единицах, а не в пикселах, то обеспечивается возможность масштабирования изображения.

На рис. 10.17 приведено изображение кораблика. Базовой точкой является точка с координатами (x_0, y_0) . Координаты остальных точек отсчитываются именно от этой точки.

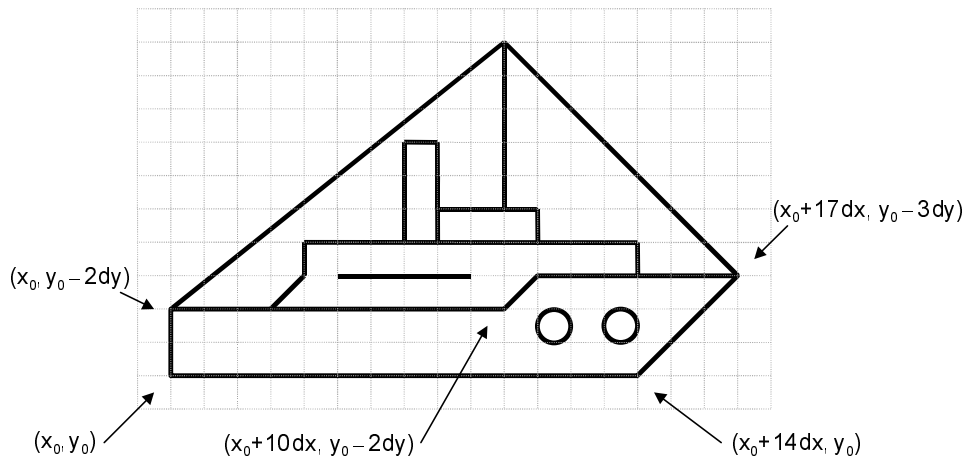


Рис. 10.17. Определение координат изображения относительно базовой точки

В листинге 10.9 приведен текст программы, которая выводит на экран изображение перемещающегося кораблика.

Листинг 10.9. Кораблик

```
unit titanik_;
```



```
interface
```



```
uses
```



```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls, ExtCtrls;
```



```
type
```



```
  TForm1 = class(TForm)  
    Timer1: TTimer;  
    procedure Timer1Timer(Sender: TObject);  
    procedure FormActivate(Sender: TObject);  
  private  
    { Private declarations }  
  public  
    { Public declarations }  
  end;
```



```
var
```



```
  Form1: TForm1;  
  x,y: integer; // координаты корабля (базовой точки)
```



```
implementation
```



```
{ $R *.DFM }
```



```
procedure Titanik(x,y: integer;    // координаты базовой точки  
                  color: TColor); // цвет корабля
```



```
const
```



```
  dx = 5;  
  dy = 5;
```



```
var
```



```
  buf: TColor;
```



```
begin
```



```
  with form1.canvas do
```



```
  begin
```



```
    buf:=pen.Color;    // сохраним текущий цвет  
    pen.Color:=color; // установим нужный цвет  
    // рисуем ...  
    // корпус  
    MoveTo(x,y);
```



```

LineTo(x, y-2*dy);
LineTo(x+10*dx, y-2*dy);
LineTo(x+11*dx, y-3*dy);
LineTo(x+17*dx, y-3*dy);
LineTo(x+14*dx, y);
LineTo(x, y);
// надстройка
MoveTo(x+3*dx, y-2*dy);
LineTo(x+4*dx, y-3*dy);
LineTo(x+4*dx, y-4*dy);
LineTo(x+13*dx, y-4*dy);
LineTo(x+13*dx, y-3*dy);
MoveTo(x+5*dx, y-3*dy);
LineTo(x+9*dx, y-3*dy);
// капитанский мостик
Rectangle(x+8*dx, y-4*dy, x+11*dx, y-5*dy);
// труба
Rectangle(x+7*dx, y-4*dy, x+8*dx, y-7*dy);
// иллюминаторы
Ellipse(x+11*dx, y-2*dy, x+12*dx, y-1*dy);
Ellipse(x+13*dx, y-2*dy, x+14*dx, y-1*dy);
// мачта
MoveTo(x+10*dx, y-5*dy);
LineTo(x+10*dx, y-10*dy);
// оснастка
MoveTo(x+17*dx, y-3*dy);
LineTo(x+10*dx, y-10*dy);
LineTo(x, y-2*dy);
pen.Color:=buf; // восстановим старый цвет карандаша

```

```
end;
```

```
end;
```

```
// обработка сигнала таймера
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
```

```
    Titanic(x, y, form1.color); // стереть рисунок
```

```
    if x < Form1.ClientWidth
```

```
        then x := x+5
```

```
    else begin // новый рейс
```

```
        x := 0;
```

```
        y := Random(50) + 100;
```

```
    end;
```

```
    Titanic(x, y, clWhite); // нарисовать в новой точке
```

```
end;
```

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    x:=0;
    y:=100;
    Form1.Color:=clNavy;
    Timer1.Interval := 50; // сигнал таймера каждые 50 миллисекунд
end;

end.
```

Отрисовку и стирание изображения кораблика выполняет процедура `Titanik`, которая получает в качестве параметров координаты базовой точки и цвет, которым надо вычертить изображение кораблика. Если при вызове процедуры цвет отличается от цвета фона формы, то процедура рисует кораблик, а если совпадает — то "стирает". В процедуре `Titanik` объявлены константы `dx` и `dy`, определяющие шаг (в пикселах), используемый при вычислении координат точек изображения. Меняя значения этих констант, можно проводить масштабирование изображения.

Использование битовых образов

В предыдущем примере изображение формировалось из графических примитивов. Теперь рассмотрим, как можно реализовать перемещение одного сложного изображения на фоне другого, например перемещение самолета на фоне городского пейзажа.

Эффект перемещения картинки может быть создан путем периодической перерисовки картинки с некоторым смещением относительно ее прежнего положения. При этом предполагается, что перед выводом картинки в новой точке сначала удаляется предыдущее изображение. Удаление картинки может быть выполнено путем перерисовки всей фоновой картинки или только той ее части, которая перекрыта битовым образом движущегося объекта.

В рассматриваемой программе используется второй подход. Картинка выводится применением метода `Draw` к свойству `Canvas` компонента `Image`, а стирается путем копирования (метод `CopyRect`) нужной части фона из буфера на поверхность компонента `Image`.

Форма программы приведена на рис. 10.18, а текст — в листинге 10.10.

Компонент `Image` используется для вывода фона, а компонент `Timer` — для организации задержки между циклами удаления и вывода на новом месте изображения самолета.

Листинг 10.10. Летящий над городом самолет

```
unit anim_;

interface
```

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls, StdCtrls, Buttons;

type

```
TForm1 = class(TForm)
    Timer1: TTimer;
    Image1: TImage;
    procedure FormActivate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

Form1: TForm1;

implementation

{ \$R *.DFM }

var

Back, bitmap, Buf : TBitmap; // фон, картинка, буфер
BackRct : TRect; // область фона, которая должна быть
// восстановлена из буфера
BufRct: TRect; // область буфера, которая используется для
// восстановления фона

x,y:integer; // текущее положение картинки
W,H: integer; // размеры картинки

procedure TForm1.FormActivate(Sender: TObject);

begin

// создать три объекта – битовых образа
Back := TBitmap.Create; // фон
bitmap := TBitmap.Create; // картинка
Buf := TBitmap.Create; // буфер

// загрузить и вывести фон
Back.LoadFromFile('factory.bmp');
Form1.Image1.canvas.Draw(0,0,Back);

```
// загрузить картинку, которая будет двигаться
bitmap.LoadFromFile('aplane.bmp');
// определим "прозрачный" цвет
bitmap.Transparent := True;
bitmap.TransparentColor := bitmap.Canvas.Pixels[1,1];

// создать буфер для сохранения копии области фона,
// на которую накладывается картинка
W:= bitmap.Width;
H:= bitmap.Height;
Buf.Width:= W;
Buf.Height:=H;
Buf.Palette:=Back.Palette; // Чтобы обеспечить соответствие палитр !!
Buf.Canvas.CopyMode:=cmSrcCopy;
// определим область буфера, которая будет использоваться
// для восстановления фона
BufRct:=Bounds(0,0,W,H);

// начальное положение картинки
x := -W;
y := 20;

// определим сохраняемую область фона
BackRct:=Bounds(x,y,W,H);
// и сохраним ее
Buf.Canvas.CopyRect(BufRct,Back.Canvas,BackRct);
end;

// обработка сигнала таймера
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    // восстановлением фона (из буфера) удалим рисунок
    Form1.Image1.Canvas.Draw(x,y,Buf);

    x:=x+2;
    if x>form1.Image1.Width then x:=-W;

    // определим сохраняемую область фона
    BackRct:=Bounds(x,y,W,H);
    // сохраним ее копию
    Buf.Canvas.CopyRect(BufRct,Back.Canvas,BackRct);

    // выведем рисунок
    Form1.Image1.Canvas.Draw(x,y,bitmap);
end;

// завершение работы программы
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
```

```
begin
```

```
  // освободим память, выделенную
```

```
  // для хранения битовых образов
```

```
Back.Free;
```

```
bitmap.Free;
```

```
Buf.Free;
```

```
end;
```

```
end.
```

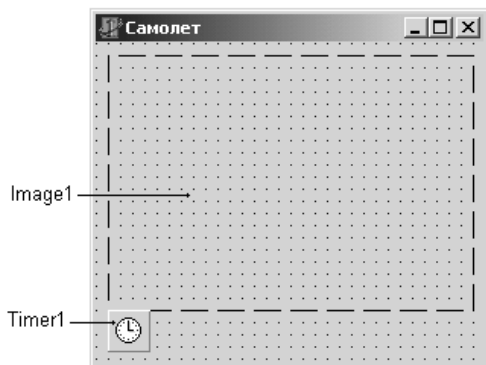


Рис. 10.18. Форма программы **Самолет**

Для хранения битовых образов (картинок) фона и самолета, а также копии области фона, перекрываемой изображением самолета, используются объекты типа `TBitmap`, которые создаются динамически процедурой `FormActivate`. Эта же процедура загружает из файлов картинки фона (`factory.bmp`) и самолета (`arplane.bmp`), а также сохраняет область фона, на которую первый раз будет накладываться картинка.

Сохранение копии фона выполняется при помощи метода `CopyRect`, который позволяет выполнить копирование прямоугольного фрагмента одного битового образа в другой. Объект, к которому применяется метод `CopyRect`, является приемником копии битового образа. В качестве параметров методу передаются координаты и размер области, куда должно быть выполнено копирование, поверхность, откуда должно быть выполнено копирование, а также положение и размер копируемой области. Информация о положении и размере копируемой в буфер области фона, на которую будет наложено изображение самолета и которая впоследствии должна быть восстановлена из буфера, находится в структуре `BackRect` типа `TRect`. Для заполнения этой структуры используется функция `Bounds`.

Следует обратить внимание на то, что начальное значение переменной `x`, которая определяет положение левой верхней точки битового образа движущейся картинки, — отрицательное число, равное ширине битового образа картинки. Поэтому в начале работы программы изображение самолета не

появляется, картинка отрисовывается за границей видимой области. С каждым событием `OnTimer` значение координаты `x` увеличивается, и на экране появляется та часть битового образа, координаты которой больше нуля. Таким образом, у наблюдателя создается впечатление, что самолет вылетает из-за левой границы окна.

Загрузка битового образа из ресурса программы

В приведенной в листинге 10.10 программе битовые образы фона и картинки загружаются из файлов. Это не всегда удобно. Delphi позволяет поместить необходимые битовые образы в виде *ресурса* в файл исполняемой программы и по мере необходимости загружать битовые образы из ресурса, т. е. из файла исполняемой программы (exe-файла).

Создание файла ресурсов

Для того чтобы воспользоваться возможностью загрузки картинки из ресурса, необходимо сначала создать *файл ресурсов*, поместив в него нужные картинки.

Файл ресурсов можно создать при помощи утилиты Image Editor (Редактор изображений), которая запускается выбором команды **Image Editor** меню **Tools**.

Для того чтобы создать новый файл ресурсов, надо из меню **File** выбрать команду **New**, а затем в появившемся подменю — команду **Resource File** (Файл ресурсов) (рис. 10.19).

В результате открывается окно нового файла ресурсов, а в строке меню окна **Image Editor** появляется новый пункт — **Resource** (рис. 10.20).

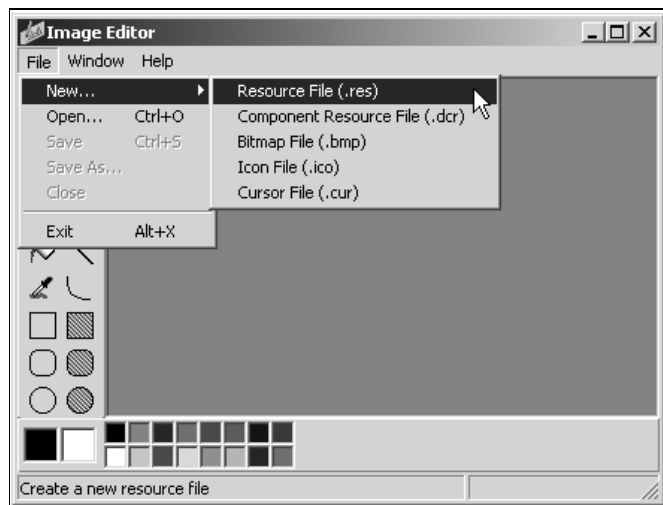


Рис. 10.19. Диалоговое окно **Image Editor**

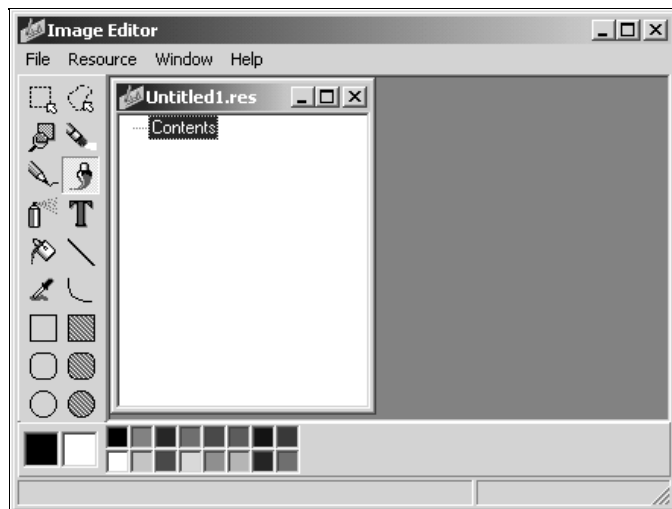


Рис. 10.20. Создание нового файла ресурсов в окне **Image Editor**

Для того чтобы в этот файл добавить новый ресурс, необходимо выбрать команду **New** меню **Resource** и из открывшегося списка — тип ресурса. В данном случае следует выбрать **Bitmap** (битовый образ). После выбора **Bitmap** открывается диалоговое окно **Bitmap Properties** (Свойства битового образа) (рис. 10.21), используя которое можно установить размер (в пикселах) и количество цветов создаваемой картинке.

Нажатие кнопки **OK** в диалоговом окне **Bitmap Properties** вызывает появление элемента **Bitmap1** в иерархическом списке **Contents**. Этот элемент соответствует новому ресурсу, добавленному в файл (рис. 10.22).

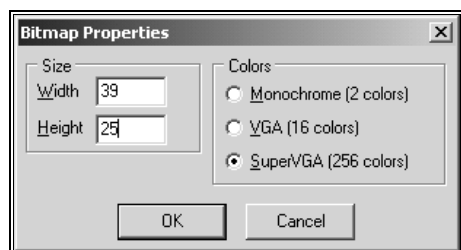


Рис. 10.21. Диалоговое окно **Bitmap Properties**

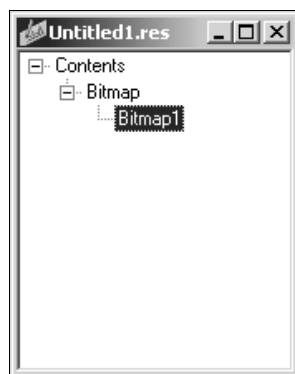


Рис. 10.22. Окно файла ресурсов после добавления ресурса **Bitmap**

Bitmap1 — это автоматически созданное имя ресурса, которое может быть изменено выбором команды **Rename** меню **Resource** и вводом нужного име-

ни. После изменения имени **Bitmap1** можно приступить к созданию битового образа. Для этого необходимо выбрать команду **Edit** меню **Resource**, в результате чего открывается окно графического редактора.

Графический редактор **Image Editor** предоставляет программисту стандартный для подобных редакторов набор инструментов, используя которые можно нарисовать нужную картинку. Если во время работы надо изменить масштаб отображения картинки, то для увеличения масштаба следует выбрать команду **Zoom In** меню **View**, а для уменьшения — команду **Zoom Out**. Увидеть картинку в реальном масштабе можно, выбрав команду **Actual Size** меню **View**.

В качестве примера на рис. 10.23 приведен вид диалогового окна **Image Editor**, в котором находится файл ресурсов для программы **aplane**. Файл содержит два битовых образа с именами **FACTORY** и **APLANE**.



Рис. 10.23. Диалоговое окно **Image Editor** с файлом ресурсов

Если нужная картинка уже существует в виде отдельного файла, то ее можно через буфер обмена (Clipboard) поместить в битовый образ файла ресурсов. Делается это следующим образом.

1. Сначала надо запустить графический редактор, например **Microsoft Paint**, загрузить в него файл картинки и выделить всю картинку или ее часть. В процессе выделения следует обратить внимание на информацию о размере (в пикселах) выделенной области (**Paint** выводит размер выделяемой области в строке состояния). Затем, выбрав команду **Копировать** меню **Правка**, следует поместить копию выделенного фрагмента в буфер.

2. Далее нужно переключиться в Image Editor, выбрать ресурс, в который надо поместить находящуюся в буфере картинку, и установить значения характеристик ресурса в соответствии с характеристиками картинки, находящейся в буфере. Значения характеристик ресурса вводятся в поля диалогового окна **Bitmap Properties**, которое открывается выбором команды **Image Properties** меню **Bitmap**. После установки характеристик ресурса можно вставить картинку в ресурс, выбрав команду **Past** меню **Edit**.
3. После добавления всех нужных ресурсов файл ресурса следует сохранить в том каталоге, где находится программа, для которой этот файл создается. Сохраняется файл ресурса обычным образом, т. е. выбором команды **Save** меню **File**. Image Editor присваивает файлу ресурсов расширение **res**.

Подключение файла ресурсов

Для того чтобы ресурсы были доступны программе, необходимо в текст программы включить инструкцию (директиву), которая сообщит компилятору, что в файл исполняемой программы следует добавить содержимое файла ресурсов.

В общем виде эта директива выглядит следующим образом:

```
{ $R ФайлРесурсов }
```

где *ФайлРесурсов* — имя файла ресурсов.

Например, директива может выглядеть так:

```
{ $R images.res }.
```

Директиву включения файла ресурсов в файл исполняемой программы обычно помещают в начале текста модуля.

Примечание

Если имена файла модуля программы и файла ресурсов совпадают, то вместо имени файла ресурсов можно поставить `"*"`. В этом случае директива включения файла ресурсов в файл исполняемой программы выглядит так:

```
{ $R *.res }
```

Загрузить картинку из ресурса в переменную типа `TBitmap` можно при помощи метода `LoadFromResourceName`, который имеет два параметра: идентификатор программы и имя ресурса. В качестве идентификатора программы используется глобальная переменная `HInstance`. Имя ресурса должно быть представлено в виде строковой константы.

Например, инструкция загрузки картинки в переменную `Pic` может выглядеть так:

```
Pic.LoadFromResourceName(HInstance, 'FACTORY');
```

В качестве примера в листинге 10.11 приведен текст программы, в которой изображение фона и самолета загружается из ресурсов.

Листинг 10.11. Пример загрузки картинок из ресурса

```
unit apanel_;
```

{ \$R images.res } // включить файл ресурсов

```
interface
```

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls, StdCtrls, Buttons;

type

TForm1 = class(TForm)
 Timer1: TTimer;
 Image1: TImage;
 procedure FormActivate(Sender: TObject);
 procedure Timer1Timer(Sender: TObject);
 procedure FormClose(Sender: TObject; **var** Action: TCloseAction);
private
 { Private declarations }
public
 { Public declarations }
end;

var

Form1: TForm1;
Back, bitmap, Buf : TBitmap; *// фон, картинка, буфер*
BackRect, BufRect: TRect; *// область фона, картинки, буфера*

x,y:integer; *// координаты левого верхнего угла картинки*
W,H: integer; *// размеры картинки*

implementation

*{ \$R *.DFM }*

procedure TForm1.FormActivate(Sender: TObject);
begin

Back := TBitmap.Create; *// фон*
bitmap := TBitmap.Create; *// картинка*
Buf := TBitmap.Create; *// буфер*

// загрузить из ресурса фон
Back.LoadFromResourceName(HInstance, 'FACTORY');
Form1.Image1.canvas.Draw(0,0,Back);

```
// загрузить из ресурса картинку, которая будет двигаться
bitmap.LoadFromResourceName(HInstance, 'APLANE') ;
bitmap.Transparent := True;
bitmap.TransparentColor := bitmap.canvas.pixels[1,1];

// создать буфер для сохранения копии области фона, на которую
// накладывается картинка
W:= bitmap.Width;
H:= bitmap.Height;
Buf.Width:= W;
Buf.Height:=H;
Buf.Palette:=Back.Palette; // Чтобы обеспечить соответствие палитр !!
Buf.Canvas.CopyMode:=cmSrcCopy;
BufRct:=Bounds(0,0,W,H);

x:=-W;
y:=20;

// определим сохраняемую область фона
BackRct:=Bounds(x,y,W,H);
// и сохраним ее
Buf.Canvas.CopyRect(BufRct,Back.Canvas,BackRct);
```

end;

procedure TForm1.Timer1Timer(Sender: TObject);

begin

```
    // восстановлением фона (из буфера) удалим рисунок
    Form1.image1.canvas.Draw(x,y,Buf);
```

```
    x:=x+2;
```

```
    if x>form1.Image1.Width then x:=-W;
```

```
    // определим сохраняемую область фона
```

```
    BackRct:=Bounds(x,y,W,H);
```

```
    // сохраним ее копию
```

```
    Buf.Canvas.CopyRect(BufRct,Back.Canvas,BackRct);
```

```
    // выведем рисунок
```

```
    Form1.image1.canvas.Draw(x,y,bitmap);
```

end;

procedure TForm1.FormClose(Sender: TObject; **var** Action: TCloseAction);

begin

```
    Back.Free;
```

```
    bitmap.Free;
```

```
    Buf.Free;
```

end;

end.

Преимущества загрузки картинок из ресурса программы очевидны: при распространении программы не надо заботиться о том, чтобы во время работы программы были доступны файлы иллюстраций, все необходимые программе картинки находятся в исполняемом файле.

Просмотр "мультика"

Теперь рассмотрим, как можно реализовать вывод в диалоговом окне программы простого "мультика", подобного тому, который можно видеть в диалоговом окне **Установка связи с <имя_провайдера>** при подключении к Internet (рис. 10.24).

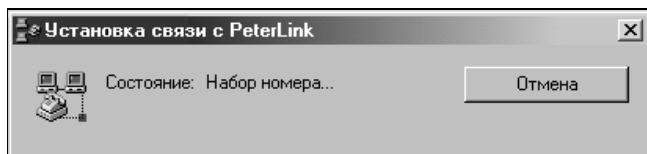


Рис. 10.24. Пример мультика в диалоговом окне **Установка связи**

Эффект бегущего между телефоном и компьютером красного квадратика достигается за счет того, что в диалоговое окно выводятся сменяющие друг друга картинки (рис. 10.25).



Рис. 10.25. Картинки, используемые для реализации мультика в диалоговом окне **Установка связи**

Кадры мультика обычно находятся в одном файле или в одном ресурсе. Перед началом работы программы они загружаются в буфер, в качестве которого удобно использовать объект типа `TBitMap`. Задача процедуры, реализующей вывод мультика, состоит в том, чтобы выделить очередной кадр и вывести его в нужное место формы.

Вывести кадр на поверхность формы можно применением метода `CopyRect` к свойству `Canvas` этой формы. Метод `CopyRect` копирует прямоугольную область одной графической поверхности на другую.

Инструкция применения метода `CopyRect` в общем виде выглядит так:

```
Canvas1.CopyRect(Область1, Canvas2, Область2)
```

где:

`Canvas1` — графическая поверхность, на которую выполняется копирование;

`Canvas2` — графическая поверхность, с которой выполняется копирование;

параметр `Область2` — задает положение и размер копируемой прямоугольной области, а параметр `Область1` — положение копии на поверхности `Canvas1`.

В качестве параметров *Область1* и *Область2* используются структуры типа `TRect`, поля которых определяют положение и размер области.

Заполнить поля структуры `TRect` можно при помощи функции `Bounds`, инструкция обращения к которой в общем виде выглядит так:

```
Bounds(x, y, Width, Height)
```

где:

`x` и `y` — координаты левого верхнего угла области;

`Width` и `Height` — ширина и высота области.

Следующая программа, текст которой приведен в листинге 10.12, выводит в диалоговое окно простой мультимедиа — эмблему Delphi, вокруг которой "летает" некоторый объект. На рис. 10.26 приведены кадры этого мультимедиа (содержимое файла `film.bmp`). Диалоговое окно программы содержит один единственный компонент — таймер.



Рис. 10.26. Кадры мультимедиа

Листинг 10.12. Мультимедиа

```
unit multik_;

interface

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

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure FormActivate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

const

```
N_KADR=12; // кадров в фильме
```

var

```
Film: TBitmap;           // фильм — все кадры  
Kadr: TBitmap;           // текущий кадр  
WKadr, HKadr: integer;   // ширина и высота кадра  
CKadr: integer;          // номер текущего кадра  
RectKadr: TRect;         // положение и размер кадра в фильме
```

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

begin

```
    Film := TBitmap.Create;  
    Film.LoadFromFile('f:\temp\film2.bmp');  
    WKadr := Round(Film.Width/N_Kadr);  
    HKadr := Film.Height;  
  
    Kadr:=TBitmap.Create;  
    Kadr.Width := WKadr;  
    Kadr.Height := HKadr;  
  
    CKadr:=0;  
    Form1.Timer1.Enabled:=True;
```

end;

```
// отрисовка кадра
```

```
procedure DrawKadr;
```

begin

```
    // определим положение текущего кадра в фильме  
    RectKadr:=Bounds(WKadr*CKadr, 0, WKadr, HKadr);  
  
    // выделим кадр из фильма  
    Kadr.Canvas.CopyRect(Rect(0,0,WKadr, HKadr), Film.Canvas, RectKadr);  
  
    // выведем кадр на форму  
    Form1.Canvas.Draw(10,10, Kadr);  
  
    // подготовимся к выводу следующего кадра  
    CKadr := CKadr+1;  
    if CKadr = N_KADR  
        then CKadr:=0;
```

end;

```
procedure TForm1.Timer1Timer(Sender: TObject);  
begin  
    DrawKadr;  
end;  
  
end.
```

Программа состоит из трех процедур. Процедура `TForm1.FormActivate` создает объект `Film` и загружает в него фильм — `bmp`-файл, в котором находятся кадры фильма. Затем, используя информацию о размере загруженного битового образа, процедура устанавливает значения характеристик кадра: высоту и ширину.

После этого создается объект `Kadr` (типа `TBitmap`), предназначенный для хранения текущего кадра. Следует обратить внимание, что после создания объекта `Kadr` принудительно устанавливаются значения свойств `Width` и `Height`. Если этого не сделать, то созданный объект будет существовать, однако память для хранения битового образа не будет выделена. В конце своей работы процедура `TForm1.FormActivate` устанавливает номер текущего кадра и запускает таймер.

Основную работу в программе выполняет процедура `DrawKadr`, которая выделяет из фильма очередной кадр и выводит его в форму. Выделение кадра выполняется копированием фрагмента картинки из объекта `Film` в объект `Kadr` при помощи метода `CopyRect`. Положение фрагмента в фильме, т. е. координата x левого верхнего угла, определяется умножением ширины кадра на номер текущего кадра. Отрисовку кадра на поверхности формы выполняет метод `Draw`. Запускает процедуру `DrawKadr` процедура `TForm1.Timer1Timer`, обрабатывающая событие `OnTimer`.



Мультимедийные возможности Delphi

Большинство современных программ, работающих в среде Windows, являются мультимедийными. Такие программы обеспечивают просмотр видеороликов и мультипликации, воспроизведение музыки, речи, звуковых эффектов. Типичными примерами мультимедийных программ являются игры и обучающие программы.

Delphi предоставляет в распоряжение программиста два компонента, позволяющие разрабатывать мультимедийные программы:

- ☐ **Animate** — обеспечивает вывод простой анимации (подобной той, которую видит пользователь во время копирования файлов).
- ☐ **MediaPlayer** — позволяет решать более сложные задачи, например, воспроизводить видеоролики, звук, сопровождаемую звуком анимацию.

Компонент *Animate*

Компонент **Animate**, значок которого находится на вкладке **Win32** (рис. 11.1), позволяет воспроизводить простую анимацию, кадры которой находятся в AVI-файле (AVI — это сокращение от Audio Video Interleave, что переводится как чередование звука и видео, т. е. AVI-файл содержит как звуковую, так и видеоинформацию).



Рис. 11.1. Значок компонента *Animate*

Примечание

Хотя анимация, находящаяся в AVI-файле может сопровождаться звуковыми эффектами (так ли это — можно проверить, например, при помощи стандартной программы **Проигрыватель Windows Media**), компонент *Animate* обеспечивает воспроизведение только изображения. Для полноценного воспроизведения сопровождаемой звуком анимации следует использовать компонент *MediaPlayer*.

Компонент *Animate* добавляется к форме обычным образом. После добавления компонента к форме следует установить значения его свойств. Свойства компонента *Animate* перечислены в табл. 11.1.

Таблица 11.1. Свойства компонента *Animate*

Свойство	Определяет
Name	Имя компонента. Используется для доступа к свойствам компонента и управления его поведением
FileName	Имя AVI-файла, в котором находится анимация, отображаемая при помощи компонента
StartFrame	Номер кадра, с которого начинается отображение анимации
StopFrame	Номер кадра, на котором заканчивается отображение анимации
Activate	Признак активизации процесса отображения кадров анимации
Color	Цвет фона компонента (цвет "экрана"), на котором воспроизводится анимация
Transparent	Режим использования "прозрачного" цвета при отображении анимации
Repetitions	Количество повторов отображения анимации

Следует еще раз обратить внимание, что компонент *Animate* предназначен для воспроизведения AVI-файлов, которые содержат **ТОЛЬКО** анимацию. При попытке записать в свойство *FileName* имя файла, который содержит звук, Delphi выводит сообщение о невозможности открытия указанного файла (Cannot open AVI). Чтобы увидеть, что находится в AVI-файле: анимация и звук или только анимация, нужно из Windows раскрыть необходимую папку, выделить AVI-файл и из контекстного меню выбрать команду **Свойства**. В результате этого откроется окно **Свойства**, в котором, на вкладке **Сводка**, будет выведена подробная информация о содержимом выбранного файла (рис. 11.2).

Следующая программа, ее текст приведен в листинге 11.1, демонстрирует использование компонента *Animate* для отображения в диалоговом окне

программы анимации. Вид формы программы приведен на рис. 11.3, а значения свойств компонента `Animate1` — в табл. 11.2.

Таблица 11.2. Значения свойств компонента `Animate1`

Свойство	Значение	Свойство	Значение
Name	Animate1	Active	False
FileName	Batr.avi	Transparent	True

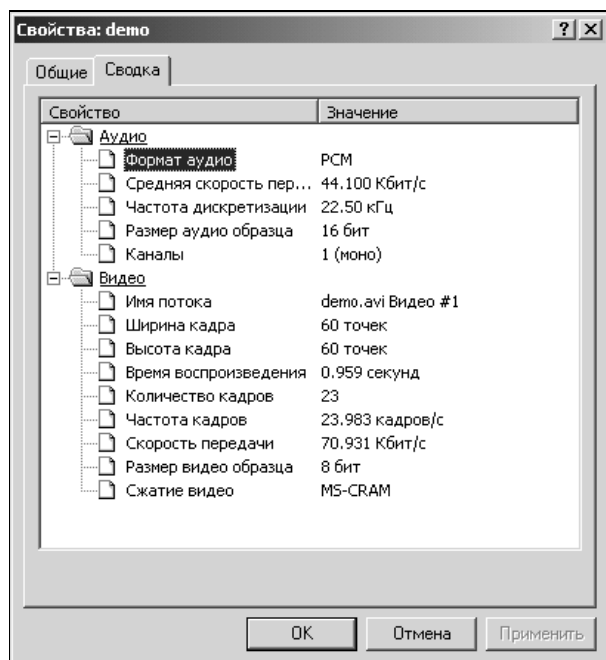


Рис. 11.2. На вкладке **Сводка** отражается информация о содержимом AVI-файла

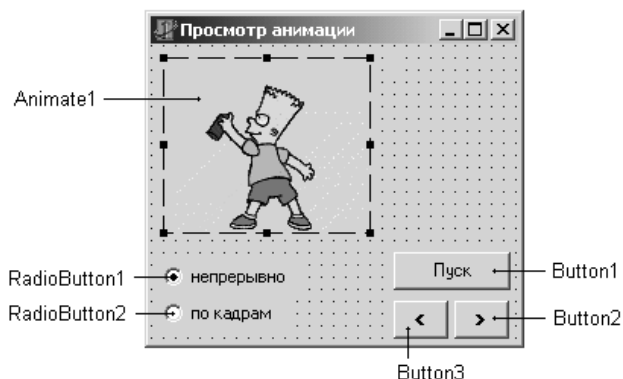


Рис. 11.3. Форма программы **Просмотр анимации**

После запуска программы в форме выводится первый кадр анимации. Программа обеспечивает два режима просмотра анимации: непрерывный и по-кадровый.

Кнопка `Button1` используется как для инициализации процесса воспроизведения анимации, так и для его приостановки. Процесс непрерывного воспроизведения анимации инициирует процедура обработки события `OnClick` на кнопке **Пуск**, которая свойству `Active` присваивает значение `True`. Эта же процедура заменяет текст на кнопке `Button1` с **Пуск** на **Стоп**. Режим воспроизведения анимации выбирается при помощи переключателей `RadioButton1` и `RadioButton2`. Процедуры обработки события `OnClick` на этих переключателях, изменением значения свойства `Enabled`, блокируют или, наоборот, делают доступными кнопки управления: активизации воспроизведения анимации (`Button1`), перехода к следующему (`Button2`) и предыдущему (`Button3`) кадру. Во время непрерывного воспроизведения анимации процедура обработки события `OnClick` на кнопке **Стоп** (`Button1`) свойству `Active` присваивает значение `False` и тем самым останавливает процесс воспроизведения анимации.

Листинг 11.1. Использование компонента `Animate`

```
unit ShowAVI_;

interface

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

type
  TForm1 = class(TForm)
    Animate1: TAnimate; // компонент Animate
    Button1: TButton;    // кнопка Пуск-Стоп
    Button2: TButton;    // следующий кадр
    Button3: TButton;    // предыдущий кадр
    RadioButton1: TRadioButton; // просмотр всей анимации
    RadioButton2: TRadioButton; // покадровый просмотр

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure RadioButton1Click(Sender: TObject);
    procedure RadioButton2Click(Sender: TObject);

  private
    { Private declarations }
```

```
public
    { Public declarations }
end;

var
    Form1: TForm1;    // форма
    CFrame: integer;  // номер отображаемого кадра
                      // в режиме покадрового просмотра

implementation
{$R *.DFM}

// к следующему кадру
procedure TForm1.Button2Click(Sender: TObject);
begin
    if CFrame = 1 then Button2.Enabled := True;
    if CFrame < Animat1.FrameCount then
    begin
        CFrame := CFrame + 1;
        // вывести кадр
        Animat1.StartFrame := CFrame;
        Animat1.StopFrame := CFrame;
        Animat1.Active := True;

        if CFrame = Animat1.FrameCount // текущий кадр – последний
        then Button2.Enabled:=False;
    end;
end;

// к предыдущему кадру
procedure TForm1.Button3Click(Sender: TObject);
begin
    if CFrame = Animat1.FrameCount
    then Button2.Enabled := True;
    if CFrame > 1 then
    begin
        CFrame := CFrame - 1;
        // вывести кадр
        Animat1.StartFrame := CFrame;
        Animat1.StopFrame := CFrame;
        Animat1.Active := True;

        if CFrame = 1 // текущий кадр – первый
        then Form1.Button3.Enabled := False;
    end;
end;
```

```

// активизация режима просмотра всей анимации
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
    Button1.Enabled:=True; // доступна кнопка Пуск
    // сделать недоступными кнопки покадрового просмотра
    Form1.Button3.Enabled:=False;
    Form1.Button2.Enabled:=False;
end;

// активизация режима покадрового просмотра
procedure TForm1.RadioButton2Click(Sender: TObject);
begin
    Button2.Enabled:=True; // кнопка Следующий кадр доступна
    Button3.Enabled:=False; // кнопка Предыдущий кадр недоступна

    // сделать недоступной кнопку Пуск — вывод всей анимации
    Button1.Enabled:=False;
end;

// пуск и остановка просмотра анимации
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Animatел.Active = False // в данный момент анимация не выводится
    then begin
        Animatел.StartFrame:=1; // вывод с первого
        Animatел.StopFrame:=Animatел.FrameCount; // по последний кадр
        Animatел.Active:=True;
        Button1.caption:='Стоп';
        RadioButton2.Enabled:=False;
    end
    else // анимация отображается
    begin
        Animatел.Active:=False; // остановить отображение
        Button1.caption:='Пуск';
        RadioButton2.Enabled:=True;
    end;
end;
end.

```

Компонент `Animate` позволяет программисту использовать в своих программах стандартные анимации Windows. Вид анимации определяется значением свойства `CommonAVI`, которое задается при помощи именованной константы. В табл. 11.3 приведены некоторые стандартные анимации, описание процесса, для иллюстрации которого они используются, и соответствующие им константы.

Таблица 11.3. Параметры некоторых стандартных анимаций

Анимация	Процесс	Константа
	Копирование файла	AviCopyFile
	Удаление файла	AviDeleteFile
	Удаление файла в корзину	AviRecycleFile

Компонент MediaPlayer

Компонент MediaPlayer, значок которого находится на вкладке **System** (рис. 11.4), позволяет воспроизводить видеоролики, звук и сопровождаемую звуком анимацию.



Рис. 11.4. Значок компонента MediaPlayer

В результате добавления к форме компонента MediaPlayer, на форме появляется группа кнопок (рис. 11.5), подобных тем, которые можно видеть на обычном аудио или видео плеере. Их назначение очевидно (табл. 11.4).



Рис. 11.5. Компонент MediaPlayer

Таблица 11.4. Кнопки компонента MediaPlayer

Кнопка	Обозначение	Действие
Воспроизведение	BtPlay	Воспроизведение звука или видео
Пауза	BtPause	Приостановка воспроизведения
Стоп	BtStop	Остановка воспроизведения

Таблица 11.4 (окончание)

Кнопка	Обозначение	Действие
Следующий	BtNext	Переход к следующему кадру
Предыдущий	BtPrev	Переход к предыдущему кадру
Шаг	BtStep	Переход к следующему звуковому фрагменту, например, к следующей песне на CD
Назад	BtBack	Переход к предыдущему звуковому фрагменту, например, к предыдущей песне на CD
Запись	BtRecord	Запись
Открыть/Закрыть	BtEject	Открытие или закрытие CD-дисковода компьютера

В табл. 11.5 приведены некоторые, наиболее важные, свойства компонента MediaPlayer.

Таблица 11.5. Свойства компонента MediaPlayer

Свойство	Описание
Name	Имя компонента. Используется для доступа к свойствам компонента и управления работой плеера
DeviceType	Тип устройства. Определяет конкретное устройство, которое представляет собой компонент MediaPlayer. Тип устройства задается именованной константой: dtVaweAudio — проигрыватель звука; dtAVIVideo — видеопроигрыватель; dtCDAudio — проигрыватель компакт-дисков
FileName	Имя файла, в котором находится воспроизводимый звуковой фрагмент или видеоролик
AutoOpen	Признак автоматического открытия, сразу после запуска программы, файла видеоролика или звукового фрагмента
Display	Определяет компонент, на поверхности которого воспроизводится видеоролик (обычно в качестве экрана для отображения видео используют компонент Panel)
VisibleButtons	Составное свойство. Определяет видимые кнопки компонента. Позволяет сделать невидимыми некоторые кнопки

Воспроизведение звука

Звуковые фрагменты находятся в файлах с расширением wav. Например, в каталоге C:\Windows\Media можно найти стандартные звуковые файлы Windows.

Следующая программа, вид ее диалогового окна приведен на рис. 11.6, а текст — в листинге 11.2, демонстрирует использование компонента MediaPlayer для воспроизведения звукового фрагмента, находящегося в WAV-файле.

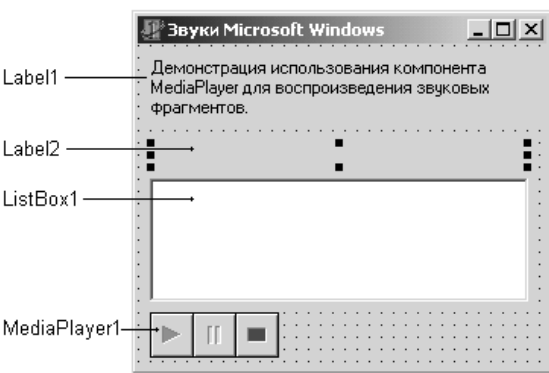


Рис. 11.6. Окно программы Звук Microsoft Windows

Помимо компонента MediaPlayer на форме находится компонент ListBox и два компонента Label, первый из которых используется для вывода информационного сообщения, второй — для отображения имени WAV-файла, выбранного пользователем из списка.

Работает программа следующим образом. После появления диалогового окна воспроизводится "Звук Microsoft" (проигрывается Microsoft.wav), затем пользователь может из списка выбрать любой из находящихся в каталоге C:\Windows\Media звуковых файлов и нажатием на кнопку **Воспроизведение** услышать, что находится в этом файле.

Значения измененных свойств компонента MediaPlayer1 приведены в табл. 11.6, значения остальных свойств оставлены без изменения.

Таблица 11.6. Свойства компонента MediaPlayer1

Компонент	Значение
Name	MediaPlayer1
DeviceType	DtWaveAudio
FileName	C:\Windows\Media\Звук Microsoft.wav
AutoOpen	True

Таблица 11.6 (окончание)

Компонент	Значение
VisibleButtons.btNext	False
VisibleButtons.BtPrev	False
VisibleButtons.btStep	False
VisibleButtons.btBack	False
VisibleButtons.btRecord	False
VisibleButtons.btEject	False

Листинг 11.2. Программа Звуки Microsoft Windows

```

unit WinSound_;

interface

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

type
  TForm1 = class(TForm)
    MediaPlayer1: TMediaPlayer; // медиаплеер
    Label1: TLabel;             // информационное сообщение
    ListBox1: TListBox;         // список WAV-файлов
    Label2: TLabel;             // выбранный из списка файл
    procedure FormActivate(Sender: TObject);
    procedure ListBox1Click(Sender: TObject);
    procedure MediaPlayer1Click(Sender: TObject; Button: TMPBtnType;
      var DoDefault: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

const
  SOUNDPATCH='c:\windows\media\'; // положение звуковых файлов

var
  Form1: TForm1;

implementation

{$R *.DFM}

```

```
procedure TForm1.FormActivate(Sender: TObject);
var
    SearchRec: TSearchRec;    // структура, содержащая информацию
                             // о файле, удовлетворяющем условию поиска
begin
    Form1.MediaPlayer1.Play;

    // сформируем список WAV-файлов, находящихся
    // в каталоге c:\windows\media
    if FindFirst(SOUNDPATCH+'*.wav', faAnyFile, SearchRec) = 0 then
    begin
        // в каталоге есть файл с расширением wav
        // добавим имя этого файла в список
        Form1.ListBox1.Items.Add(SearchRec.Name);
        // пока в каталоге есть другие файлы с расширением wav
        while (FindNext(SearchRec) = 0) do
            Form1.ListBox1.Items.Add(SearchRec.Name);
    end;
end;

// щелчок на элементе списка
procedure TForm1.ListBox1Click(Sender: TObject);
begin
    // вывести в поле метки Label2 имя выбранного файла
    Label2.Caption:=ListBox1.Items[ListBox1.ItemIndex];
end;

// щелчок на кнопке компонента MediaPlayer
procedure TForm1.MediaPlayer1Click(Sender: TObject; Button: TMPBtnType;
    var DoDefault: Boolean);
begin
    if (Button = btPlay) and (Label2.Caption <> '') then
    begin
        // нажата кнопка Play
        with MediaPlayer1 do
        begin
            FileName:=SOUNDPATCH+Label2.Caption; // имя выбранного файла
            Open; // открыть звуковой файл
            Play; // проиграть звуковой файл
        end;
    end;
end;

end;
```

Воспроизведение звука сразу после запуска программы активизирует процедура обработки события `onFormActivate` путем применения метода `Play` к компоненту `MediaPlayer1` (действие этого метода аналогично нажатию на кнопку **Воспроизведение**). Эта же процедура формирует список WAV-файлов, находящихся в каталоге `C:\Windows\Media`. Для формирования списка используются функции `FindFirst` и `FindNext`, которые выполняют поиск соответственно первого и следующего (по отношению к последнему, найденному функцией `FindFirst` или `FindNext`) файла, удовлетворяющего указанному при вызове функций критерию. Обеим функциям в качестве параметров передаются маска WAV-файла (критерий поиска) и переменная-структура `SearchRec`, поле `Name` которой в случае успешного поиска будет содержать имя файла, удовлетворяющего критерию поиска.

Щелчок на элементе списка обрабатывается процедурой `TForm1.ListBox1Click`, которая выводит в поле метки `Label2` имя файла, выбранного пользователем (во время работы программы свойство `ItemIndex` содержит номер элемента списка, на котором выполнен щелчок).

В результате нажатия одной из кнопок компонента `MediaPlayer1` активизируется процедура `TForm1.MediaPlayer1Click`, которая проверяет, какая из кнопок компонента была нажата. Если нажата кнопка **Воспроизведение** (`btPlay`), то в свойство `Name` записывается имя выбранного пользователем файла, затем, вызовом метода `Open`, данный файл открывается. После этого метод `Play` активизирует воспроизведение WAV-файла.

Наличие у компонента `MediaPlayer` свойства `Visible` позволяет скрыть компонент от пользователя и при этом использовать его для воспроизведения звука без участия пользователя. Например, следующая программа пересчитывает вес из фунтов в килограммы и сопровождает выдачу результата звуковым сигналом. В случае, если пользователь забудет ввести исходные данные или введет их неверно, программа выводит сообщение об ошибке, также сопровождаемое звуковым сигналом. Вид диалогового окна программы во время ее разработки приведен на рис. 11.7, значения свойств компонента `MediaPlayer1` — в табл. 11.7. Текст модуля программы представлен в листинге 11.3.

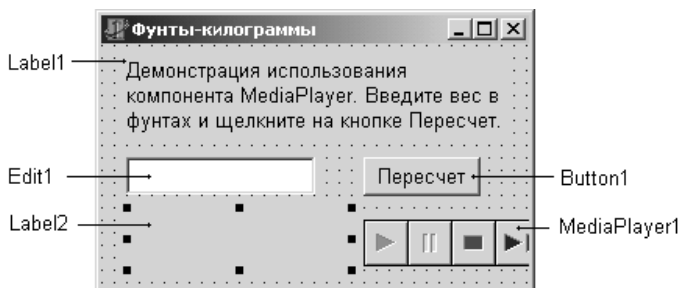


Рис. 11.7. Диалоговое окно программы

Таблица 11.7. Значения свойств компонента *MediaPlayer1*

Свойство	Значение
Name	MediaPlayer1
DeviceType	DtWaveAudio
FileName	c:\windows\media\ding.wav
AutoOpen	True
Visible	False

Листинг 11.3. Использование *MediaPlayer* для вывода звука

```

unit FuntToKg1_;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // поле ввода веса в фунтах
    Button1: TButton;       // кнопка Пересчет
    Label2: TLabel;        // поле вывода результата
    Label1: TLabel;        // поле информационного сообщения
    MediaPlayer1: TMediaPlayer; // медиаплеер
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

// щелчок на кнопке Пересчет
procedure TForm1.Button1Click(Sender: TObject);

```

```

var
  f: real; // вес в фунтах
  k: real; // вес в килограммах

begin
  form1.Label2.Caption:='';
  try // возможна ошибка, если в поле
      // Edit1 будет не число
    f:=StrToFloat(Edit1.Text);
    Form1.MediaPlayer1.Play; // звуковой сигнал
    k:=f*0.4095;
    label2.caption:=Edit1.text+' ф. — это ' +
      FloatToStrF(k,ffGeneral,4,2)+' кг.';
  except
    on EConvertError do // ошибка преобразования
      begin
        // определим и проиграем звук "Ошибка"
        Form1.MediaPlayer1.FileName:='c:\windows\media\chord.wav';
        Form1.MediaPlayer1.Open;
        Form1.MediaPlayer1.Play; // звуковой сигнал
        ShowMessage('Ошибка! Вес следует ввести числом.');
```

form1.Edit1.SetFocus; // курсор в поле ввода

// восстановим звук

Form1.MediaPlayer1.FileName:='c:\windows\media\ding.wav';

Form1.MediaPlayer1.Open;

end;

end;

end;

end.

Запись звука

В некоторых случаях программисту могут потребоваться специфические звуки или музыкальные фрагменты, которые не представлены на диске компьютера в виде WAV-файла. В этом случае возникает задача создания, или, как говорят, записи WAV-файла.

Наиболее просто получить представление нужного звукового фрагмента в виде WAV-файла можно при помощи входящей в состав Windows программы **Звукозапись**. Программа **Звукозапись**, вид ее диалогового окна приведен на рис. 11.8, запускается из главного меню Windows при помощи команды **Пуск/Программы/Стандартные/Развлечения/Звукозапись**.

Источником звука для программы **Звукозапись** может быть микрофон, аудио CD или любое другое подключенное к линейному входу звуковой платы

компьютера устройство, например аудио магнитофон. Кроме того, возможно микширование (смешение) звуков различных источников.

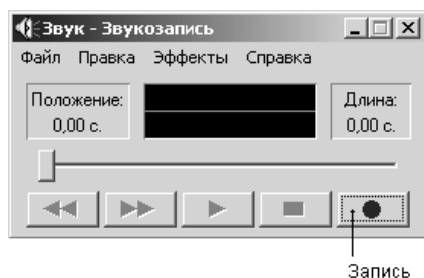


Рис. 11.8. Диалоговое окно программы **Звукозапись**

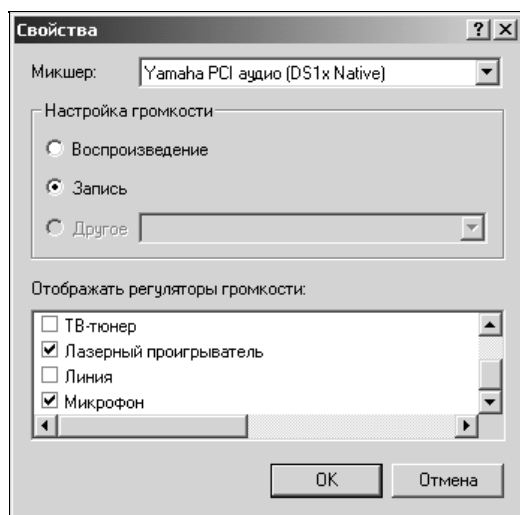


Рис. 11.9. Диалоговое окно **Свойства**

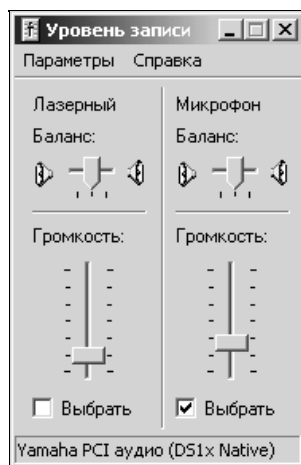


Рис. 11.10. Диалоговое окно **Уровень записи** позволяет управлять записываемым сигналом

Создается WAV-файл следующим образом. Сначала нужно определить источник (или источники) звука. Для этого надо щелкнуть правой кнопкой мыши на находящемся на панели задач изображении динамика и из появившегося меню выбрать команду **Регулятор громкости**, а из меню **Параметры** — команду **Свойства**. Затем в появившемся окне **Свойства** (рис. 11.9) выбрать переключатель **Запись** и в списке **Отображать регуляторы громкости** — установить флажки, соответствующие тем устройствам, сигнал с которых нужно записать. После нажатия кнопки **ОК** на экране появляется окно **Уровень записи** (рис. 11.10), используя которое можно управлять как уровнем сигнала (громкостью) каждого источника звука в общем звуке, так и величиной общего, суммарного сигнала, поступающего на вход програм-

мы **Звукозапись**. Величина сигнала задается путем перемещения движков соответствующих регуляторов. Следует обратить внимание на то, что движки регуляторов группы **Уровень записи** доступны только во время процесса записи звука. На этом подготовительные действия заканчиваются. Теперь можно приступить непосредственно к записи звука.

Чтобы записать музыкальный или речевой фрагмент, надо запустить программу **Звукозапись**, активизировать диалоговое окно **Уровень записи**, выбрать устройство-источник звука, инициировать процесс звучания (если запись осуществляется, например, с CD) и в нужный момент времени нажать кнопку **Запись**.

Во время записи в диалоговых окнах можно наблюдать изменение сигнала на выходе микшера (индикатор **Громкость** диалогового окна **Уровень записи**) и на входе программы записи. На рис. 11.11 в качестве примера приведен вид диалогового окна **Звукозапись** во время записи звука.

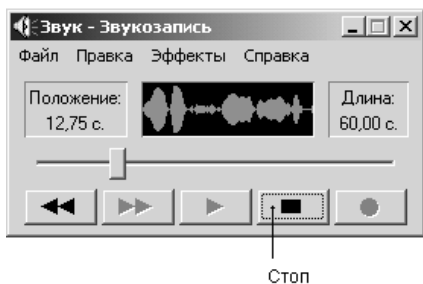


Рис. 11.11. Диалоговое окно **Звукозапись** во время записи

Для остановки процесса записи следует нажать кнопку **Стоп**.

Сохраняется записанный фрагмент в файле обычным образом, т. е. выбором из меню **Файл** команды **Сохранить** или **Сохранить как**. При выборе команды **Сохранить как** можно выбрать формат, в котором будет сохранен записанный звуковой фрагмент.

Существует несколько форматов звуковых файлов. В частности, возможно сохранение с различным качеством как стерео, так и моно звука. Здесь следует понимать, что чем выше качество записи, тем больше места на диске компьютера требуется для хранения соответствующего WAV-файла. Считается, что для речи приемлемым является формат "22050 Гц, 8 бит, моно", а для музыки — "44 100 Гц, 16 бит, моно" или "44 100 Гц, 16 бит, стерео".

Просмотр видеороликов и анимации

Помимо воспроизведения звука, компонент `MediaPlayer` позволяет просматривать видеоролики и мультипликации, представленные как AVI-файлы.

Процесс использования компонента `MediaPlayer` для вывода находящейся в AVI-файле информации рассмотрим на примере программы, которая вос-

производит на поверхности формы сопровождаемую звуковым эффектом мультипликацию — вращающееся по часовой стрелке слово `Delphi` (файл `delphi.avi`, содержащий этот мультик находится на прилагаемом к книге сопроводительном диске).

Вид диалогового окна программы приведен на рис. 11.12, а значения свойств компонента `MediaPlayer1` — в табл. 11.8.



Рис. 11.12. Диалоговое окно программы

Таблица 11.8. Значения свойств компонента `MediaPlayer1`

Свойство	Значение	Свойство	Значение
Name	MediaPlayer1	AutoOpen	True
FileName	delphi.avi	Display	Panel1
DeviceType	DtAVIVideo	Visible	False

Создается форма приложения обычным образом. Компонент `Panel1` применяется в качестве экрана, на который осуществляется вывод анимации, и его имя используется в качестве значения свойства `Display` компонента `MediaPlayer1`. Поэтому сначала к форме лучше добавить компонент `Panel`, а затем — `MediaPlayer`. Такой порядок создания формы позволяет установить значение свойства `Display` путем выбора из списка.

Следует особо обратить внимание на то, что размер области вывода анимации на панели определяется не значениями свойств `Width` и `Height` панели (хотя их значения должны быть как минимум такими же, как ширина и высота анимации). Размер области задается значением свойства `DisplayRect` компонента `MediaPlayer`. Свойство `DisplayRect` во время разработки программы недоступно (его значение не выводится в окне **Object Inspector**). Поэтому значение свойства `DisplayRect` устанавливается во время работы программы в результате выполнения следующей инструкции:

```
MediaPlayer1.DisplayRect:=Rect(0, 0, 60, 60).
```


Примечание

Чтобы получить информацию о размере кадров AVI-файла, надо, используя возможности Windows, открыть папку, в которой находится этот файл, щелкнуть правой кнопкой мыши на имени файла, выбрать команду **Свойства**, а в появившемся диалоговом окне — вкладку **Сводка**, в которой выводится подробная информация о файле, в том числе и размер кадров.

Текст программы приведен в листинге 11.4.

Листинг 11.4. Воспроизведение сопровождаемой звуком анимации

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, MPlayer, StdCtrls, ExtCtrls;

type

TForm1 = **class**(TForm)

Label1: TLabel; *// информационное сообщение*

Panel1: TPanel; *// панель, на которую выводится анимация*

Button1: TButton; *// кнопка Ok*

MediaPlayer1: TMediaPlayer; *// универсальный проигрыватель*

procedure Button1Click(Sender: TObject);

procedure FormCreate(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{ \$R *.DFM }

procedure TForm1.Button1Click(Sender: TObject);

begin

MediaPlayer1.Play; *// воспроизведение анимации*

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

```
// зададим размер области вывода анимации  
// на поверхности формы  
MediaPlayer1.DisplayRect:=Rect(0, 0, 60, 60);
```

```
end;
```

```
end.
```

Процесс воспроизведения анимации активизируется применением метода `Play`, что эквивалентно нажатию кнопки **Play** в случае, если кнопки компонента `MediaPlayer` доступны пользователю.

Создание анимации

Процесс создания файла анимации (AVI-файла) рассмотрим на примере. Пусть надо создать анимацию, которая воспроизводит процесс рисования эскиза Дельфийского храма (окончательный вид рисунка представлен на рис. 11.13, несколько кадров анимации — на рис. 11.14).



Рис. 11.13. Эскиз Дельфийского храма

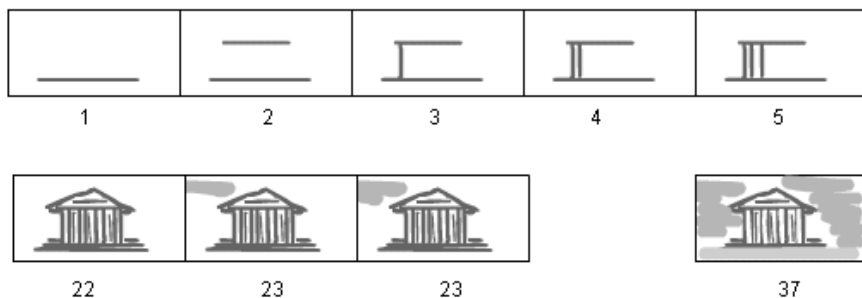


Рис. 11.14. Кадры анимации процесса рисования Дельфийского храма

Для решения поставленной задачи можно воспользоваться популярной программой `Macromedia Flash 5`.

В `Macromedia Flash` анимация, которую также довольно часто называют роликом (*Movie*), состоит из *слоев*. В простейшем случае ролик представляет собой один-единственный слой (*Layer*). Слой — это последовательность кадров (*Frame*), которые в процессе воспроизведения анимации выводятся последовательно, один за другим. Если ролик состоит из нескольких слоев, то кадры анимации получаются путем наложения кадров одного слоя на кадры другого. Например, один слой может содержать изображение фона,

на котором разворачивается действие, а другой — изображение персонажей. Возможность формирования изображения путем наложения слоев существенно облегчает процесс создания анимации. Таким образом, чтобы создать анимацию, нужно распределить изображение по слоям и для каждого слоя создать кадры.

После запуска Macromedia Flash на фоне главного окна программы появляется окно **Movie1** (рис. 11.15), которое используется для создания анимации. В верхней части окна, которая называется *Timeline*, отражена структура анимации, в нижней части, именуемой рабочей областью, находится изображение текущего кадра выбранного слоя. После запуска Macromedia Flash анимация состоит из одного единственного слоя (*Layer 1*), который в свою очередь представляет один-единственный пустой (чистый) кадр.

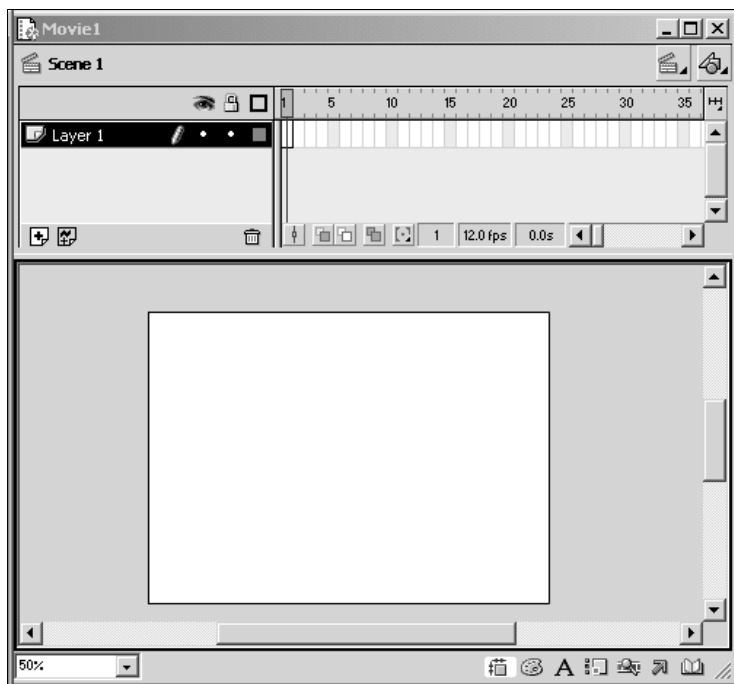


Рис. 11.15. Окно **Movie** в начале работы над новой анимацией

Перед тем как приступить непосредственно к созданию кадров анимации, нужно задать общие характеристики анимации (ролика): размер кадров и скорость их воспроизведения. Характеристики вводятся в поля диалогового окна **Movie Properties** (рис. 11.16), которое появляется в результате выбора из меню **Modify** команды **Movie**. В поле **Frame Rate** нужно ввести скорость воспроизведения ролика, которая измеряется в кадрах в секунду (*fps* — *frame per second* — кадров в секунду), в поля **Width** и **Height** — ширину и

высоту кадров. В этом же окне можно выбрать фон кадров (список **Background Color**).

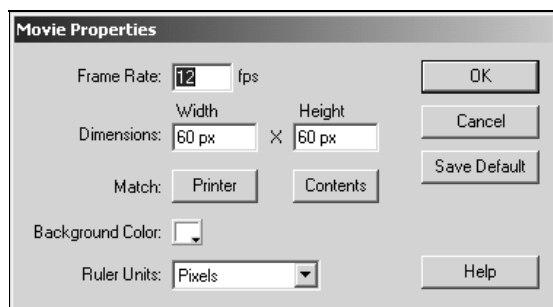


Рис. 11.16. Характеристики ролика отображаются в окне **Movie Properties**

После того, как установлены характеристики ролика, можно приступить к созданию кадров анимации.

Первый кадр нужно просто нарисовать. Технология создания изображений Macromedia Flash обычная, используется стандартный набор инструментов: кисть, карандаш, пульверизатор, резинка и др.

Чтобы создать следующий кадр, нужно из меню **Insert** выбрать команду **Keyframe**. В результате в текущий слой будет добавлен кадр, в который скопировано содержимое предыдущего кадра (т. к. в большинстве случаев следующий кадр создается путем изменения предыдущего). Теперь можно нарисовать второй кадр. Аналогичным образом создаются остальные кадры анимации.

Иногда не нужно, чтобы новый кадр содержал изображение предыдущего, в этом случае вместо команды **Keyframe** следует воспользоваться командой **Blank Keyframe**.

Если некоторое изображение должно оставаться статичным в течение времени кратного выводу нескольких кадров, то вместо того, чтобы вставлять в слой несколько одинаковых кадров (**Keyframe**), нужно сделать кадр статичным. Если кадр, изображение которого должно быть статичным, является последним кадром ролика, то в окне **Timeline** нужно выделить кадр, до которого изображение должно оставаться статичным, и из меню **Insert** выбрать команду **Frame**. Если кадр, изображение которого должно быть статичным, не является последним, то нужно выделить этот кадр и несколько раз из меню **Insert** выбрать команду **Frame**.

Можно значительно облегчить работу по созданию анимации, если разделить изображение на основное и фоновое, поместив каждое в отдельный слой (именно так поступают при создании мультфильмов). Сначала нужно создать кадры слоя фона так, как было описано выше. Затем, выбрав из меню **Insert** команду **Layer**, надо добавить слой основного действия.

Следует обратить внимание, что все действия по редактированию изображения направлены на текущий кадр выбранного слоя. В списке слоев выбранный слой выделен цветом, номер текущего кадра помечен маркером — красным квадратиком.

Чтобы выводимая анимация сопровождалась звуком, нужно сначала сделать доступным соответствующий звуковой файл. Для этого надо из меню **File** выбрать команду **Import** и добавить в проект звуковой файл (рис. 11.17).

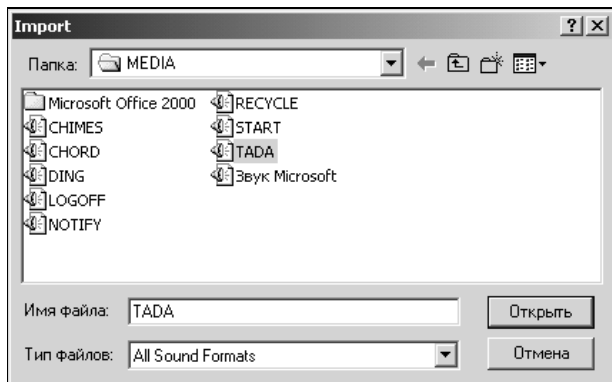


Рис. 11.17. Импорт звукового файла



Рис. 11.18. Диалоговое окно **Sound**

Затем в окне **Timeline** нужно выделить кадр, при отображении которого должно начаться воспроизведение звукового фрагмента, и, используя диалоговое окно **Sound** (рис. 11.18), выбрать звуковой фрагмент и задать, если необходимо, параметры его воспроизведения. Количество повторов нужно ввести в поле **Loops**, эффект, используемый при воспроизведении, можно выбрать из списка **Effect**.

В качестве примера на рис. 11.19 приведен вид окна **Timeline** в конце работы над анимацией. Анимация состоит из двух слоев. Слой *Layer 2* — содержит фон. Детали фона появляются постепенно, в течение 9 кадров. После этого фон не меняется, поэтому 9-й кадр является статичным. Слой *Layer 1* — слой основного действия, которое начинается после того, как будет выведен фон. Вывод анимации заканчивается стандартным звуком **TADA** (его длительность равна одной секунде). Начало воспроизведения звука совпадает с выводом последнего (49, если считать от начала ролика) кадра основного действия, поэтому этот кадр сделан статическим в течение вывода следующих 12 кадров (скорость вывода анимации — 12-ти кадров в секунду). Сделано это для того, чтобы процесс вывода анимации завершился одновременно с окончанием звукового сигнала.

После того как ролик будет готов, его надо сохранить. Делается это обычным образом, т. е. выбором из меню **File** команды **Save**.

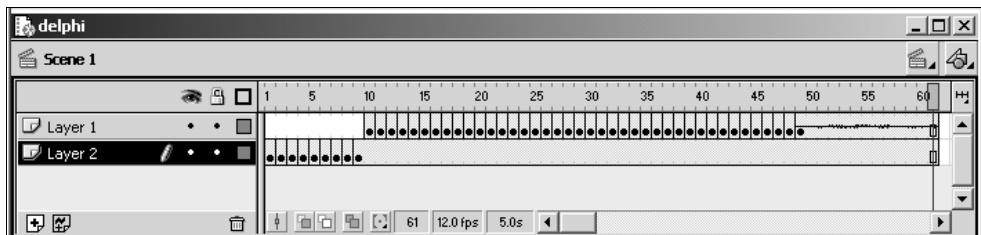


Рис. 11.19. Пример анимации

Для преобразования файла из формата *Macromedia Flash* в формат *AVI* нужно из меню **File** выбрать команду **Export Movie** и задать имя файла. Затем в появившемся диалоговом окне **Export Windows AVI** (рис. 11.20) надо задать размер кадра (поля **Width** и **Height**), из списка **Video Format** выбрать формат, в котором будет записана видео часть ролика, а из поля **Sound Format** — формат звука.

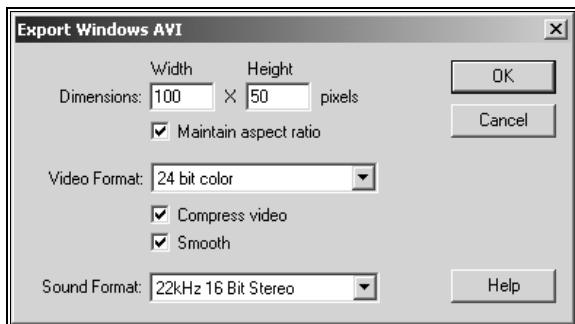
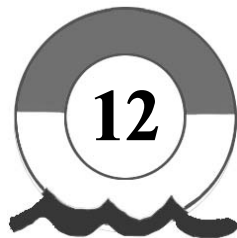


Рис. 11.20. Окно **Export Windows AVI**

Если установлен переключатель **Compress video**, то после нажатия кнопки **OK** появится диалоговое окно, в котором можно будет выбрать один из стандартных методов сжатия видео. При выборе видео и звукового формата нужно учитывать, что чем более высокие требования будут предъявлены к качеству записи звука и изображения, тем больше места на диске займет *AVI*-файл. Здесь нужно обратить внимание, что завышенные требования не всегда оправданы.



Рекурсия

Понятие рекурсии

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя. Факториал — это классический пример рекурсивного объекта. Факториал числа n — это произведение целых чисел от 1 до n . Обозначается факториал числа n так: $n!$.

Согласно определению

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

приведенное выражение можно переписать следующим образом:

$$n! = n \times ((n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1) = n \times (n-1)!$$

Следовательно, факториал числа n равен произведению числа n на факториал числа $(n-1)$. В свою очередь, факториал числа $(n-1)$ — это произведение числа $(n-1)$ на факториал числа $(n-2)$ и т. д.

Таким образом, если вычисление факториала n реализовать как функцию, то в теле этой функции будет инструкция вызова функции вычисления факториала числа $(n-1)$, т. е. функция будет вызывать сама себя. Такой способ вызова называется *рекурсией*, а функция, которая обращается сама к себе, — *рекурсивной функцией*.

В листинге 12.1 приведена рекурсивная функция вычисления факториала.

Листинг 12.1. Рекурсивная функция вычисления факториала

```
function factorial(n: integer): integer;  
begin  
    if n <> 1
```

```

then factorial:= n * factorial(n-1) // функция вызывает сама себя
else factorial:= 1; // рекурсивный процесс закончен
end;

```

Обратите внимание, что функция вызывает сама себя только в том случае, если значение полученного параметра *n* не равно единице. Если значение параметра равно единице, то функция сама себя не вызывает, а возвращает значение, и рекурсивный процесс завершается.

На рис. 12.1 приведен вид диалогового окна программы, которая для вычисления факториала числа использует рекурсивную функцию *factorial*. Текст программы приведен в листинге 12.2.



Рис. 12.1. Окно программы вычисления факториала

Листинг 12.2. Использование рекурсивной функции

```

unit factor_;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Button1: TButton;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

```



```

var
    Form1: TForm1;

implementation

{$R *.DFM}

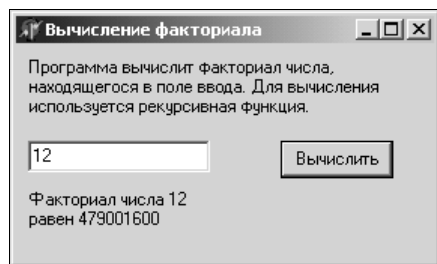
// рекурсивная функция
function factorial(n: integer): integer;
begin
    if n > 1
    then factorial := n * factorial(n-1) // функция вызывает сама себя
    else factorial:= 1; // факториал 1 равен 1
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    k:integer; // число, факториал которого надо вычислить
    f:integer; // значение факториала числа k
begin
    k := StrToInt(Edit1.Text);
    f := factorial(k);
    label2.caption:='Факториал числа '+Edit1.Text
                    +' равен '+IntToStr(f);
end;

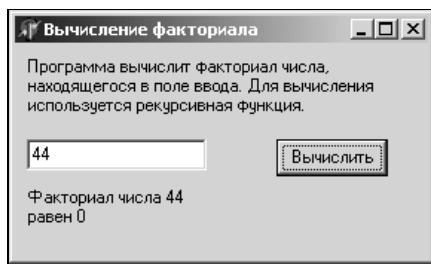
end.

```

На рис. 12.2 приведены два примера работы программы. Результат вычисления факториала, представленный на рис. 12.2, а, соответствует ожидаемому.



а)



б)

Рис. 12.2. Примеры работы программы вычисления факториала

Результат, представленный на рис. 12.2, б, не соответствует ожидаемому. Факториал числа 44 равен нулю! Произошло это потому, что факториал числа 44 настолько велик, что превысил максимальное значение для пере-

менной типа `integer`, и, как говорят программисты, "произошло переполнение с потерей значения".

Примечание

Delphi может включать в исполняемую программу инструкции контроля диапазона значений переменных. Чтобы инструкции контроля были добавлены в программу, нужно на вкладке **Compiler** диалогового окна **Project Options** (рис. 12.3) установить флажок **Overflow checking** (Контроль переполнения), который находится в группе **Runtime errors** (Ошибки времени выполнения).

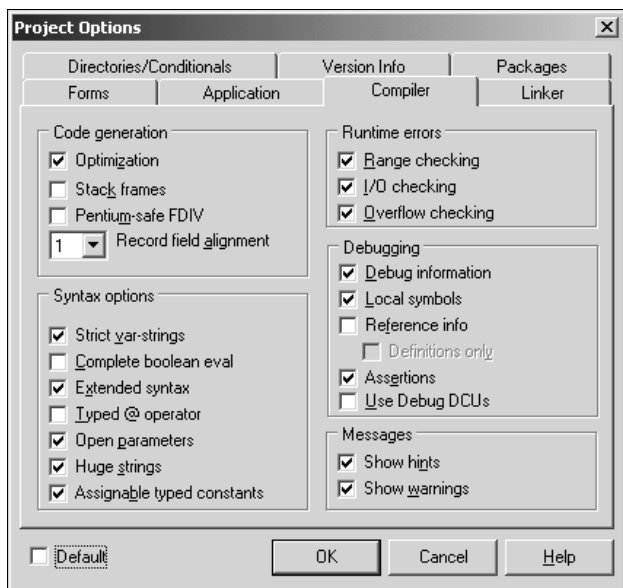


Рис. 12.3. Вкладка **Compiler** диалогового окна **Project Options**

Примеры программ

Поиск файлов

В качестве примера использования рекурсии рассмотрим задачу поиска файлов. Пусть нужно получить список всех файлов, например, с расширением `bmp`, которые находятся в указанном пользователем каталоге и во всех подкаталогах этого каталога.

Словесно алгоритм обработки каталога может быть представлен так:

- ☐ вывести список всех файлов, удовлетворяющих критерию запроса;
- ☐ если в каталоге есть подкаталоги, то обработать каждый из этих подкаталогов.

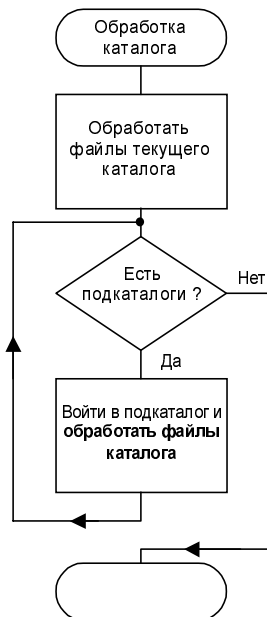


Рис. 12.4. Рекурсивный алгоритм поиска файлов

Приведенный алгоритм, его блок-схема приведена на рис. 12.4, является рекурсивным: для того чтобы обработать подкаталог, процедура обработки текущего каталога должна вызвать сама себя.

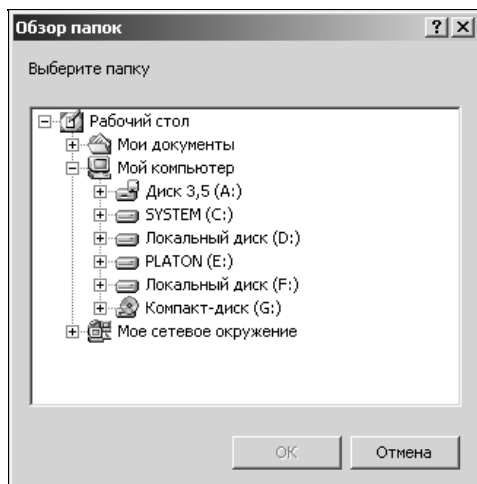
Вид диалогового окна программы приведен на рис. 12.5, текст — в листинге 12.3.

Поле **Файл** (Edit1) используется для ввода имени искомого файла или маски (для поиска файлов одного типа). Имя каталога, в котором нужно выполнить поиск, можно ввести непосредственно в поле **Папка** или выбрать из стандартного диалогового окна **Обзор папок**, которое появляется в результате нажатия кнопки **Папка**. Окно **Обзор папок** (рис. 12.6) выводится на экран стандартной функцией

SelectDirectory. Следует обратить внимание, что имя каталога, который используется в диалоговом окне **Обзор папок** в качестве корневого, должно передаваться функции SelectDirectory как строка WideString. Для преобразования обычной строки в строку WideString, использована функция StringToWideChar.



Рис. 12.5. Окно программы Поиск файлов

Рис. 12.6. Диалоговое окно **Обзор папок** появляется в результате щелчка на кнопке **Папка**

Основную работу выполняет рекурсивная функция Find. У функции Find один-единственный параметр — структура SearchRec, которая используется функциями FindFirst и FindNext для поиска соответственно первого и следующего файла, удовлетворяющего критерию поиска. Следует обратить внимание на то, как осуществляется перебор каталогов в текущем каталоге. Если текущий каталог не корневой, то помимо обычных, т. е. имеющих имя, в каталоге есть еще два каталога: .. и ., которые обозначают каталог предыдущего уровня. Эти два каталога не обрабатываются, т. к. при входе в них фактически выполняется выход, переход в родительский каталог. Если это не учесть, то программа заиклится.

Листинг 12.3. Программа поиска файлов

```
// поиск файла в указанном каталоге и его подкаталогах
// используется рекурсивная процедура Find
unit FindFile_;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;           // что искать
    Edit2: TEdit;           // где искать
    Memo1: TMemo;           // результат поиска
    Button1: TButton;        // кнопка Найти
    Button2: TButton;        // кнопка Папка
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

```

var
  FileName: string; // имя или маска искомого файла
  cDir: string;
  n: integer; // кол-во файлов, удовлетворяющих запросу

// поиск файла в текущем каталоге
procedure Find;
var
  SearchRec: TSearchRec; // информация о файле или каталоге
begin
  GetDir(0,cDir); // получить имя текущего каталога
  if cDir[length(cDir)] <> '\' then cDir := cDir+'\\';

  if FindFirst(FileName, faArchive,SearchRec) = 0 then
    repeat
      if (SearchRec.Attr and faAnyFile) = SearchRec.Attr then
        begin
          Form1.Memo1.Lines.Add(cDir + SearchRec.Name);
          n := n + 1;
        end;
    until FindNext(SearchRec) <> 0;

  // обработка подкаталогов текущего каталога
  if FindFirst('*', faDirectory, SearchRec) = 0 then
    repeat
      if (SearchRec.Attr and faDirectory) = SearchRec.Attr then
        begin
          // каталоги .. и . тоже каталоги,
          // но в них входить не надо !!!
          if SearchRec.Name[1] <> '.' then
            begin
              ChDir(SearchRec.Name); // войти в каталог
              Find; // выполнить поиск в подкаталоге
              ChDir('..'); // выйти из каталога
            end;
          end;
        until FindNext(SearchRec) <> 0;
    end;

// возвращает каталог, выбранный пользователем
function GetPath(mes: string):string;
var
  Root: string; // корневой каталог
  pwRoot : PWideChar;
  Dir: string;

```

```

begin
  Root := '';
  GetMem(pwRoot, (Length(Root)+1) * 2);
  pwRoot := StringToWideChar(Root, pwRoot, MAX_PATH*2);
  if SelectDirectory(mes, pwRoot, Dir)
  then
    if length(Dir) = 2 // пользователь выбрал корневой каталог
    then GetPath := Dir+'\'
    else GetPath := Dir
  else
    GetPath := '';
end;

// щелчок на кнопке Поиск
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memol.Clear; // очистить поле Мемо1
  Label4.Caption := '';
  FileName := Edit1.Text; // что искать
  cDir := Edit2.Text; // где искать
  n:=0; // кол-во найденных файлов
  ChDir(cDir); // войти в каталог начала поиска
  Find; // начать поиск
  if n = 0 then
    ShowMessage('Файлов, удовлетворяющих критерию поиска нет.')
  else Label4.Caption := 'Найдено файлов:' + IntToStr(n);
end;

// щелчок на кнопке Папка
procedure TForm1.Button2Click(Sender: TObject);
var
  Path: string;
begin
  Path := GetPath('Выберите папку');
  if Path <> ''
  then Edit2.Text := Path;
end;

end.

```

Кривая Гильберта

Еще один пример рекурсии — программа, которая вычерчивает в диалоговом окне кривую Гильберта. На рис. 12.7 приведены кривые Гильберта первого, второго и третьего порядков. Если присмотреться, то видно, что кри-

вая второго порядка получается путем соединения прямыми линиями четырех кривых первого порядка, две из которых повернуты на 90 градусов: одна по часовой стрелке, другая — против. Аналогичным образом строится кривая третьего порядка, но при этом в качестве "кирпичиков" используются кривые второго порядка. Таким образом, чтобы нарисовать кривую третьего порядка, надо нарисовать четыре кривых второго порядка. В свою очередь, чтобы нарисовать кривую второго порядка, нужно нарисовать четыре кривых первого порядка. Следовательно, алгоритм вычерчивания кривой Гильберта является рекурсивным.

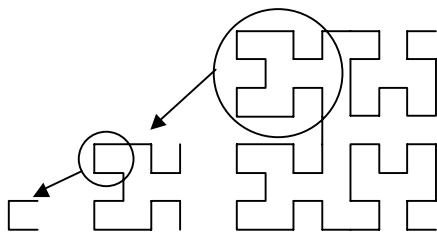


Рис. 12.7. Кривые Гильберта первого, второго и третьего порядков



Рис. 12.8. Кривая Гильберта пятого порядка

Диалоговое окно программы **Кривая Гильберта**, в котором находится кривая пятого порядка, приведено на рис. 12.8, текст программы — в листинге 12.4.

Листинг 12.4. Кривая Гильберта

```
unit gilbert_;

interface

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

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
    Form1: TForm1;

implementation

{$R *.dfm}

var
    p: integer = 5;  // порядок кривой
    u: integer = 5;  // длина штриха

{ Кривая Гильберта состоит из четырех элементов a, b, c и d,
  соединенных прямыми.
  Каждый элемент строит соответствующая процедура. }

procedure a(i:integer; canvas: TCanvas); forward;
procedure b(i:integer; canvas: TCanvas); forward;
procedure c(i:integer; canvas: TCanvas); forward;
procedure d(i:integer; canvas: TCanvas); forward;

// Элементы кривой
procedure a(i: integer; canvas: TCanvas);
begin
    if i > 0 then begin
        d(i-1, canvas); canvas.LineTo(canvas.PenPos.X-u, canvas.PenPos.Y);
        a(i-1, canvas); canvas.LineTo(canvas.PenPos.X, canvas.PenPos.Y+u);
        a(i-1, canvas); canvas.LineTo(canvas.PenPos.X+u, canvas.PenPos.Y);
        b(i-1, canvas);
    end;
end;

procedure b(i: integer; canvas: TCanvas);
begin
    if i > 0 then
        begin
            c(i-1, canvas); canvas.LineTo(canvas.PenPos.X, canvas.PenPos.Y-u);
            b(i-1, canvas); canvas.LineTo(canvas.PenPos.X+u, canvas.PenPos.Y);
            b(i-1, canvas); canvas.LineTo(canvas.PenPos.X, canvas.PenPos.Y+u);
            a(i-1, canvas);
        end;
end;

procedure c(i: integer; canvas: TCanvas);
begin
    if i > 0 then
        begin
            b(i-1, canvas); canvas.LineTo(canvas.PenPos.X+u, canvas.PenPos.Y);
            c(i-1, canvas); canvas.LineTo(canvas.PenPos.X, canvas.PenPos.Y-u);
```



```

    c(i-1, canvas); canvas.LineTo(canvas.PenPos.X-u, canvas.PenPos.Y);
    d(i-1, canvas);
end;
end;

procedure d(i: integer; canvas: TCanvas);
begin
    if i > 0 then
    begin
        a(i-1, canvas); canvas.LineTo(canvas.PenPos.X, canvas.PenPos.Y+u);
        d(i-1, canvas); canvas.LineTo(canvas.PenPos.X-u, canvas.PenPos.Y);
        d(i-1, canvas); canvas.LineTo(canvas.PenPos.X, canvas.PenPos.Y-u);
        c(i-1, canvas);
    end;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    Form1.Canvas.MoveTo(Form1.ClientWidth-u, u);
    a(p, Form1.Canvas); // вычертить кривую Гильберта
end;

end.
```

Следует обратить внимание на следующую особенность реализации программы. Процедура, которая вычерчивает элемент *a*, помимо самой себя (для вычерчивания элемента *a* кривой более низкого порядка), вызывает процедуры *d* и *b*, описание (текст) которых в тексте программы находится после процедуры *a*. Чтобы компилятор не вывел сообщение об ошибке, в текст программы помещено *объявление* процедуры с ключевым словом *forward*, означающим, что это только объявление, а *описание* (реализация) находится дальше. Таким образом, уже в процессе компиляции процедуры *a* компилятор "знает", что имена *b* и *d* означают процедуры.

Поиск пути

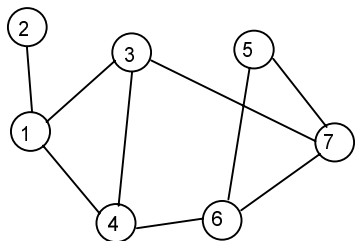
Механизм рекурсии весьма эффективен при программировании задач поиска. В качестве еще одного примера рассмотрим задачу поиска пути между двумя городами. Если несколько городов соединены дорогами, то очевидно, что попасть из одного города в другой можно различными маршрутами. Задача состоит в нахождении всех возможных маршрутов.

Карта дорог между городами может быть изображена в виде графа — набора вершин, означающих города, и ребер, обозначающих дороги (рис. 12.9).

Процесс поиска может быть представлен как последовательность шагов. На каждом шаге с использованием некоторого критерия выбирается точка,

в которую можно попасть из текущей. Если очередная выбранная точка совпала с заданной конечной точкой, то маршрут найден. Если не совпала, то делаем из этой точки еще шаг. Так как текущая точка может быть соединена с несколькими другими, то нужен какой-то формальный критерий выбора. В простейшем случае можно выбрать точку с наименьшим номером.

Пусть, например, надо найти все возможные пути из точки 1 в точку 5. Согласно принятому правилу, сначала выбираем точку 2. На следующем шаге выясняем, что точка 2 тупиковая, поэтому возвращаемся в точку 1 и делаем шаг в точку 3. Из точки 3 — в точку 4, из 4 — в 6 и из точки 6 — в точку 5. Один маршрут найден. После этого возвращаемся в точку 6 и проверяем, возможен ли шаг в точку, отличную от 5. Так как это возможно, то делаем шаг в точку 7 и затем в 5. Найден еще один путь. Таким образом, процесс поиска состоит из шагов вперед и возвратов назад. Поиск завершается, если из узла начала движения уже некуда идти.



Алгоритм поиска имеет рекурсивный характер: чтобы сделать шаг, мы выбираем точку и опять делаем шаг, и так продолжаем до тех пор, пока не достигнем цели.

Рис. 12.9. Представление карты дорог в виде графа

Таким образом, задача поиска маршрута может рассматриваться как задача выбора очередной точки (города) и поиска оставшейся части маршрута, т. е. имеет место рекурсия.

Граф можно представить двумерным массивом, который назовем `map` (карта). Значение элемента массива `map[i, j]` — это расстояние между городами `i` и `j`, если города соединены дорогой, или ноль, если города не соединены прямой дорогой. Для приведенного графа массив `map` можно изобразить в виде таблицы следующим образом:

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	1	0	0	1
4	1	0	1	0	0	1	0
5	0	0	0	0	0	1	1
6	0	0	0	1	1	0	1
7	0	0	1	0	1	1	0

Содержимое ячейки таблицы на пересечении строки i и столбца j соответствует значению `map[i, j]`.

Помимо массива `map` нам потребуются массив `road` (дорога) и массив `incl` (от `include` — включать). В массив `road` мы будем записывать номера пройденных городов. В момент достижения конечной точки он будет содержать номера всех пройденных точек, т. е. описание маршрута.

В `incl[i]` будем записывать `true`, если точка с номером i включена в маршрут. Делается это для того, чтобы не включать в маршрут уже пройденную точку (не ходить по кругу).

Так как мы используем рекурсивную процедуру, то надо обратить особое внимание на условие завершения рекурсивного процесса. Процедура должна прекратить вызывать сама себя, если текущая точка совпала с заданной конечной точкой.

На рис. 12.10 приведена блок-схема алгоритма процедуры выбора очередной точки формируемого маршрута, а на рис. 12.11 — диалоговое окно программы.

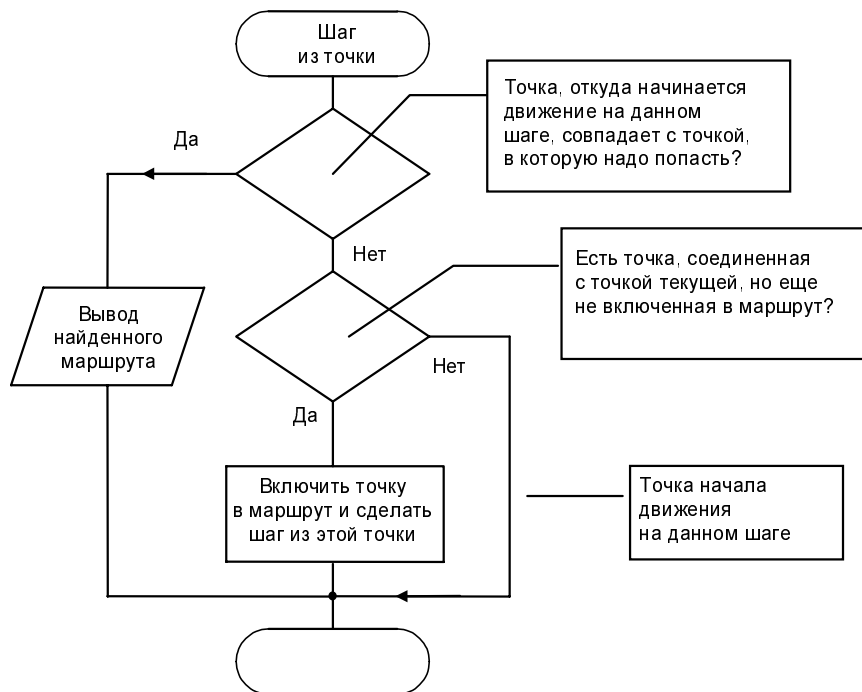


Рис. 12.10. Блок-схема процедуры выбора точки маршрута

Для ввода массива, представляющего описание карты, используется компонент `StringGrid1` (значения его свойств приведены в табл. 12.1), для вывода

результата (найденного маршрута) — поле метки Label1. Начальная и конечная точки маршрута задаются вводом значений в поля редактирования Edit1 и Edit2. Процедура поиска запускается нажатием кнопки **Поиск** (Button1). Поля меток Label2, Label3 и Label4 используются для вывода поясняющего текста.

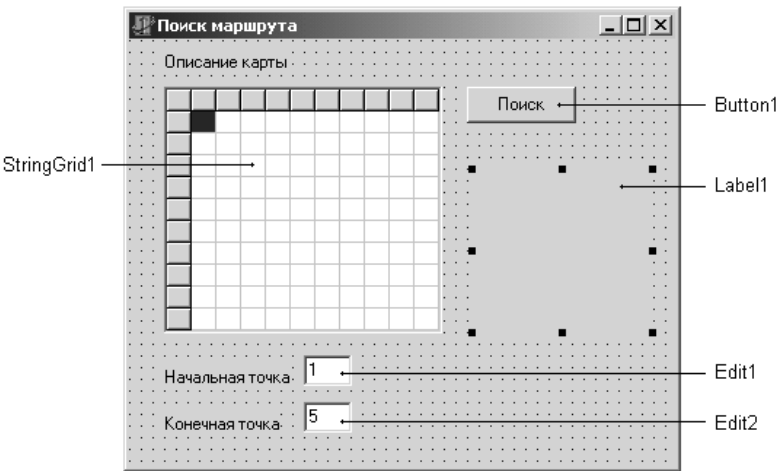


Рис. 12.11. Окно программы **Поиск маршрута**

Таблица 12.1. Значения свойств компонента *StringGrid1*

Свойство	Значение	Свойство	Значение
Name	StringGrid1	FixedRows	1
ColCount	11	Options.goEditing	TRUE
RowCount	11	DefaultColWidth	16
FixedCols	1	DefaultRowHeight	14

Текст программы приведен в листинге 12.5.

Листинг 12.5. Поиск маршрута

```
unit road_;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
```

```
  Dialogs, StdCtrls, Grids;
```

type

```

TForm1 = class(TForm)
  StringGrid1: TStringGrid;
  Edit1: TEdit;
  Edit2: TEdit;
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Button1: TButton;
  Label4: TLabel;
  procedure FormActivate(Sender: TObject);
  procedure Button1Click(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;

```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

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

var

```
i:integer;
```

begin

```
  // нумерация строк
```

```
  for i:=1 to 10 do
```

```
    StringGrid1.Cells[0,i]:=IntToStr(i);
```

```
  // нумерация колонок
```

```
  for i:=1 to 10 do
```

```
    StringGrid1.Cells[i,0]:=IntToStr(i);
```

```
  // описание предопределенной карты
```

```
  StringGrid1.Cells[1,2]:='1';
```

```
  StringGrid1.Cells[2,1]:='1';
```

```
  StringGrid1.Cells[1,3]:='1';
```

```
  StringGrid1.Cells[3,1]:='1';
```

```
  StringGrid1.Cells[1,4]:='1';
```

```
  StringGrid1.Cells[4,1]:='1';
```

```
  StringGrid1.Cells[3,7]:='1';
```

```
  StringGrid1.Cells[7,3]:='1';
```

```
  StringGrid1.Cells[4,6]:='1';
```

```
  StringGrid1.Cells[6,4]:='1';
```

```

StringGrid1.Cells[5,6]:='1';
StringGrid1.Cells[6,5]:='1';
StringGrid1.Cells[5,7]:='1';
StringGrid1.Cells[7,5]:='1';
StringGrid1.Cells[6,7]:='1';
StringGrid1.Cells[7,6]:='1';

end;

procedure TForm1.Button1Click(Sender: TObject);
const
    N=10; // кол-во вершин графа
var
    map:array[1..N,1..N]of integer; // Карта.map[i,j] не 0,
                                     // если точки i и j соединены
    road:array[1..N]of integer;    // Дорога – номера точек карты
    incl:array[1..N]of boolean;   // incl[i] равен TRUE, если точка
                                     // с номером i включена в road
    start,finish:integer;         // Начальная и конечная точки
    found:boolean;
    i,j:integer;

procedure step(s,f,p:integer);
var
    c:integer; // Номер точки, в которую делаем очередной шаг
    i:integer;
begin
    if s=f then
        begin
            // Точки s и f совпали !
            found:=TRUE;
            Label1.caption:=Label1.caption+#13+'Путь: ';
            for i:=1 to p-1 do
                Label1.caption:=Label1.caption+' '+IntToStr(road[i]);
        end
    else begin
        // выберем очередную точку
        for c:=1 to N do
            begin // проверяем все вершины
                if (map[s,c]<> 0) and (NOT incl[c])
                    // точка соединена с текущей и не включена в маршрут
                    then begin
                        road[p]:=c;    // добавим вершину в путь
                        incl[c]:=TRUE; // пометим вершину как включенную
                        step(c,f,p+1);
                    end
            end
    end

```

```

        incl[c]:=FALSE;
        road[p]:=0;

    end;

end;

end;

end; // конец процедуры step

begin
    Label1.caption:='';
    // инициализация массивов
    for i:=1 to N do road[i]:=0;
    for i:=1 to N do incl[i]:=FALSE;

    // ввод описания карты из SrtngGrid.Cells
    for i:=1 to N do
        for j:=1 to N do
            if StringGrid1.Cells[i,j] <> ''
                then map[i,j]:=StrToInt(StringGrid1.Cells[i,j])
                else map[i,j]:=0;

start:=StrToInt(Edit1.text);
finish:=StrToInt(Edit2.text);

road[1]:=start; // внесем точку в маршрут
incl[start]:=TRUE; // пометим ее как включенную

step(start,finish,2); // ищем вторую точку маршрута

// проверим, найден ли хотя бы один путь
if not found
    then Label1.caption:='Указанные точки не соединены!';

end;

end.
```

При запуске программы в момент активизации формы приложения происходит событие `OnActivate`, процедура обработки которого заполняет массив `StringGrid1.Cells` значениями, представляющими описание карты. Эта же процедура нумерует строки и столбцы таблицы, заполняя зафиксированные ячейки первого столбца и первой строки `StringGrid1`.

Поиск маршрута инициирует процедура `TForm1.Button1Click`, которая запускается нажатием кнопки **Поиск**. Для поиска точки, соединенной с исходной, процедура `TForm1.Button1Click` вызывает процедуру `Step`, которая, после выбора первой точки, соединенной с начальной, и включения ее в маршрут, вызывает сама себя. При этом в качестве начальной точки задается уже не исходная, а текущая, только что включенная в маршрут точка.

Поиск кратчайшего пути

Обычно задача поиска пути на графе формулируется следующим образом: найти наилучший маршрут. Под наилучшим маршрутом, как правило, понимают кратчайший. Найти кратчайший маршрут можно выбором из всех найденных. Однако совсем не обязательно искать все маршруты.

Можно поступить иначе: во время выбора очередной точки проверить, не превысит ли длина формируемого маршрута длину уже найденного пути, если эта точка будет включена в маршрут; если превысит, то эту точку следует пропустить и выбрать другую.

Таким образом, после того как будет найден первый маршрут, программа будет вести поиск только по тем ветвям графа, которые могут улучшить найденное решение, отсекая пути, делающие формируемый маршрут длиннее уже найденного.

В листинге 12.6 приведена процедура, которая использует процедуру `step`, выполняющую выбор очередной точки маршрута таким образом, что обеспечивается поиск пути минимальной длины.

Листинг 12.6. Поиск кратчайшего пути

```
procedure TForm1.Button1Click(Sender: TObject);
const
    N=10; { кол-во вершин графа }
var
    map:array[1..N,1..N]of integer; // Карта.map[i,j] не 0, если
                                   // точки i и j соединены
    road:array[1..N]of integer;    // Дорога — номера точек карты
    incl:array[1..N]of boolean;    // incl[i] равен TRUE, если точка
                                   // с номером i включена в road
    start,finish:integer;          // Начальная и конечная точки
    found:boolean;
    len:integer;                   // длина найденного (минимального)
                                   // маршрута
    c_len:integer;                 // длина текущего (формируемого)
                                   // маршрута
    i,j:integer;

// выбор очередной точки
procedure step(s,f,p:integer);
var
    c:integer; { Номер точки, в которую делаем очередной шаг }
    i:integer;
```



```

begin
  if s=f then
    begin
      len:=c_len; { сохраним длину найденного маршрута }
      { вывод найденного маршрута }
      for i:=1 to p-1 do
        Label1.caption:=Label1.caption+' '+IntToStr(road[i]);
      Label1.caption:=Label1.caption
        +', длина: '+IntToStr(len)+'#13;
    end
  else
    { выбираем очередную точку }
    for c:=1 to N do { проверяем все вершины }
      if (map[s,c]<> 0) and (NOT incl[c])
        and ((len=0) or (c_len+map[s,c] < len))
      then begin
        // точка соединена с текущей, но не включена
        // в маршрут
        road[p]:=c; { добавим вершину в путь }
        incl[c]:=TRUE; { пометим вершину как включенную }
        c_len:=c_len+map[s,c];
        step(c,f,p+1);
        incl[c]:=FALSE;
        road[p]:=0;
        c_len:=c_len-map[s,c];
      end;
    end; { конец процедуры step }

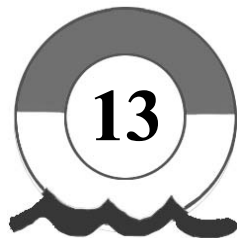
begin
  Label1.caption:='';
  { инициализация массивов }
  for i:=1 to N do road[i]:=0;
  for i:=1 to N do incl[i]:=FALSE;
  { ввод описания карты из SrtingGrid.Cells}
  for i:=1 to N do
    for j:=1 to N do
      if StringGrid1.Cells[i,j] <> ''
        then map[i,j]:=StrToInt(StringGrid1.Cells[i,j])
        else map[i,j]:=0;

  len:=0; // длина найденного (минимального) маршрута
  c_len:=0; // длина текущего (формируемого) маршрута
  start:=StrToInt(Edit1.text);
  finish:=StrToInt(Edit2.text);
  road[1]:=start; { внесем точку в маршрут }

```

```
incl[start]:=TRUE;{ пометим ее как включенную }  
step(start,finish,2);{ищем вторую точку маршрута }  
// проверим, найден ли хотя бы один путь  
if not found  
    then Labell.caption:='Указанные точки не соединены!';  
end;
```

Диалоговое окно программы поиска кратчайшего пути и процедура обработки события `OnActivate` ничем не отличаются от диалогового окна (рис. 12.11) и соответствующей процедуры `OnActivate` (листинг 12.5) программы поиска всех возможных маршрутов, рассмотренной в предыдущем разделе.



Отладка программы

Под отладкой программы понимается процесс поиска и устранения ошибок.

Классификация ошибок

Ошибки, которые могут быть в программе, принято делить на три группы:

- ☐ Синтаксические
- ☐ Ошибки времени выполнения
- ☐ Алгоритмические

Синтаксические ошибки, их также называют *ошибками времени компиляции* (Compile-time error), наиболее легко устранимы. Их обнаруживает компилятор, а программисту остается только внести изменения в текст программы и выполнить повторную компиляцию.

Ошибки времени выполнения, в Delphi они называются *исключениями* (exception), тоже, как правило, легко устранимы. Они обычно проявляются уже при первых запусках программы и во время тестирования.

При возникновении ошибки в программе, запущенной из Delphi, среда разработки прерывает работу программы, о чем свидетельствует заключенное в скобки слово **Stopped** в заголовке главного окна Delphi, и на экране появляется диалоговое окно **Error** (Ошибка), которое содержит сообщение об ошибке и информацию о типе (классе) ошибки. На рис. 13.1 приведен пример сообщения об ошибке, возникающей при попытке открыть несуществующий файл.

После возникновения ошибки программист может либо прервать выполнение программы, для этого надо из меню **Run** выбрать команду **Program Reset**, либо продолжить ее выполнение, например, по шагам (для этого из

меню **Run** надо выбрать команду **Step**), наблюдая результат выполнения каждой инструкции.

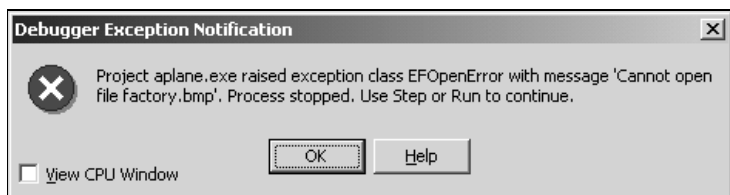


Рис. 13.1. Сообщение об ошибке при запуске программы из Delphi

Если программа запущена из Windows, то при возникновении ошибки на экране также появляется диалоговое окно **Error**, однако оно содержит только сообщение об ошибке (рис. 13.2). После нажатия кнопки **ОК** программа, в которой проявилась ошибка, продолжает (если сможет) работу.

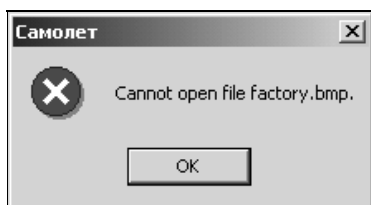


Рис. 13.2. Сообщение об ошибке при запуске программы из Windows

С *алгоритмическими ошибками* дело обстоит иначе. Компиляция программы, в которой есть алгоритмическая ошибка, завершается успешно. При пробных запусках программа ведет себя нормально, однако при анализе результата выясняется, что он неверный. Для того чтобы устранить алгоритмическую ошибку, приходится анализировать алгоритм, вручную "прокручивать" его выполнение.

Предотвращение и обработка ошибок

Как было сказано выше, в программе, во время ее работы, могут возникать ошибки, причиной которых, как правило, являются действия пользователя. Например, пользователь может ввести неверные данные или, что бывает довольно часто, удалить нужный программе файл.

Нарушение в работе программы называется *исключением*. Обработку исключений (ошибок) берет на себя автоматически добавляемый в выполняемую программу код, который обеспечивает, в том числе, вывод информационного сообщения. Вместе с тем Delphi дает возможность программе самой выполнить обработку исключения.

Инструкция обработки исключения в общем виде выглядит так:

try

// здесь инструкции, выполнение которых может вызвать исключение

except *// начало секции обработки исключений*

on *ТипИсключения1 do Обработка1;*

on *ТипИсключения2 do Обработка2;*

on *ТипИсключенияJ do ОбработкаJ;*

else

// здесь инструкции обработки остальных исключений

end;

где:

try — ключевое слово, обозначающее, что далее следуют инструкции, при выполнении которых возможно возникновение исключений, и что обработку этих исключений берет на себя программа;

except — ключевое слово, обозначающее начало секции обработки исключений. Инструкции этой секции будут выполнены, если в программе возникнет ошибка;

on — ключевое слово, за которым следует тип исключения, обработку которого выполняет инструкция, следующая за **do**;

else — ключевое слово, за которым следуют инструкции, обеспечивающие обработку исключений, тип которых не указан в секции **except**.

Как было сказано выше, каждое исключение характеризуется типом. В табл. 13.1 перечислены наиболее часто возникающие исключения и указаны причины, которые могут привести к их возникновению.

Таблица 13.1. Типичные исключения

Исключение	Возникает
EZeroDivide	При выполнении операции деления, если делитель равен нулю
EConvertError	При выполнении преобразования, если преобразуемая величина не может быть приведена к требуемому виду. Наиболее часто возникает при преобразовании строки символов в число
EFileError	При обращении к файлу. Наиболее частой причиной является отсутствие требуемого файла или, в случае использования сменного диска, отсутствие диска в накопителе

Следующая программа, вид ее диалогового окна приведен на рис. 13.3, а текст — в листинге 13.1, демонстрирует обработку исключений при помощи инструкции **try**.

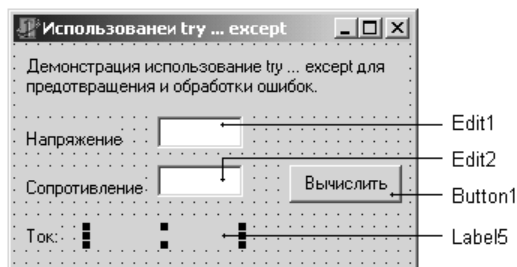


Рис. 13.3. Диалоговое окно программы

Листинг 13.1. Обработка исключения

```

unit UsTry_;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Edit1: TEdit;      // напряжение
    Edit2: TEdit;      // сопротивление
    Label5: TLabel;    // результат расчета — ток
    Button1: TButton;  // кнопка Вычислить
  procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var

```

```
u: real; // напряжение
r: real; // сопротивление
i: real; // ток

begin
  Label5.Caption := ' ';
  try
    // инструкции, которые могут вызвать исключение (ошибку)
    u := StrToFloat(Edit1.Text);
    r := StrToFloat(Edit2.Text);
    i := u/r;
  except // секция обработки исключений
    on EZeroDivide do // деление на ноль
      begin
        ShowMessage('Сопротивление не может быть равно нулю!');
        exit;
      end;
    on EConvertError do // ошибка преобразования строки в число
      begin
        ShowMessage('Напряжение и сопротивление должны быть ' +
          'заданы числом.' + #13+
          'При записи дробного числа используйте запятую.');
```

В приведенной программе исключения могут возникнуть при вычислении величины тока. Если пользователь задаст, что сопротивление равно нулю, то при выполнении инструкции `i:=u/r` возникнет исключение `EZeroDivide`. Если неверно будет введено числовое значение, например, для разделения целой и дробной частей числа вместо запятой будет использована точка, то возникнет исключение типа `EConvertError`. Оба исключения обрабатываются одинаково: выводится сообщение, после чего процедура обработки события `OnClick` завершает свою работу.

Отладчик

Интегрированная среда разработки Delphi предоставляет программисту отладчик — мощное средство поиска и устранения ошибок в программе. Отладчик позволяет выполнять *трассировку* программы, наблюдать значения переменных, контролировать выводимые программой данные.

Трассировка программы

Во время работы программы инструкции программы выполняются одна за другой со скоростью работы процессора компьютера. При этом программист не может определить, какая инструкция выполняется в данный момент, и, следовательно, определить, соответствует ли реальный порядок выполнения инструкций разработанному им алгоритму.

В случае неправильной работы программы необходимо видеть реальный порядок выполнения инструкций. Это можно сделать, выполнив трассировку программы. *Трассировка* — это процесс выполнения программы по шагам (*step-by-step*), инструкция за инструкцией. Во время трассировки программист дает команду: выполнить очередную инструкцию программы.

Delphi обеспечивает два режима трассировки: без захода в процедуру (*Step over*) и с заходом в процедуру (*Trace into*). Режим трассировки без захода в процедуру выполняет трассировку только главной процедуры, при этом трассировка подпрограмм не осуществляется, вся подпрограмма исполняется за один шаг. В режиме трассировки с заходом в процедуру трассируется вся программа, т. е. по шагам выполняется не только главная программа, но и все подпрограммы.

Для того чтобы начать трассировку, необходимо из меню **Run** выбрать команду **Step over** или **Trace into**. В результате в окне редактора кода будет выделена инструкция, выполняемая первой. Для того чтобы выполнить выделенную инструкцию, следует из меню **Run** выбрать команду **Step over** или **Trace into**. После выполнения инструкции будет выделена следующая. Таким образом, выбирая нужную команду из меню **Run**, можно выполнить трассировку программы.

Активизировать и выполнить трассировку можно при помощи функциональной клавиатуры. Команде **Step over** соответствует клавиша <F8>, а команде **Trace into** — клавиша <F7>.

В любой момент времени можно завершить трассировку и продолжить выполнение программы в реальном темпе. Для этого надо из меню **Run** выбрать команду **Run**.

При необходимости выполнить трассировку части программы следует установить курсор на инструкцию программы, с которой надо начать трассировку, и из меню **Run** выбрать команду **Run to cursor** или нажать клавишу <F4>. Затем, нажимая клавишу <F7> или клавишу <F8>, выполнить трассировку нужного фрагмента программы.

Во время трассировки можно наблюдать не только порядок выполнения инструкций программы, но и значения переменных. О том, как это сделать, рассказывается в одном из следующих разделов.

Точки останова программы

При отладке широко используется метод, который называют *методом точек останова*. Суть метода заключается в том, что программист помечает некоторые инструкции программы (ставит точки останова), при достижении которых программа приостанавливает свою работу и программист может начать трассировку или проконтролировать значения переменных.

Добавление точки останова

Для того чтобы поставить в программу *точку останова* (breakpoint), нужно из меню **Run** выбрать команду **Add Breakpoint** (Добавить точку останова), затем из меню следующего уровня — команду **Source Breakpoint**.

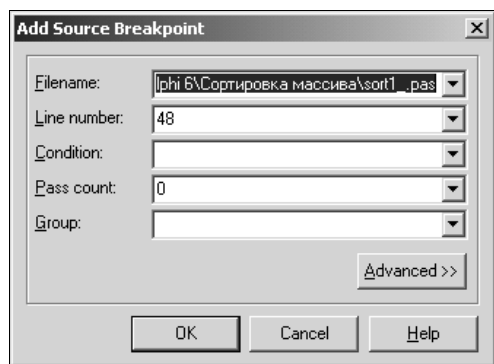


Рис. 13.4. Диалоговое окно **Add Source Breakpoint**

В результате открывается диалоговое окно **Add Source Breakpoint** (рис. 13.4), в котором выводится информация о добавляемой точке останова.

- ☐ Поле **Filename** содержит имя файла программы, куда добавляется точка останова.
- ☐ Поле **Line number** — номер строки программы, в которую добавляется точка останова.
- ☐ О назначении полей **Condition** (Условие) и **Pass count** (Число пропусков) будет сказано позже.

После нажатия кнопки **ОК** точка останова добавляется в программу и строка, в которой находится точка останова, помечается красной точкой и выделяется цветом (рис. 13.5).

Примечание

Точку останова можно добавить, щелкнув мышью на синей точке, помечающей ту инструкцию программы, перед которой надо поместить точку останова (если в программе нет ошибок, то компилятор помечает выполняемые инструкции программы синими точками).

Для точки останова можно задать условие, при выполнении которого программа приостановит свою работу в данной точке (например, если значение переменной равно определенной величине). Условие (логическое выражение) вводится в поле **Condition** диалогового окна **Add Source Breakpoint**.

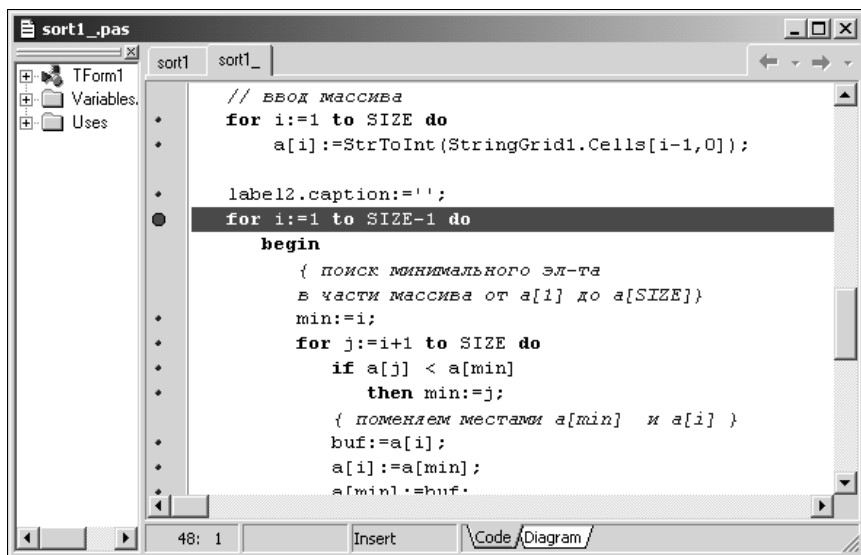


Рис. 13.5. Окно редактора кода после добавления точки останова

Если для точки останова задано условие, то программа приостанавливает свою работу только в том случае, если выражение, находящееся в поле **Condition**, истинно (его значение равно **TRUE**).

Кроме условия для точки останова можно задать количество пропусков данной точки. Если во время добавления в программу точки останова в поле **Pass count** (Число пропусков) диалогового окна **Add Source Breakpoint** записать отличное от нуля число, то программа приостановит свою работу в этой точке только после того, как инструкция, находящаяся в строке, помеченной точкой останова, будет выполнена указанное число раз.

Изменение характеристик точки останова

Программист может изменить характеристики точки останова. Для этого надо из меню **View** выбрать команду **Debug Windows**, затем из меню следующего уровня — команду **Breakpoints**. В открывшемся диалоговом окне **Breakpoint List** (рис. 13.6) нужно щелкнуть правой кнопкой мыши в строке, содержащей информацию о нужной точке останова, и в появившемся контекстном меню выбрать команду **Properties**. В результате открывается диалоговое окно **Source Breakpoint Properties**, в котором можно изменить харак-

теристики точки останова, например, модифицировать условие (содержимое поля **Condition**) остановки программы в данной точке. Используя это же контекстное меню, можно быстро перейти к инструкции, в которой находится точка останова; для этого надо выбрать команду **Edit Source**.

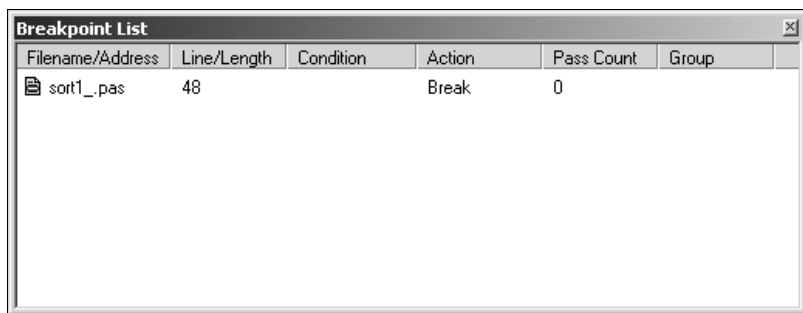


Рис. 13.6. Окно **Breakpoint List**

Удаление точки останова

Для того чтобы удалить точку останова, нужно в диалоговом окне **Breakpoint List** щелкнуть правой кнопкой мыши в строке, содержащей информацию о точке, которую надо удалить, и в появившемся контекстном меню выбрать команду **Delete**.

Можно также в окне редактора кода щелкнуть мышью на красной точке, помечающей строку, в которой находится точка останова.

Наблюдение значений переменных

Во время отладки, в частности, при выполнении программы по шагам, довольно часто бывает полезно знать, чему равно значение той или иной переменной. Отладчик позволяет наблюдать значения переменных программы.

Для того чтобы во время выполнения программы по шагам иметь возможность контролировать значение переменной, нужно добавить имя этой переменной в список наблюдаемых элементов (**Watch List**). Для этого надо из меню **Run** выбрать команду **Add Watch** (Добавить наблюдаемый элемент) и в поле **Expression** появившегося диалогового окна **Watch Properties** (рис. 13.7) ввести имя переменной.

В результате в список **Watch List**, содержимое которого отражается в диалоговом окне **Watch List** (рис. 13.8), будет добавлен новый элемент. Так как переменные программы существуют (и, следовательно, доступны) только во время выполнения программы, то после имени переменной выводится сообщение: `process not accessible` (процесс недоступен).

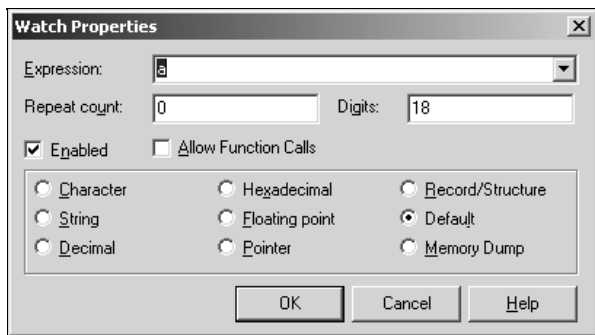


Рис. 13.7. Добавление имени переменной в список **Watch List**

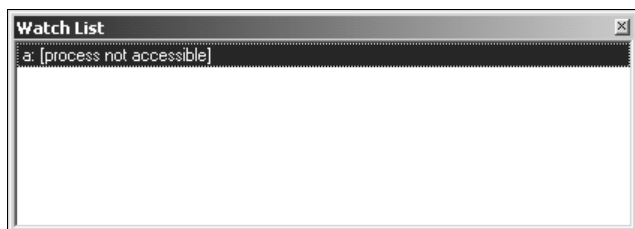


Рис. 13.8. Результат добавления имени переменной в список **Watch List**

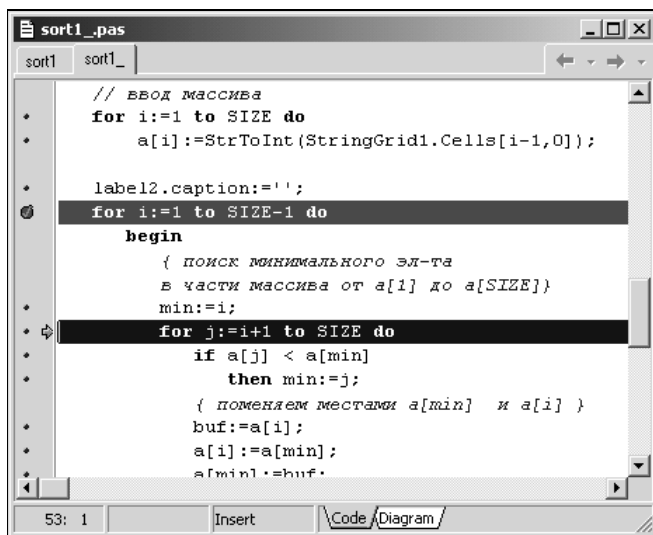


Рис. 13.9. Контроль значений переменных во время пошагового выполнения программы

В качестве примера на рис. 13.9 приведен вид окна редактора кода и окна **Watch List** во время пошагового выполнения программы сортировки массива.

В окне редактора кода стрелкой помечена инструкция, которая будет выполнена на следующем шаге выполнения программы (при нажатии клавиши

<F8> или при выборе команды **Step over** из меню **Run**), в диалоговом окне **Watch List** выведены значения переменных.

Существует еще один способ, позволяющий проверить значение переменной, не добавляя ее имя в список **Watch List**. Заключается он в следующем. После того, как программа достигнет точки останова, в результате чего откроется окно редактора кода, нужно установить курсор мыши на имя переменной, значение которой надо проверить (рис. 13.10).

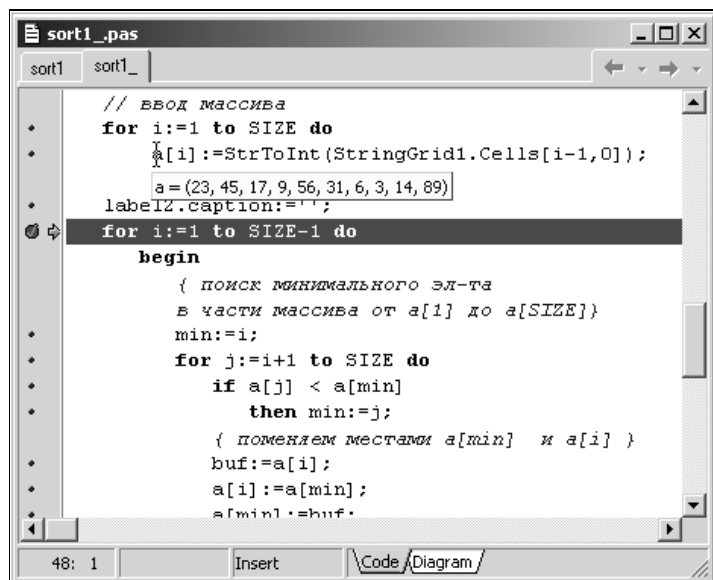
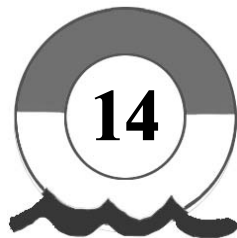


Рис. 13.10. Контроль значения переменной без добавления имени в список **Watch List**



Справочная система

Желательно, чтобы каждая программа имела справочную систему, при помощи которой пользователь мог получить исчерпывающую информацию о программе, о том, как с ней работать.

Справочная система представляет собой набор файлов определенной структуры, используя которые программа Winhelp, являющаяся составной частью Windows, выводит справочную информацию, которая находится в hlp-файле.

Создать hlp-файл можно, например, при помощи поставляемой вместе с Delphi программы Microsoft Help Workshop, исходными данными для которой является текст справки, представленный в виде rtf-файла.

Процесс создания справочной системы (hlp-файла) можно представить как последовательность следующих двух шагов:

- ☐ Подготовка справочной информации (создание файла документа справочной информации).
- ☐ Преобразование файла справочной информации в файл справочной системы.

Файл документа справочной информации

Файл документа справочной системы представляет собой rtf-файл определенной структуры. Создать rtf-файл справочной информации можно при помощи Microsoft Word. Сначала нужно набрать текст разделов справки, оформив заголовки разделов одним из стилей **Заголовков**, например **Заголовок 1**. При этом текст каждого раздела должен находиться на отдельной странице документа (заканчиваться символом "разрыв страницы").

После того как текст разделов будет набран, нужно, используя сноски, пометить заголовки разделов справочной информации (сноски используются

компилятором справочной системы в процессе преобразования ttf-файла в hlp-файл, файл справки).

Некоторые, наиболее часто используемые, сноски перечислены в табл. 14.1.

Таблица 14.1. Часто используемые сноски

Сноска	Назначение
#	Задаёт идентификатор раздела справки, который может использоваться в других разделах для перехода к помеченному этой сноской разделу
\$	Задаёт имя раздела, которое будет применяться для идентификации раздела справки в списке поиска и в списке просмотренных тем во время использования справочной системы
K	Задаёт список ключевых слов, при выборе которых из списка диалога поиска осуществляется переход к разделу справки, заголовок которой помечен этой сноской

Для того чтобы пометить заголовок раздела сноской, нужно установить курсор перед первой буквой заголовка раздела и из меню **Вставка** выбрать команду **Сноска**. В открывшемся диалоговом окне **Сноски** (рис. 14.1) в группе **Вставить сноску** нужно установить переключатель в положение **обычную**, а в группе **Нумерация** — в положение **другая**. В поле ввода номера сноски следует ввести символ # и нажать кнопку **ОК**.

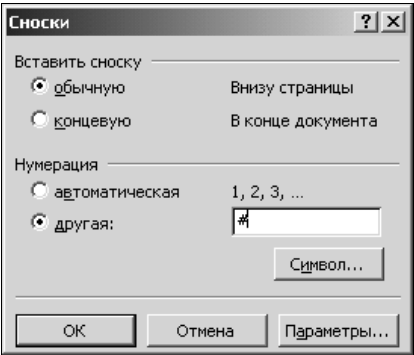


Рис. 14.1. Диалоговое окно Сноски

В результате в документ будет вставлена сноска # и в нижней части окна документа появится окно ввода текста сноски, в котором рядом со значком сноски следует ввести идентификатор помечаемого раздела справки (рис. 14.2).

В качестве идентификатора можно использовать аббревиатуру заголовка раздела справки или сквозной номер раздела, поставив перед ним, например, буквы TI (topic identifier).

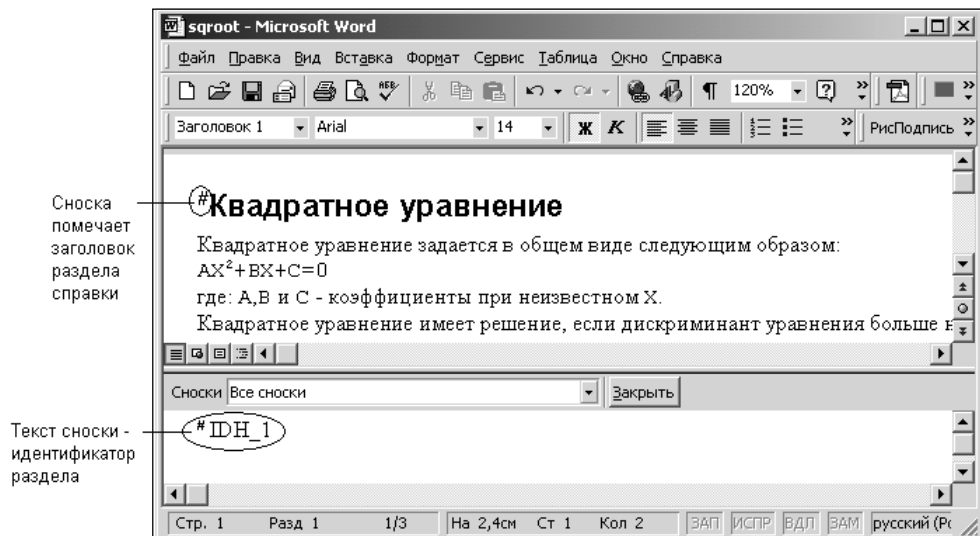


Рис. 14.2. Вставка в документ сноски, помечающей заголовок раздела справки

Примечание

Однако лучше, чтобы идентификатор раздела справки начинался с префикса IDH_. В этом случае во время компиляции rtf-файла будет проверена корректность ссылок на разделы справки. Компилятор выведет список идентификаторов, которые перечислены в разделе [MAP] файла проекта (см. ниже), но которых нет в rtf-файле.

Как правило, разделы справки содержат ссылки на другие разделы. В окне справочной системы понятия (слова), выбор которых вызывает переход к другому разделу справки, выделяются отличным от основного текста справки цветом и подчеркиваются.

Во время подготовки текста справочной информации, слово-ссылку, при выборе которой происходит переход к другому разделу справки, следует подчеркнуть двойной линией. Сразу за ссылкой, без пробела, поместить идентификатор раздела справки, к которому должен быть выполнен переход. Вставленный идентификатор необходимо оформить как скрытый текст.

На рис. 14.3 приведен вид окна редактора текста во время подготовки файла справочной информации для программы решения квадратного уравнения. Слово "дискриминант" помечено как ссылка на другой раздел справки (здесь предполагается, что раздел справки, в котором находятся сведения о дискриминанте, помечен сноской #, имеющей идентификатор IDH_2).

Помимо ссылки, обеспечивающей переход к другому разделу справки, в документ можно вставить ссылку на комментарий — текст, появляющийся

в всплывающем окне. Во время работы справочной системы ссылки на комментарии выделяются цветом и подчеркиваются пунктирной линией. При подготовке документа справочной системы комментарии, как и разделы справки, располагают на отдельной странице, однако текст комментария не должен иметь заголовка. Сноска # должна быть поставлена перед текстом комментария. Ссылка на комментарий оформляется следующим образом: сначала надо подчеркнуть одинарной линией слово, при выборе которого появляется комментарий, затем сразу после этого слова вставить идентификатор комментария, оформив его как скрытый текст.

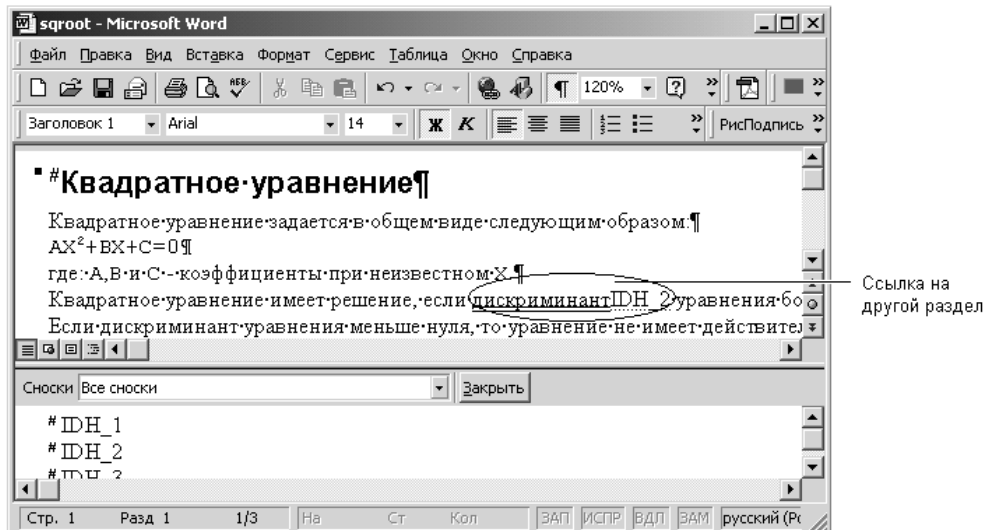


Рис. 14.3. Оформление ссылки на другой раздел справки

Создание справочной системы

После того как создан файл справочной информации системы (rtf-файл), можно приступить к созданию справочной системы (hlp-файла). Для этого удобно воспользоваться программой Microsoft Help Workshop, которая поставляется вместе с Delphi и находится в файле Hcw.exe.

Запустить Microsoft Help Workshop можно из Windows или из Delphi, выбрав из меню **Tools** команду **Help Workshop**.

Примечание

Если в меню **Tools** команды **Help Workshop** нет, то надо из этого же меню выбрать команду **Configure Tools** и в открывшемся диалоговом окне **Tool Options** (рис. 14.4) щелкнуть на кнопке **Add**. В результате этого откроется диалоговое

окно **Tool Properties** (рис. 14.5), в поле **Title** которого надо ввести название программы — Help Workshop, а в поле **Program** — полное (т. е. с указанием пути) имя исполняемого файла программы Microsoft Help Workshop. Для ввода имени файла можно воспользоваться кнопкой **Browse**.

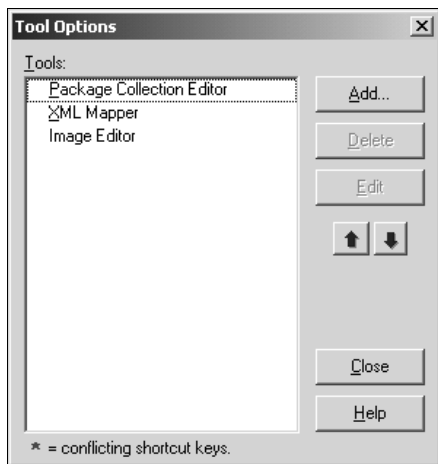


Рис. 14.4. Диалоговое окно **Tool Options**

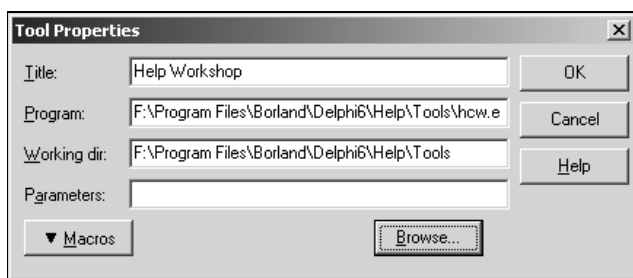


Рис. 14.5. Диалоговое окно **Tool Properties**

После запуска программы Help Workshop на экране появляется главное окно программы (рис. 14.6).

Для того чтобы приступить к созданию справочной системы, нужно из меню **File** выбрать команду **New**, затем в открывшемся диалоговом окне **Project File Name** (рис. 14.7) указать тип создаваемого файла — **Help Project**. Далее следует выбрать папку, где находится программа, для которой создается справочная система, и где уже должен находиться файл документа справочной системы (rtf-файл). Затем в поле **Имя файла** нужно ввести имя файла проекта справочной системы. После нажатия кнопки **Сохранить** открывается окно проекта справочной системы (рис. 14.8).

Используя окно проекта справочной системы, можно добавить необходимые компоненты в проект, задать характеристики окна справочной системы, вы-

полнить компиляцию проекта и пробный запуск созданной справочной системы.

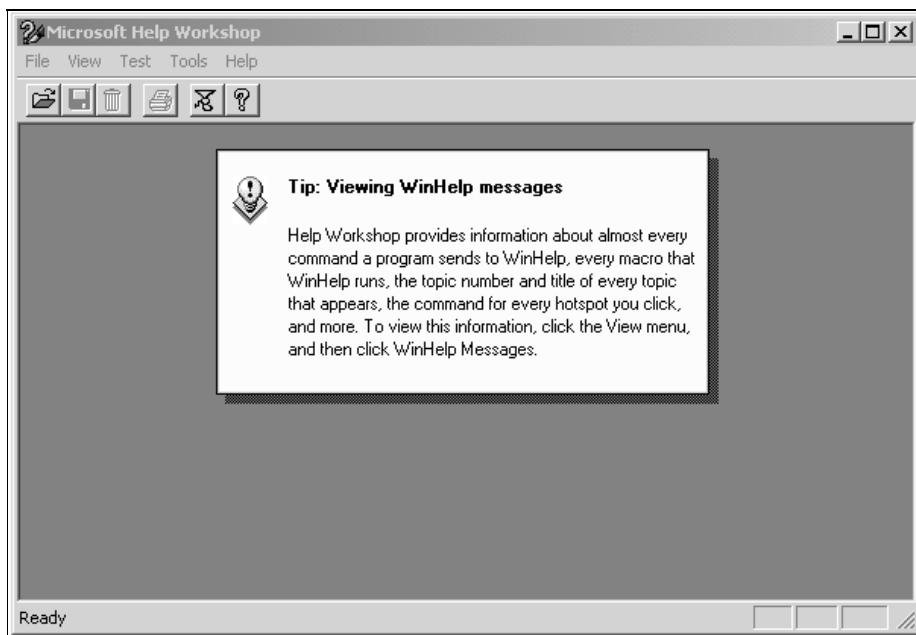


Рис. 14.6. Главное окно программы Help Workshop

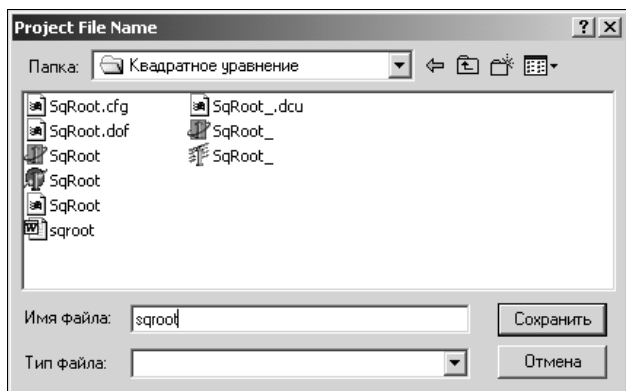


Рис. 14.7. Начало работы над новым проектом

Для того чтобы добавить в проект файл справочной информации, нужно нажать на кнопку **Files** и в открывшемся диалоговом окне **Topic Files** — кнопку **Add** (рис. 14.9). В итоге откроется стандартное окно **Открытие файла**, позволяющее выбрать нужный rtf-файл. В результате этих действий в окне проекта появится раздел **FILES**, в котором будет указано имя файла

справочной информации. Если справочная информация распределена по нескольким файлам, то операцию добавления файла нужно повторить.

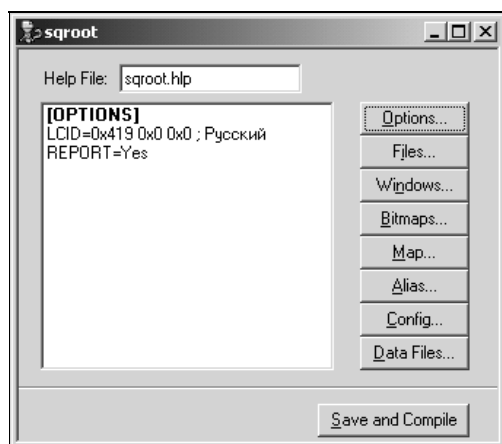


Рис. 14.8. Окно проекта справочной системы

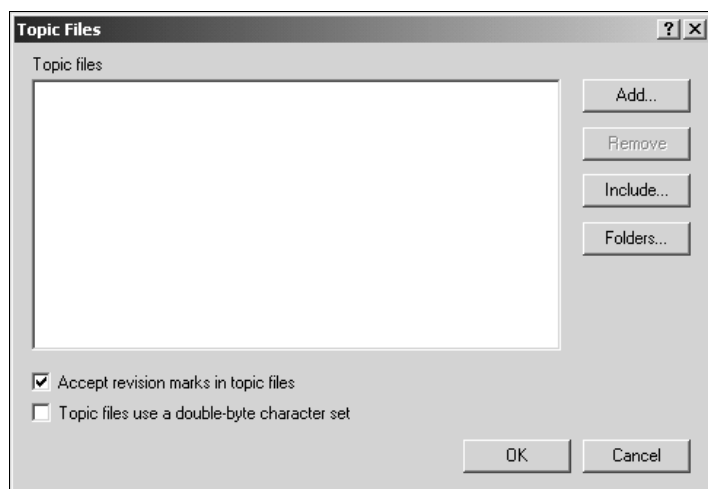


Рис. 14.9. Диалоговое окно **Topic Files**

Для того чтобы задать характеристики главного окна справочной системы, надо в окне проекта нажать кнопку **Windows**. Появится окно **Create a window**. В поле **Create a window named** (рис. 14.10) введите слово **main**.

В результате нажатия на кнопку **OK** появляется окно **Window Properties**, в поле **Title bar text** вкладки **General** которого нужно ввести заголовок главного окна создаваемой справочной системы (рис. 14.11).

Используя вкладку **Position** диалогового окна **Window Properties**, можно задать положение и размер окна справочной системы (рис. 14.12). На вкладке

Position находится кнопка **Auto-Sizer**, при нажатии которой открывается окно **Help Window Auto-Sizer** (рис. 14.13), размер и положение которого определяется содержимым полей вкладки **Position**. При помощи мыши можно менять размер и положение этого окна. После нажатия кнопки **ОК** координаты и размер окна **Help Window Auto-Sizer** будут записаны в поля вкладки **Position**.

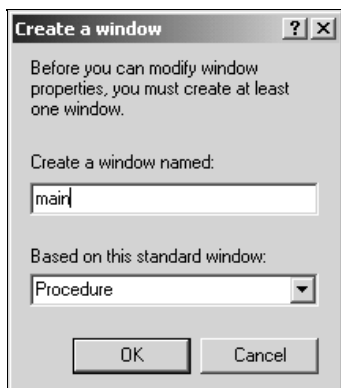


Рис. 14.10. Диалоговое окно **Create a window**



Рис. 14.11. Вкладка **General** диалогового окна **Window Properties**

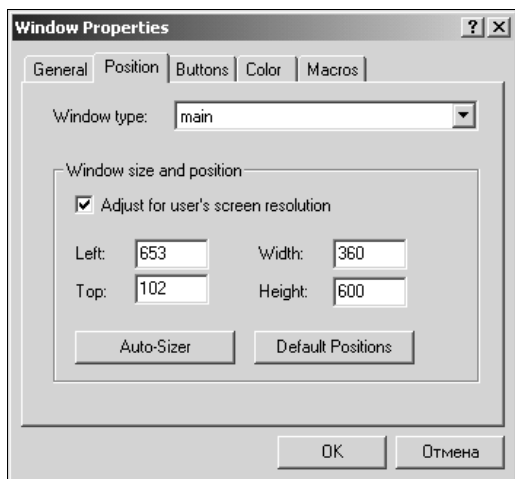


Рис. 14.12. Вкладка **Position** диалогового окна **Window Properties**

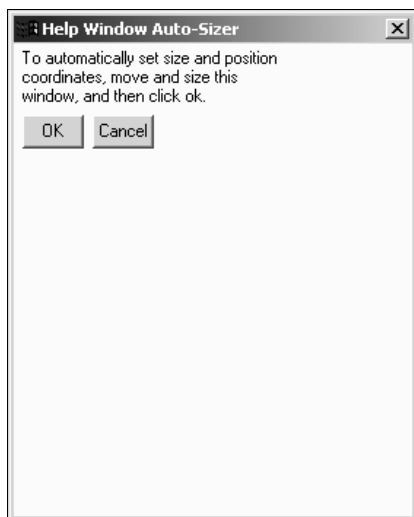


Рис. 14.13. Окно **Help Window Auto-Sizer**

Используя вкладку **Color** (рис. 14.14), можно задать цвет фона области заголовка раздела справки (**Nonscrolling area color**) и области текста справки (**Topic area color**). Для этого надо нажать соответствующую кнопку **Change** и в стандартном окне **Цвет** выбрать нужный цвет.

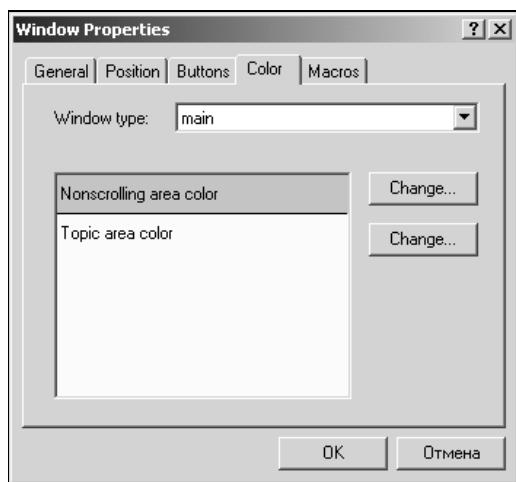


Рис. 14.14. Вкладка **Color** диалогового окна **Window Properties**

Чтобы программа, использующая справочную систему, могла получить доступ к конкретному разделу справочной информации, нужно определить числовые значения для идентификаторов разделов. Чтобы это сделать, надо в окне проекта справочной системы нажать кнопку **Map**, в результате чего откроется диалоговое окно **Map** (рис. 14.15). В этом окне нужно нажать кнопку **Add** и в поле **Topic ID**, открывшегося диалогового окна **Add Map Entry** (рис. 14.16), ввести идентификатор раздела справки, а в поле **Mapped numeric value** — соответствующее идентификатору числовое значение. В поле **Comment** можно ввести комментарий — название раздела справочной системы, которому соответствует идентификатор.

На рис. 14.17 приведено окно проекта справочной системы после добавления rtf-файла, установки характеристик главного окна справочной системы и назначения числовых значений идентификаторам разделов.

После того как будет подготовлен файл проекта, можно выполнить компиляцию, щелкнув на находящейся в окне проекта кнопке **Save and Compile**. Однако первый раз компиляцию проекта справочной системы лучше выполнить выбором из меню **File** команды **Compile**, в результате выполнения которой открывается диалоговое окно **Compile a Help File** (рис. 14.18).

В этом окне следует установить флажок **Automatically display Help file in WinHelp when done** (Автоматически показывать созданную справочную систему по завершении компиляции), а затем нажать кнопку **Compile**. По завершении компиляции на экране появляется окно с информационным

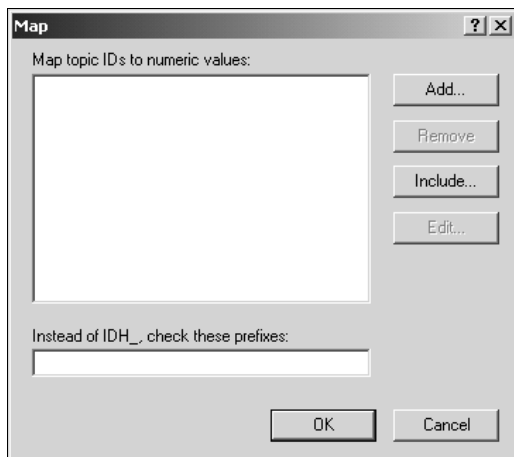
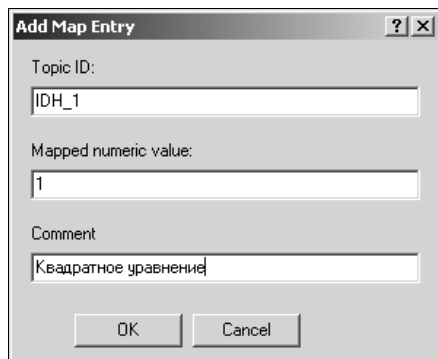
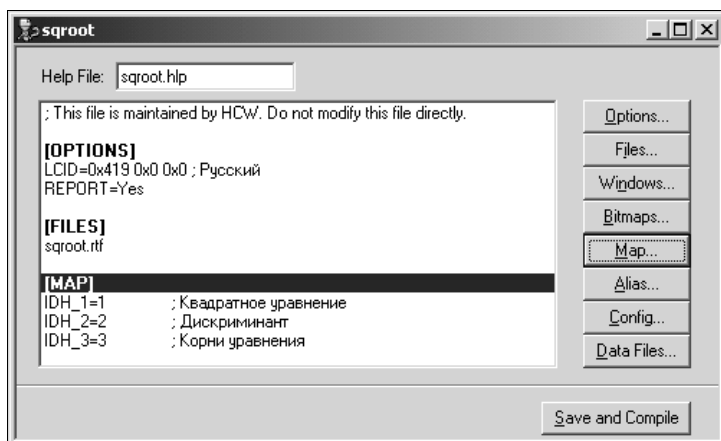
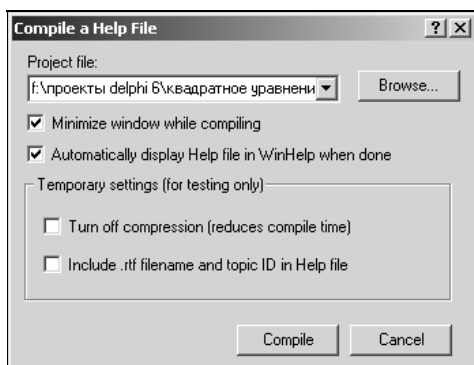
Рис. 14.15. Диалоговое окно **Map**Рис. 14.16. Диалоговое окно **Add Map Entry**

Рис. 14.17. Окно проекта справочной системы

Рис. 14.18. Диалоговое окно **Compile a Help File**

сообщением о результатах компиляции и, если компиляция выполнена успешно, окно созданной справочной системы. Созданный компилятором файл справочной системы (hlp-файл) будет помещен в ту папку, в которой находится файл проекта.

Использование справочной системы

Для того чтобы во время работы программы пользователь, нажав клавишу <F1>, мог получить справочную информацию, надо чтобы свойство

HelpFile главного окна приложения (рис. 14.19) содержало имя файла справочной системы, а свойство HelpContext — числовой идентификатор раздела справочной информации, который должен быть введен.

Файл справочной системы приложения лучше поместить в ту папку, в которой находится файл исполняемой программы.

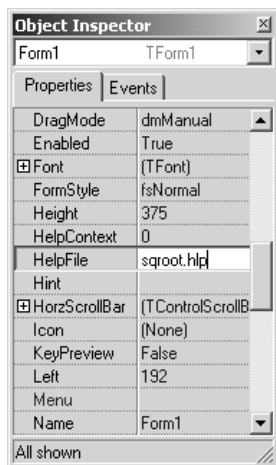


Рис. 14.19. Свойство HelpFile должно содержать имя файла справки

Для каждого компонента формы, например поля ввода, можно задать свой раздел справки. Раздел справки, который появляется, если фокус находится на компоненте и пользователь нажимает клавишу <F1>, определяется значением свойства HelpContext этого компонента. Если значение свойства HelpContext элемента управления равно нулю, то при нажатии клавиши <F1> появляется тот раздел справки, который задан для формы приложения.

Если в диалоговом окне есть кнопка **Справка**, то справочная информация выводится по-другому: для кнопки создается процедура обработки события OnClick, которая обращением к функции winhelp запускает программу Windows Help (winhlp32.exe). При вызове функции winhelp в качестве параметров указываются: идентификатор окна, которое запрашивает справочную информацию; имя файла справочной системы; константа, определяющая действие, которое должна выполнить программа Windows Help и уточняющий действие параметр.

Примечание

Идентификатор окна — это свойство Handle, которое доступно только во время работы программы.

Если необходимо вывести конкретный раздел справки, то в качестве параметра, определяющего действие, используется константа `HELP_CONTEXT`. Уточняющий параметр в этом случае задает раздел справки, который будет выведен на экран.

Ниже, в качестве примера, приведена процедура обработки события `OnClick` для кнопки **Справка** (`Button2`) диалогового окна программы решения квадратного уравнения.

```
// щелчок на кнопке Справка
procedure TForm1.Button2Click(Sender: TObject);
begin
    winhelp(Form1.Handle, 'sqroot.hlp', HELP_CONTEXT, 1);
end;
```

HTML Help Workshop

Современные программы выводят справочную информацию в стиле Internet. Окно, которое используется для вывода справки, напоминает окно **Internet Explorer**. И это не удивительно, т. к. для вывода справочной информации используются компоненты, составляющие основу Internet Explorer. Система отображения справочной информации является частью операционной системы, поэтому никакие дополнительные средства для вывода справочной информации не нужны.

Физически справочная информация находится в файлах с расширением `htm`. СНМ-файл — это так называемый компилированный HTML-документ. СНМ-файл получается путем компиляции (объединения) файлов, составляющих HTML-документ, который, как правило, состоит из нескольких HTML-файлов.

Процесс преобразования HTML-документа в справочную систему называют компиляцией. Исходной информацией для компилятора справочной системы являются HTML-файлы и файлы иллюстраций. В результате компиляции получается СНМ-файл, содержащий всю справочную информацию.

Наиболее просто справочную систему можно создать при помощи программы Microsoft HTML Help Workshop.

Чтобы создать справочную систему нужно:

- ☐ подготовить файлы справочной информации;
- ☐ создать файл проекта;
- ☐ создать файл контекста (содержания);
- ☐ выполнить компиляцию.

Последние три из перечисленных выше шагов могут быть выполнены непосредственно в программе HTML Help Workshop.

Подготовка справочной информации

Подготовить HTML-файл можно при помощи любого редактора текста. Наиболее быстро это можно сделать, если редактор позволяет сохранить набранный текст в HTML-формате. Если использовать обычный редактор, например **Блокнот**, то в этом случае придется изучить основы языка HTML.

В простейшем случае вся справочная информация может быть помещена в один-единственный HTML-файл. Однако если для навигации по справочной системе предполагается использовать вкладку **Содержание**, на которой будут перечислены разделы справки, то в этом случае информацию каждого раздела нужно поместить в отдельный HTML-файл. В качестве примера на рис. 14.20 приведено окно справочной системы программы **Квадратное уравнение**. На вкладке **Содержание** три пункта. Это значит, что исходная справочная информация была представлена тремя HTML-файлами.

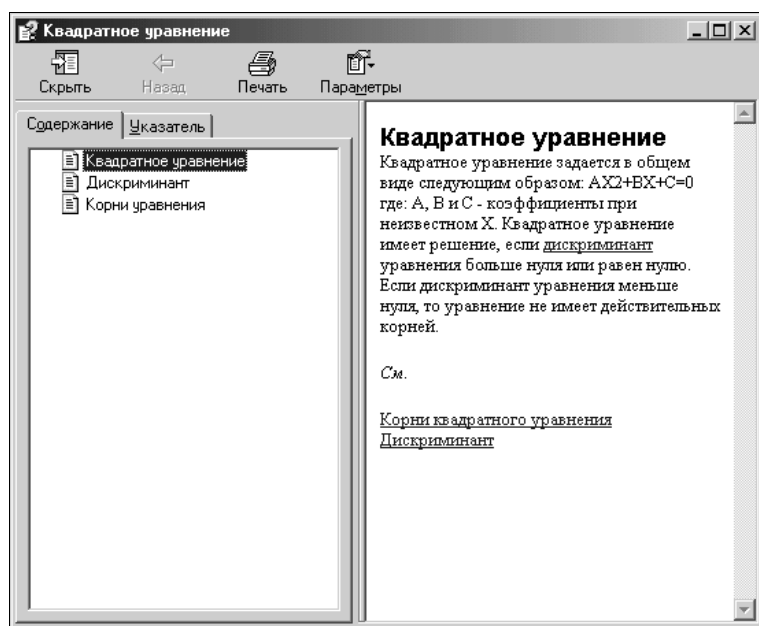


Рис. 14.20. Для навигации по справочной информации можно использовать вкладку **Содержание**

Использование редактора Microsoft Word

При использовании редактора Microsoft Word сначала нужно набрать текст разделов справки (каждый раздел в отдельном файле). Заголовки разделов и подразделов нужно оформить одним из стилей **Заголовок**. Заголовки разделов, как правило, оформляют стилем **Заголовок 1**, подразделов — **Заголовок 2**.

Следующее, что надо сделать, — вставить закладки в те точки документа, в которые предполагаются переходы из других частей документа. Чтобы вставить в документ закладку, нужно установить курсор в точку текста, в которой должна быть закладка, из меню **Вставка** выбрать команду **Закладка** и в поле **Имя закладки** диалогового окна **Закладка** (рис. 14.21) ввести имя закладки.

Имя закладки должно отражать суть предполагаемого перехода к закладке, содержимое помечаемого фрагмента текста. В имени закладки пробел использовать НЕЛЬЗЯ. Вместо пробела можно поставить символ подчеркивания. Заголовки, оформленные стилем **Заголовок**, помечать закладками не надо. Таким образом, если в создаваемой справочной системе предполагаются переходы только к заголовкам разделов справочной информации, закладки можно не вставлять. После этого следует приступить к расстановке гиперссылок.

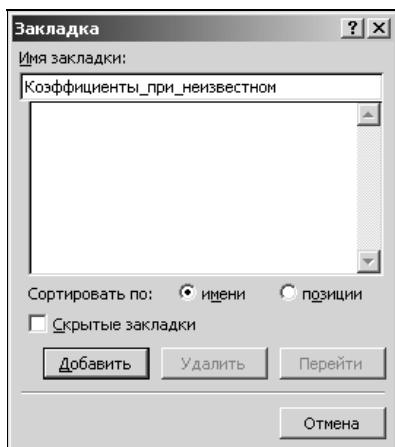


Рис. 14.21. Добавление закладки

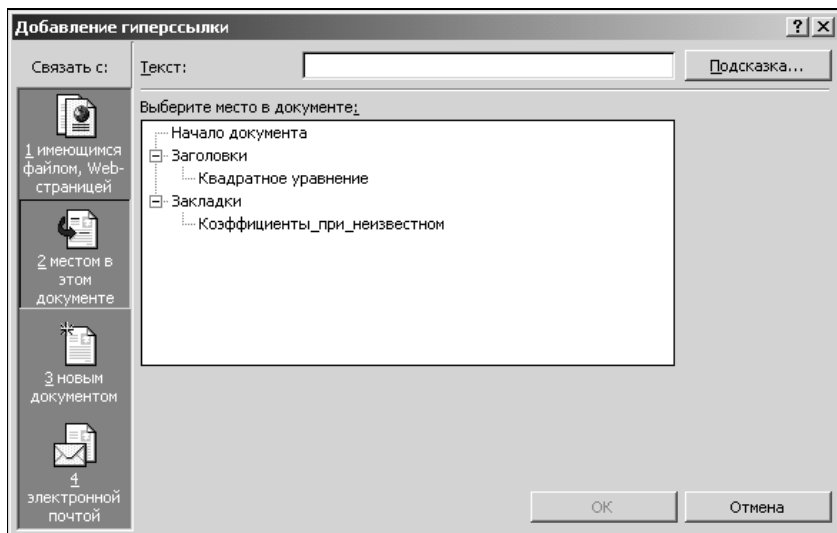


Рис. 14.22. Выбор точки документа для перехода по ссылке

Чтобы вставить в документ ссылку на закладку или заголовок, находящиеся в этом же документе, надо выделить фрагмент текста (слово или фразу), который должен быть гиперссылкой. Затем из меню **Вставка** выберите команду **Гиперссылка**, в появившемся окне **Добавление гиперссылки** (рис. 14.22) нажмите кнопку **Связать с местом в этом документе**, затем выберите заголовок или закладку, к которой должен быть выполнен переход.

Если нужно вставить в документ ссылку на раздел справки, который находится в другом файле, то в диалоговом окне **Добавление гиперссылки** нужно нажать кнопку **Файл** и в появившемся стандартном окне выбрать имя нужного HTML-файла.

После того как в документ будут помещены все необходимые гиперссылки, документ нужно сохранить в HTML-формате.

Использование HTML Help Workshop

Подготовить HTML-файл можно и при помощи HTML-редактора, входящего в состав HTML Help Workshop. Однако для этого надо знать хотя бы основы HTML — языка гипертекстовой разметки (далее приведены краткие сведения о HTML, которых достаточно для того, чтобы создать вполне приличную справочную систему).

Чтобы создать HTML-файл, надо запустить HTML Help Workshop, из меню **File** выбрать команду **New/HTML File** и в появившемся окне **HTML Title** (рис. 14.23) задать название раздела справки, текст которого будет находиться в создаваемом файле.

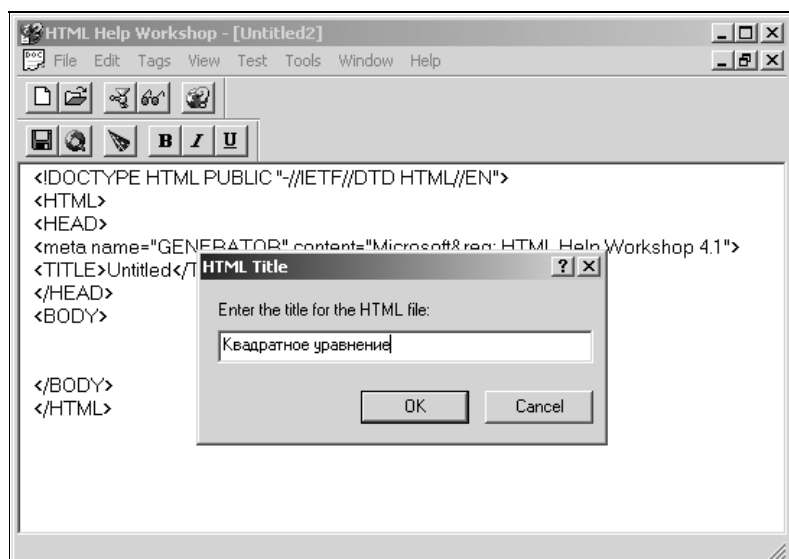


Рис. 14.23. Начало работы над новым HTML-файлом

После нажатия кнопки **ОК** становится доступным окно HTML-редактора, в котором находится шаблон HTML-документа. В этом окне, после строки `<BODY>`, можно набирать текст.

Основы HTML

HTML-документ представляет собой текст, в который помимо обычного текста включены специальные последовательности символов — *тэги*. Тэг начинается символом `<` и заканчивается символом `>`. Тэги используются программами отображения HTML-текста для форматирования текста в окне просмотра (тэги в окне просмотра не отображаются).

Большинство тэгов парные. Например, пара тэгов `<h2> </h2>` сообщает программе отображения HTML-документа, что текст, который находится между этими тэгами, является заголовком второго уровня и должен быть отображен соответствующим стилем.

В табл. 14.2 представлен минимальный набор тэгов, используя которые можно подготовить HTML-файл с целью дальнейшего его преобразования в СНМ-файл справочной системы.

Таблица 14.2. Тэги

Тэг	Пояснение
<code><TITLE> Название </TITLE></code>	Задает название HTML-документа. Программы отображения HTML-документов, как правило, выводят название документа в заголовке окна, в котором документ отображается. Если название не задано, то в заголовке окна будет выведено название файла
<code><BODY</code> <code>BACKGROUND = "Файл"</code> <code>BGCOLOR = "Цвет"</code> <code>TEXT="Цвет "></code>	Параметр BACKGROUND задает фоновый рисунок, BGCOLOR — цвет фона. TEXT — цвет символов HTML-документа
<code><BASEFONT FACE="Шрифт"</code> <code>SIZE=n></code>	Задает основной шрифт, который используется для отображения текста: FACE — название шрифта, SIZE — размер в относительных единицах. По умолчанию значение параметра SIZE равно 3. Размер шрифта заголовков (см. тэг <code><H></code>) берется от размера, заданного параметром SIZE
<code><H1></H1></code>	Определяет текст, находящийся между тэгами <code><H1></code> и <code></H1></code> как заголовок уровня 1. Пара тэгов <code><H2></H2></code> определяет заголовок второго уровня, а пара <code><H3></H3></code> — третьего

Таблица 14.2 (окончание)

Тэг	Пояснение
 	Конец строки. Текст, находящийся после этого тэга, будет выведен с начала новой строки
 	Текст, находящийся внутри этой пары тэгов, будет выделен полужирным
<I> </I>	Текст, находящийся внутри этой пары тэгов, будет выделен курсивом
	Помечает фрагмент документа закладкой. Имя закладки задает параметр NAME. Это имя используется для перехода к закладке
	Выделяет фрагмент документа как гиперссылку, при выборе которой происходит перемещение к закладке, имя которой указано в параметре HREF
	Выводит иллюстрацию, имя файла которой указано в параметре SRC
<!-- -->	Комментарий. Текст, находящийся между дефисами, на экран не выводится

```

<HTML>
<TITLE>Квадратное уравнение</TITLE>
<BODY BGCOLOR=#FFFFFF>
<BASEFONT FACE="Tahoma"SIZE=2>
<A NAME="Квадратное уравнение"><H2>Квадратное
уравнение</H2></A>Квадратное уравнение задается в общем виде следующим
образом:<BR>

$$AX^2+BX+C=0$$

<BR>
где: А,В и С — коэффициенты при неизвестном Х.<BR>
Квадратное уравнение имеет <A
HREF="sqrroot_02.htm#Корни_уравнения">корни </A>(решение), если
<A HREF="sqrroot_03.htm#Дискриминант">дискриминант </A>уравнения больше
или равен нулю. Если дискриминант уравнения меньше нуля, то уравнение
не имеет решения.<BR>
<BR>
<I>См.</I><BR>
<A HREF="sqrroot_03.htm#Дискриминант">Дискриминант</A><BR>
<A HREF="sqrroot_02.htm#Корни_уравнения">Корни уравнения</A><BR>
<BR>
</BODY>
</HTML>

```

Рис. 14.24. HTML-текст раздела справочной системы

Набирается HTML-текст обычным образом. Тэги можно набирать как прописными, так и строчными буквами. Однако, чтобы лучше была видна структура документа, рекомендуется записывать все тэги строчными (большими) буквами. Следующее, на что надо обратить внимание, это то, что программы отображения HTML-документов игнорируют "лишние" пробелы и другие "невидимые" символы (табуляция, новая строка). Это значит, что для того, чтобы фрагмент документа начинался с новой строки, в конце предыдущей строки надо поставить тэг `<BR>`, а чтобы между строками текста появилась пустая строка, в HTML-текст нужно вставить два тэга `<BR>` подряд.

Работая с HTML-редактором в программе HTML Help Workshop, в процессе набора HTML-текста, можно увидеть, как будет выглядеть набираемый текст. Для этого надо из меню **View** выбрать команду **In Browser** или нажать командную кнопку, на которой изображен стандартный значок Internet Explorer.

В качестве примера на рис. 14.24 приведен текст одного из разделов справочной системы программы **Квадратное уравнение**.

Создание файла справки

После того как будут созданы HTML-файлы справочной информации, в которых находятся все необходимые для навигации по справочной системе гиперссылки, можно приступить к непосредственному созданию справочной системы.

После запуска HTML Help Workshop надо из меню **File** выбрать команду **New/Project** и в окне **New Project — Destination** (рис. 14.25) ввести имя файла проекта справочной системы. После нажатия кнопки **Далее** в этом и следующем окне HTML Help Workshop должно выглядеть так, как показано на рис. 14.26.

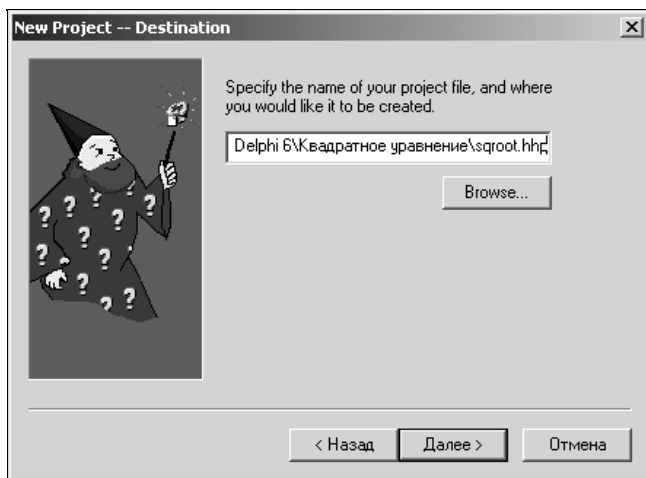


Рис. 14.25. Начало работы над новым проектом

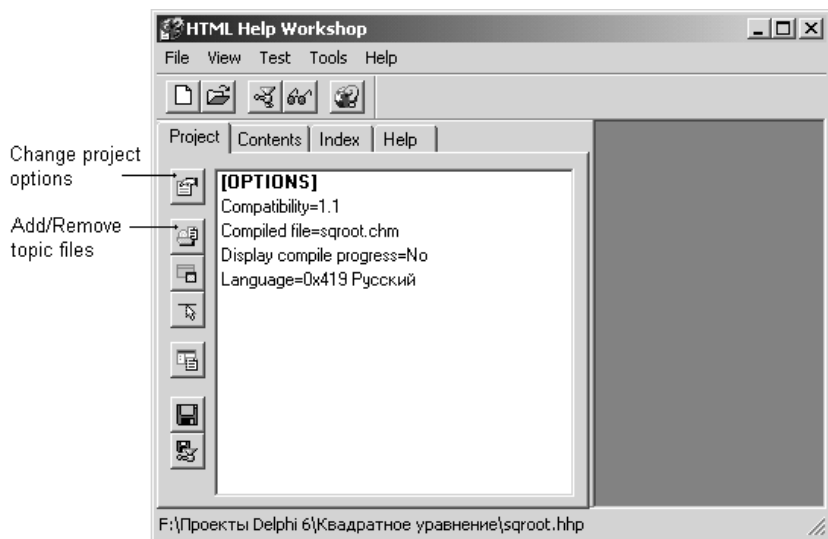


Рис. 14.26. Окно HTML Help Workshop в начале работы над новым проектом

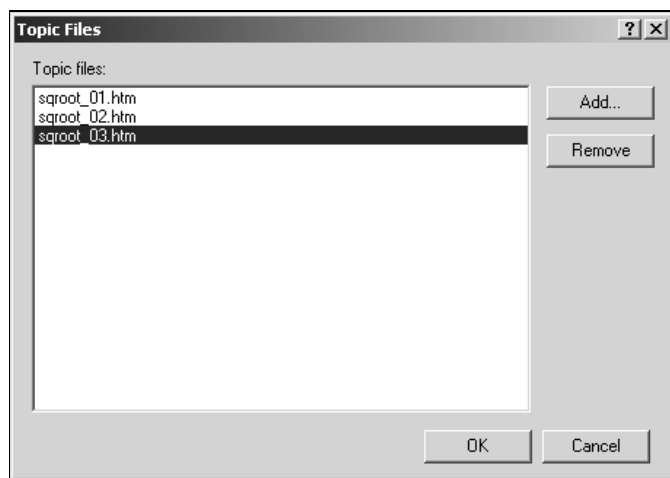


Рис. 14.27. Диалоговое окно **Topic Files**

Первое, что надо сделать, — сформировать раздел [FILES], который должен включать имена HTML-файлов, содержащих справочную информацию по разделам. Чтобы добавить в раздел [FILES] имя файла, надо нажать кнопку **Add/Remove topic files**, затем в появившемся диалоговом окне **Topic Files** (рис. 14.27) — кнопку **Add** и в появившемся стандартном диалоговом окне **Открыть** выбрать HTML-файл раздела справки. Если справочная информация распределена по нескольким файлам, то операцию добавления нужно повторить несколько раз. После того как в диалоговом окне **Topic Files** будут

перечислены все необходимые для создания справочной информации HTML-файлы, нужно нажать кнопку **ОК**. В результате этих действий в файле проекта появится раздел **[FILES]**, в котором будут перечислены HTML-файлы, используемые для создания справочной системы (рис. 14.28).

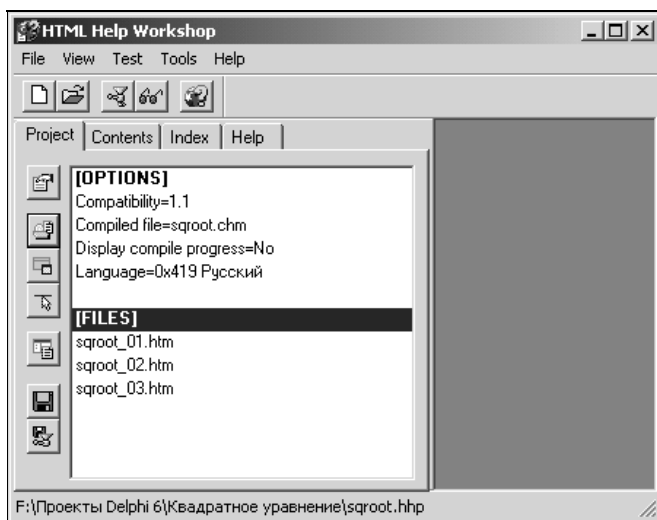


Рис. 14.28. В разделе **[FILES]** перечислены файлы, используемые для создания СНМ-файла

Следующее, что надо сделать, — задать главный (стартовый) раздел и заголовок окна, в котором будет выводиться справочная информация. Текст заголовка и имя файла главного раздела вводятся соответственно в поля **Title** и **Default file** вкладки **General** диалогового окна **Options** (рис. 14.29), которое появляется в результате нажатия кнопки **Change project options**.

Если для навигации по справочной системе предполагается использовать вкладку **Содержание**, то надо создать файл контекста. Чтобы это сделать, нужно щелкнуть на вкладке **Contents**, подтвердить создание нового файла контекста и задать имя файла контекста, в качестве которого можно использовать имя проекта. В результате станет доступной вкладка **Contents** (рис. 14.30), в которую нужно ввести содержание — названия разделов справочной системы.

Примечание

Содержание справочной системы принято изображать в виде иерархического списка. Элементы верхнего уровня соответствуют разделам, а подчиненные им элементы — подразделам и темам.

Чтобы на вкладку **Contents** добавить элемент, соответствующий разделу справочной системы, нужно нажать кнопку **Insert a heading**, в поле **Entry title** появившегося диалогового окна **Table of Contents Entry** (рис. 14.31) ввести название раздела и нажать кнопку **Add**. На экране появится окно **Path or**

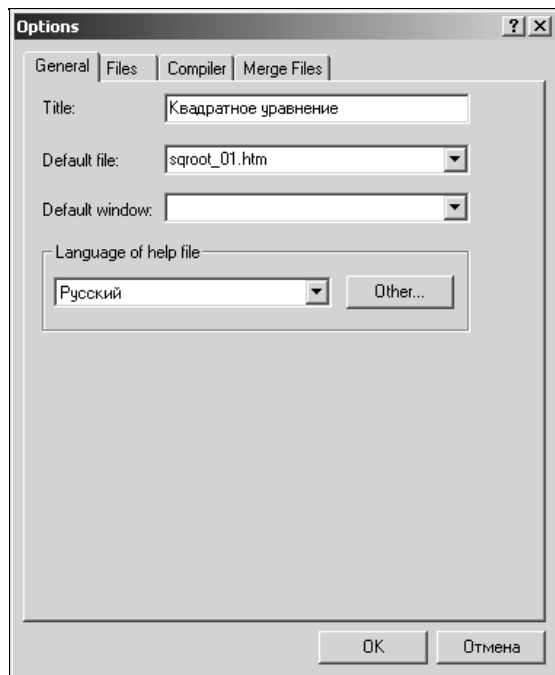


Рис. 14.29. Диалоговое окно Options

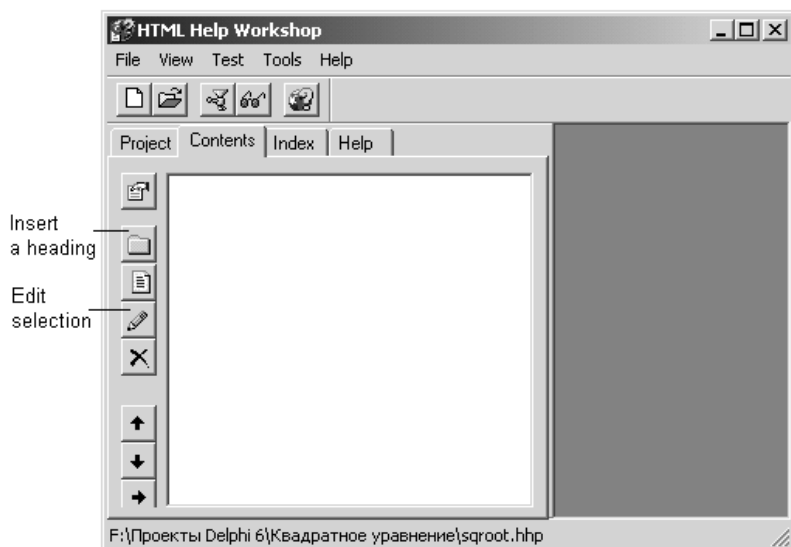


Рис. 14.30. Вкладка Contents

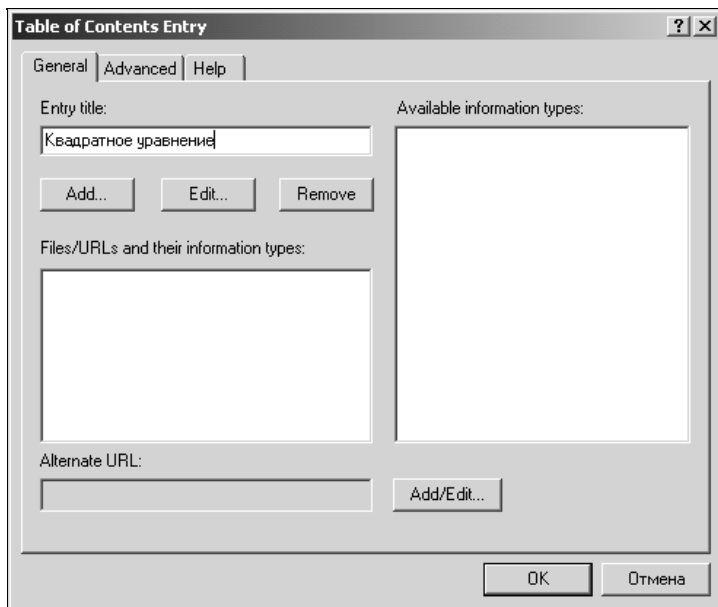


Рис. 14.31. Добавление элемента в список разделов

URL (рис. 14.32). В поле **HTML titles** этого окна будут перечислены названия разделов (заголовки HTML-файлов) справочной информации, которая находится во включенных в проект файлах (имена этих файлов указаны в разделе [FILES] вкладки **Project**). Если вместо названия раздела справочной информации будет указано имя файла, то это значит, что в этом файле нет тэга <TITLE>. Выбрав (по заголовку или по имени) нужный файл, надо нажать кнопку **OK**. В результате перечисленных выше действий на вкладке **Contents** появится строка с названием раздела справочной информации.

Если нужно изменить значок, соответствующий добавленному разделу, то надо нажать кнопку **Edit selection** и, используя список **Image index** вкладки **Advanced** окна **Table of Contents**, выбрать нужный значок (обычно рядом с названием раздела или подраздела изображена книжка).

Подраздел добавляется точно так же, как и раздел, но после того как подраздел будет добавлен, нужно нажать кнопку **Move selection right**. В результате этого уровень заголовка понизится, т. е. раздел станет подразделом.

Элементы содержания, соответствующие темам справочной информации, добавляются аналогичным образом, но этот процесс начинается нажатием кнопки **Insert a page**.

Иногда возникает необходимость изменить порядок следования элементов списка содержания или уровень иерархии элемента списка. Сделать это можно при помощи командных кнопок, на которых изображены стрелки.

Кнопки **Move selection up** и **Move selection down** перемещают выделенный элемент списка соответственно вверх и вниз. Кнопка **Move selection right** перемещает выделенный элемент вправо, т. е. делает его подчиненным предыдущему элементу списка. Кнопка **Move selection left** выводит элемент из подчиненности предыдущему элементу.

В качестве примера на рис. 14.33 приведена вкладка **Contents** справочной системы программы **Квадратное уравнение**.

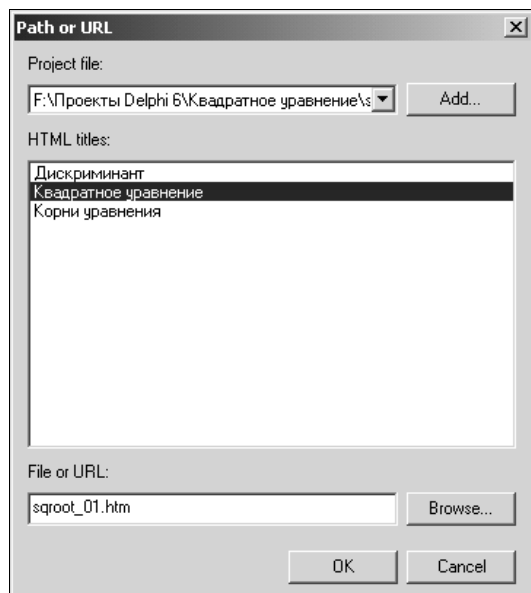


Рис. 14.32. Выбор файла, соответствующего элементу списка разделов

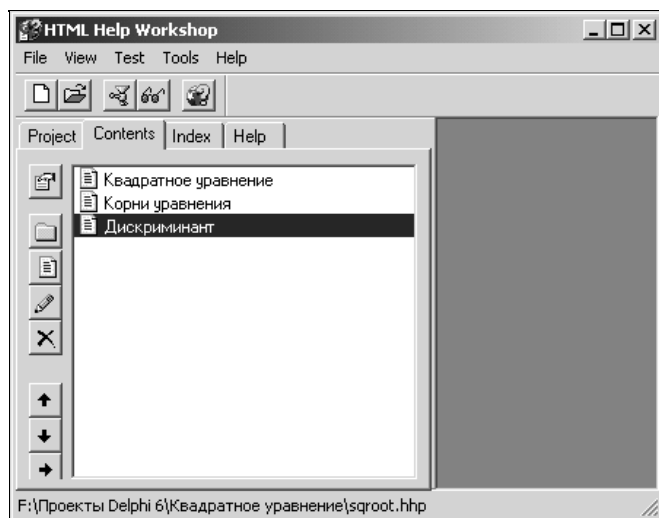


Рис. 14.33. Вкладка **Contents** содержит названия разделов справочной системы

Компиляция

Компиляция — это процесс преобразования исходной справочной информации в файл справочной системы (СНМ-файл).

Исходной информацией для HTML Help компилятора служат:

- ☐ Файл проекта (ННР-файл)
- ☐ Файл контекста (ННС-файл)
- ☐ Файлы справочной информации (НТМ-файлы)
- ☐ Файлы иллюстраций (GIF- и JPG-файлы)

Результатом компиляции является файл справочной системы (СНМ-файл).

Чтобы выполнить компиляцию, надо из меню **File** выбрать команду **Compile**, в появившемся диалоговом окне **Create a compiled file** (рис. 14.34) установить переключатель **Automatically display compiled help file when done** (После компиляции показать созданный файл справки) и нажать кнопку **Compile**. В результате этого будет создан файл справки, и на экране появится окно справочной системы, в котором будет выведена информация главного раздела.

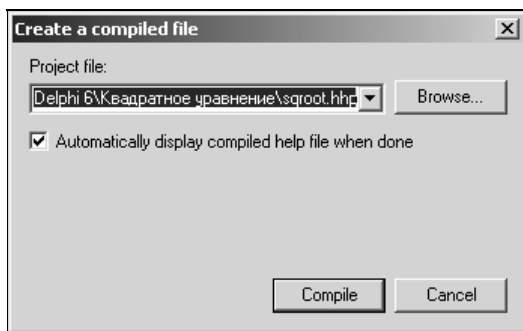


Рис. 14.34. Диалоговое окно **Create a compiled file**

Вывод справочной информации

Чтобы вывести справочную информацию, которая находится в СНМ-файле, нужно воспользоваться компонентом `ActiveX Hhopen`, который входит в состав Windows и представляет собой специальную динамическую библиотеку.

Первое, что надо сделать, — установить компонент `Hhopen` на одну из вкладок палитры компонентов. Для этого нужно из меню **Component** выбрать команду **Import ActiveX Control**. На экране появится окно **Import ActiveX**, в котором будут перечислены все зарегистрированные в реестре Windows компоненты.

В окне **Import ActiveX**, в списке зарегистрированных компонентов, нужно выбрать строку **hhopen OLE Control module** (рис. 14.35) и нажать кнопку **Create Unit**. В результате этого Delphi создаст файл представления — модуль, содержащий описание методов, свойств и событий компонента, и поместит значок компонента **Hhopen** на вкладку **ActiveX** (рис. 14.36).

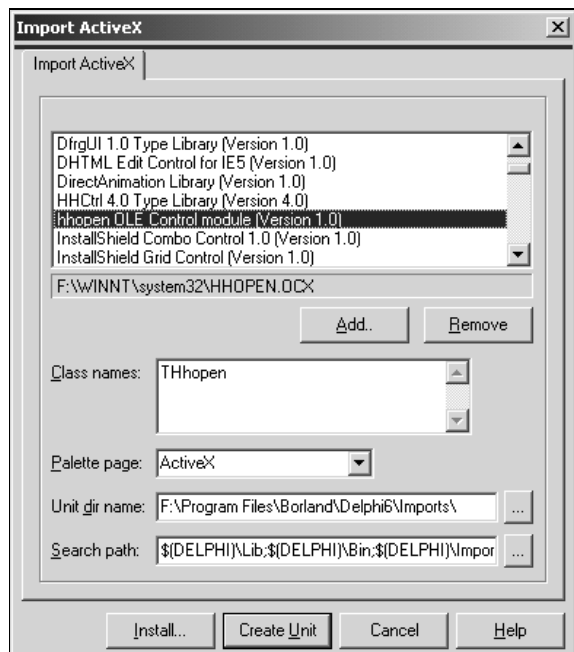


Рис. 14.35. Установка компонента **Hhopen**

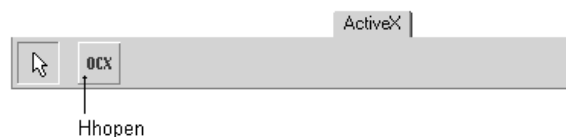


Рис. 14.36. Значок компонента **Hhopen**

Пролистав в окне редактора кода модуль **HHOPENLib_TLIB.pas**, который представляет собой интерфейс для доступа к элементу управления, можно найти описание класса **THhopen** (листинг 14.1).

Листинг 14.1. Описание класса **THhopen**

```
THhopen = class(ToleControl)
private
  FIntf: _DHhopen;
  function GetControlInterface: _DHhopen;
```

```

protected
    procedure CreateControl;
    procedure InitControlData; override;
public
    function OpenHelp(const HelpFile: WideString;
                     const HelpSection: WideString): Integer;

    procedure CloseHelp;

    property ControlInterface: _DHopen read GetControlInterface;
    property DefaultInterface: _DHopen read GetControlInterface;
published
    property isHelpOpened: WordBool index 1
        read GetWordBoolProp
        write SetWordBoolProp stored False;

end;

```

Класс `TNlopen` предоставляет два метода: `OpenHelp` и `CloseHelp`.

Метод `OpenHelp` обеспечивает вывод справочной информации, метод `CloseHelp` — закрытие окна справочной системы.

Метод `OpenHelp` имеет два параметра — имя файла справочной информации и имя раздела, содержимое которого будет выведено. В качестве имени раздела надо использовать имя HTML-файла, который использовался программой HTML Help Workshop в процессе создания СНМ-файла. Следует обратить внимание на то, что оба параметра должны быть строками типа `WideChar`.

Следующая программа, диалоговое окно которой приведено на рис. 14.37, а текст — в листинге 14.2, демонстрирует использование компонента `ActiveXNlopen` для вывода справочной информации. Компонент `Nlopen` добавляется в форму обычным образом. Так как во время работы программы он не отображается, то его можно поместить в любое место формы.

Листинг 14.2. Использование компонента `Nlopen`

```

unit ushh_;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, OleCtrls, HHOPENLib_TLB, StdCtrls;

type
    TForm1 = class(TForm)
        Label1: TLabel;

```

```

Edit1: TEdit;      // файл справки
Edit2: TEdit;      // раздел справки (имя HTML-файла)
Button1: TButton;  // кнопка Справка
Hhopen1: THhopen;  // ActiveX компонент Hhopen
Label2: TLabel;
Label3: TLabel;

```

```

procedure Button1Click(Sender: TObject);

```

```

private

```

```

    { Private declarations }

```

```

public

```

```

    { Public declarations }

```

```

end;

```

```

var

```

```

    Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

// щелчок на кнопке Справка

```

```

procedure TForm1.Button1Click(Sender: TObject);

```

```

var

```

```

    HelpFile : string;      // файл справки

```

```

    HelpTopic : string;     // раздел справки

```

```

    pwHelpFile : PWideChar;  // файл справки (указатель на строку WideChar)

```

```

    pwHelpTopic : PWideChar; // раздел (указатель на строку WideChar)

```

```

begin

```

```

    HelpFile := Edit1.Text;

```

```

    HelpTopic := Edit2.Text;

```

```

    // выделить память для строк WideChar

```

```

    GetMem(pwHelpFile, Length(HelpFile) * 2);

```

```

    GetMem(pwHelpTopic, Length(HelpTopic)*2);

```

```

    // преобразовать строку Ansi в строку WideString

```

```

    pwHelpFile := StringToWideChar(HelpFile,pwHelpFile,MAX_PATH*2);

```

```

    pwHelpTopic := StringToWideChar(HelpTopic,pwHelpTopic,32);

```

```

    // вывести справочную информацию

```

```

    Form1.Hhopen1.OpenHelp(pwHelpFile,pwHelpTopic);

```

```

end;

```

```

end.

```

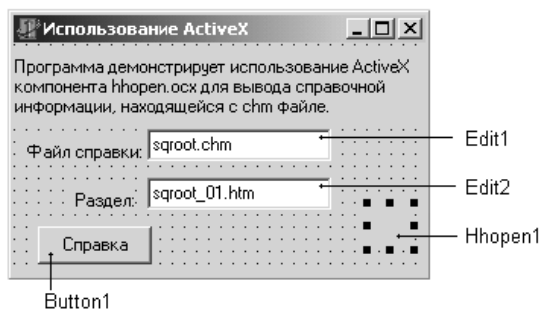
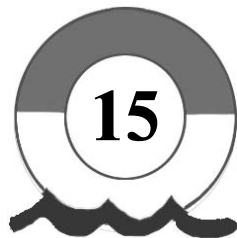



Рис. 14.37. Окно программы **Использование ActiveX**

Вывод справочной информации выполняет процедура обработки события `OnClick` на кнопке **Справка**. Так как параметры метода `OpenHelp` должны быть строками типа `WideChar`, то сначала необходимо выполнить преобразование строки `Ansi` в строку `WideChar`.



Примеры программ

Система проверки знаний

Тестирование широко применяется для оценки уровня знаний в учебных заведениях, при приеме на работу, для оценки квалификации персонала учреждений, т. е. практически во всех сферах деятельности человека. Испытуемому предлагается ряд вопросов (тест), на которые он должен ответить.

Обычно к каждому вопросу дается несколько вариантов ответа, из которых надо выбрать правильный. Каждому варианту ответа соответствует некоторая оценка. Общий балл, на основе которого делается вывод об уровне подготовленности испытуемого, получается суммированием оценок за ответы.

В этом разделе рассматривается программа, позволяющая автоматизировать процесс тестирования.

Требования к программе

В результате анализа различных тестов были сформулированы следующие требования к программе:

1. Программа должна обеспечить работу с тестом произвольной длины, т. е. не должно быть ограничения на количество вопросов в тесте.
2. Вопрос может сопровождаться иллюстрацией.
3. Для каждого вопроса может быть представлено до четырех возможных вариантов ответа со своей оценкой в баллах.
4. Результат тестирования должен быть отнесен к одному из четырех уровней, например, "отлично", "хорошо", "удовлетворительно" или "плохо".
5. Вопросы теста должны находиться в текстовом файле.

6. Программа должна быть инвариантна к различным тестам, т. е. изменения в тесте не должны вызывать требование изменения программы.
7. В программе должна быть заблокирована возможность возврата к предыдущему вопросу и пропуск текущего. Если вопрос предложен, то на него должен быть дан ответ.

На рис. 15.1 приведен пример диалогового окна программы тестирования во время ее работы.

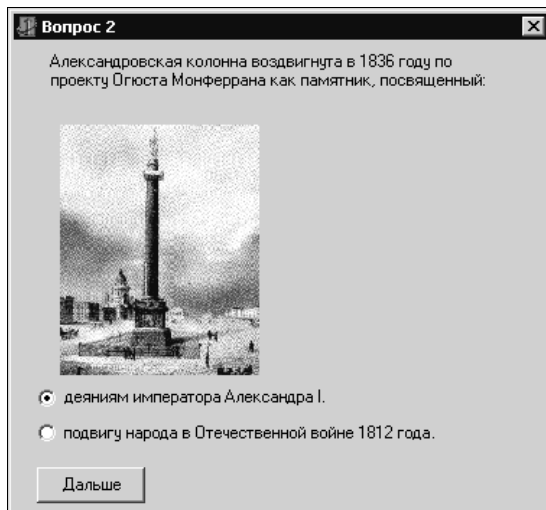


Рис. 15.1. Диалоговое окно программы тестирования

Файл теста

Тест представляет собой последовательность вопросов, на которые испытуемый должен ответить путем выбора правильного ответа из нескольких предложенных вариантов.

Файл теста состоит из трех разделов:

- ☐ раздел заголовка;
- ☐ раздел оценок;
- ☐ раздел вопросов.

Заголовок содержит общую информацию о тесте, например, о его назначении. Заголовок может состоять из нескольких строк. Признаком конца заголовка является точка, стоящая в начале строки.

Вот пример заголовка файла теста:

Сейчас Вам будут предложены вопросы о знаменитых памятниках и архитектурных сооружениях Санкт-Петербурга. Вы должны из предложенных вариантов ответа выбрать правильный.

За заголовком следует раздел оценок, в котором приводятся названия оценочных уровней и количество баллов, необходимое для достижения этих уровней. Название уровня должно располагаться в одной строке. Признаком конца раздела оценок является точка, стоящая в начале строки.

Вот пример раздела оценок:

```
Отлично
100
Хорошо
85
Удовлетворительно
60
Плохо
50
.
```

За разделом оценок следует раздел вопросов теста. Каждый вопрос начинается текстом вопроса, за которым может следовать имя файла иллюстрации, размещаемое на отдельной строке и начинающееся символом \. Имя файла иллюстрации является признаком конца текста вопроса. Если к вопросу нет иллюстрации, то вместо имени файла ставится точка.

После вопроса следуют альтернативные ответы. Текст альтернативного ответа может занимать несколько строк. В строке, следующей за текстом ответа, располагается количество баллов, которое испытуемый получит за выбор этого ответа. Если альтернативный ответ не является последним для текущего вопроса, то перед оценкой ставится запятая, если является — то точка.

Вот пример вопроса:

```
Какую формулу следует записать в ячейку B5, чтобы
вычислить сумму выплаты?
\tabl.bmp
=сумма (B2-B4)
, 0
=сумма (B2:B4)
, 2
=B2+B3+B4
. 1
```

В приведенном вопросе второй и третий ответы помечены как правильные (оценка за их выбор не равна нулю). При этом видно, что выбор второго альтернативного ответа дает более весомый вклад в общую сумму баллов.

Ниже, в качестве примера, приведен текст файла вопросов для контроля знания истории памятников и архитектурных сооружений Санкт-Петербурга.

Сейчас Вам будут предложены вопросы о знаменитых памятниках и архитектурных сооружениях Санкт-Петербурга. Вы должны из предложенных нескольких вариантов ответа выбрать правильный.

.

Вы прекрасно знаете историю Санкт-Петербурга!

8

Вы много знаете о Санкт-Петербурге, но на некоторые вопросы ответили неверно.

7

Вы недостаточно хорошо знаете историю Санкт-Петербурга.

6

Вы, наверное, только начали знакомиться с историей Санкт-Петербурга?

5

.

Архитектор Исаакиевского собора:

\isaak.bmp

Доменико Трезини

,0

Огюст Монферран

,1

Карл Росси

.0

Александровская колонна воздвигнута как памятник, посвященный:

\aleks.bmp

деяниям императора Александра I

,0

подвигу народа в Отечественной войне 1812 года

.1

Архитектор Зимнего дворца:

\herm.bmp

Бартоломео Растрелли

,1

Огюст Монферран

,0

Карл Росси

.0

Памятник русской военной славы собор Божией Матери Казанской (Казанский собор) построен по проекту русского зодчего:

.

А. Н. Воронихина

,1

И. Е. Старова

,0

В. И. Баженова

.0

Остров, на котором находится Ботанический сад, основанный императором Петром I, называется:

\bot.bmp

Заячий

,0

Медицинский

,0

Аптекарский

.1

Невский проспект получил свое название:

.

по имени реки, на берегах которой расположен Санкт-Петербург

,0

по имени близко расположенной Александро-Невской лавры

,1

в память о знаменитом полководце Александре Невском

.0

Создатель скульптурных групп Аничкова моста "Укрощение коня человеком":

\klodt.bmp

П. Клодт

,1

Э. Фальконе

.0

Скульптор знаменитого монумента "Медный всадник":

.

Э. Фальконе

,1

П. Клодт

.0

Примечание

Файл теста может быть подготовлен в текстовом редакторе Notepad или Microsoft Word. В случае использования Microsoft Word при сохранении текста следует указать, что надо сохранить только текст. Для этого в диалоговом окне **Сохранить** в списке **Тип файла** следует выбрать вариант **Только текст (*.txt)**.

Форма приложения

На рис. 15.2 приведен вид стартовой формы **Form1** во время разработки программы. Эта форма будет использоваться как для вывода вопросов теста и ввода ответов пользователя, так и для вывода начальной информации о тесте и результатов тестирования.

Поле метки Label5 предназначено для вывода текста вопроса, начальной информации о тесте и результатов тестирования.

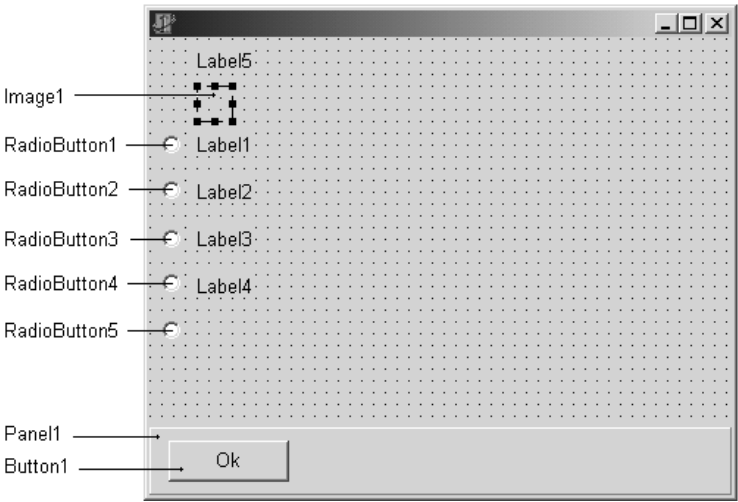


Рис. 15.2. Форма приложения **Test**

Поля Label1, Label2, Label3 и Label4 предназначены для вывода текста альтернативных ответов, а переключатели RadioButton1, RadioButton2, RadioButton3 и RadioButton4 — для выбора ответа.

Командная кнопка Button1 предназначена для подтверждения выбора ответа и перехода к следующему вопросу.

Следует обратить внимание на недоступный (невидимый) во время работы переключатель RadioButton5. Перед выводом очередного вопроса этот переключатель программно устанавливается в выбранное положение, что обеспечивает сброс (установку в невыбранное состояние) опций выбора ответа (RadioButton1, RadioButton2, RadioButton3 и RadioButton4).

Значения свойств стартовой формы приведены в табл. 15.1.

Таблица 15.1. Свойства стартовой формы

Свойство	Значение	Пояснение
Caption		
Height	362	
Width	562	
Color	clWhite	
Font.Name	Arial Cyr	
BorderIcons.biSystemMenu	True	Есть кнопка системного меню
BorderIcons.biMinimize	False	Нет кнопки "Свернуть окно"

Таблица 15.1 (окончание)

Свойство	Значение	Пояснение
BorderIcons.biMaximize	False	Нет кнопки "Развернуть окно"
BorderStyle	bsSingle	Тонкая граница окна, размер окна нельзя изменить

Примечание

Обратите внимание, что несмотря на то, что свойства `BorderIcons.biMinimize` и `BorderIcons.biMaximize` имеют значение `False`, т. е. форма не должна иметь кнопок минимизации и разворачивания окна, эти кнопки отображены в форме. Реальное воздействие значения этих свойств на вид окна проявляется лишь во время работы программы. Значение свойства `BorderStyle` также проявляет себя только во время работы программы.

В табл. 15.2—15.5 приведены значения свойств компонентов формы.

Таблица 15.2. Значения свойств компонентов `Label1` — `Label5`

Свойство	Компонент				
	Label1	Label2	Label3	Label4	Label5
Name	Label1	Label2	Label3	Label4	Label5
Caption	Label1	Label2	Label3	Label4	Label5
Left	32	32	32	32	32
Top	64	96	128	160	8
AutoSize	True	True	True	True	True
WordWrap	True	True	True	True	True

Таблица 15.3. Значения свойств компонентов `RadioButton1` — `RadioButton5`

Свойство	Компонент				
	RadioButton1	RadioButton2	RadioButton3	RadioButton4	RadioButton5
Name	RadioButton1	RadioButton2	RadioButton3	RadioButton4	RadioButton5
Caption					
Left	8	8	8	8	8
Top	64	96	128	160	174
Visible	True	True	True	True	False

Таблица 15.4. Значения свойств кнопки *Button1*

Свойство	Значение	Свойство	Значение
Name	Button1	Top	273
Caption	Ok	Height	28
Left	13	Width	82

Таблица 15.5. Значения свойств панели *Panel1*

Свойство	Значение	Свойство	Значение
Name	Panel1	Height	46
Caption		Align	alBottom

Вывод иллюстрации

Для вывода иллюстрации в форму добавлен компонент `Image`, значок которого (рис. 15.3) находится на вкладке **Additional** палитры компонентов. В табл. 15.6 приведены свойства компонента `Image`.



Рис. 15.3. Значок компонента `Image`

Таблица 15.6. Свойства компонента *Image*

Свойство	Определяет
Name	Имя компонента
Picture	Свойство, являющееся объектом типа <code>TBitmap</code> . Определяет выводимую картинку
Left	Расстояние от левого края формы до левой границы области картинки
Top	Расстояние от верхней границы формы до верхней границы области картинки

Таблица 15.6 (окончание)

Свойство	Определяет
Height	Высоту картинки
Width	Ширину картинки
AutoSize	Признак автоматического изменения размера компонента в соответствии с реальным размером картинки. Если значение свойства <code>AutoSize</code> равно <code>True</code> , то при изменении значения свойства <code>Picture</code> автоматически меняется размер области вывода иллюстрации так, чтобы была видна вся картинка. Если значение свойства <code>AutoSize</code> равно <code>False</code> , а размер картинки превышает размер области, то отображается только часть картинки
Stretch	Признак автоматического сжатия или растяжения картинки таким образом, чтобы она была видна полностью в области, размер которой задан свойствами <code>Width</code> и <code>Height</code>

Картинку, отображаемую в области `Image`, можно задать во время создания формы или во время работы программы. Во время создания формы картинка задается установкой значения свойства `Picture`. Во время работы программы — применением метода `LoadFromFile`.

Например, для разрабатываемого приложения инструкция вывода иллюстрации, находящейся в файле `isaak.bmp` (изображение Исаакиевского собора), может быть такой:

```
Image1.Picture.LoadFromFile('isaak.bmp');
```

Очевидно, что размер области формы, которая может использоваться для вывода иллюстрации, зависит от длины (количества слов) вопроса, длины и количества альтернативных ответов. Чем длиннее вопрос и ответы, тем больше места в поле формы они занимают и тем меньше места остается для иллюстрации.

При проектировании формы можно задать жесткие ограничения на размер областей, предназначенных для вопроса и альтернативных ответов, и жестко задать предельный размер иллюстрации. Однако можно поступить иначе. После прочтения из файла очередного вопроса вычислить, сколько места займут тексты вопроса и ответов и сколько места можно выделить для вывода иллюстрации (рис. 15.4).

Если реальный размер иллюстрации превышает размер области, выделенной для ее вывода, то необходимо вычислить коэффициент масштабирования и установить максимально возможные, пропорциональные ширине и высоте иллюстрации значения свойств `Width` и `Height` области вывода иллюстрации.

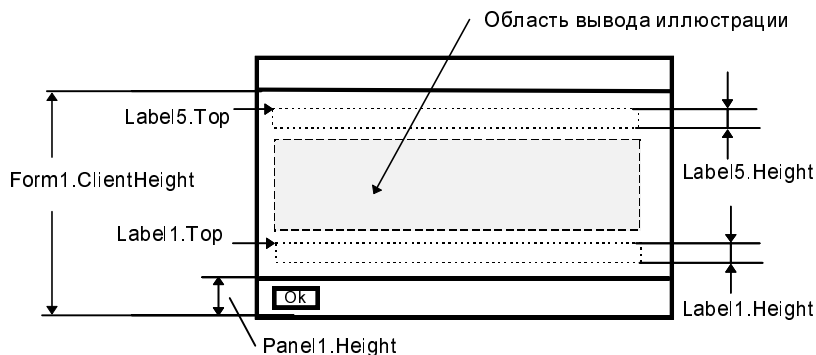


Рис. 15.4. Вычисление размера области вывода иллюстрации

Реальные размеры иллюстрации, загруженной в область `Image1`, можно получить из свойств `Image1.Picture.Bitmap.Width` и `Image1.Picture.Bitmap.Height`.

Загрузка файла теста

Передать имя файла теста программе тестирования можно через параметр командной строки путем настройки свойств значка, изображающего программу тестирования на рабочем столе или в папке.

Например, для настройки программы тестирования, значок запуска которой находится на рабочем столе, на работу с файлом теста `peterb.txt` необходимо щелкнуть правой кнопкой мыши на значке программы, из появившегося контекстного меню выбрать команду **Свойства** и в поле **Объект**, после имени файла программы (`Test1.exe`), ввести имя файла теста (`peterb.txt`), заключив его в двойные кавычки (рис. 15.5).

Примечание

Текст, находящийся в поле **Объект** вкладки **Ярлык** диалогового окна **Свойства**, называется командной строкой.

Программа может получить параметр, указанный в командной строке запуска программы, как значение функции `ParamStr(n)`, где n — номер параметра. Количество параметров командной строки находится в глобальной переменной `ParamCount`. Для приведенного выше примера командной строки запуска программы тестирования значение переменной `ParamCount` равно 1, а функции `ParamStr(1)` — `peterb.txt`.

Ниже приведен фрагмент программы, обеспечивающий прием параметра из командной строки:

```
if ParamCount = 0 then
begin
```

```
ShowMessage('Ошибка! Не задан файл вопросов теста.');
```

```
goto bye; // аварийное завершение программы
```

```
end;
```

```
FileName := ParamStr(1); // имя файла – параметр командной строки
```

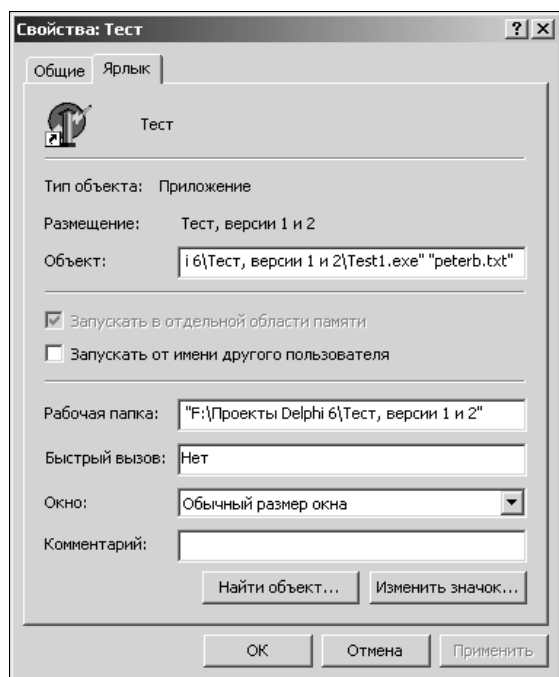


Рис. 15.5. Настройка программы тестирования

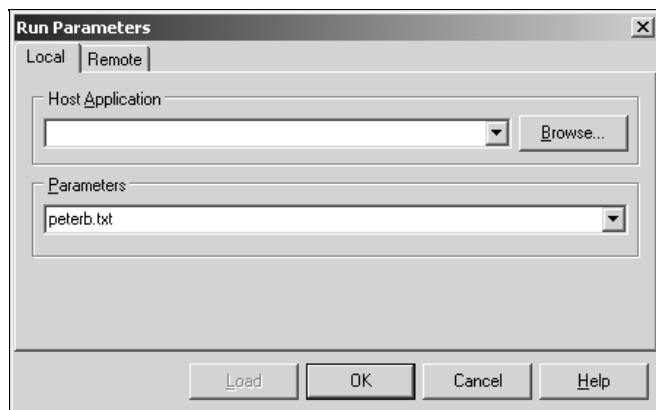


Рис. 15.6. Диалоговое окно **Run Parameters**

При запуске программы, использующей параметры командной строки, из среды разработки параметры нужно ввести в поле **Parameters** диалогового ок-

на **Run Parameters** (рис. 15.6), которое открывается в результате выбора из меню **Run** команды **Parameters**.

Текст программы

После создания формы в окно редактора кода, в секцию `implementation` следует поместить описание глобальных констант (раздел `const`) и переменных (раздел `var`). Затем можно приступить к созданию процедур обработки событий.

Их в программе три:

1. Обработка события `OnActivate` для стартовой формы.
2. Обработка события `OnClick` для командной кнопки `Button1`.
3. Процедура обработки события `OnClick` — одна, общая для переключателей выбора ответа.

В листинге 15.1 приведен полный текст программы.

Листинг 15.1. Программа тестирования

```
unit test1_;

interface

uses

  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type

  TForm1 = class(TForm)
    // вопрос
    Label5: TLabel;
    // альтернативные ответы
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    // переключатели выбора ответа
    RadioButton1: TRadioButton;
    RadioButton2: TRadioButton;
    RadioButton3: TRadioButton;
    RadioButton4: TRadioButton;

    Image1: TImage;    // область вывода иллюстрации
    Button1: TButton;  // кнопки ОК, Далее
```

```

RadioButton5: TRadioButton; // "служебная" кнопка

Panell1: TPanel;

procedure FormActivate(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure RadioButtonClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;      // форма

implementation

const
    N_LEV=4; // четыре уровня оценки
    N_ANS=4; // четыре варианта ответов

var
    f:TextFile;
    fn:string; // имя файла вопросов

    level:array[1..N_LEV] of integer; // сумма, соответствующая уровню
    mes:array[1..N_LEV] of string;      // сообщение, соответствующее уровню

    score:array[1..N_ANS] of integer; // балл за выбор ответа
    summa:integer; // набрано очков
    vopros:integer; // номер текущего вопроса
    n_otv:integer; // число вариантов ответа
    otv:integer; // номер выбранного ответа

// вывод начальной информации о тесте
procedure info(var f:TextFile;l:TLabel);
var
    s,buf:string;
begin
    buf:='';
    repeat
        readln(f,s);
        if s[1] <> '.'
            then buf := buf + s + ' ';
    until s[1] = '.';
    l.caption:=buf;
end;

```

// прочитать информацию об оценках за тест

Procedure GetLevel (**var** f:TextFile);

var

i:integer;

buf:string;

begin *// заполняем значения глобальных массивов*

i:=1;

repeat

readln(f,buf);

if buf[1] <> '.' **then begin**

mes[i]:=buf;

readln(f,level[i]);

i:=i+1;

end;

until buf[1]='.';

end;

// масштабирование иллюстрации

Procedure ScaleImage (Image1:TImage);

var

w,h:integer; *// максимально допустимые размеры картинки*

scaleX:real; *// коэф. масштабирования по X*

scaleY:real; *// коэф. масштабирования по Y*

scale:real; *// общий коэф. масштабирования*

begin

// вычислить максимально допустимые размеры картинки

w:=Form1.ClientWidth-10;

h:=Form1.ClientHeight

— Form1.Panel1.Height -5

— Form1.Label5.Top

— Form1.Label5.Height - 5;

if Form1.Label1.Caption <> ''

then h:=h-Form1.Label1.Height-5;

if Form1.Label2.Caption <> ''

then h:=h-Form1.Label2.Height-5;

if Form1.Label3.Caption <> ''

then h:=h-Form1.Label3.Height-5;

if Form1.Label4.Caption <> ''

then h:=h-Form1.Label4.Height-5;

// определить масштаб

if w>Image1.Picture.Bitmap.Width

then scaleX:=1

else scaleX:=w/Image1.Picture.Bitmap.Width;

```

if h>Image1.Picture.Bitmap.Height
    then scaleY:=1
    else scaleY:=h/Image1.Picture.Bitmap.Height;
if ScaleY<ScaleX
    then scale:=scaleY
    else scale:=scaleX;
// здесь масштаб определен
Image1.Top:=Form1.Label5.Top+Form1.Label5.Height+5;
Image1.Width:=Round(Image1.Picture.Bitmap.Width*scale);
Image1.Height:=Round(Image1.Picture.Bitmap.Height*scale);
end;

// вывод вопроса на экран
Procedure VoprosToScr(var f:TextFile;frm:TForm1;var vopros:integer);
var
    i:integer;
    code:integer;
    s,buf:string;
    ifn:string; // файл иллюстрации
begin
    vopros:=vopros+1;
    str(vopros:3,s);
    frm.caption:='Вопрос' + s;
    // выведем текст вопроса
    buf:='';
    repeat
        readln(f,s);
        if (s[1] <> '.') and (s[1] <> '\')
            then buf:=buf+s+' ';
    until (s[1] = '.') or (s[1] = '\');
    frm.Label5.caption:=buf;

    if s[1] <> '\'
        then Form1.Image1.Tag:=0
        else // к вопросу есть иллюстрация
            begin
                Form1.Image1.Tag:=1;
                ifn:=copy(s,2,length(s));
                try
                    Form1.Image1.Picture.LoadFromFile(ifn);
                except
                    on E:EFOpenError do
                        frm.tag:=0;
                end; // try
            end;
end;

```



```

// Читаем варианты ответов
i:=1;
repeat
    buf:='';
    repeat // читаем текст варианта ответа
        readln(f,s);
        if (s[1]<>'.' ) and (s[1] <> ',')
            then buf:=buf+s+' ';
    until (s[1]='.') or (s[1]=',');
    // прочитан альтернативный ответ
    val(s[2],score[i],code);
    case i of
        1: frm.Label1.caption:=buf;
        2: frm.Label2.caption:=buf;
        3: frm.Label3.caption:=buf;
        4: frm.Label4.caption:=buf;
    end;
    i:=i+1;
until s[1]='.';
// здесь прочитана иллюстрация и альтернативные ответы
// текст вопроса уже выведен
if Form1.Imagel.Tag =1 // есть иллюстрация к вопросу
    then begin
        ScaleImage(Form1.Imagel);
        Form1.Imagel.Visible:=TRUE;
    end;
// вывод альтернативных ответов
if Form1.Label1.Caption <> ''
then begin
    if Form1.Imagel.Tag =1
        then frm.Label1.top:=frm.Imagel.Top+frm.Imagel.Height+5
        else frm.Label1.top:=frm.Label5.Top+frm.Label5.Height+5;
    frm.RadioButton1.top:=frm.Label1.top;
    frm.Label1.visible:=TRUE;
    frm.RadioButton1.visible:=TRUE;
end;

if Form1.Label2.Caption <> ''
then begin
    frm.Label2.top:=frm.Label1.top+ frm.Label1.height+5;
    frm.RadioButton2.top:=frm.Label2.top;
    frm.Label2.visible:=TRUE;
    frm.RadioButton2.visible:=TRUE;
end;
end;

```

```
if Form1.Label3.Caption <> ''
then begin
    frm.Label3.top:=frm.Label2.top+ frm.Label2.height+5;
    frm.RadioButton3.top:=frm.Label3.top;
    frm.Label3.visible:=TRUE;
    frm.RadioButton3.visible:=TRUE;
end;

if Form1.Label4.Caption <> ''
then begin
    frm.Label4.top:=frm.Label3.top+ frm.Label3.height+5;
    frm.RadioButton4.top:=frm.Label4.top;
    frm.Label4.visible:=TRUE;
    frm.RadioButton4.visible:=TRUE;
end;
end;
```

Procedure ResetForm(frm:TForm1);

begin

// сделать невидимыми все метки и переключатели

frm.Label1.Visible:=FALSE;

frm.Label1.caption:='';

frm.Label1.width:=frm.ClientWidth-frm.Label1.left-5;

frm.RadioButton1.Visible:=FALSE;

frm.Label2.Visible:=FALSE;

frm.Label2.caption:='';

frm.Label2.width:=frm.ClientWidth-frm.Label2.left-5;

frm.RadioButton2.Visible:=FALSE;

frm.Label3.Visible:=FALSE;

frm.Label3.caption:='';

frm.Label3.width:=frm.ClientWidth-frm.Label3.left-5;

frm.RadioButton3.Visible:=FALSE;

frm.Label4.Visible:=FALSE;

frm.Label4.caption:='';

frm.Label4.width:=frm.ClientWidth-frm.Label4.left-5;

frm.RadioButton4.Visible:=FALSE;

frm.Label5.width:=frm.ClientWidth-frm.Label5.left-5;

frm.Image1.Visible:=FALSE;

end;

// определение достигнутого уровня

procedure Itog(summa:integer;frm:TForm1);

```

var
  i:integer;
  buf:string;
begin
  buf:='';
  str(summa:5,buf);
  buf:='Результаты тестирования'+chr(13)
    +'Всего баллов: '+buf;
  i:=1;
  while (summa < level[i]) and (i<N_LEV) do
    i:=i+1;
  buf:=buf+chr(13)+mes[i];
  frm.Label5.caption:=buf;
end;

{$R *.DFM}

procedure TForm1.FormActivate(Sender: TObject);
begin
  ResetForm(Form1);
  if ParamCount = 0
  then begin
    Label5.caption:= 'Не задан файл вопросов теста.';
    Button1.caption:='Ok';
    Button1.tag:=2;
    Button1.Enabled:=TRUE
  end
  else begin
    fn := ParamStr(1);
    assignfile(f,fn);
    {$I-}
    reset(f);
    {$I+}
    if IOResult=0 then
      begin
        Info(f,Label5); // прочитать и вывести информацию о тесте
        GetLevel(f);    // прочитать информацию об уровнях оценок
      end;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  case Button1.tag of
    0: begin

```

```

        Button1.caption:='Дальше';
        Button1.tag:=1;
        RadioButton5.Checked:=TRUE;
        // вывод первого вопроса
        Button1.Enabled:=False;
        ResetForm(Form1);
        VoprosToScr(f,Form1,vopros)
    end;
1: begin // вывод остальных вопросов
        summa:=summa+score[otv];
        RadioButton5.Checked:=TRUE;
        Button1.Enabled:=False;
        ResetForm(Form1);
        if not eof(f)
            then VoprosToScr(f,Form1,vopros)
            else
                begin
                    summa:=summa+score[otv];
                    closefile(f);
                    Button1.caption:='Ok';
                    Form1.caption:='Результат';
                    Button1.tag:=2;
                    Button1.Enabled:=TRUE;
                    Itog(summa,Form1);
                end;
        end;
2: begin // завершение работы
        Form1.Close;
    end;
end;
end;

procedure TForm1.RadioButtonClick(Sender: TObject);
begin
    if sender = RadioButton1
    then otv:=1
    else if sender = RadioButton1
    then otv:=2
    else if sender = RadioButton3
    then otv:=3
    else otv:=4;

    Button1.enabled:=TRUE;
end;

end.

```

После запуска программы и вывода на экран стартовой формы происходит событие `OnActivate`. Процедура `FormActivate` сначала вызывает процедуру `ResetForm`, которая, присваивая значение `False` свойству `Visible`, делает невидимыми поля вывода альтернативных ответов и все переключатели. Аналогично делается невидимой область иллюстрации. Кроме того, процедура устанавливает максимально возможную ширину полей меток альтернативных ответов.

После очистки формы проверяется, указан ли при запуске программы параметр — имя текстового файла с вопросами теста.

Если параметр не указан (значение `ParamCount` в этом случае равно нулю), то присвоением значения свойству `Caption` метки `Label5` выводится сообщение:

Не задан файл вопросов теста и свойству `Tag` кнопки `Button1` присваивается значение 2 (`Button1.Tag:=2;`)

Если параметр задан, то открывается файл теста.

Программа тестирования получает имя файла теста как результат функции `ParamStr(1)`. Реализация программы предполагает, что если имя файла теста задано без указания пути доступа к нему, то файл теста и файлы с иллюстрациями находятся в том же каталоге, что и программа тестирования. Если путь доступа указан, то файлы с иллюстрациями должны находиться в том же каталоге, что и файл теста. Такой подход позволяет сгруппировать все файлы одного теста в одном каталоге.

Открывается файл теста обычным образом. Сначала обращением к процедуре `AssignFile` имя файла связывается с файловой переменной, а затем вызывается инструкция открытия файла для чтения.

После успешного открытия файла вызывается процедура `Info`, которая считывает из файла информацию о тесте и выводит ее в форму присваиванием прочитанного текста свойству `Caption` поля метки `Label5`.

Затем вызывается процедура `GetLevel`, которая считывает из файла теста информацию об уровнях оценки. Эта процедура заполняет массивы `level` и `mes`.

После вывода информационного сообщения программа ждет, когда пользователь нажмет кнопку **Ok** (`Button1`).

Командная кнопка `Button1` используется для:

- ☐ аварийного завершения работы приложения (в случае, если не задано имя файла теста);
- ☐ начала тестирования (после прочтения информационного сообщения);
- ☐ перехода к следующему вопросу (после выбора одного из ответов);
- ☐ завершения работы программы (после прочтения результатов тестирования).

Свойство `Tag` кнопки `Button1` используется для идентификации текущего состояния формы и выбора действия при нажатии кнопки `Button1`.

После вывода информации о тесте значение свойства `Tag` кнопки `Button1` равно нулю. Поэтому в результате первого нажатия кнопки `Button1` выполняется та часть программы, которая обеспечивает:

- ❑ вывод первого вопроса;
- ❑ замену текста, находящегося на командной кнопке (свойство `Caption` с **Ok** на **Дальше**;
- ❑ установку невидимого служебного переключателя `RadioButton5` в выбранное состояние.

Кроме того, присваиванием значения `False` свойству `Enabled` кнопка `Button1` делается недоступной, тем самым блокируется переход к следующему вопросу до тех пор, пока не будет выбран один из ответов. Значению свойства `Button1.Tag` присваивается единица, тем самым выполняется подготовка к обработке следующего нажатия кнопки `Button1`.

После выбора ответа и нажатия кнопки **Дальше** (`Button1`) (в этом случае значение свойства `Button1.Tag` равно единице) к набранной сумме баллов добавляется количество баллов за выбранный ответ. Затем, если конец файла не достигнут, вызывается процедура вывода очередного вопроса. Если конец файла достигнут, то сначала закрывается файл теста, текст на кнопке `Button1` изменяется на **Ok**, а значение `Button1.Tag` устанавливается на 2. Затем посредством процедуры `Itog` выводятся результаты тестирования.

Если значение `Button1.Tag` равно двум, то применением метода `Close` к форме `Form1` закрывается окно программы, в результате чего программа завершает работу.

Вывод вопроса и альтернативных ответов выполняет процедура `VoprosToScr`. Сначала процедура увеличивает счетчик вопросов `vopros` и присвоением значения свойству формы `Caption` выводит номер текущего вопроса в заголовок окна. Затем процедура читает строки из файла теста до тех пор, пока первым символом очередной прочитанной строки не будет точка или "обратная наклонная черта".

После вывода текста вопроса делается проверка: какой символ используется в качестве признака конца вопроса. Если обратная наклонная черта, что свидетельствует о том, что к вопросу есть иллюстрация, то свойству `Form1.Image1.Tag` присваивается единица и из прочитанной строки выделяется имя файла иллюстрации.

Загрузка иллюстрации осуществляется применением метода `LoadFromFile` к свойству `Image1.Picture`. Однако после загрузки иллюстрация на экране не появляется, т. к. значение свойства `Image1.Visible` равно `False`.

После считывания иллюстрации процедура считывает вопросы. После обработки последнего вопроса, если была загружена иллюстрация, вызовом про-

цедуры `ScaleImage` вычисляется и устанавливается размер области иллюстрации. После этого установкой значения свойства `Image1.Top` задается положение верхней границы области иллюстрации, а присваиванием значения `True` свойству `Image1.Visible` иллюстрация делается видимой.

Так как количество символов в тексте вопроса и число альтернативных ответов от вопроса к вопросу могут меняться, и, следовательно, на экране они могут занимать разное количество строк, то каждый раз перед выводом текста очередного ответа устанавливается значение свойства `Top` как расстояние от нижней границы предыдущего альтернативного ответа. Для поля вывода первого альтернативного ответа (`Label1`) значение `Top` вычисляется от нижней границы поля вопроса (`Label5`) или, если к вопросу есть иллюстрация, от нижней границы поля иллюстрации (`Image1`).

Выбор ответа пользователь осуществляет щелчком одного из переключателей. После вывода вопроса ни один из переключателей, соответствующих альтернативному ответу, не является выбранным. Выбран только переключатель `RadioButton5`, который находится за панелью `Panel1` и поэтому не виден пользователю.

Для обработки события `OnClick` переключателей `RadioButton1`, `RadioButton2`, `RadioButton3` и `RadioButton4` в программе используется общая процедура — `TForm1.RadioButtonClick`. Эта процедура получает в качестве параметра объект, на котором произошло событие. Сравнивая полученное значение с именами объектов-кнопок выбора, процедура присваивает значение глобальной переменной `otv`, которая используется процедурой `VoprosToScr` для увеличения набранной суммы баллов. Кроме того, процедура `TForm1.RadioButtonClick` делает доступной кнопку перехода к следующему вопросу **Дальше** (`Button1`), которая после вывода очередного вопроса является недоступной.

Процедура `Itog`, сравнивая набранную сумму баллов `summa` со значением элементов массива `level`, определяет, какого уровня достиг испытуемый, и выводит соответствующее сообщение присвоением значения свойству `Label5.Caption`.

Усовершенствование программы

Очевидно, что приведенный выше текст программы был бы намного проще и изящней, если бы поля вывода альтернативных ответов и переключателя выбора ответов были бы объединены в массивы. Тогда программа могла бы обращаться к полям и переключателям не по имени, а по индексу.

Delphi позволяет объединить компоненты в массив, однако создаваться такие компоненты должны не во время разработки формы приложения, а динамически — во время работы программы.

На рис. 15.7 приведен вид формы усовершенствованного приложения.

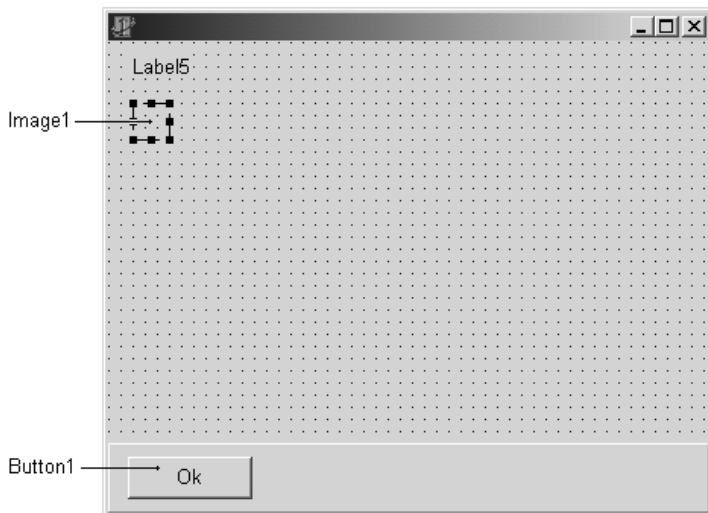


Рис. 15.7. Форма приложения **Тест, версия 2**

На форме отсутствуют поля вывода альтернативных ответов и переключатели выбора правильного ответа. Они будут созданы во время работы программы.

Объявление массива компонентов ничем не отличается от объявления обычного массива — указывается имя массива, диапазон изменения индекса и тип элементов массива. Ниже приведено объявление массивов компонентов формы разрабатываемой программы:

```
answer: array[1..N_ANSWERS] of TLabel; // альтернативные ответы
selector: array[1..N_ANSWERS+1] of TRadioButton; // кнопки выбора ответа
```

Однако, для того чтобы компонент появился в форме, одного объявления недостаточно. Компонент — это объект Delphi, и его объявление — это только указатель на область памяти, который без наличия объекта ни на что не указывает. Создается компонент путем применения метода `Create` к указателю на компонент, в нашем случае — к элементу массива.

Например, инструкции

```
answer[1] := TLabel.Create(self);
answer[1].Parent := Form1;
```

создают компонент (`TLabel`) и помещают его в форму приложения (`Form1`).

После создания компонента программа должна выполнить его настройку, т. е. ту работу, которую во время создания формы приложения выполняет программист при помощи окна **Object Inspector**. Под настройкой понимается присваивание начальных значений тем свойствам компонента, предопределенные значения которых не отвечают предъявляемым требованиям.

Если компонент должен реагировать на некоторое событие, то нужно написать процедуру обработки этого события и поместить объявление созданной процедуры в объявление типа формы. Например, объявление типа формы разрабатываемой программы должно выглядеть так:

type

```
TForm1 = class (TForm)
    Label5: TLabel;    // поле вывода вопроса
    Image1: TImage;    // область вывода иллюстрации
    Panel1: TPanel;
    Button1: TButton;   // кнопка Ok, Далее, Завершить

    procedure FormActivate(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure SelectorClick(Sender: TObject);

private
    { Private declarations }
public
    { Public declarations }
end;
```

В отличие от других строк, сгенерированных Delphi, объявление

```
procedure SelectorClick(Sender: TObject)
```

вставлено в текст программы вручную.

Примечание

При создании процедуры обработки события для обычного компонента (компонента, который добавлен в форму во время разработки формы программы) Delphi автоматически генерирует заготовку процедуры обработки события и ее объявление. Программист должен написать только инструкции процедуры. В случае создания процедуры обработки события для компонента, который создается динамически, программист должен полностью написать текст процедуры и поместить ее объявление в объявление формы.

После того как будет написана процедура обработки события, нужно связать эту процедуру с конкретным компонентом. Делается это путем присвоения имени процедуры обработке свойству, имя которого совпадает с именем обрабатываемого события. Например, инструкция

```
selector[1].OnClick := SelectorClick;
```

задает процедуру обработки события OnClick для компонента selector[1].

В листинге 15.2 приведен полный текст программы **Тест, версия 2**.

Листинг 15.2. Программа тестирования, версия 2

```

unit test2_;

interface

uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Dialogs, StdCtrls, ExtCtrls;

type
    TForm1 = class (TForm)
        Label5: TLabel;    // поле вывода вопроса
        Image1: TImage;    // область вывода иллюстрации
        Panel1: TPanel;
        Button1: TButton;  // кнопки Ok, Далее, Завершить

        procedure FormActivate(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure Button1Click(Sender: TObject);
        procedure SelectorClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1; // форма

implementation

const
    N_ANSWERS=4; // четыре варианта ответов
    N_LEVEL=4;   // четыре уровня оценки

var
    // динамически создаваемые компоненты
    answer: array[1..N_ANSWERS] of TLabel; // альтернативные ответы
    selector: array[1..N_ANSWERS+1] of TRadioButton; // кнопки выбора ответа
    f: TextFile;
    fn: string; // имя файла вопросов

    level: array[1..N_LEVEL] of integer; // сумма, соответствующая уровню
    mes: array[1..N_LEVEL] of string;   // сообщение, соответствующее уровню
    score: array[1..N_ANSWERS] of integer; // очки за выбор ответа

```

```
summa:integer; // набрано очков
vopros:integer; // номер текущего вопроса
n_otv:integer; // число вариантов ответа
otv:integer;    // номер выбранного ответа
```

```
// установка формы в исходное состояние
```

```
Procedure ResetForm(frm:TForm1);
```

```
var
    i:integer;
begin
    for i:=1 to N_ANSWERS do
    begin
        answer[i].width:=frm.ClientWidth-answer[i].left-5;
        answer[i].Visible:=FALSE;
        Selector[i].Visible:=FALSE;
    end;
    frm.Label5.width:=frm.ClientWidth-frm.Label5.left-5;
    frm.Image1.Visible:=False;
end;
```

```
// определение достигнутого уровня
```

```
procedure Itog(summa:integer;frm:TForm1);
```

```
var
    i:integer;
    buf:string;
begin
    buf:='';
    str(summa:5,buf);
    buf:='Результаты тестирования'+chr(13)
        +'Всего баллов: '+buf;
    i:=1;
    while (summa < level[i]) and (i<N_LEVEL) do
        i:=i+1;
    buf:=buf+chr(13)+mes[i];
    frm.Label5.caption:=buf;
end;
```

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

```
var
    i: integer;
begin
    // создадим пять меток для вывода вопроса и альтернативных ответов
    for i:=1 to N_ANSWERS do
    begin
        answer[i]:=TLabel.Create(self);
```

```

    answer[i].Parent:=Form1;
    answer[i].Left:=36;
    answer[i].WordWrap:=True;
end;

// создадим переключатели для выбора ответа
for i:=1 to N_ANSWERS+1 do
begin
    selector[i]:=TRadioButton.Create(self);
    selector[i].Parent:=self;
    selector[i].Caption:='';
    selector[i].Width:=17;
    selector[i].Left:=16;
    selector[i].Visible:=False;
    selector[i].Enabled:=True;
    selector[i].OnClick:=SelectorClick;
end;
ResetForm(Form1);
end;

// вывод начальной информации о тесте
procedure info(var f:TextFile;l:TLabel);
var
    s,buf:string;
begin
    buf:='';
    repeat
        readln(f,s);
        if s[1] <> '.'
            then buf:=buf+s+' ';
    until s[1] = '.';
    Form1.Label5.caption:=buf;
end;

// прочитать информацию об оценках за тест
Procedure GetLevel(var f:TextFile);
var
    i:integer;
    buf:string;
begin // заполняем значения глобальных массивов
    i:=1;
    repeat
        readln(f,buf);
        if buf[1] <> '.' then begin
            mes[i]:=buf;

```

```

        readln(f, level[i]);
        i:=i+1;
    end;
until buf[1]='.';
end;

// масштабирование иллюстрации
Procedure ScalePicture;
var
    w,h:integer; // максимально допустимые размеры картинки
    scaleX:real; // коэф. масштабирования по X
    scaleY:real; // коэф. масштабирования по Y
    scale:real; // общий коэф. масштабирования
    i:integer;
begin
    // вычислить максимально допустимые размеры картинки
    w:=Form1.ClientWidth-Form1.Label5.Left;
    h:=Form1.ClientHeight
        - Form1.Panel1.Height -5
        - Form1.Label5.Top
        - Form1.Label5.Height - 5;
    for i:=1 to N_ANSWERS do
        if answer[i].Caption <> ''
            then h:=h-answer[i].Height-5;

    // здесь определена максимально допустимая величина иллюстрации
    // определить масштаб
    if w>Form1.Imagel.Picture.Width
        then scaleX:=1
        else scaleX:=w/Form1.Imagel.Picture.Width;
    if h>Form1.Imagel.Picture.Height
        then scaleY:=1
        else scaleY:=h/Form1.Imagel.Picture.Height;
    if ScaleY<ScaleX
        then scale:=scaleY
        else scale:=scaleX;

    // здесь масштаб определен
    Form1.Imagel.Top:=Form1.Label5.Top+Form1.Label5.Height+5;
    Form1.Imagel.Left:=Form1.Label5.Left;
    Form1.Imagel.Width:=Round(Form1.Imagel.Picture.Width*scale);
    Form1.Imagel.Height:=Round(Form1.Imagel.Picture.Height*scale);
    Form1.Label5.Visible:=TRUE;
end;

// вывод вопроса на экран
Procedure VoprosToScr(var f:TextFile;frm:TForm1;var vopros:integer);

```

```

var
  i:integer;
  code:integer;
  s,buf:string;
  ifn:string; // файл иллюстрации
begin
  vopros:=vopros+1;
  str(vopros:3,s);
  frm.caption:='Вопрос' + s;
  //Выведем текст вопроса
  buf:='';
  repeat
    readln(f,s);
    if (s[1] <> '.') and (s[1] <> '\')
      then buf:=buf+s+' ';
  until (s[1] = '.') or (s[1] = '\');
  frm.Label5.caption:=buf;

  if s[1] = '\'
  then // к вопросу есть иллюстрация
  begin
    frm.Image1.Tag:=1;
    ifn:=copy(s,2,length(s));
    try
      frm.Image1.Picture.LoadFromFile(ifn);
    except
      on E:EOFOpenError do
        frm.tag:=0;
      end // try
    end
  else frm.Image1.Tag:=0;

  // Читаем варианты ответов
  for i:=1 to N_ANSWERS do
  begin
    answer[i].caption:='';
    answer[i].Width:=frm.ClientWidth-Form1.Label5.Left-5;
  end;
  i:=1;
  repeat
    buf:='';
    repeat // читаем текст варианта ответа
      readln(f,s);
      if (s[1]<>'.') and (s[1] <> ',')
        then buf:=buf+s+' ';
    until (s[1]='.') or (s[1]=',');

```

```

        // прочитан альтернативный ответ
        val(s[2],score[i],code);
        answer[i].caption:=buf;
        i:=i+1;
until s[1]='.';
// здесь прочитана иллюстрация и альтернативные ответы

if Form1.Image1.Tag =1  // есть иллюстрация к вопросу ?
then begin
    ScalePicture;
    Form1.Image1.Visible:=TRUE;
end;

// вывод альтернативных ответов
i:=1;
while (answer[i].caption <> '') and (i <= N_ANSWERS) do
begin
    if i = 1
    then
        if frm.Image1.Tag =1
        then answer[1].top:=frm.Image1.Top+frm.Image1.Height+5
        else answer[i].top:=frm.Label5.Top+frm.Label5.Height+5
    else
        answer[i].top:=answer[i-1].top+ answer[i-1].height+5;
    selector[i].top:=answer[i].top;
    selector[i].visible:=TRUE;
    answer[i].visible:=TRUE;
    i:=i+1;
end;
end;

{$R *.DFM}

procedure TForm1.FormActivate(Sender: TObject);
begin
    ResetForm(Form1);
    if ParamCount = 0
    then
        begin
            Label5.font.color:=clRed;
            Label5.caption:='Не задан файл вопросов теста.';
            Button1.caption:='Ok';
            Button1.tag:=2;
            Button1.Enabled:=TRUE
        end
end

```

```

else begin
    fn:=ParamStr(1);
    assignfile(f,fn);
    {$I-}
    reset(f);
    {I+}
    if IOResult=0 then
        begin
            Info(f,Label5);
            GetLevel(f);
        end;
    summa:=0;
end;

end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    case Button1.tag of
        0: begin
            Button1.caption:='Дальше';
            Button1.tag:=1;
            Selector[N_ANSWERS+1].Checked:=TRUE;
            // вывод первого вопроса
            Button1.Enabled:=False;
            ResetForm(Form1);
            VoprosToScr(f,Form1,vopros)
        end;
        1: begin // вывод остальных вопросов
            summa:=summa+score[otv];
            Selector[N_ANSWERS+1].Checked:=TRUE;
            Button1.Enabled:=False;
            ResetForm(Form1);
            if not eof(f)
            then VoprosToScr(f,Form1,vopros)
            else
                begin
                    closefile(f);
                    Button1.caption:='Ok';
                    Form1.caption:='Результат';
                    Button1.tag:=2;
                    Button1.Enabled:=TRUE;
                    Itog(summa,Form1);
                end;
        end;
    end;
end;

```



```

2: begin // завершение работы
    Form1.Close;
end;
end;

end;

// щелчок на кнопке выбора ответа
procedure TForm1.SelectorClick(Sender: TObject);
var
    i:integer;
begin
    i:=1;
    while selector[i].Checked = FALSE do
        i:=i+1;
    otv:=i;
    Button1.enabled:=TRUE;
end;
end.

```

По сравнению с первым вариантом программа **Тест, версия 2** обладает существенным преимуществом. Для ее модернизации, например для увеличения количества альтернативных ответов, достаточно изменить только описание именованной константы `N_ANSWERS`.

Игра Сапер 2001

Всем пользователям Windows хорошо знакома игра **Сапер**. В этом разделе рассматривается аналогичная программа — игра **Сапер 2001**. Пример окна программы в конце игры приведен на рис. 15.8.

Правила

Игровое поле состоит из клеток, в каждой из которых может быть мина. Задача игрока — найти все мины и пометить их флажками.

Используя кнопки мыши, игрок может открыть клетку или поставить в нее флажок, указав тем самым, что в клетке находится мина. Клетка открывается щелчком левой кнопки мыши, флажок ставится щелчком правой. Если в клетке, которую открыл игрок, есть мина, то происходит взрыв (сапер ошибся, а он, как известно, ошибается только один раз) и игра заканчивается. Если в клетке мины нет, то в этой клетке появляется число, соответствующее количеству мин, находящихся в соседних клетках. Анализируя информацию о количестве мин в клетках, соседних с уже открытыми, игрок может обнаружить, пометить флажками все мины. Ограничений на коли-

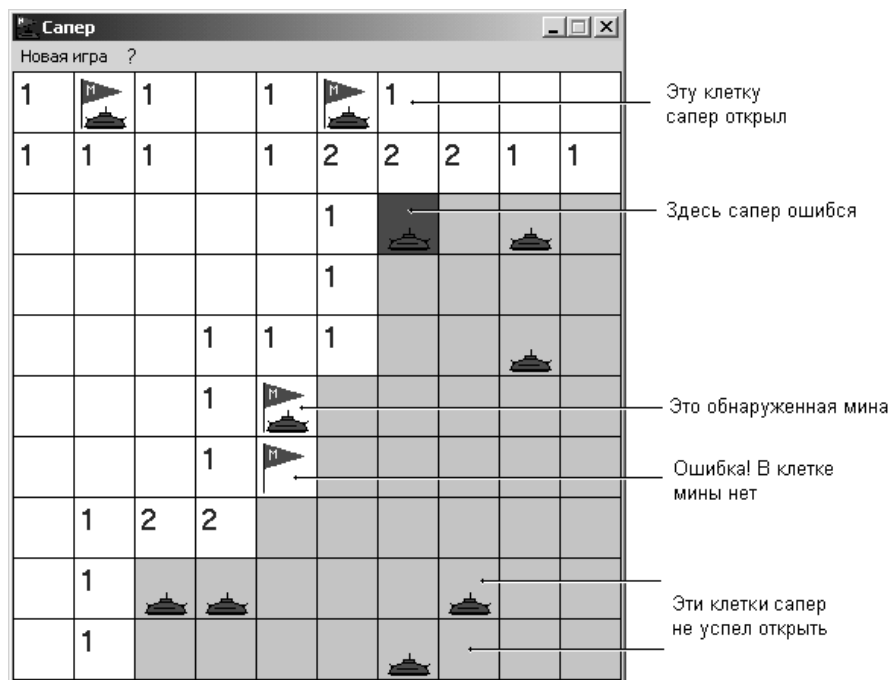


Рис. 15.8. Окно программы **Сапер 2001**

чество клеток, помеченных флажками, нет. Однако для завершения игры (выигрыша) флажки должны быть установлены только в тех клетках, в которых есть мины. Ошибочно установленный флажок можно убрать, щелкнув правой кнопкой мыши в клетке, в которой он находится.

Представление данных

В программе игровое поле представлено массивом размерности $N+2$ на $M+2$, где $N \times M$ — размер игрового поля. Элементы массива с номерами строк от 1 до N и номерами столбцов от 1 до M соответствуют клеткам игрового поля (рис. 15.9), первые и последние столбцы и строки соответствуют границе игрового поля.

В начале игры каждый элемент массива, соответствующий клетке игрового поля, может содержать число от 0 до 9. Если элемент массива содержит нуль, то он задает пустую клетку, рядом с которой мин нет. Для пустых (не содержащих мин) клеток элементы массива принимают значения от 1 до 8. Если же клетка содержит мину, то соответствующий ей элемент массива имеет значение 9.

Элементы массива, определяющие границы поля, содержат значения -3 .

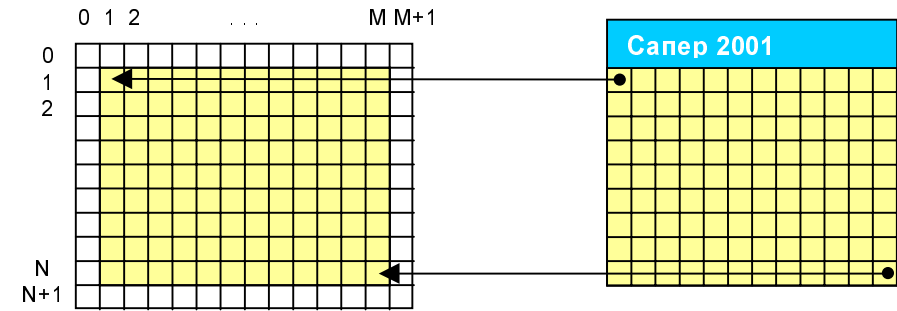


Рис. 15.9. Клетке игрового поля соответствует элемент массива

-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3
-3	9	1	0	0	0	0	0	0	0	0	-3
-3	1	1	0	0	0	0	0	0	0	0	-3
-3	1	2	2	1	0	0	0	1	1	1	-3
-3	1	9	9	1	0	0	0	2	9	2	-3
-3	1	2	2	1	0	0	0	2	9	3	-3
-3	0	0	0	0	0	0	0	2	3	9	-3
-3	0	1	2	2	1	0	0	1	9	2	-3
-3	0	2	9	9	1	0	0	1	1	1	-3
-3	0	2	9	3	1	0	0	0	0	0	-3
-3	0	1	1	1	0	0	0	0	0	0	-3
-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3

Рис. 15.10. Массив в начале игры

В качестве примера на рис. 15.10 изображен массив, описывающий состояние поля в начале игры.

В процессе игры состояние игрового поля меняется (игрок открывает клетки и ставит флажки), и соответственно меняются значения элементов массива. Если игрок поставил в клетку флажок, то значение соответствующего элемента массива увеличивается на 100. Например, если флажок поставлен правильно, т. е. в клетку, в которой есть мина, то значение соответствующего элемента массива будет 109. Если флажок поставлен ошибочно, например в пустую клетку, элемент массива будет содержать число 100. Если игрок открыл клетку, то значение элемента массива увеличивается на 200. Такой способ кодирования позволяет сохранить информацию об исходном состоянии клетки.

Форма приложения

Главная (стартовая) форма игры **Сапер 2001** приведена на рис. 15.11.

Следует обратить внимание, что размер формы не соответствует размеру игрового поля. Нужный размер формы будет установлен во время работы программы. Делает это процедура обработки события `OnFormActivate`, которая, на основе информации о размере игрового поля (количестве клеток по вертикали и горизонтали) и о размере клеток, устанавливает значение свойств `ClientHeight` и `ClientWidth`, определяющих размер клиентской области главного окна программы.

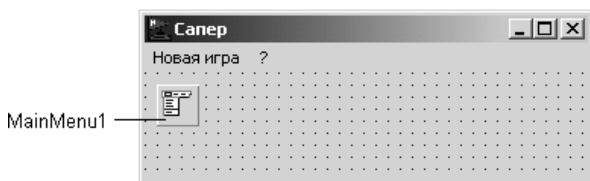


Рис. 15.11. Главная форма программы **Сапер 2001**

Главное окно программы содержит компонент `MainMenu1`, который представляет собой главное меню программы. Значок компонента `MainMenu` находится на вкладке **Standard** (рис. 15.12).



Рис. 15.12. Компонент `MainMenu`

Значок компонента `MainMenu` можно поместить в любое место формы, т. к. во время работы программы он не виден. Пункты меню появляются в верхней части формы в результате настройки меню. Для настройки меню используется редактор меню, который запускается двойным щелчком левой кнопки мыши на значке компонента или выбором из контекстного меню компонента команды **Menu Designer**. В начале работы над новым меню, сразу после добавления компонента к форме, в окне редактора находится один единственный прямоугольник — заготовка пункта меню. Чтобы превратить эту заготовку в меню, нужно в окне **Object Inspector**, в поле **Caption** ввести название меню.

Примечание

Если перед какой-либо буквой в названии меню ввести знак `&`, то во время работы программы можно будет активизировать этот пункт меню путем нажатия `<Alt>` и клавиши, соответствующей символу, перед которым стоит знак `&`. В названии меню эта буква будет подчеркнута.

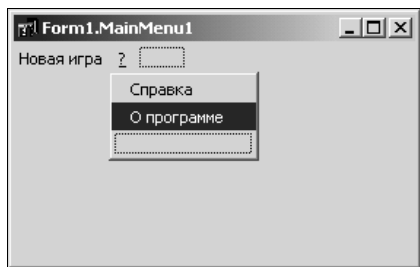


Рис. 15.13. Структура меню программы **Сапер 2001**

Чтобы добавить в главное меню элемент, нужно в окне редактора меню выбрать последний (пустой) элемент меню и ввести название нового пункта.

Чтобы добавить в меню команду, надо выбрать пункт меню, в который нужно добавить команду, переместить указатель активного элемента меню в конец списка команд меню и ввести название команды.

На рис. 15.13 приведено окно редактора меню, в котором находится меню программы **Сапер 2001**.

После того как будет сформирована структура меню, можно, используя **Object Inspector**, выполнить настройку элементов меню. Каждый элемент меню (пункты и команды) — это объект типа `TMenuItem`. Свойства элементов меню (табл. 15.7) определяют его вид во время работы программы.

Таблица 15.7. Свойства элементов меню

Свойство	Определяет
Name	Имя элемента меню. Используется для доступа к свойствам
Caption	Название меню или команды
Bitmap	Значок, который отображается слева от названия элемента меню
Enabled	Признак доступности элемента меню. Если значение свойства равно <code>False</code> , то элемент меню недоступен, при этом название меню отображается серым цветом

При выборе элемента меню во время работы программы происходит событие `Click`. Чтобы создать процедуру обработки этого события, нужно в окне формы выбрать пункт меню и щелкнуть левой кнопкой мыши, Delphi создаст шаблон процедуры обработки этого события.

Начало игры

В начале игры программа расставляет мины и для каждой клетки поля подсчитывает, сколько мин находится в соседних клетках. Процедура `NewGame`, ее текст приведен в листинге 15.3, решает эту задачу.

Листинг 15.3. Процедура NewGame

```
// новая игра — генерирует новое поле
procedure NewGame();

var
    row,col : integer; // координаты клетки (индексы массива)
    n : integer;        // количество поставленных мин
    k : integer;        // кол-во мин в соседних клетках
begin
    // Очистим эл-ты массива, соответствующие клеткам
    // игрового поля.
    for row :=1 to MR do
        for col :=1 to MC do
            Pole[row,col] := 0;

    // расставим мины
    Randomize(); // инициализация генератора случайных чисел (ГСЧ)
    n := 0; // кол-во мин
    repeat
        row := Random(MR) + 1;
        col := Random(MC) + 1;
        if ( Pole[row,col] <> 9) then
            begin
                Pole[row,col] := 9;
                n := n+1;
            end;
    until ( n = NM);

    // для каждой клетки вычислим
    // кол-во мин в соседних клетках
    for row := 1 to MR do
        for col := 1 to MC do
            if ( Pole[row,col] <> 9) then
                begin
                    k :=0;
                    if Pole[row-1,col-1] = 9 then k := k + 1;
                    if Pole[row-1,col] = 9 then k := k + 1;
                    if Pole[row-1,col+1] = 9 then k := k + 1;
                    if Pole[row,col-1] = 9 then k := k + 1;
                    if Pole[row,col+1] = 9 then k := k + 1;
                    if Pole[row+1,col-1] = 9 then k := k + 1;
                    if Pole[row+1,col] = 9 then k := k + 1;
                    if Pole[row+1,col+1] = 9 then k := k + 1;
                    Pole[row,col] := k;
                end;
```

```
status := 0; // начало игры
nMin   := 0; // нет обнаруженных мин
nFlag  := 0; // нет поставленных флагов
end;
```

После того как процедура `NewGame` расставит мины, процедура `ShowPole`, ее текст приведен в листинге 15.4, выведет изображение игрового поля.

Листинг 15.4. Процедура `ShowPole`

```
// Показывает поле
Procedure ShowPole(Canvas : TCanvas; status : integer);
var
    row,col : integer;
begin
    for row := 1 to MR do
        for col := 1 to MC do
            Kletka(Canvas, row, col, status);
end;
```

Процедура `ShowPole` выводит изображение поля последовательно, клетка за клеткой. Вывод изображения отдельной клетки выполняет процедура `Kletka`, ее текст приведен в листинге 15.5. Процедура `Kletka` используется для вывода изображения поля в начале игры, во время игры и в ее конце. В начале игры (значение параметра `status` равно 0) процедура выводит только контур клетки, во время игры (значение параметра `status` равно 1) — количество мин в соседних клетках или флажок, а в конце (значение параметра `status` равно 2) отображает исходное состояние клетки и действия пользователя. Информацию о фазе игры процедура `Kletka` получает через параметр `status`.

Листинг 15.5. Процедура `Kletka`

```
// выводит на экран изображение клетки
Procedure Kletka(Canvas : TCanvas; row, col, status : integer);
var
    x,y : integer; // координаты области вывода
begin
    x := (col-1)* W + 1;
    y := (row-1)* H + 1;

    if status = 0 then
        begin
            Canvas.Brush.Color := clLtGray;
```

```

        Canvas.Rectangle(x-1,y-1,x+W,y+H);
        exit;
    end;

    if Pole[row,col] < 100 then
        begin
            Canvas.Brush.Color := clLtGray; // не открытые – серые
            Canvas.Rectangle(x-1,y-1,x+W,y+H);
            // если игра завершена (status = 2), то показать мины
            if (status = 2) and (Pole[row,col] = 9)
                then Mina(Canvas, x, y);
            exit;
        end;

    // открытая клетка
    Canvas.Brush.Color := clWhite; // открытые белые
    Canvas.Rectangle(x-1,y-1,x+W,y+H);
    if ( Pole[row,col] = 100)
        then exit; // клетка открыта, но она пустая

    if ( Pole[row,col] >= 101) and (Pole[row,col] <= 108) then
        begin // в соседних клетках есть мины
            Canvas.Font.Size := 14;
            Canvas.Font.Color := clBlue;
            Canvas.TextOut(x+3,y+2,IntToStr(Pole[row,col] -100));
            exit;
        end;

    if ( Pole[row,col] >= 200) then
        Flag(Canvas, x, y);

    if (Pole[row,col] = 109) then // на этой мине подорвались!
        begin
            Canvas.Brush.Color := clRed;
            Canvas.Rectangle(x-1,y-1,x+W,y+H);
        end;

    if ( (Pole[row,col] mod 10) = 9) and (status = 2) then
        Mina(Canvas, x, y);

end;
```

Игра

Во время игры программа воспринимает нажатия кнопок мыши и, в соответствии с правилами игры, открывает клетки или ставит в них флажки.

Основную работу выполняет процедура обработки события `OnMouseDown` (ее текст приведен в листинге 15.6). Сначала процедура преобразует координаты точки, в которой игрок нажал кнопку мыши, в координаты клетки игрового поля. Затем делает необходимые изменения в массиве `Pole` и, если нажата правая кнопка, рисует в клетке флажок. Если нажата левая кнопка в клетке, в которой нет мины, то эта клетка открывается, на экран выводится ее содержимое. Если нажата левая кнопка в клетке, в которой есть мина, то игра заканчивается и вызывается процедура `ShowPole`, которая показывает все мины, в том числе и те, которые игрок не успел найти.

Листинг 15.6. Обработка события `OnMouseDown` на поверхности игрового поля

```
// нажатие кнопки мыши на игровом поле
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
                               Shift: TShiftState; X, Y: Integer);

var
    row, col : integer;
begin
    if status = 2 // игра завершена
        then exit;

    if status = 0 then // первый щелчок
        status := 1;

    // преобразуем координаты мыши в индексы
    // клетки поля
    row := Trunc(Y/H) + 1;
    col := Trunc(X/W) + 1;

    if Button = mbLeft then
        begin
            if Pole[row,col] = 9 then
                begin // открыта клетка, в которой есть мина
                    Pole[row,col] := Pole[row,col] + 100;
                    status := 2; // игра закончена
                    ShowPole(Form1.Canvas, status);
                end
            else if Pole[row,col] < 9 then
                Open(row,col);
        end
    else
        if Button = mbRight then
            if Pole[row,col] > 200 then
                begin
```

```

// уберем флаг и закроем клетку
nFlag := nFlag - 1;
Pole[row,col] := Pole[row,col]-200; // уберем флаг
x := (col-1)* W + 1;
y := (row-1)* H + 1;
Canvas.Brush.Color := clLtGray;
Canvas.Rectangle(x-1,y-1,x+W,y+H);
end
else
begin // поставить в клетку флаг
nFlag := nFlag + 1;
if Pole[row,col] = 9
then nMin := nMin + 1;
Pole[row,col]:=Pole[row,col]+200; // поставили флаг
if (nMin = NM) and (nFlag = NM) then
begin
status := 2; // игра закончена
ShowPole(Form1.Canvas, status);
end
else Kletka(Form1.Canvas, row, col, status);
end;
end;
end;

```

Справочная информация

При выборе из меню ? команды **Справка** появляется справочная информация — правила игры (рис. 15.14).

Процесс создания СНМ-файла подробно описан в 14-й главе. Процедура, обеспечивающая вывод справочной информации, приведена в листинге 15.7.

Примечание

Перед непосредственным созданием процедуры, обеспечивающей вывод справочной информации, в главную форму необходимо добавить компонент HhOpen.

Листинг 15.7. Вывод справочной информации

```

// выбор из меню ? команды Справка
procedure TForm1.N3Click(Sender: TObject);

var
  HelpFile : string;           // файл справки
  HelpTopic : string;         // раздел справки

```

```

pwHelpFile : PWideChar; // файл справки (указатель на строку WideChar)
pwHelpTopic : PWideChar; // раздел (указатель на строку WideChar)
begin
  HelpFile := 'saper.chm';
  HelpTopic := 'saper_02.htm';

  // выделить память для строк WideChar
  GetMem(pwHelpFile, Length(HelpFile) * 2);
  GetMem(pwHelpTopic, Length(HelpTopic)*2);

  // преобразовать строку Ansi в строку WideString
  pwHelpFile := StringToWideChar(HelpFile,pwHelpFile,MAX_PATH*2);
  pwHelpTopic := StringToWideChar(HelpTopic,pwHelpTopic,32);

  // вывести справочную информацию
  Form1.Nhopen1.OpenHelp(pwHelpFile,pwHelpTopic);
end;
```

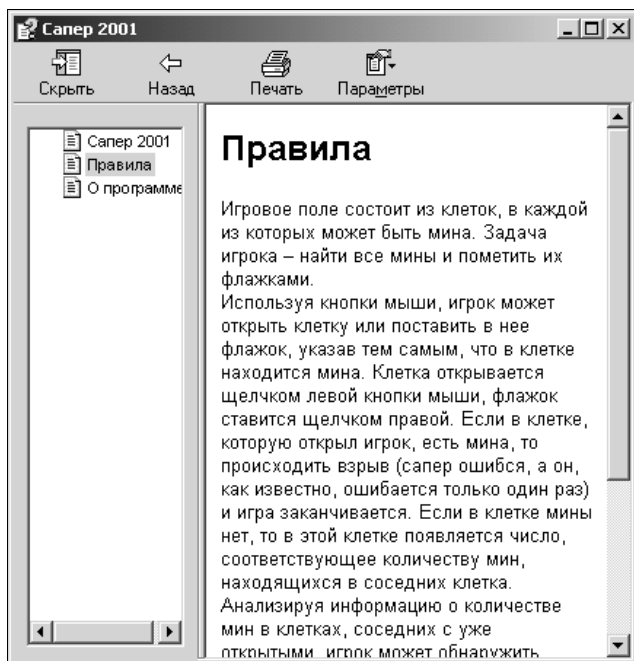


Рис. 15.14. Окно справочной информации

Информация о программе

При выборе из меню ? команды **О программе** на экране должно появиться одноименное окно (рис. 15.15).

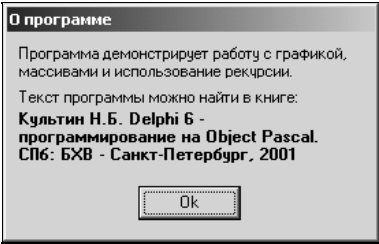


Рис. 15.15. Окно **О программе**

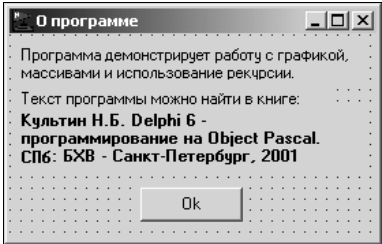


Рис. 15.16. Форма **О программе**

Чтобы программа во время своей работы могла вывести на экран окно, отличное от главного (стартового), нужно создать это окно. Делается это выбором из меню **File** команды **New form**. В результате выполнения команды **New form** в проект добавляется новая форма и соответствующий ей модуль. Вид формы `AboutForm` после добавления необходимых компонентов приведен на рис. 15.16, значения ее свойств — в табл. 15.8.

Таблица 15.8

Свойство	Значение
Name	AboutForm
Caption	О программе
BorderStyle	bsSingle
BorderIcons.biSystemMenu	False
BorderIcons.biMinimize	False
BorderIcons.biMaximize	False

Событие, которое происходит в результате выбора из меню ? команды **О программе**, обрабатывается процедурой `Click`, которая выводит окно **О программе**. Непосредственно вывод окна выполняет метод `ShowModal`, который выводит окно как *модальный* диалог.

Листинг 15.8. Вывод окна **О программе**

```
// выбор из меню ? команды О программе
procedure TForm1.N4Click(Sender: TObject);
begin
    AboutForm.Top := Trunc(Form1.Top + Form1.Height/2
                          - AboutForm.Height/2);
    AboutForm.Left := Trunc(Form1.Left +Form1.Width/2
                          - AboutForm.Width/2);
    AboutForm.ShowModal;
end;
```

Примечание

Модальный диалог перехватывает все события, адресованные другим окнам приложения. Пока модальный диалог находится на экране, другие окна приложения не реагируют на действия пользователя. Для продолжения работы с приложением нужно закрыть модальный диалог. В большинстве программ, в том числе и в Delphi, информация о программе реализована как модальный диалог.

Если не предпринимать никаких усилий, то окно **О программе** появится в той точке экрана, в которой находилась форма во время ее разработки. Вместе с тем, можно "привязать" это окно к главному окну программы так, чтобы оно появлялось в центре главного окна. Привязка осуществляется на основании информации о текущем положении главного окна программы (свойства `Top` и `Left`) и размере окна **О программе**. Окно **О программе** должно быть удалено с экрана в результате нажатия кнопки **ОК**. Процедура обработки этого события приведена ниже.

```
procedure TAboutForm.Button1Click(Sender: TObject);
begin
    ModalResult := mrOk;
end;
```

Листинги

Полный текст программы **Сапер 2001** представлен ниже. В листинге 15.9 приведен модуль, соответствующий главной форме, в листинге 15.10 — форме **О программе**.

Листинг 15.9. Модуль главного окна программы Сапер 2001

```
unit saper_1;
interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, Menus, StdCtrls, OleCtrls, HHOPENLib_TLB;

type
    TForm1 = class(TForm)
        MainMenu1: TMainMenu;
        N1: TMenuItem;
        N2: TMenuItem;
        N3: TMenuItem;
        N4: TMenuItem;
        Hhopen1: THhopen;
```

```

procedure Form1Create(Sender: TObject);
procedure Form1Paint(Sender: TObject);
procedure Form1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
procedure N1Click(Sender: TObject);

procedure N4Click(Sender: TObject);
procedure N3Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

uses saper_2;

{$R *.DFM}

const
    MR = 10; // кол-во клеток по вертикали
    MC = 10; // кол-во клеток по горизонтали
    NM = 10; // кол-во мин

    W    = 40; // ширина клетки поля
    H    = 40; // высота клетки поля

var
    Pole: array[0..MR+1, 0.. MC+1] of integer; // минное поле
    // значение элемента массива:
    // 0..8 — количество мин в соседних клетках
    // 9 — в клетке мина
    // 100..109 — клетка открыта
    // 200..209 — в клетку поставлен флаг

    nMin  : integer; // кол-во найденных мин
    nFlag : integer; // кол-во поставленных флагов

    status : integer; // 0 — начало игры; 1- игра; 2 -результат

Procedure NewGame(); forward; // генерирует новое поле
Procedure ShowPole(Canvas : TCanvas; status : integer);
    forward; // Показывает поле

```

```

Procedure Kletka(Canvas : TCanvas; row, col, status : integer);
    forward; // выводит содержимое клетки

Procedure Open( row, col : integer); forward;// открывает текущую
    // и все соседние клетки, в которых нет мин

Procedure Mina(Canvas : TCanvas; x, y : integer); forward; // Рисует мину
Procedure Flag( Canvas : TCanvas; x, y : integer); forward;// Рисует флаг

// выводит на экран содержимое клетки

Procedure Kletka(Canvas : TCanvas; row, col, status : integer);
    var
        x,y : integer; // координаты области вывода
    begin
        x := (col-1)* W + 1;
        y := (row-1)* H + 1;
        if status = 0 then
            begin
                Canvas.Brush.Color := clLtGray;
                Canvas.Rectangle(x-1,y-1,x+W,y+H);
                exit;
            end;
        if Pole[row,col] < 100 then
            begin
                Canvas.Brush.Color := clLtGray; // не открытые - серые
                Canvas.Rectangle(x-1,y-1,x+W,y+H);
                // если игра завершена (status = 2), то показать мины
                if (status = 2) and (Pole[row,col] = 9)
                    then Mina(Canvas, x, y);
                exit;
            end;

        // открываем клетку
        Canvas.Brush.Color := clWhite; // открытые белые
        Canvas.Rectangle(x-1,y-1,x+W,y+H);
        if ( Pole[row,col] = 100)
            then exit; // клетка открыта, но она пустая

        if ( Pole[row,col] >= 101) and (Pole[row,col] <= 108) then
            begin
                Canvas.Font.Size := 14;
                Canvas.Font.Color := clBlue;
                Canvas.TextOut(x+3,y+2,IntToStr(Pole[row,col] -100));
                exit;
            end;

        if ( Pole[row,col] >= 200) then
            Flag(Canvas, x, y);

```

```
if (Pole[row,col] = 109) then // на этой мине подорвались!
begin
    Canvas.Brush.Color := clRed;
    Canvas.Rectangle(x-1,y-1,x+W,y+H);
end;

if ( (Pole[row,col] mod 10) = 9) and (status = 2) then
    Mina(Canvas, x, y);

end;

// Показывает поле
Procedure ShowPole(Canvas : TCanvas; status : integer);
var
    row,col : integer;
begin
    for row := 1 to MR do
        for col := 1 to MC do
            Kletka(Canvas, row, col, status);
        end;
    end;
end;

// рекурсивная функция открывает текущую и все соседние
// клетки, в которых нет мин
Procedure Open( row, col : integer);
begin
    if Pole[row,col] = 0 then
        begin
            Pole[row,col] := 100;
            Kletka(Form1.Canvas, row,col, 1);
            Open(row,col-1);
            Open(row-1,col);
            Open(row,col+1);
            Open(row+1,col);
            //примыкающие диагонально
            Open(row-1,col-1);
            Open(row-1,col+1);
            Open(row+1,col-1);
            Open(row+1,col+1);
        end
    else
        if (Pole[row,col] < 100) and ( Pole[row,col] <> -3) then
            begin
                Pole[row,col] := Pole[row,col] + 100;
                Kletka(Form1.Canvas, row, col, 1);
            end;
        end;
    end;
end;
```


// новая игра — генерирует новое поле

```

procedure NewGame();
var
    row,col : integer; // координаты клетки
    n : integer;       // количество поставленных мин
    k : integer;       // кол-во мин в соседних клетках
begin
    // Очистим эл-ты массива, соответствующие клеткам
    // игрового поля.
    for row :=1 to MR do
        for col :=1 to MC do
            Pole[row,col] := 0;
    // расставим мины
    Randomize(); // инициализация ГСЧ
    n := 0; // кол-во мин
    repeat
        row := Random(MR) + 1;
        col := Random(MC) + 1;
        if ( Pole[row,col] <> 9) then
            begin
                Pole[row,col] := 9;
                n := n+1;
            end;
    until ( n = NM);

    // для каждой клетки вычислим
    // кол-во мин в соседних клетках
    for row := 1 to MR do
        for col := 1 to MC do
            if ( Pole[row,col] <> 9) then
                begin
                    k :=0;
                    if Pole[row-1,col-1] = 9 then k := k + 1;
                    if Pole[row-1,col]   = 9 then k := k + 1;
                    if Pole[row-1,col+1] = 9 then k := k + 1;
                    if Pole[row,col-1]   = 9 then k := k + 1;
                    if Pole[row,col+1]   = 9 then k := k + 1;
                    if Pole[row+1,col-1] = 9 then k := k + 1;
                    if Pole[row+1,col]   = 9 then k := k + 1;
                    if Pole[row+1,col+1] = 9 then k := k + 1;
                    Pole[row,col] := k;
                end;
    status := 0; // начало игры
    nMin   := 0; // нет обнаруженных мин
    nFlag  := 0; // нет флагов
end;

```

```
// Рисует мину
```

```
Procedure Mina(Canvas : TCanvas; x, y : integer);
```

```
begin
```

```
    with Canvas do
```

```
        begin
```

```
            Brush.Color := clGreen;
```

```
            Pen.Color := clBlack;
```

```
            Rectangle(x+16,y+26,x+24,y+30);
```

```
            Rectangle(x+8,y+30,x+16,y+34);
```

```
            Rectangle(x+24,y+30,x+32,y+34);
```

```
            Pie(x+6,y+28,x+34,y+44,x+34,y+36,x+6,y+36);
```

```
            MoveTo(x+12,y+32); LineTo(x+26,y+32);
```

```
            MoveTo(x+8,y+36); LineTo(x+32,y+36);
```

```
            MoveTo(x+20,y+22); LineTo(x+20,y+26);
```

```
            MoveTo(x+8,y+30); LineTo(x+6,y+28);
```

```
            MoveTo(x+32,y+30); LineTo(x+34,y+28);
```

```
        end;
```

```
end;
```

```
// Рисует флаг
```

```
Procedure Flag(Canvas : TCanvas; x, y : integer);
```

```
    var
```

```
        p : array [0..3] of TPoint; // координаты точек флага
```

```
        m : array [0..4] of TPoint; // буква М
```

```
    begin
```

```
        // зададим координаты точек флага
```

```
        p[0].x:=x+4; p[0].y:=y+4;
```

```
        p[1].x:=x+30; p[1].y:=y+12;
```

```
        p[2].x:=x+4; p[2].y:=y+20;
```

```
        p[3].x:=x+4; p[3].y:=y+36; // нижняя точка древка
```

```
        m[0].x:=x+8; m[0].y:=y+14;
```

```
        m[1].x:=x+8; m[1].y:=y+8;
```

```
        m[2].x:=x+10; m[2].y:=y+10;
```

```
        m[3].x:=x+12; m[3].y:=y+8;
```

```
        m[4].x:=x+12; m[4].y:=y+14;
```

```
    with Canvas do
```

```
        begin
```

```
            // установим цвет кисти и карандаша
```

```
            Brush.Color := clRed;
```

```
            Pen.Color := clRed;
```

```
            Polygon(p); // флажок
```

```

    // древко
    Pen.Color := clBlack;
    MoveTo(p[0].x, p[0].y);
    LineTo(p[3].x, p[3].y);

```

```

    // буква М
    Pen.Color := clWhite;
    Polyline(m);
    Pen.Color := clBlack;

```

```

end;

```

```

end;

```

```

// выбор из меню ? команды О программе

```

```

procedure TForm1.N4Click(Sender: TObject);

```

```

    begin

```

```

        AboutForm.Top := Trunc(Form1.Top + Form1.Height/2
                                - AboutForm.Height/2);

```

```

        AboutForm.Left := Trunc(Form1.Left + Form1.Width/2
                                - AboutForm.Width/2);

```

```

        AboutForm.ShowModal;

```

```

    end;

```

```

procedure TForm1.Form1Create(Sender: TObject);

```

```

var

```

```

    row,col : integer;

```

```

begin

```

```

    // В неотображаемые эл-ты массива, которые соответствуют

```

```

    // клеткам по границе игрового поля, запишем число -3.

```

```

    // Это значение используется функцией Open для завершения

```

```

    // рекурсивного процесса открытия соседних пустых клеток.

```

```

    for row :=0 to MR+1 do

```

```

        for col :=0 to MC+1 do

```

```

            Pole[row,col] := -3;

```

```

    NewGame(); // "разбросать" мины

```

```

    Form1.ClientHeight := H*MR + 1;

```

```

    Form1.ClientWidth := W*MC + 1;

```

```

end;

```

```

// нажатие кнопки мыши на игровом поле

```

```

procedure TForm1.Form1MouseDown(Sender: TObject; Button: TMouseButton;
                                Shift: TShiftState; X, Y: Integer);

```

```

var

```

```

    row, col : integer;

```

```

begin

```

```
if status = 2 // игра завершена
    then exit;

if status = 0 then // первый щелчок
    status := 1;

// преобразуем координаты мыши в индексы
// клетки поля
row := Trunc(y/H) + 1;
col := Trunc(x/W) + 1;

if Button = mbLeft then
    begin
        if Pole[row,col] = 9 then
            begin // открыта клетка, в которой есть мина
                Pole[row,col] := Pole[row,col] + 100;
                status := 2; // игра закончена
                ShowPole(Form1.Canvas, status);
            end
        else if Pole[row,col] < 9 then
            Open(row,col);
        end
    else
        if Button = mbRight then
            if Pole[row,col] > 200 then
                begin
                    // уберем флаг и закроем клетку
                    nFlag := nFlag - 1;
                    Pole[row,col] := Pole[row,col] - 200; // уберем флаг
                    x := (col-1)* W + 1;
                    y := (row-1)* H + 1;
                    Canvas.Brush.Color := clLtGray;
                    Canvas.Rectangle(x-1,y-1,x+W,y+H);
                end
            else
                begin // поставить в клетку флаг
                    nFlag := nFlag + 1;
                    if Pole[row,col] = 9
                        then nMin := nMin + 1;
                    Pole[row,col] := Pole[row,col] + 200; // поставили флаг
                    if (nMin = NM) and (nFlag = NM) then
                        begin
                            status := 2; // игра закончена
                            ShowPole(Form1.Canvas, status);
                        end
                    end
                end
            end
        end
    end
```

```

        else Kletka(Form1.Canvas, row, col, status);
    end;

end;

// Выбор меню Новая игра
procedure TForm1.N1Click(Sender: TObject);
begin
    NewGame();
    ShowPole(Form1.Canvas, status);
end;

// выбор из меню ? команды Справка
procedure TForm1.N3Click(Sender: TObject);
var
    HelpFile : string;           // файл справки
    HelpTopic : string;         // раздел справки
    pwHelpFile : PWideChar;     // файл справки (указатель на строку WideChar)
    pwHelpTopic : PWideChar;    // раздел (указатель на строку WideChar )
begin
    HelpFile := 'saper.chm';
    HelpTopic := 'saper_02.htm';
    // выделить память для строк WideChar
    GetMem(pwHelpFile, Length(HelpFile) * 2);
    GetMem(pwHelpTopic, Length(HelpTopic)*2);
    // преобразовать строку Ansi в строку WideString
    pwHelpFile := StringToWideChar(HelpFile, pwHelpFile, MAX_PATH*2);
    pwHelpTopic := StringToWideChar(HelpTopic, pwHelpTopic, 32);
    // вывести справочную информацию
    Form1.Hopen1.OpenHelp(pwHelpFile, pwHelpTopic);
end;

procedure TForm1.Form1Paint(Sender: TObject);
begin
    ShowPole(Form1.Canvas, status);
end;

end.

```

Листинг 15.10. Модуль окна О программе

```

unit saper_2;

interface

uses

```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls,  
saper_1;
```

type

```
TAboutForm = class(TForm)  
    Button1: TButton;  
    Label1: TLabel;  
    Label2: TLabel;  
    Label3: TLabel;  
    procedure Button1Click(Sender: TObject);  
private  
    { Private declarations }  
public  
    { Public declarations }  
end;
```

var

```
AboutForm: TAboutForm;
```

implementation

```
{ $R *.DFM }
```

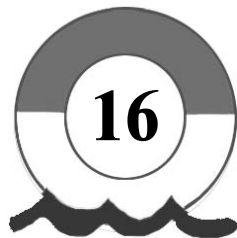
```
procedure TAboutForm.Button1Click(Sender: TObject);
```

begin

```
    ModalResult := mrOk;
```

```
end;
```

```
end.
```



Компонент программиста

Delphi предоставляет возможность программисту создать свой собственный компонент, поместить его на одну из вкладок палитры компонентов и использовать при разработке приложений точно так же, как и другие компоненты Delphi.

Процесс создания компонента может быть представлен как последовательность следующих этапов:

- ☐ Выбор базового класса
- ☐ Создание модуля компонента
- ☐ Тестирование компонента
- ☐ Добавление компонента в пакет компонентов

Рассмотрим процесс создания компонента программиста на примере разработки компонента с именем `NkEdit`, предназначенного для ввода и редактирования дробного числа.

Примечание

Перед началом работы по созданию компонента следует создать новый каталог для модуля компонента.

Выбор базового класса

Приступая к разработке нового компонента, следует четко сформулировать назначение компонента. Затем необходимо определить, какой из компонентов Delphi наиболее близок к разрабатываемому компоненту по назначению, виду и функциональным возможностям. Именно этот существующий компонент Delphi нужно выбрать в качестве базового.

Создание модуля компонента

Для того чтобы создать модуль компонента, необходимо из меню **Component** выбрать команду **New Component** и в поля открывшегося диалогового окна **New Component** (рис. 16.1) ввести информацию о создаваемом компоненте.

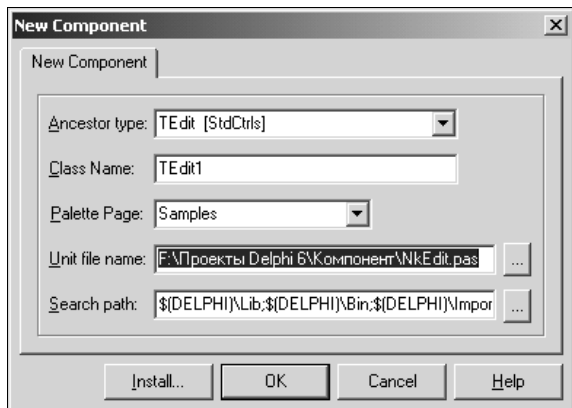


Рис. 16.1. Диалоговое окно **New Component**

Поле **Ancestor type** должно содержать базовый тип для создаваемого компонента, который можно задать непосредственным вводом имени типа или выбором из раскрывающегося списка. Для разрабатываемого компонента базовым компонентом является компонент `Edit` (поле редактирования). Поэтому базовым типом для типа разрабатываемого компонента является тип `TEdit`.

В поле **Class Name** необходимо ввести имя класса разрабатываемого компонента, например, `TNkEdit`.

В поле **Palette Page** нужно ввести имя вкладки палитры компонентов, на которую после создания компонента будет добавлен его значок. Название вкладки палитры компонентов можно выбрать из раскрывающегося списка. Если в поле **Palette Page** ввести имя еще несуществующей вкладки палитры компонентов, то непосредственно перед добавлением компонента вкладка с указанным именем будет создана.

В поле **Unit file name** находится автоматически сформированное имя файла модуля создаваемого компонента. Delphi присваивает модулю компонента имя, совпадающее с именем типа компонента, но без буквы `T`. Нажав кнопку с тремя точками, можно выбрать каталог, в котором должен быть сохранен модуль компонента.

После нажатия кнопки **ОК** к текущему проекту добавляется сформированный Delphi-модуль, представляющий собой заготовку (шаблон) модуля компонента. Текст этого модуля приведен в листинге 16.1.

Листинг 16.1. Шаблон модуля компонента

```
unit NkEdit;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, StdCtrls;

type
  TEdit1 = class(TEdit)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TNkEdit]);
end;

end.
```

В объявлении нового класса указан только тип родительского класса. В раздел реализации помещена процедура `Register`, которая используется Delphi для регистрации нового класса во время установки созданного программистом компонента на указанную вкладку палитры компонентов.

В сформированное Delphi объявление класса нового компонента нужно внести дополнения: объявить свойство, поле данных этого свойства, функцию доступа к полю данных, процедуру установки значения поля данных, конструктор и деструктор. Если на некоторые события компонент должен реагировать не так, как базовый, то в объявление класса нужно поместить описание соответствующих процедур обработки событий.

В листинге 16.2 приведен текст модуля `NkEdit` после внесения всех необходимых изменений.

Листинг 16.2. Модуль компонента NkEdit

```
unit NkEdit;

interface

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

type
  TNkEdit = class(TEdit)
  private
    FNumb: single; // число, находящееся в поле редактирования
    // Это описание функции доступа
    // и процедуры установки поля FNumb
    function GetNumb: single;
    procedure SetNumb(value:single);
  protected

    procedure KeyPress(var Key: Char); override;
  public

  published
    constructor Create(AOwner:TComponent); override;
    property Numb : single // СВОЙСТВО КОМПОНЕНТА
      read GetNumb
      write SetNumb;
  end;

procedure Register;

implementation

// процедура регистрации компонента
procedure Register;
begin
  RegisterComponents('Samples',[TNkEdit]);
end;

// конструктор компонента
constructor TNkEdit.Create(AOwner:TComponent);
begin
  // don't forget to call the ancestors' constructor
  inherited Create(AOwner);
end;
```

```

// функция доступа к полю FNumb
function TNkEdit.GetNumb:single;
begin
    try // поле Text может быть пустым
        Result:=StrToFloat(text);
    except
        on EConvertError do
            begin
                Result:=0;
                text:='';
            end;
        end;
    end;
end;

// процедура записи в поле FNumb
procedure TNkEdit.SetNumb(Value:single);
begin
    FNumb:=Value;
    Text:=FloatToStr(value);
end;

// процедура обработки события KeyPress
procedure TNkEdit.KeyPress(var key:char);
begin
    case key of
        '0'.. '9', #8, #13:;
        '-': if Length(text)<>0 then key:=#0;
        else
            if not ((key = DecimalSeparator) and
                (Pos(DecimalSeparator,text)=0))
            then key:= #0;
    end;

    inherited KeyPress(key); // вызов процедуры обработки события
                             // OnKeyPress родительского класса
end;

end.

```

В описание класса `TNkEdit` добавлено объявление свойства `Numb`, которое представляет собой число, находящееся в поле редактирования. Для хранения значения свойства `Numb` используется поле `FNumb`. Функция `GetNumb` необходима для доступа к полю `FNumb`, а процедура `SetNumb` — для установки значения свойства.

Конструктор класса `TNkEdit` сначала вызывает конструктор родительского класса (`TEdit`), присваивает значение свойству `Text`, затем устанавливает значение свойства `Numb`.

Реакция компонента `NkEdit` на нажатие клавиши клавиатуры определяется процедурой обработки события `TNkEdit.KeyPress`, которая замещает соответствующую процедуру базового класса. В качестве параметра процедура `TNkEdit.KeyPress` получает код нажатой клавиши. Перед вызовом процедуры обработки события `OnKeyPress` родительского класса код нажатой клавиши проверяется на допустимость. Если нажата недопустимая для компонента `NkEdit` клавиша, то код символа заменяется на ноль. Допустимыми для компонента `NkEdit` являются цифровые клавиши, разделитель целой и дробной частей числа (в зависимости от настройки Windows: точка или запятая), "минус", клавиша `<Back Space>` (забой, позволяет удалить ошибочно введенный символ) и `<Enter>`.

Примечание

Следует помнить, что в тексте программы дробная часть числовой константы отделяется от целой части точкой. Во время работы программы, при вводе исходных данных пользователь должен использовать тот символ, который задан в настройке Windows. В качестве разделителя обычно используют запятую (стандартная настройка для России) или точку. Приведенная процедура обработки события `OnKeyPress` учитывает, что настройка Windows может меняться, и поэтому введенный пользователем символ сравнивается не с константой, а со значением глобальной переменной `DecimalSeparator`, которая содержит символ-разделитель, используемый в Windows в данный момент.

Тестирование модуля компонента

Перед добавлением нового компонента в палитру компонентов необходимо всесторонне его проверить. Для этого надо создать приложение, использующее компонент и убедиться, что компонент работает без ошибок.

Во время создания формы приложения нельзя добавить в форму компонент, значка которого нет в палитре компонентов. Однако такой компонент может быть добавлен в форму динамически, т. е. во время работы приложения.

Тестовое приложение создается обычным образом: сначала создается форма приложения, а затем — его модуль.

Вид формы приложения тестирования компонента `NkEdit` приведен на рис. 16.2.

Форма (рис. 16.2) содержит две метки и командную кнопку. Первая метка предназначена для вывода информационного сообщения, вторая метка используется для вывода числа, введенного в поле редактирования. Самого поля редактирования компонента `NkEdit` в форме нет. Этот компонент будет

создан динамически во время работы программы, но для него оставлено место над полем метки.

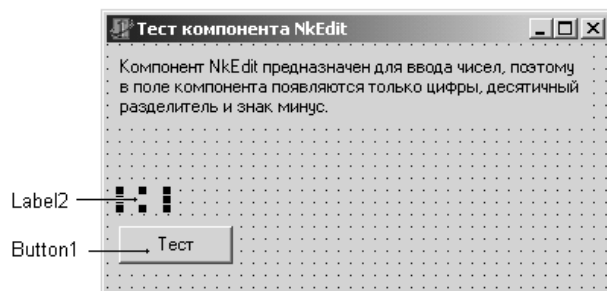


Рис. 16.2. Форма приложения **Тест компонента NkEdit**

После создания формы в модуль приложения, автоматически сформированный Delphi, необходимо внести следующие дополнения:

1. В список используемых модулей (раздел `uses`) добавить имя модуля тестируемого компонента.
2. В раздел объявления приложения добавить инструкцию объявления компонента. Здесь следует помнить, что компонент является объектом, поэтому объявление компонента в разделе переменных не обеспечивает создание компонента, а только создает указатель на компонент, поэтому необходима инструкция вызова конструктора объекта, которая действительно создает компонент (объект).
3. Для формы приложения создать процедуру обработки события `OnCreate`, которая вызовом конструктора тестируемого компонента создаст компонент и установит значения его свойств.

В листинге 16.3 приведен модуль приложения тестирования компонента `NkEdit`.

Листинг 16.3. Тест компонента `NkEdit`

```
unit tstNkEdit_;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,
  NkEdit; // ссылка на модуль компонента

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
```

```

    Button1: TButton;
procedure FormCreate(Sender: TObject);
procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;
    myEdit: TnEdit; // компонент NkEdit
implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
    // создадим компонент и поместим его на форму
    myEdit := TNkEdit.Create(self);
    myEdit.Parent := self;
    myEdit.Left := 8;
    myEdit.Top := 64;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    label2.Caption := FloatToStr(myEdit.Numb);
end;

end.

```

Тестируемый компонент создается процедурой обработки события `FormCreate` (Создание формы) посредством вызова конструктора компонента, которому в качестве параметра передается значение `self`, показывающее, что владельцем компонента является форма приложения.

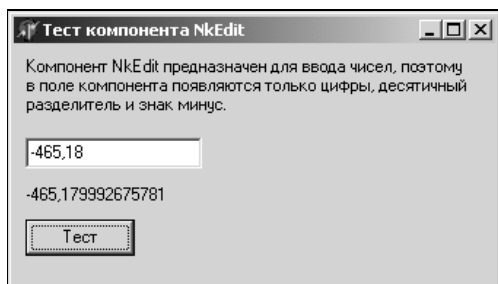


Рис. 16.3. Тестирование компонента

После создания компонента обязательно должен быть выполнен важный шаг: свойству `Parent` должно быть присвоено значение. В данном случае тестируемый компонент находится в форме приложения, поэтому свойству `Parent` присваивается значение `self`.

На рис. 16.3 приведено окно программы **Тест компонента NkEdit** после ввода числа в поле редактирования и нажатия кнопки **Тест**.

Установка компонента

Для того чтобы значок компонента появился в палитре компонентов, компонент должен быть добавлен в один из пакетов (`Packages`) компонентов Delphi. *Пакет компонентов* — это файл с расширением `dpc` (Delphi PacKage file). Например, компоненты, созданные программистом, находятся в пакете `dclusr60.dpc`.

Во время добавления компонента в пакет Delphi использует модуль компонента и *файл ресурсов компонента*, в котором должен находиться битовый образ значка компонента. Имя файла ресурсов компонента должно обязательно совпадать с именем файла модуля компонента. Файл ресурсов имеет расширение `dcr` (Dynamic Component Resource). Имя битового образа, находящегося внутри файла ресурсов, должно совпадать с именем класса компонента.

Ресурсы компонента

Файл ресурсов компонента можно создать при помощи утилиты **Image Editor**, которая запускается выбором из меню **Tools** команды **Image Editor**.

Для того чтобы создать новый файл ресурса компонента, нужно из меню **File** выбрать команду **New** и из появившегося списка выбрать тип создаваемого файла — **Component Resource File** (рис. 16.4).

В результате открывается окно файла ресурсов **Untitled1.dcr**, а в меню диалогового окна **Image Editor** появляется новый пункт — **Resource**. Теперь нужно из меню **Resource** выбрать команду **New/Bitmap** и в открывшемся окне **Bitmap Properties** (рис. 16.5) установить характеристики битового образа значка компонента: **Size** — 24×24 пиксела, **Colors** — 16.

В результате этих действий в создаваемый файл ресурсов компонента будет добавлен новый ресурс — битовый образ с именем **Bitmap1**. Двойной щелчок на имени ресурса (**Bitmap1**) раскрывает окно редактора битового образа, в котором можно нарисовать нужную картинку.

Изображение в окне графического редактора можно увеличить. Для этого необходимо выбрать команду **Zoom In** меню **View**.

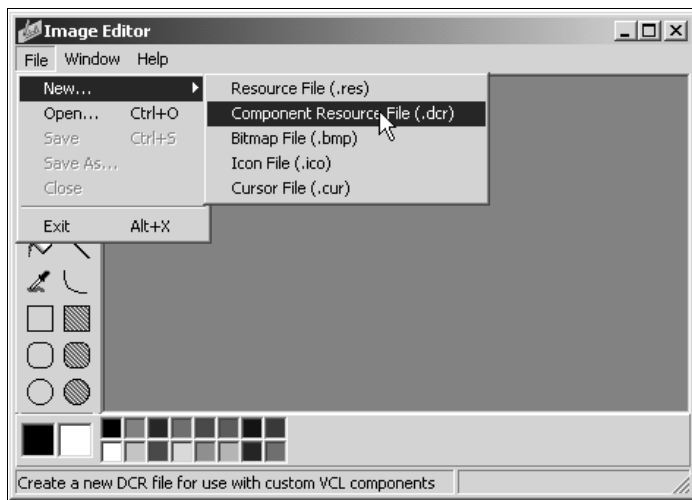


Рис. 16.4. Выбор типа создаваемого файла

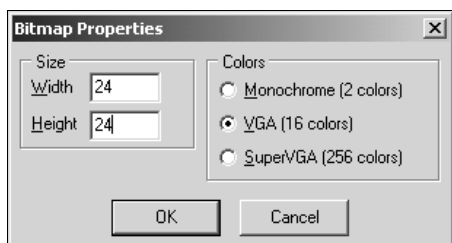


Рис. 16.5. Диалоговое окно Bitmap Properties

Примечание

Цвет правой нижней точки рисунка определяет "прозрачный" цвет. Элементы значка компонента, закрасенные этим цветом, на палитре компонентов Delphi не видны.

Перед тем как сохранить файл ресурсов компонента, битовому образу надо присвоить имя. Имя должно совпадать с именем класса компонента. Чтобы задать имя битового образа, необходимо щелкнуть правой кнопкой мыши на имени битового образа (**Bitmap1**), выбрать в появившемся контекстном меню команду **Rename** и ввести новое имя.

Созданный файл ресурсов компонента нужно сохранить в том же каталоге, в котором находится файл модуля компонента. Для этого надо из меню **File** выбрать команду **Save**.

На рис. 16.6 приведен вид окна **Image Editor**, в левой части которого находится файл ресурсов компонента **TNkEdit (nkedit.dcr)**, а в правой части — окно редактора битового образа, в котором располагается изображение значка для создаваемого компонента.

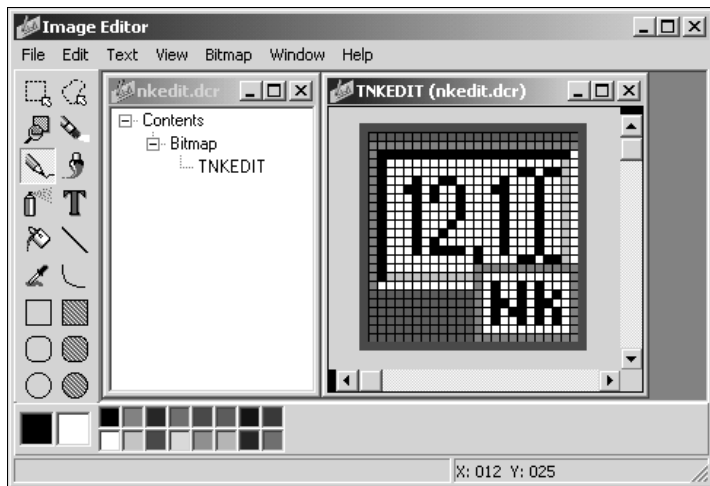


Рис. 16.6. Значок компонента NkEdit

Внимание!

Имя файла ресурсов компонента (NEdit.dcr) должно совпадать с именем модуля компонента (NEdit.pas), а имя битового образа (TNEdit) — с именем класса компонента (TNEdit).

Установка

После создания файла ресурсов компонента, в котором находится битовый образ значка компонента, можно приступить к установке компонента. Для этого надо из меню **Component** выбрать команду **Install Component** и заполнить поля открывшегося окна **Install Component** (рис. 16.7).

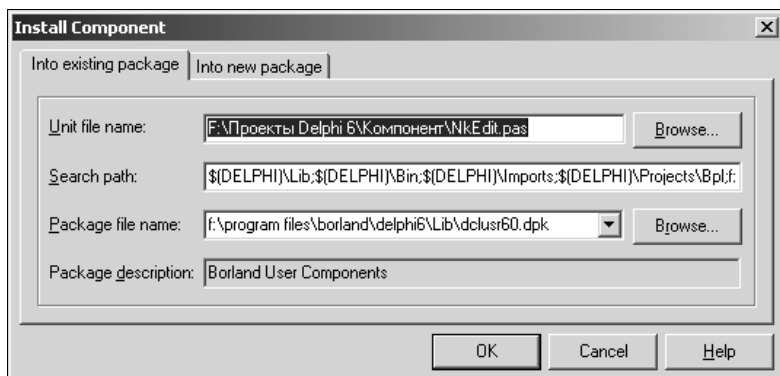


Рис. 16.7. Диалоговое окно Install Component

В поле **Unit file name** (Имя файла модуля) нужно ввести имя файла модуля. Для этого удобно воспользоваться кнопкой **Browse**.

Поле **Search path** (Путь поиска) должно содержать разделенные точкой с запятой имена каталогов, в которых Delphi во время установки компонента будет искать необходимые файлы, в частности файл ресурсов компонента. Если имя файла модуля было введено в поле **Unit file name** выбором файла из списка, полученного при помощи кнопки **Browse**, то Delphi автоматически добавляет в поле **Search path** имена необходимых каталогов.

Примечание

Файл ресурса компонента должен находиться в одном из каталогов, перечисленных в поле **Search path**. Если его там нет, то компоненту будет назначен значок его родительского класса.

Поле **Package file name** должно содержать имя пакета, в который будет установлен компонент. По умолчанию компоненты, создаваемые программистом, добавляются в пакет dclusr60.dpk.

Поле **Package description** содержит название пакета. Для пакета dclusr50.dpk это текст: Borland User's Components.

После заполнения перечисленных полей и нажатия кнопки **OK** начинается процесс установки. Сначала на экране появляется окно **Confirm** (рис. 16.8), в котором Delphi просит подтвердить обновление пакета.

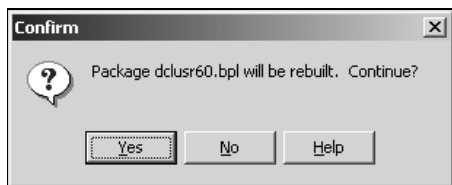


Рис. 16.8. Запрос подтверждения обновления пакета в процессе установки компонента

После выбора кнопки **Yes** процесс установки продолжается. Если он завершается успешно, то на экране появляется информационное сообщение (рис. 16.9) о том, что в результате обновления пакета палитра компонентов обновлена, т. е. в нее добавлен значок компонента, и новый компонент зарегистрирован.

После установки компонента в пакет открывается диалоговое окно **Package** (Редактор пакета компонентов) (рис. 16.10), в котором перечислены компоненты, находящиеся в пакете.

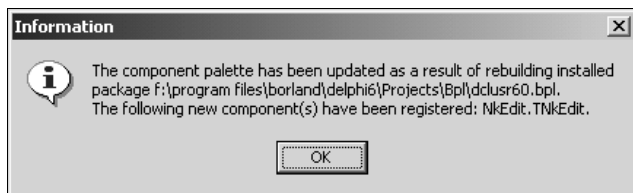


Рис. 16.9. Сообщение об успешной установке компонента

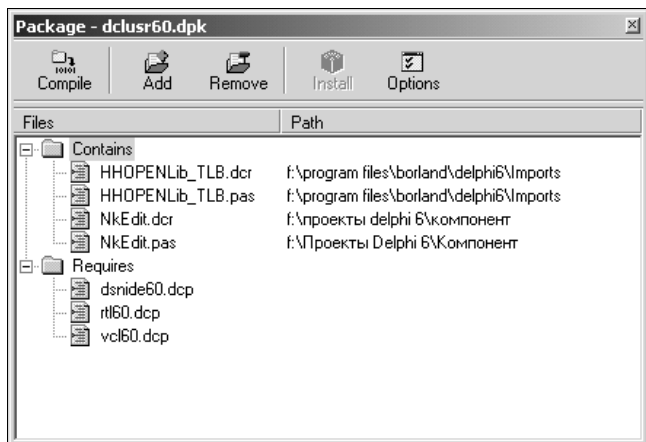


Рис. 16.10. Окно редактора пакета компонентов

На этом процесс установки компонента заканчивается. В результате на вкладке палитры компонентов, имя которой было задано при создании модуля компонента, появляется значок установленного компонента (рис. 16.11).



Рис. 16.11. Вкладка **Samples** после установки компонента NkEdit

Ошибки при установке компонента

Во время работы над новым компонентом наиболее частой ошибкой является попытка установить (переустановить) компонент, который уже находится в одном из пакетов (обычно такое желание возникает после внесения изменений в модуль компонента).

В этом случае Delphi выводит сообщение: `The package already contains unit named...` (Пакет уже содержит модуль, который называется...) и процесс установки завершается. Для того чтобы преодолеть эту ошибочную ситуацию и добавить компонент в нужный пакет или обновить уже установленную версию компонента, необходимо сначала удалить компонент из пакета, а затем установить его снова.

Тестирование компонента

После добавления компонента в пакет, необходимо проверить поведение компонента во время разработки приложения, использующего этот компонент.

Примечание

Поведение компонента во время работы приложения было протестировано ранее, когда компонент добавлялся в форму динамически, т. е. во время работы программы.

Можно считать, что компонент работает правильно, если во время разработки приложения удалось поместить этот компонент в форму разрабатываемого приложения и, используя окно **Object Inspector**, установить значения свойств компонента, причем как новых, так и унаследованных от родительского класса.

Работоспособность компонента `NkEdit` можно проверить, используя его, например, в приложении **Поездка на дачу**, вид формы которого приведен на рис. 16.12.

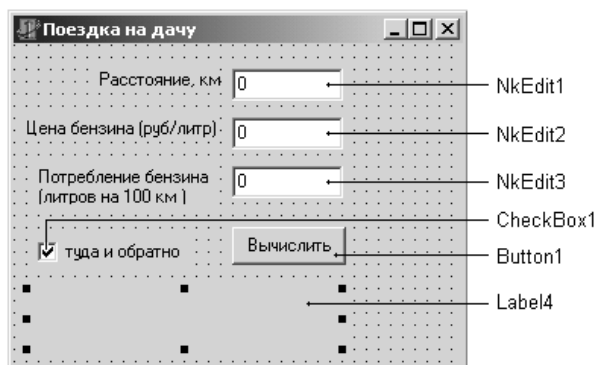


Рис. 16.12. Форма приложения **Поездка на дачу** (тест компонента `NkEdit`)

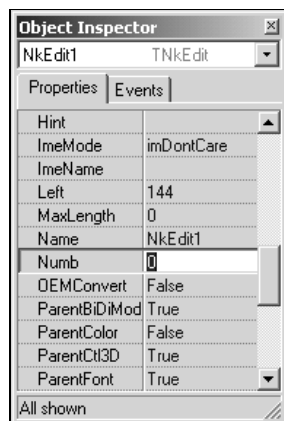


Рис. 16.13. Свойства компонента класса `TNkEdit` в диалоговом окне **Object Inspector**

Внешне форма разрабатываемого приложения почти ничем не отличается от формы приложения **Поездка на дачу**, рассмотренного в главе 6. Однако если выбранным компонентом является поле ввода одного из коэффициентов уравнения, то в заголовке диалогового окна **Object Inspector** указано, что текущим компонентом будет компонент класса `TNkEdit`, а в списке свойств можно увидеть новое (по сравнению со списком свойств стандартного компонента `Edit`) свойство — `Numb` (рис. 16.13).

В листинге 16.4 приведен модуль приложения **Поездка на дачу**. Очевидно, что текст программы значительно уменьшился, по сравнению с первоначальным вариантом, в котором для ввода данных использовался компонент `Edit`.

Листинг 16.4. Поездка на дачу (тест компонента NkEdit)

```

unit fazenda_;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,
  NkEdit; // ссылка на модуль компонента !!!

type
  TForm1 = class(TForm)
    NkEdit1: TNkEdit;      // расстояние
    NkEdit2: TNkEdit;      // цена литра бензина
    NkEdit3: TNkEdit;      // потребление бензина на 100 км
    CheckBox1: TCheckBox;  // True – поезда туда и обратно
    Button1: TButton;      // кнопка Вычислить
    Label4: TLabel;        // поле вывода результата расчета
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;

    procedure Button1Click(Sender: TObject);
    procedure NkEdit1KeyPress(Sender: TObject; var Key: Char);
    procedure NkEdit2KeyPress(Sender: TObject; var Key: Char);
    procedure NkEdit3KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// нажатие клавиши в поле Расстояние
procedure TForm1.NkEdit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = Char(VK_RETURN)
  then NkEdit2.SetFocus; // переместить курсор в поле Цена
end;

```

```

// нажатие клавиши в поле Цена
procedure TForm1.NkEdit2KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = Char(VK_RETURN)
    then NkEdit3.SetFocus; // переместить курсор в поле Потребление
end;

// нажатие клавиши в поле Потребление
procedure TForm1.NkEdit3KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = Char(VK_RETURN)
    then Button1.SetFocus; // // сделать активной кнопку Вычислить
end;

// щелчок на кнопке Вычислить
procedure TForm1.Button1Click(Sender: TObject);
var
    rast : real; // расстояние
    cena : real; // цена
    potr : real; // потребление на 100 км
    summ : real; // сумма
    mes: string;
begin
    rast := StrToFloat(NkEdit1.Text);
    cena := StrToFloat(NkEdit2.Text);
    potr := StrToFloat(NkEdit3.Text);
    summ := rast / 100 * potr * cena;
    if CheckBox1.Checked then
        summ := summ * 2;
    mes := 'Поездка на дачу';
    if CheckBox1.Checked then
        mes := mes + ' и обратно';
    mes := mes + 'обойдется в ' + FloatToStrF(summ, ffGeneral, 4, 2)
        + ' руб.';
    Label4.Caption := mes;
end;

end.

```

Удаление компонента

Иногда возникает необходимость удалить компонент из пакета. Сделать это можно при помощи редактора пакета компонентов.

Для того чтобы запустить редактор пакета компонентов, надо из меню **Component** выбрать команду **Install Packages**, в открывшемся диалоговом ок-

не **Project Options** (рис. 16.14), в списке **Design packages**, выбрать нужный пакет и нажать кнопку **Edit**.

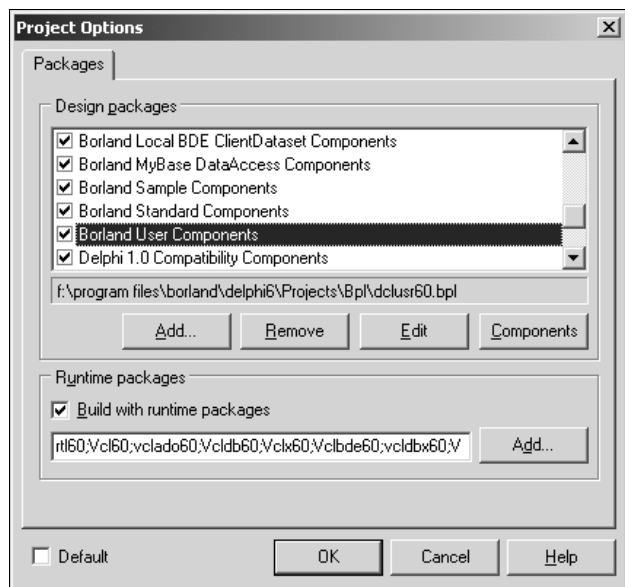


Рис. 16.14. Выбор пакета для редактирования

В открывшемся окне **Confirm** (рис. 16.15) в ответ на запрос: *Cancel this dialog box and open...* (Заккрыть этот диалог и открыть пакет...) надо нажать кнопку **Yes**.

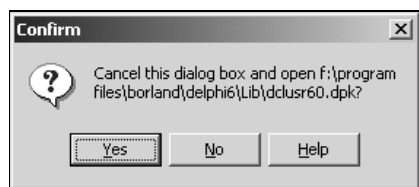


Рис. 16.15. Диалоговое окно **Confirm**

В результате открывается окно редактора пакета **Package** (рис. 16.16), в котором в списке **Contains** (Содержимое) перечислены компоненты пакета.

Для того чтобы удалить компонент из пакета, необходимо нажать кнопку **Remove** (Переместить). В открывшемся диалоговом окне **Remove From Project** (рис. 16.17) следует выбрать удаляемый компонент и нажать кнопку **OK**.

После удаления компонента из пакета нужно обязательно выполнить перекомпиляцию пакета. Для этого необходимо в окне редактора пакета нажать кнопку **Compile**. После перекомпиляции пакета Delphi информирует о том, что удаленный компонент больше не зарегистрирован (рис. 16.18).

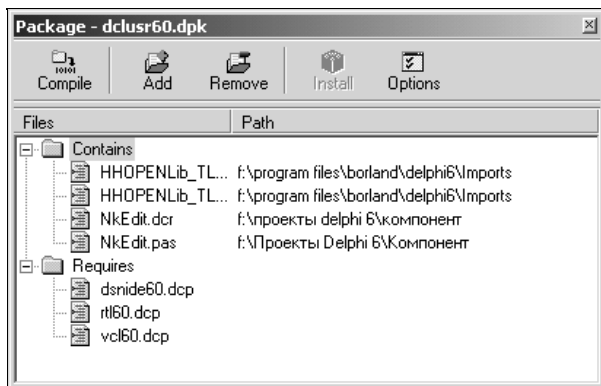


Рис. 16.16. Окно редактора пакета

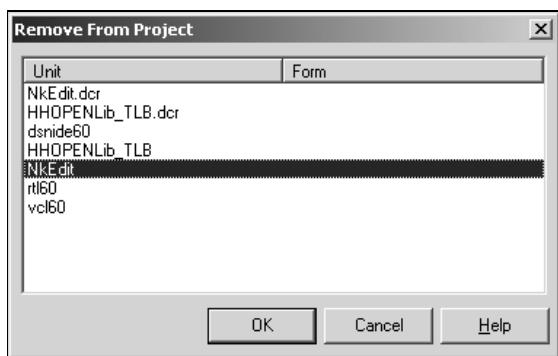


Рис. 16.17. Выбор удаляемого из пакета компонента

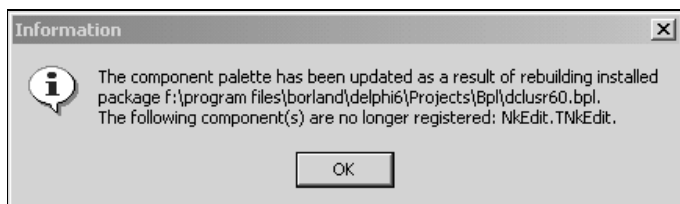


Рис. 16.18. Информационное сообщение о том, что компонент больше недоступен

После перекомпиляции пакета необходимо закрыть окно редактора пакета и в открывшемся окне подтвердить сохранение изменений в пакете, из которого был удален компонент.

Настройка палитры компонентов

Delphi позволяет менять порядок следования вкладок палитры компонентов, названия этих вкладок, а также порядок следования значков компонентов на вкладках. Настройка палитры компонентов выполняется в диалоговом

окне **Palette Properties** (рис. 16.19), которое открывается выбором из меню **Component** команды **Configure Palette**.

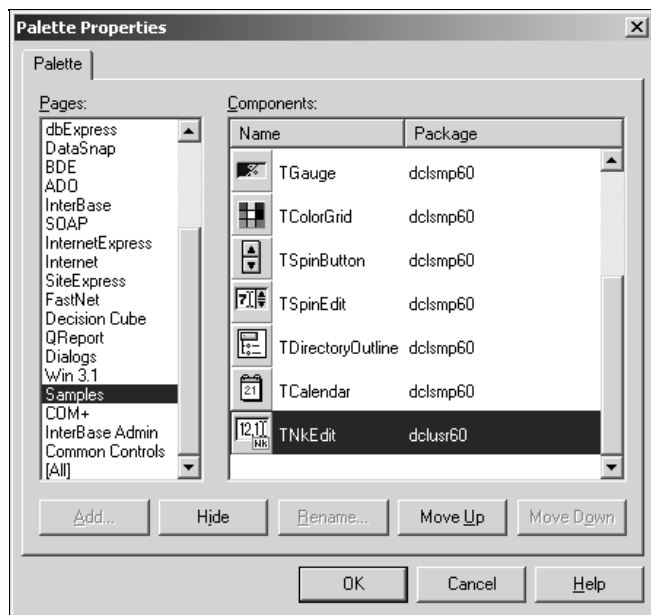


Рис. 16.19. Диалоговое окно **Palette Properties**

Сначала в списке **Pages** необходимо выделить нужную вкладку палитры компонентов. Затем, если нужно изменить порядок следования вкладок палитры компонентов, надо воспользоваться кнопками **Move Up** и **Move Down** и путем перемещения выбранного имени по списку **Pages** добиться нужного порядка следования вкладок.

Если нужно изменить порядок следования значков компонентов на вкладке, то в списке **Components** следует выбрать нужный значок компонента и кнопками **Move Up** и **Move Down** переместить значок на новое место.

При необходимости изменить имя вкладки палитры следует в списке **Pages** выбрать имя нужной вкладки, нажать кнопку **Rename** и в поле **Page Name** открывшегося диалогового окна **Rename Page** (рис. 16.20) ввести новое имя.

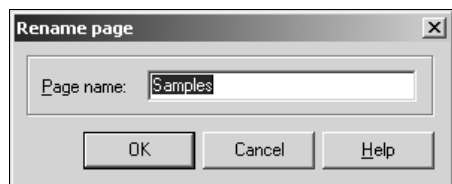


Рис. 16.20. Диалоговое окно **Rename page**



Базы данных

С точки зрения пользователя, *база данных* — это программа, которая обеспечивает работу с информацией. При запуске такой программы на экране, как правило, появляется таблица, просматривая которую пользователь может найти интересующие его сведения. Если система позволяет, то он может внести изменения в базу данных: добавить новую информацию или удалить ненужную.

С точки зрения программиста, *база данных* — это набор файлов, содержащих информацию. Разрабатывая базу данных для пользователя, программист создает программу, которая обеспечивает работу с файлами данных.

В настоящее время существует достаточно большое количество программных систем, позволяющих создавать и использовать локальные (dBASE, FoxPro, Access, Paradox) и удаленные (Interbase, Oracle, Sysbase, Infomix, Microsoft SQL Server) базы данных.

В состав Delphi входят компоненты, позволяющие писать программы работы с файлами данных различных систем: от dBASE до Informix и Oracle. Кроме того, Delphi предоставляет утилиту Borland Database Desktop, позволяющую программисту создавать файлы баз данных в различных форматах.

Классификация баз данных

В зависимости от расположения программы, использующей данные, и самих данных, а также способа разделения данных между несколькими пользователями различают *локальные* и *удаленные* базы данных.

Локальная база данных

Данные локальной базы данных (файлы данных) находятся на одном (локальном) устройстве, в качестве которого может выступать диск компьютера или сетевой диск (диск другого компьютера, работающего в сети).

Для обеспечения разделения данных (доступа к данным) между несколькими пользователями, в качестве которых выступают программы, работающие на одном или нескольких компьютерах, в локальных базах данных используется метод, получивший название *блокировка файлов*. Суть этого метода заключается в том, что пока данные используются одним пользователем, другой пользователь не может применять эти данные, т. е. данные для него закрыты, заблокированы.

Paradox, dBase, FoxPro и Access — это локальные базы данных.

Удаленная база данных

Данные (файлы) удаленной базы данных находятся на удаленном компьютере. (Следует обратить внимание, что каталоги удаленного компьютера не могут рассматриваться как сетевые диски.)

Программа работы с удаленной базой данных состоит из двух частей: клиентской и серверной. *Клиентская* часть программы, работающая на компьютере пользователя, обеспечивает взаимодействие с серверной частью программы; посредством запросов, передаваемых на удаленный компьютер, обеспечивается доступ к данным.

Серверная часть программы, работающая на удаленном компьютере, принимает запросы, выполняет их и пересылает данные клиентской программе. *Запросы* представляют собой команды, представленные на языке SQL (Structured Query Language) — языке структурированных запросов.

Программа, работающая на удаленном сервере, проектируется таким образом, чтобы обеспечить одновременный доступ к информации нескольким пользователям. При этом для обеспечения доступа к данным вместо механизма блокировки файлов используют механизм транзакций.

Транзакция — это некоторая последовательность действий, которая должна быть обязательно выполнена над данными перед тем, как они будут переданы. В случае обнаружения ошибки во время выполнения любого из действий вся последовательность действий, составляющая транзакцию, повторяется снова. Таким образом, механизм транзакций обеспечивает защиту от аппаратных сбоев. Он также обеспечивает возможность многопользовательского доступа к данным.

Примечание

Разработка удаленной базы данных — довольно сложная и трудоемкая задача. Ее решение предполагает наличие у разработчика глубоких знаний и достаточно большого опыта. Поэтому в данной книге задача разработки удаленных баз данных не рассматривается.

Структура базы данных

База данных — это набор однородной, как правило, упорядоченной по некоторому критерию информации. База данных может быть представлена в "бумажном" или в электронном виде.

Типичным примером "бумажной" базы данных является каталог библиотеки — набор бумажных карточек, содержащих информацию о книгах. Информация в этой базе однородная (содержит сведения только о книгах) и упорядоченная (карточки расставлены, например, в соответствии с алфавитным порядком фамилий авторов). Другими примерами "бумажной" базы данных являются телефонный справочник и расписание движения поездов.

Компьютерная база данных представляет собой файл (или набор файлов), содержащий информацию.

База данных состоит из *записей*. Каждая запись содержит информацию об одном экземпляре. Например, каждая запись базы данных "Архитектурные памятники Санкт-Петербурга" хранит информацию только об одном экземпляре — историческом памятнике.

Памятник	Архитектор	Историческая справка
1 Адмиралтейство	А. Д. Захаров	Здание Адмиралтейства таким, как оно выглядит сейчас, стало после перестройки в 1806—1823 годах. Автор проекта — гениальный русский зодчий А. Д. Захаров. Высота шпиля: 72 метра
2 Александровская колонна	Огюст Монферран	Памятник победы России над войсками Наполеона в Отечественной войне 1812 года. Открыта 30 августа 1834 года. Высота: 47,5 метра; вес гранитного ствола: 600 тонн
3 Зимний дворец	Ф. Б. Растрелли	Зимний дворец много раз менял свой облик. Последний раз он перестраивался по проекту Растрелли. Строительство дворца продолжалось более семи лет (1754—1762 годы)
4 Ростральные колонны	Тома де Томон	32-метровые ростральные колонны, органично вошедшие в архитектурный ансамбль Стрелки Васильевского острова, были сооружены в 1810 году. Они напоминают о существовавшем в древнем Риме обычае — украшать триумфальные колонны рострами захваченных кораблей
5 Исаакиевский собор	Огюст Монферран	Исаакиевский собор, четвертый по счету, стали возводить в 1818 году. Строился собор 40 лет и был окончен в 1858 году. Автор проекта — Огюст Монферран

Рис. 17.1. Представление базы данных в виде таблицы

Записи состоят из *полей*. Каждое поле содержит информацию об одной характеристике экземпляра. Например, запись базы данных "Архитектурные памятники Санкт-Петербурга" состоит из следующих полей: "Памятник", "Архитектор" и "Историческая справка", где "Памятник", "Архитектор" и "Историческая справка" — это имена полей. Содержимое этих полей характеризует конкретный памятник.

Примечание

Обратите внимание, что каждая запись состоит из одинаковых полей. Некоторые поля могут быть не заполнены, однако они все равно присутствуют в записи.

На бумаге базу данных удобно представить в виде таблицы (рис. 17.1). Каждая строка таблицы соответствует записи, а ячейка таблицы — полю. При этом заголовок столбца таблицы — это имя поля, а номер строки таблицы — номер записи.

Примечание

Информацию компьютерных баз данных обычно выводят на экран в виде таблиц. Поэтому в литературе довольно часто вместо словосочетания "файл данных" используется словосочетание "таблица данных" или просто "таблица".

Модель базы данных в Delphi

Каждая таблица физически хранится в отдельном файле. Однако отождествлять базу данных и таблицу нельзя, т. к. довольно часто поля одной записи распределены по таблицам и, следовательно, находятся в нескольких файлах.

В простейшем случае источником информации для программы, работающей с базой данных, может быть вся таблица. Однако, как правило, пользователя интересует не вся информация, находящаяся в базе данных, а только какая-то ее часть. Он выбирает и просматривает лишь некоторые, удовлетворяющие его запросу записи. Поэтому в модель базы данных помимо таблицы, представляющей собой всю базу данных, было введено понятие *запроса*, являющегося выборкой, т. е. группой записей базы данных.

Псевдоним базы данных

Разрабатывая программу работы с базой данных, программист не знает, на каком диске и в каком каталоге будут находиться файлы базы данных во время ее использования. Например, пользователь может поместить базу данных в один из каталогов дисков C, D или на сетевой диск. Поэтому возникает проблема передачи в программу информации о размещении файлов базы данных.

В Delphi проблема передачи в программу информации о путях к файлам базы данных решается за счет использования *псевдонима* базы данных. Псевдоним (*Alias*) — это, как правило, короткое имя, поставленное в соответствие полному имени каталога базы данных, т. е. каталога, в котором находятся файлы базы данных. Например, псевдонимом каталога `C:\data\SPetersburg` может быть имя `Peter`.

Примечание

Имя псевдонима должно представлять собой строку не более чем из восьми символов — букв латинского алфавита и цифр.

Для доступа к информации программа, обеспечивающая работу с базой данных, применяет библиотеку Borland Database Engine (BDE), которая, в свою очередь, использует конфигурационный файл, содержащий информацию обо всех зарегистрированных в системе псевдонимах.

Псевдоним базы данных может быть создан (зарегистрирован) при помощи утилиты BDE Administrator. Эта же утилита позволяет изменить каталог, связанный с псевдонимом.

Создание базы данных

База данных — это набор файлов (таблиц), в которых находится информация. Как правило, база данных состоит из нескольких таблиц, которые размещают в одном каталоге. Каталог для новой базы данных создается обычным образом, например, при помощи Проводника, а таблица — посредством входящей в состав Delphi утилиты Borland Database Desktop.

Так как для доступа к файлам (таблицам) базы данных библиотека BDE использует не имя каталога, в котором находятся файлы, а его псевдоним, то перед тем, как приступить к созданию таблиц новой базы данных, необходимо создать псевдоним для этой базы данных.

Таким образом, процесс создания базы данных может быть представлен как последовательность следующих шагов:

- ☐ Создание каталога
- ☐ Создание псевдонима
- ☐ Создание таблиц

Создание каталога

Каталог (папка) для файлов базы данных создается обычным образом, например, при помощи Проводника. Как правило, файлы базы данных помещают в отдельном подкаталоге каталога программы работы с базой данных.

Примечание

Для дальнейшей работы с рассматриваемой в качестве примера базой данных "Архитектурные памятники Санкт-Петербурга" следует в каталоге \Проекты создать каталог \Петербург и в нем — подкаталог \Data.

Создание псевдонима

Псевдоним базы данных создается при помощи входящей в Delphi утилиты BDE Administrator, которая запускается из Windows выбором из меню **Программы/Borland Delphi 5** команды **BDE Administrator**.

Вид диалогового окна **BDE Administrator** после запуска приведен на рис. 17.2.

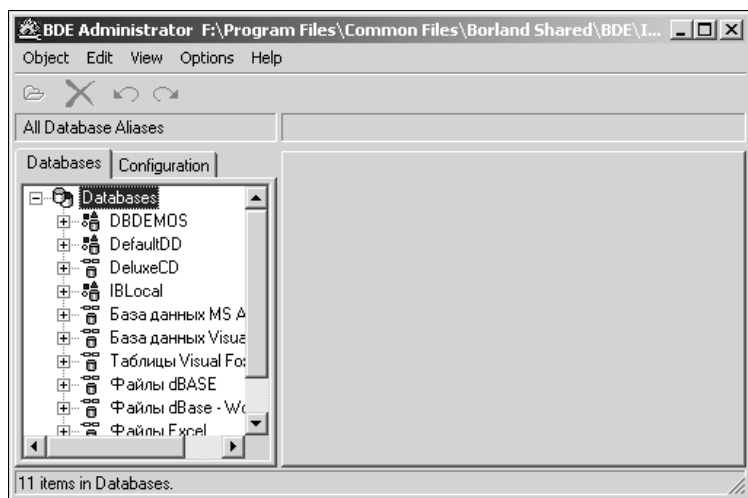


Рис. 17.2. Диалоговое окно **BDE Administrator**

В левой части окна, на вкладке **Databases**, перечислены зарегистрированные на данном компьютере псевдонимы. Для того чтобы создать новый псевдоним, необходимо из меню **Object** выбрать команду **New**. Затем в открывшемся диалоговом окне **New Database Alias** (Новый псевдоним базы данных) из раскрывающегося списка **Database Driver Name**, в котором перечислены зарегистрированные во время установки Delphi драйверы доступа к базам данных, нужно выбрать драйвер доступа к данным создаваемой базы данных (рис. 17.3).

Примечание

При создании псевдонима по умолчанию предлагается драйвер STANDARD (default driver), который обеспечивает доступ к таблицам в формате Paradox.

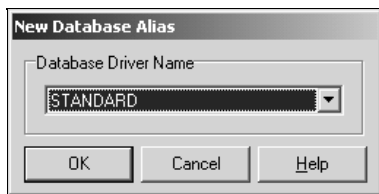


Рис. 17.3. Диалоговое окно **New Database Alias**

После выбора драйвера и нажатия кнопки **ОК** в список псевдонимов будет добавлен новый элемент (рис. 17.4).

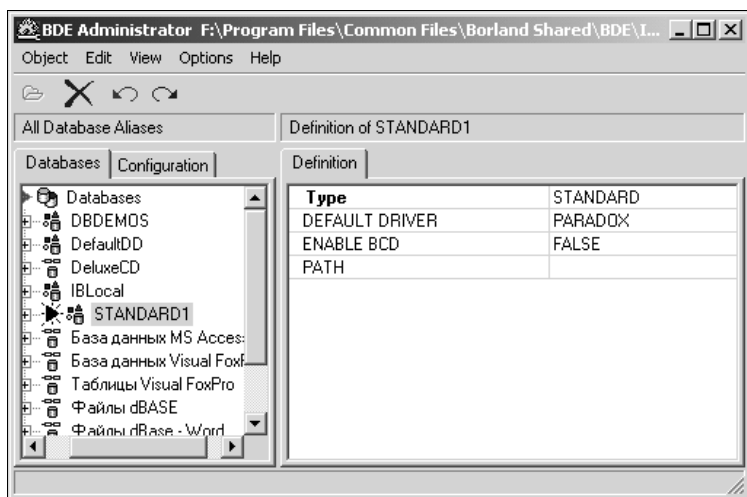


Рис. 17.4. Регистрации нового псевдонима

После этого нужно изменить автоматически созданное администратором имя псевдонима и задать путь к файлам базы данных, для которой создается псевдоним.

Имя псевдонима можно изменить обычным для Windows способом: щелкнуть правой кнопкой мыши на имени псевдонима (на вкладке **Databases**), в появившемся контекстном меню выбрать команду **Rename** (Переименовать) и в открывшемся диалоговом окне ввести новое имя.

Путь к файлам базы данных можно ввести на вкладке **Definition** в поле **Path** с клавиатуры или воспользоваться стандартным диалоговым окном **Select Directory** (Выбор каталога), которое открывается нажатием на кнопке с тремя точками, находящейся в конце поля **Path**.

В качестве примера на рис. 17.5 приведен вид окна администратора после создания псевдонима *Peterburg* для базы данных "Архитектурные памятники Санкт-Петербурга".

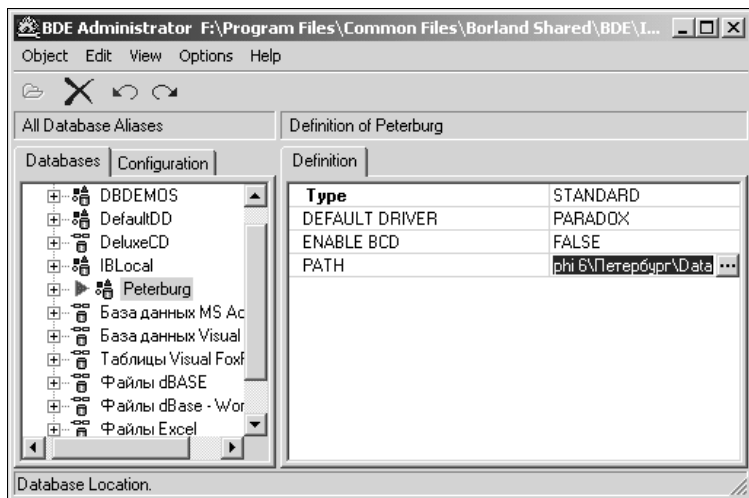


Рис. 17.5. Результат создания псевдонима

Для того чтобы созданный псевдоним был зарегистрирован в файле конфигурации (idapi.cfg), необходимо в меню **Object** выбрать команду **Apply** (Применить). В открывшемся диалоговом окне **Confirm** следует подтвердить необходимость сохранения изменений в файле конфигурации.

Создание таблицы

Важным моментом при создании базы данных является распределение информации между полями записи. Очевидно, что информация может быть распределена между полями различным образом.

Например, сведения об исторических памятниках Санкт-Петербурга могут быть организованы в виде записей, состоящих из полей "Памятник" и "Историческая справка", или из полей "Памятник", "Архитектор", "Год" и "Историческая справка".

В первом варианте поле "Памятник" будет содержать название памятника, например, Эрмитаж, а поле "Историческая справка" — всю остальную информацию. При этом пользователь сможет найти информацию об интересующем его памятнике только по названию. При втором варианте организации записи пользователь сможет найти информацию о памятниках, архитектором которых является конкретный зодчий, или о памятниках, возведенных в конкретный исторический период.

Можно сформулировать следующее правило: если предполагается, что во время использования базы данных будет выполняться выборка информации по некоторому критерию, то информацию, обеспечивающую возможность этой выборки, следует поместить в отдельное поле.

После того как определены поля записи, необходимо выполнить распределение полей по таблицам. В простой базе данных все поля можно разместить в одной таблице. В сложной базе данных поля распределяют по нескольким таблицам, связь между которыми обеспечивается вводом некоторой дополнительной информации, однозначно идентифицирующей каждую запись.

Примечание

Базы данных, состоящие из нескольких, связанных между собой таблиц, называются *реляционными*. В реляционных базах данных, для того чтобы избежать дублирования информации в таблицах, к основной информации добавляется некоторая служебная информация, которая однозначно идентифицирует запись. Подробное рассмотрение организации реляционных баз данных не входит в задачу данной книги. Читатель может самостоятельно ознакомиться с вопросами организации реляционных баз данных, обратившись к литературе.

После того как определена структура записей базы данных, можно приступить непосредственно к созданию таблицы. Таблицы создаются при помощи входящей в состав Delphi утилиты Database Desktop.

Утилита Database Desktop позволяет выполнять все необходимые при работе с базами данных действия. Она обеспечивает создание, просмотр и модификацию таблиц баз данных различных форматов (Paradox, dBASE, Microsoft Access). Кроме того, утилита позволяет выполнять выборку информации путем создания запросов.

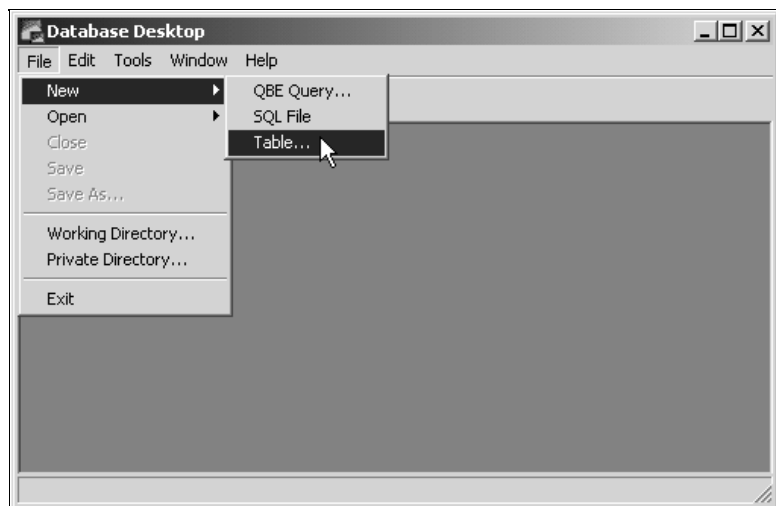


Рис. 17.6. Диалоговое окно Database Desktop

Для того чтобы создать новую таблицу, нужно выбором из меню **Tools** команды **Database Desktop** запустить утилиту Database Desktop. Затем в по-

явившемся окне утилиты Database Desktop надо из меню **File** выбрать команду **New** и в появившемся списке выбрать тип создаваемого файла — **Table** (рис. 17.6). Затем в открывшемся диалоговом окне **Create Table** (рис. 17.7) следует выбрать тип создаваемой таблицы (значением по умолчанию является тип Paradox 7).

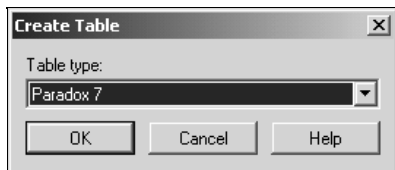


Рис. 17.7. Выбор типа таблицы

В результате открывается диалоговое окно **Create Paradox 7 Table** (рис. 17.8), в котором можно определить структуру записей таблицы.

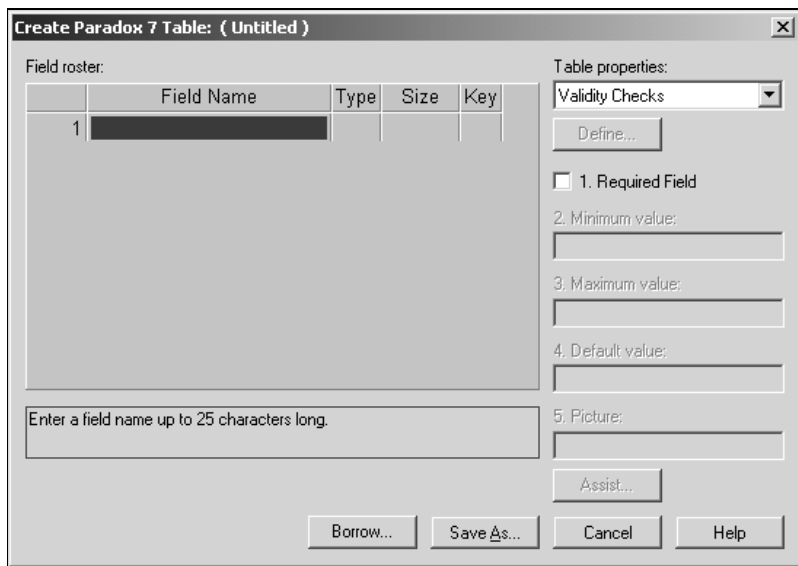


Рис. 17.8. Диалоговое окно **Create <Table Type> Table**

Для каждого поля таблицы необходимо задать имя, тип и, если нужно, размер поля. Имя поля используется для доступа к данным. В качестве имени поля, которое вводится в колонку **Field name**, можно использовать последовательность из букв латинского алфавита и цифр длиной не более 25 символов.

Тип поля определяет тип данных, которые могут быть помещены в поле. Тип задается вводом в колонку **Type** символьной константы. Типы полей и соответствующие им константы приведены в табл. 17.1.

Таблица 17.1. Типы полей таблицы

Тип	Константа	Содержимое поля
Alpha	A	Строка символов. Максимальная длина строки определяется характеристикой <code>Size</code> , значения которой находятся в диапазоне 1..255
Number	N	Число из диапазона $10^{-307}..10^{308}$ с 15-ю значащими цифрами
Money	\$	Число в денежном формате. (Цифры числа делятся на группы при помощи разделителя групп разрядов. Также выводится знак денежной единицы)
Short	S	Целое число из диапазона -32767.. 32767
Long Integer	I	Целое число из диапазона -2147483648..2147483647
Date	D	Дата
Time	T	Время с полуночи, выраженное в миллисекундах
Timestamp	@	Время и дата
Memo	M	Строка символов произвольной длины. Поле типа Memo используется для хранения текстовой информации, которая не может быть сохранена в поле типа Alpha. Размер поля (1.. 240) определяет, сколько символов хранится в таблице. Остальные символы хранятся в файле, имя которого совпадает с именем файла таблицы, а расширение файла — <code>mb</code>
Formatted Memo	F	Строка символов произвольной длины (как у типа Memo). Имеется возможность указать тип и размер шрифта, способ оформления и цвет символов
Graphic	G	Графика
Logical	L	Логическое значение "истина" (True) или "ложь" (False)
Autoincrement	+	Целое число. При добавлении к таблице очередной записи в поле записывается число на единицу большее, чем находится в соответствующем поле последней добавленной записи
Bytes	Y	Двоичные данные. Поле этого типа используется для хранения данных, которые не могут быть интерпретированы Database Desktop

Таблица 17.1 (окончание)

Тип	Константа	Содержимое поля
Binary	B	Двоичные данные. Поле этого типа используется для хранения данных, которые не могут быть интерпретированы Database Desktop. Как и данные типа Memo, эти данные не находятся в файле таблицы. Поля типа Binary, как правило, содержат audio-данные

Константа, определяющая тип поля, может быть введена с клавиатуры или путем выбора типа поля из списка (рис. 17.9), который появляется при щелчке правой кнопкой мыши в колонке **Тип** или при нажатии клавиши <пробел>.

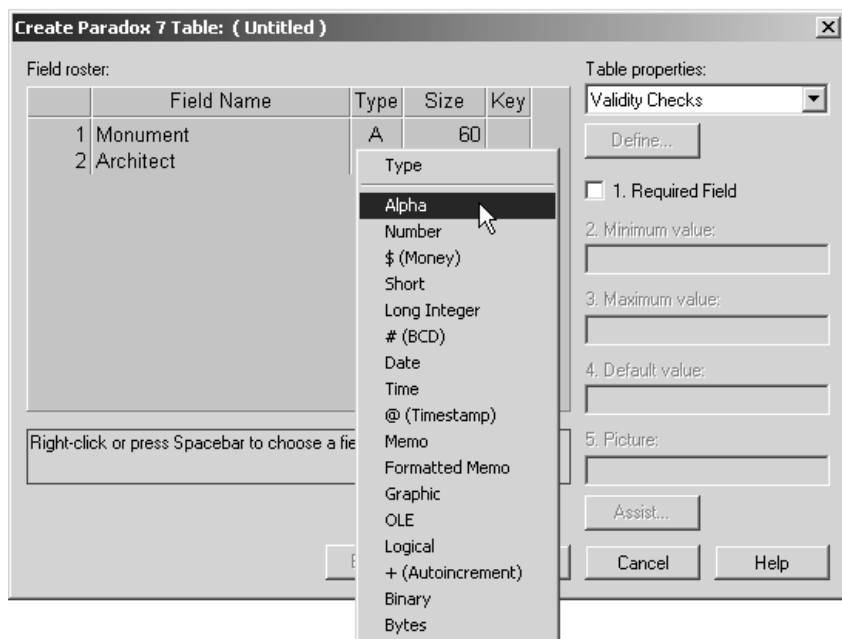


Рис. 17.9. Тип поля можно выбрать из списка

Одно или несколько полей можно пометить как *ключевые*. Ключевое поле определяет *логический* порядок следования записей в таблице. Например, если символьное (тип Alpha) поле Fam (Фамилия) пометить как ключевое, то при выводе таблицы записи будут упорядочены в соответствии с алфавитным порядком фамилий. Если поле Fam не пометить как ключевое, то записи будут выведены в том порядке, в котором они были введены в таблицу.

Примечание

Обратите внимание на то, что в таблице не может быть двух записей с одинаковым содержимым ключевых полей.

Поэтому в рассматриваемом примере ключевыми полями должны быть поля **Fam** (Фамилия) и **Name** (Имя). Тогда в таблицу можно будет ввести информацию об однофамильцах. Однако по-прежнему нельзя будет ввести однофамильцев, у которых совпадают имена. Поэтому в качестве ключевого поля обычно выбирают поле, которое содержит уникальную информацию. Для таблицы, содержащей список людей, в качестве ключевого можно выбрать поле **Pasp** (Паспорт).

Для того чтобы пометить поле как ключевое, необходимо выполнить двойной щелчок в колонке **Key**. Следует обратить внимание на то, что ключевые поля должны быть сгруппированы в верхней части таблицы.

Если данные, для хранения которых предназначено поле, должны обязательно присутствовать в записи, то следует установить флажок **Required Field**. Например, очевидно, что поле **Fam** (Фамилия) обязательно должно быть заполнено, в то время как поле **Tel** (телефон) может оставаться пустым.

Если значение, записываемое в поле, должно находиться в определенном диапазоне, то вводом значений в поля **Minimum value** (Минимальное значение) и **Maximum value** (Максимальное значение) можно задать границы диапазона.

Поле **Default value** позволяет задать значение по умолчанию, которое будет автоматически записываться в поле при добавлении к таблице новой записи.

Поле **Picture** позволяет задать шаблон, используя который можно контролировать правильность вводимой в поле информации. Шаблон представляет собой последовательность обычных и специальных символов. Специальные символы перечислены в табл. 17.2.

Во время ввода информации в позиции поля, которой соответствует специальный символ, будут появляться только символы, допустимые для данного шаблона. Например, если в позиции шаблона стоит символ #, то в соответствующую этому символу позицию можно ввести только цифру. Если в позиции шаблона стоит обычный символ, то во время ввода информации в данной позиции будет автоматически появляться указанный символ.

Например, пусть поле **Tel** типа **A** (строка символов) предназначено для хранения номера телефона, и программа, работающая с базой данных, предполагает, что номер телефона должен быть представлен в обычном виде, т. е. в виде последовательности сгруппированных, разделенных дефисами цифр. В этом случае в поле **Picture** следует записать шаблон: ###-##-##. При вводе информации в поле **Tel** будут появляться только цифры (нажатия клавиш с другими символами игнорируются), причем после ввода третьей и пятой цифр в поле будут автоматически добавлены дефисы.

Таблица 17.2. Специальные символы, используемые при записи шаблонов

Символ шаблона	Допустимый при вводе символ
#	Цифра
?	Любая буква (прописная или строчная)
&	Любая буква (автоматически преобразуется в прописную)
~	Любая буква (автоматически преобразуется в строчную)
@	Любой символ
!	Любой символ (если введена буква, то она автоматически преобразуется в прописную)
;	Символ, следующий за символом "точка с запятой", интерпретируется как обычный символ, а не символ шаблона
*	Любое количество повторяющихся символов, определяемых следующим за "звездочкой" символом шаблона

Некоторые элементы данных поля могут быть необязательными, например, код города для номера телефона. Элементы шаблона, обеспечивающие ввод необязательных данных, заключают в квадратные скобки. Например, шаблон [(###)]###-##-## позволяет вводить в поле номер телефона как с заключенным в скобки кодом города, так и без кода.

Шаблоны позволяют не только контролировать правильность вводимых в поле данных путем блокирования ввода неверных символов, но и обеспечивают автоматизацию ввода данных. Это достигается путем указания в шаблоне в квадратных или фигурных скобках списка допустимых значений содержимого поля.

Например, если для поля Address задать шаблон {Санкт-Петербург, Москва, Воронеж}*@ или [Санкт-Петербург, Москва, Воронеж]*@, то во время ввода данных в это поле название соответствующего города будет появляться сразу после ввода одной из букв: с, м или в. Отличие фигурных скобок от квадратных, и следовательно этих шаблонов друг от друга, состоит в том, что в первом шаблоне содержимое поля обязательно должно начинаться с названия одного из перечисленных в списке городов, а во втором — город может называться по-другому, однако его название придется вводить полностью.

После того как определена структура записей, таблицу следует сохранить. Для этого необходимо нажать кнопку **Save As**, которая находится в диалоговом окне **Create <Table Type> Table**. В результате открывается диалоговое окно **Save Table As**, в котором следует из списка **Alias** выбрать псевдоним базы данных, частью которой является созданная таблица, задать имя файла, в котором будет сохранена таблица (рис. 17.10).

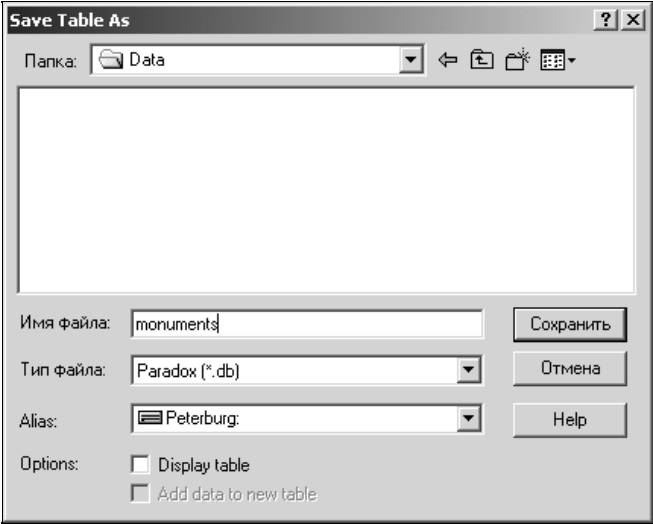


Рис. 17.10. Диалоговое окно **Save Table As**

Каталог может быть выбран обычным образом или путем выбора псевдонима из списка **Alias**, в котором перечислены все зарегистрированные псевдонимы баз данных.

Программа управления базой данных

Процесс создания программы управления базой данных рассмотрим на примере создания базы данных "Архитектурные памятники Санкт-Петербурга".

Перед тем как приступить непосредственно к разработке приложения управления базой данных, необходимо, используя утилиту Database Desktop, создать файл данных (таблицу) и добавить в нее несколько записей. В табл. 17.3 перечислены поля таблицы `monuments` (`monuments` — монументы, памятники). В таблицу `monuments` можно внести информацию о памятниках Санкт-Петербурга (табл. 17.4).

Таблица 17.3. Поля таблицы `monuments`

Поле	Тип	Размер	Содержание
Monument	A	60	Название архитектурного памятника
Architect	A	40	Имя архитектора
Note	A	255	Краткая историческая справка
Photo	A	12	Имя файла иллюстрации

Таблица 17.4. Содержание таблицы *monuments*

Памятник	Архитектор	Историческая справка	Имя файла
Адмиралтейство	А. Д. Захаров	Здание Адмиралтейства таким, как оно выглядит сейчас, стало после перестройки в 1806—1823 годах. Автор проекта — гениальный русский зодчий А. Д. Захаров. Высота шпиля: 72 метра	admiral.bmp
Александровская колонна	Огюст Монферран	Памятник победы России над войсками Наполеона в Отечественной войне 1812 года. Открыта 30 августа 1834 года. Высота: 47,5 метра; вес гранитного ствола: 600 тонн	aleks.bmp
Зимний дворец	Ф. Б. Растрелли	Зимний дворец много раз менял свой облик. Последний раз он перестраивался по проекту Растрелли. Строительство дворца продолжалось более семи лет (1754—1762 годы)	herm.bmp
Исаакиевский собор	Огюст Монферран	Исаакиевский собор, четвертый по счету, стали возводить в 1818 году. Строился собор 40 лет и был окончен в 1858 году. Автор проекта — Огюст Монферран	isaak.bmp
Ростральные колонны	Тома де Томон	32-метровые ростральные колонны, органично вошедшие в архитектурный ансамбль Стрелки Васильевского острова, были сооружены в 1810 году. Они напоминают о существовавшем в древнем Риме обычае: украшать триумфальные колонны рострами захваченных кораблей	rostr.bmp

Примечание

На прилагаемой к книге дискете есть файлы, содержащие изображения исторических памятников Санкт-Петербурга.

После создания и заполнения таблицы нужно, используя утилиту BDE Administrator, создать псевдоним для разрабатываемой базы данных. В качестве псевдонима можно использовать, например, имя *Peterburg*.

Теперь можно приступить к разработке приложения. Методика разработки приложения работы с базой данных ничем не отличается от методики создания обычной программы: к форме добавляются нужные компоненты, устанавливаются значения свойств компонентов, разрабатываются необходимые процедуры обработки событий.

Приложение работы с базой данных должно содержать компоненты, обеспечивающие доступ к данным и возможность просмотра и редактирования содержимого полей. Компоненты доступа к данным находятся на вкладке **Data Access** палитры компонентов, а компоненты отображения данных — на вкладке **Data Controls**.

Доступ к файлу данных (таблице)

Доступ к файлу данных (таблице) обеспечивают компоненты `DataSource`, `Table` и `Database`, значки которых расположены на вкладках **Data Access** и **BDE** (рис. 17.11) палитры компонентов.

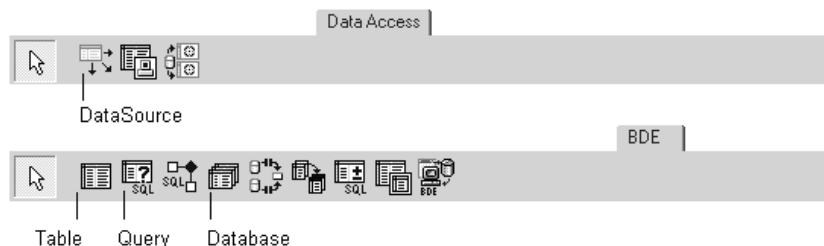


Рис. 17.11. Компоненты вкладок **Data Access** и **BDE** обеспечивают доступ к данным

Компонент `Database` представляет базу данных как единое целое, т. е. совокупность таблиц, а компонент `Table` — одну из таблиц базы данных. Следует обратить внимание на компонент `DataSource` (источник данных), который обеспечивает связь таблиц базы данных с компонентами просмотра и редактирования содержимого полей базы данных.

В простейшем случае, когда база данных представляет собой одну-единственную таблицу, приложение работы с базой данных должно содержать один компонент `Table` и один компонент `DataSource`.

В табл. 17.5 перечислены свойства компонента `Table`, а в табл. 17.6 — свойства компонента `DataSource`. Свойства перечислены в том порядке, в котором следует устанавливать их значения после добавления компонентов в форму приложения.

Таблица 17.5. Свойства компонента *Table*

Свойство	Определяет
Name	Имя компонента. Используется для доступа к свойствам компонента
DatabaseName	Имя базы данных, частью которой является таблица (файл данных), для доступа к которой используется компонент. В качестве значения свойства следует использовать псевдоним базы данных
TableName	Имя файла данных (таблицы данных), для доступа к которому используется компонент
TableType	Тип таблицы. Таблица может быть набором данных в формате Paradox (ttParadox), dBase (ttDBase), FoxPro (ttFoxPro) или представлять собой форматированный текстовый файл (ttASCII)
Active	Признак активизации файла данных (таблицы). В результате присваивания свойству значения True, происходит открытие файла таблицы

Примечание

Во время разработки формы приложения, значения свойств `DatabaseName` и `TableName` задаются путем выбора из списков. В списке `DatabaseName` перечислены все ЗАРЕГИСТРИРОВАННЫЕ псевдонимы, а в списке `TableName` — имена файлов таблиц, которые находятся в соответствующем псевдониму каталоге.

Таблица 17.6. Свойства компонента *DataSource*

Свойство	Определяет
Name	Имя компонента. Используется для доступа к свойствам компонента
DataSet	Имя компонента, представляющего собой входные данные

Свойство `DataSet` обеспечивает связь между компонентом, представляющим собой таблицу или запрос, и компонентами, предназначенными для доступа к записям. Наличие этого свойства позволяет выбирать источник данных. Например, база данных может быть организована таким образом, что таблица, состоящая из большого числа записей, разделена на несколько подтаблиц, имеющих одинаковую структуру. В этом случае в приложении каждой подтаблице будет соответствовать свой компонент `Table`, а выбор конкретной подтаблицы можно осуществить установкой значения свойства `DataSet`. В табл. 17.7 и 17.8 приведены значения свойств компонентов `Table` и `DataSource` для разрабатываемого приложения.

Таблица 17.7. Значения свойств компонента Table

Свойство	Значение
Name	Table1
DatabaseName	Peterburg
TableName	monuments.db
Active	True

Таблица 17.8. Значения свойств компонента DataSource

Свойство	Значение
Name	DataSource1
DataSet	Table1

Просмотр базы данных

Пользователь может просматривать базу данных в режиме формы или в режиме таблицы. В режиме формы можно видеть только одну запись, а в режиме таблицы — несколько записей одновременно. Довольно часто эти два режима комбинируют. Краткая информация (содержимое некоторых ключевых полей) выводится в табличной форме, а при необходимости видеть содержимое всех полей записи выполняется переключение в режим формы.

Компоненты, обеспечивающие просмотр и редактирование содержимого полей базы данных, находятся на вкладке **Data Controls** (рис. 17.12).

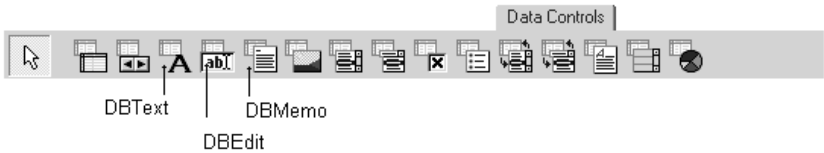


Рис. 17.12. Компоненты просмотра и редактирования полей базы данных

Режим формы

Для того чтобы обеспечить просмотр базы данных в режиме формы, в форму приложения нужно добавить компоненты, обеспечивающие просмотр и, если нужно, редактирование содержимого полей записи, причем по одному компоненту для каждого поля.

Компонент `DBText` позволяет только просматривать содержимое поля, а компоненты `DBEdit` и `DBMemo` — просматривать и редактировать. В табл. 17.9 перечислены некоторые свойства этих компонентов. Свойства перечислены в том порядке, в котором следует устанавливать их значения после добавления в форму приложения.

Таблица 17.9. Свойства компонентов просмотра и редактирования содержимого полей

Свойство	Определяет
<code>Name</code>	Имя компонента. Используется для доступа к свойствам компонента
<code>DataSource</code>	Имя компонента, являющегося источником данных
<code>DataField</code>	Имя поля, для отображения (редактирования) содержимого которого используется компонент

Следует обратить внимание на то, что если значение свойства `Active` таблицы, для отображения содержимого поля которой используется компонент, равно `True`, то после установки значения свойства `DataField` будет выведено содержимое соответствующего поля первой записи файла данных (таблицы).

Для разрабатываемого приложения в форму следует добавить три компонента `DBEdit` (для просмотра и редактирования полей `Name`, `Architect` и `Photo`) и компонент `DBMemo` (для просмотра и редактирования поля `Note`). Кроме того, в форму нужно добавить компонент `Image` для просмотра иллюстрации и четыре компонента `Label` для вывода текста, поясняющего назначение полей ввода.

На рис. 17.13 изображен вид формы разрабатываемого приложения, а в табл. 17.10 приведены значения свойств компонентов просмотра и редактирования содержимого полей. Значению свойства `Visible` компонентов `Image1`, `Label4` и `DBEdit3` следует присвоить значение `False`.

Таблица 17.10. Значения свойств компонентов просмотра и редактирования содержимого полей

Компонент	Значение свойства <code>DataSource</code>	Значение свойства <code>DataField</code>
<code>DBEdit1</code>	<code>DataSource1</code>	<code>Monument</code>
<code>DBEdit2</code>	<code>DataSource1</code>	<code>Architect</code>
<code>DBEdit3</code>	<code>DataSource1</code>	<code>Photo</code>
<code>DBMemo1</code>	<code>DataSource1</code>	<code>Note</code>



Рис. 17.13. Форма приложения **Архитектурные памятники Санкт-Петербурга**

Примечание

Если значение свойства `Active` таблицы, с которой связывается компонент просмотра-редактирования поля записи базы данных, имеет значение `True`, т. е. установлен режим автоматического открытия таблицы, то после установки значения свойства `DataField` в поле компонента автоматически появляется содержимое первой записи таблицы или, если таблица не содержит данных, поле остается незаполненным. Если значение `Active` равно `False`, то в поле компонента появляется имя компонента (содержимое поля `Name`).

Теперь, если откомпилировать и запустить программу, на экране появится форма, в полях которой будет находиться содержимое первой записи файла данных.

Для того чтобы иметь возможность просматривать другие записи файла данных, в форму приложения нужно добавить компонент `DBNavigator`, значок которого находится на вкладке **Data Controls** (рис. 17.14). Компонент `DBNavigator` (рис. 17.15) представляет собой набор кнопок, при щелчке на которых во время работы программы происходит перемещение указателя текущей записи к следующей, предыдущей, первой или последней записи базы данных, а также добавление к файлу данных новой записи, удаление текущей записи.



Рис. 17.14. Значок компонента `DBNavigator` находится на вкладке **Data Controls**

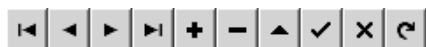


Рис. 17.15. Компонент `DBNavigator`

Табл. 17.11 содержит описания действий, которые выполняются в результате нажатия кнопок компонента *DBNavigator*.

В табл. 17.12 перечислены некоторые свойства компонента *DBNavigator*.

Таблица 17.11. 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	NbInsert В файл данных добавляется новая запись
	Удалить	NbDelete Текущая запись файла данных удаляется
	Редактиро- вание	NbEdit Устанавливается режим редактирования текущей записи
	Сохранить	NbPost Изменения, внесенные в текущую запись, записываются в файл данных
	Отменить	Cancel Изменения, внесенные в текущую запись, отменяются
	Обновить	NbRefresh Внесенные изменения записываются в файл

Таблица 17.12. Свойства компонента *DBNavigator*

Свойство	Определяет
Name	Имя компонента. Используется для доступа к свойствам компонента
DataSource	Имя компонента, являющегося источником данных. В качестве источника данных может выступать база данных (компонент Database), таблица (компонент Table) или результат выполнения запроса (компонент Query)
VisibleButtons	Видимые командные кнопки

Следует обратить внимание на свойство `VisibleButtons`. Оно позволяет скрывать некоторые кнопки компонента `DBNavigator` и тем самым запрещать соответствующие операции над файлом данных. Например, присвоив значение `False` свойству `VisibleButtons.nbDelete`, можно скрыть кнопку `nbDelete` и тем самым запретить удаление записей.

На рис. 17.16 приведен вид формы приложения **Архитектурные памятники Санкт-Петербурга** после добавления компонента `DBNavigator`. Свойству `DataSource` компонента `DBNavigator1` следует присвоить значение `Table1`.

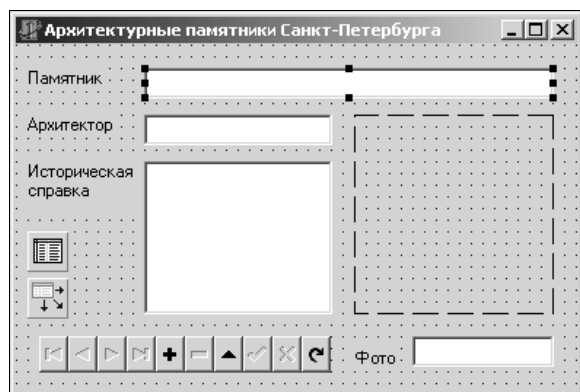


Рис. 17.16. Окончательный вид формы приложения **Архитектурные памятники Санкт-Петербурга**

В принципе, после добавления в форму компонента `DBNavigator` простейшая программа управления базой данных готова. Эта программа обеспечивает просмотр, редактирование, добавление новых и удаление ненужных записей.

Теперь рассмотрим, что надо сделать, чтобы в поле `Image1` появилось изображение памятника, информация о котором выведена в форме. Разрабатываемое приложение предполагает, что изображения (фотографии) архитектурных памятников находятся в файлах, в том же каталоге, что и таблица базы данных. Во время добавления информации в базу данных пользователь вводит в поле `Photo` имя файла фотографии, а во время просмотра фотография автоматически появляется в поле `Image1`.

В листинге 17.1 приведен текст модуля программы **Архитектурные памятники Санкт-Петербурга**.

Листинг 17.1. Архитектурные памятники Санкт-Петербурга

```
unit peter_;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
```



```
Dialogs, ExtCtrls, StdCtrls, DBCtrls, Mask, Db, DBTables,
jpeg; // чтобы можно было выводить jpg-иллюстрации;
```

type

```
TForm1 = class(TForm)
    Table1: TTable;           // база данных – таблица
    DataSource1: TDataSource; // источник данных для полей
                                // редактирования-просмотра

    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBMemo1: TDBMemo;
    Image1: TImage;
    DBEdit3: TDBEdit;
    DBNavigator1: TDBNavigator;
    Label4: TLabel;
    procedure Table1AfterScroll(DataSet: TDataSet);
    procedure DBEdit3KeyPress(Sender: TObject; var Key: Char);
    procedure DBNavigator1Click(Sender: TObject; Button: TNavigateBtn);
    procedure Table1BeforeOpen(DataSet: TDataSet);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
BmpPath: string; // Путь к файлам иллюстраций. Иллюстрации
                  // находятся в подкаталоге Data каталога программы.
```

implementation

```
{ $R *.DFM }
```

```
// выводит фотографию в поле Image1
```

```
procedure ShowFoto(foto: string);
begin
    try
        Form1.Image1.Picture.LoadFromFile(BmpPath+foto);
        Form1.Image1.Visible:=True;
    except
        on EFOpenError do
            begin
```

```
MessageDlg('Файл иллюстрации '+foto+' не найден.',
           mtInformation, [mbOk], 0);

end;
end;
end;

// переход к другой записи (следующей, предыдущей,
// первой или последней)
procedure TForm1.Table1AfterScroll(DataSet: TDataSet);
begin
    if form1.DBEdit3.Visible then
        begin
            form1.DBEdit3.Visible := False;
            form1.Label4.Visible:=False;
        end;
    if Form1.DBEdit3.Text <> ''
        then ShowFoto(Form1.DBEdit3.Text)
        else form1.Image1.Visible:=False;
end;

// нажатие клавиши в поле Фото
procedure TForm1.DBEdit3KeyPress(Sender: TObject; var Key: Char);
begin
    if (key = #13) then
        if Form1.DBEdit3.Text <> ''
            then ShowFoto(Form1.DBEdit3.Text) // показать иллюстрацию
            else form1.Image1.Visible:=False;
end;

// щелчок на компоненте Навигатор
procedure TForm1.DBNavigator1Click(Sender: TObject; Button: TNavigatorBtn);
begin
    case Button of
        nbInsert: begin
            Image1.Visible:=False; // скрыть область вывода иллюстрации
            DBEdit3.Visible:=True; // показать поле Фото
            Label4.Visible:=True; // показать метку Фото
        end;
        nbEdit: begin // редактирование записи
            DBEdit3.Visible:=True; // показать поле Фото
            Label4.Visible:=True; // показать метку Фото
        end;
    end;
end;
end;
```

```
// перед открытием таблицы
```

```
procedure TForm1.Table1BeforeOpen(DataSet: TDataSet);  
begin  
    BmpPath:=ExtractFilePath(ParamStr(0))+'data\';  
end;  
  
end.
```

Вызов процедуры вывода фотографии (ShowFoto) во время просмотра базы данных выполняет процедура TForm1.Table1AfterScroll, которая обеспечивает обработку события AfterScroll для компонента Table1. Событие AfterScroll происходит всякий раз после перехода к другой (следующей, предыдущей, первой, последней) записи таблицы как результат нажатия пользователем на соответствующей кнопке компонента DBNavigator. Процедура TForm1.Table1AfterScroll анализирует содержимое поля (Photo) Form1.DBEdit3.Text и, если оно не пустое, что свидетельствует о наличии ссылки на файл фотографии, выводит иллюстрацию.

При просмотре базы данных поле имени файла иллюстрации (DBEdit3) и его заголовок (Label4) на форме не отображаются. Если пользователь нажимает одну из кнопок компонента DBNavigator, то как результат обработки события OnClick вызывается процедура TForm1.DBNavigator1Click, которая при нажатии кнопки **Добавить** или **Редактировать** делает доступным поле DBEdit3, тем самым позволяя пользователю ввести или изменить имя файла иллюстрации.

Процедура TForm1.DBEdit3KeyPress обрабатывает событие OnKeyPress для компонента DBEdit3. Если пользователь ввел в поле Edit3 (Photo) имя файла иллюстрации и нажал клавишу <Enter> (ее код равен 13), то процедура TForm1.DBEdit3KeyPress выводит иллюстрацию путем вызова процедуры ShowFoto.

Режим таблицы

Программа работы с базой данных, хранящей информацию об архитектурных памятниках Петербурга, выводила информацию в режиме формы. При этом пользователь может одновременно просматривать только одну запись. Такой режим работы с базой данных не всегда удобен. Если надо одновременно видеть несколько записей базы данных, то нужно обеспечить табличный режим просмотра.

Процесс создания приложения, обеспечивающего просмотр базы данных в режиме таблицы, рассмотрим на примере работы с базой данных **Школа**. Пусть база данных **Школа**, имеющая псевдоним Школа, представляет собой таблицу, которая находится в файле school.db. Записи таблицы school состоят из полей: Name (Имя), Fam (Фамилия), Class (Класс), Adr (Адрес) и N (Личный номер). Поля Name, Fam, Class и Adr являются полями символьного типа (тип A), а поле N — числовое, с автоувеличением.

Примечание

Псевдоним *Школа* следует создать при помощи BDE Administrator, а таблицу (файл school.db) — при помощи Database Desktop.

Для обеспечения просмотра данных в режиме таблицы в форму приложения сначала надо добавить и настроить компоненты *Table* и *DataSource*, обеспечивающие доступ к файлу данных и связь с компонентами просмотра-редактирования полей записей, а затем — компонент *DBGrid*, значок которого находится на вкладке **Data Controls** палитры компонентов (рис. 17.17).

Вид формы разрабатываемого приложения после добавления компонента *DBGrid* изображен на рис. 17.18. В табл. 17.13 и 17.14 приведены значения свойств компонентов *Table1* и *DataSource1*.

Таблица 17.13. Значения свойств компонента *Table1*

Свойство	Значение
Name	Table1
DatabaseName	Школа
TableName	school.db
Active	True



Рис. 17.17. Значок компонента *DBGrid* на вкладке **Data Controls**

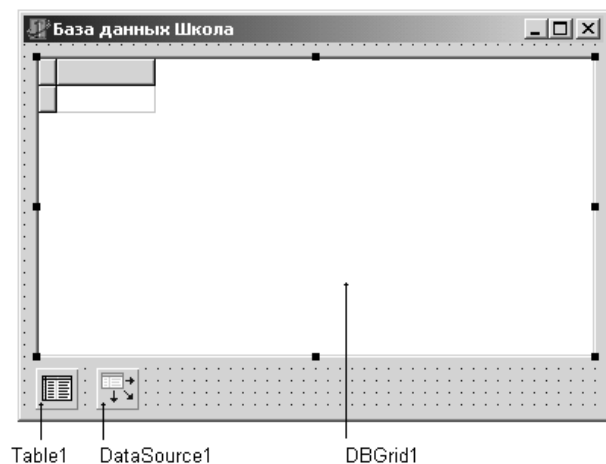


Рис. 17.18. Форма приложения после добавления компонента *DBGrid*

Таблица 17.14. Значения свойств компонента *DataSource1*

Свойство	Значение
Name	DataSource1
DataSet	Table1

Компонент `DBGrid` обеспечивает представление базы данных в виде таблицы. Свойства компонента `DBGrid1` определяют вид таблицы и действия, которые могут быть выполнены над данными во время работы программы. В табл. 17.15 перечислены некоторые свойства компонента `DBGrid`.

Таблица 17.15. Свойства компонента *DBGrid*

Свойство	Определяет
Name	Имя компонента
DataSource	Источник отображаемых в таблице данных
Columns	Отображаемую в таблице информацию
Options.dgTitles	Разрешает вывод строки заголовка столбцов
Options.DgIndicator	Разрешает вывод колонки индикатора. Во время работы с базой данных текущая запись помечается в колонке индикатора треугольником, новая запись — звездочкой, редактируемая — специальным значком
Options.dgColumnResize	Разрешает менять во время работы программы ширину колонок таблицы
Options.dgColLines	Разрешает выводить линии, разделяющие колонки таблицы
Options.dgRowLines	Разрешает выводить линии, разделяющие строки таблицы

Для того чтобы задать, какая информация будет отображена в таблице во время работы программы, нужно сначала определить источник данных для таблицы (установить значения свойства `DataSource`), затем — установить значения уточняющих параметров свойства `Columns`. Значение свойства `DataSource` задается обычным образом, т. е. в окне **Object Inspector**. Чтобы установить значение свойства `Columns`, надо в окне **Object Inspector** выбрать это свойство и щелкнуть на кнопке с тремя точками. В результате открывается окно редактора колонок (рис. 17.19).

Для того чтобы в таблицу (в компонент `DBGrid`) добавить колонку, обеспечивающую просмотр содержимого поля записи файла данных, необходимо

нажать кнопку **Add New**, находящуюся на панели инструментов в верхней части окна (это единственная доступная после запуска редактора кнопка). Затем нужно выбрать строку, соответствующую добавленной колонке и, используя **Object Inspector**, установить значения свойств этой колонки. Свойство `Columns` компонента `DBGrid` представляет собой массив компонентов типа `TColumn`. Каждой колонке соответствует элемент массива. Устанавливая значения свойств компонента `Columns[i]`, программист задает вид *i*-й колонки компонента `DBGrid`, тем самым определяя вид всей таблицы.



Рис. 17.19. Редактор колонок

В табл. 17.16 перечислены свойства компонента `Columns`.

Таблица 17.16. Свойства компонента `Columns`

Свойство	Определяет
FieldName	Поле записи, содержимое которого выводится в колонке
Width	Ширина колонки в пикселах
Font	Шрифт, используемый для вывода текста в ячейках колонки
Color	Цвет фона колонки
Alignment	Способ выравнивания текста в ячейках колонки. Текст может быть выровнен по левому краю (<code>taLeftJustify</code>), по центру (<code>taCenter</code>) или по правому краю (<code>taRightJustify</code>)
Title.Caption	Заголовок колонки. Значением по умолчанию является имя поля записи
Title.Alignment	Способ выравнивания заголовка колонки. Заголовок может быть выровнен по левому краю (<code>taLeftJustify</code>), по центру (<code>taCenter</code>) или по правому краю (<code>taRightJustify</code>)
Title.Color	Цвет фона заголовка колонки
Title.Font	Шрифт заголовка колонки

В простейшем случае для каждой колонки достаточно установить значение свойства `FieldName`, которое определяет имя поля записи, содержимое которого будет отображаться в колонке, а также значение свойства `Title.Caption`, определяющего заголовок колонки.

В табл. 17.17 указаны значения свойств `Columns` компонента `DBGrid1`, а на рис. 17.20 приведен окончательный вид формы.

Таблица 17.17. Значения свойств компонента `DBGrid1.Columns`

Компонент	FieldName	Title.Caption
<code>DBGrid1.Columns[0]</code>	Fam	Фамилия
<code>DBGrid1.Columns[1]</code>	Name	Имя
<code>DBGrid1.Columns[2]</code>	Class	Класс
<code>DBGrid1.Columns[3]</code>	Adr	Адрес, телефон

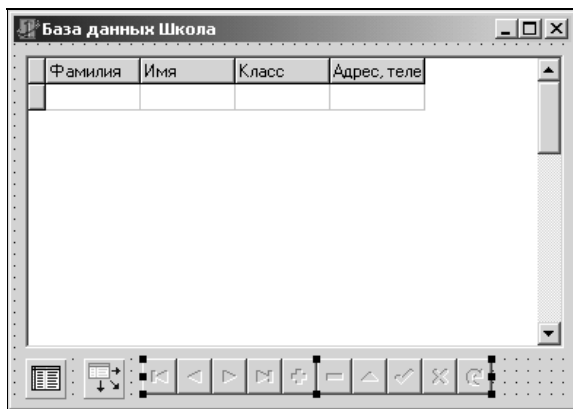


Рис. 17.20. Вид формы после настройки компонента `DBGrid1`

Последнее, что надо сделать, — добавить к форме компонент `DBNavigator`, настроив его на работу с таблицей-источником данных (в рассматриваемом примере свойству `DataSource` компонента `DBNavigator` следует присвоить значение `Table1`).

После этого программу можно откомпилировать и запустить. Чтобы после запуска программы иметь возможности просмотра информации и ввода новых записей в базу данных, свойство `Active` таблицы-источника данных должно иметь значение `True`.

Работа с базой данных, представленной в виде таблицы, во многом похожа на работу с электронной таблицей, например, Microsoft Excel.

Выбор информации из базы данных

Большинство систем управления базами данных позволяют выполнять выборку нужной информации путем выполнения *запросов*. Пользователь в соответствии с определенными правилами формулирует запрос, указывая, каким критериям должна удовлетворять интересующая его информация, а система выводит записи, удовлетворяющие запросу.

Для выборки из базы данных записей, удовлетворяющих некоторому критерию, предназначен компонент `Query`, значок которого находится на вкладке **Data Access** (рис. 17.21).



Рис. 17.21. Значок компонента **Query**

Компонент `Query` во многом похож на компонент `Table`, но, в отличие от последнего, он представляет не всю базу данных (все записи), а только ее часть — записи, удовлетворяющие запросу.

В табл. 17.18 перечислены некоторые свойства компонента Query.

Таблица 17.18. Свойства компонента Query

Свойство	Определяет
Name	Имя компонента. Используется компонентом <code>Datasource</code> для связи результата выполнения запроса (набора записей) с компонентом, обеспечивающим просмотр записей, например <code>DBGrid</code>
SQL	Записанный на языке SQL запрос к базе данных (к таблице)

Таблица 17.18 (окончание)

Свойство	Определяет
Active	При присвоении свойству значения True активизирует выполнение запроса

Для того чтобы во время разработки программы задать, какая информация будет выделена из базы данных в результате выполнения запроса, свойство SQL должно содержать представленный на языке SQL запрос на выборку данных.

В общем виде запрос на выборку данных из таблицы выглядит так:

```
SELECT СписокПолей
FROM Таблица
WHERE
( Критерий)
ORDER BY СписокПолей
```

где:

SELECT — команда выбрать из таблицы записи и вывести содержимое полей, имена которых указаны в списке;

FROM — параметр команды, который определяет имя таблицы, из которой нужно сделать выборку;

WHERE — параметр, который задает критерий выбора. В простейшем случае критерий — это инструкция сравнения содержимого поля с константой;

ORDER BY — параметр, который задает условие, в соответствии с которым будут упорядочены записи, удовлетворяющие критерию запроса.

Например, запрос

```
SELECT Fam, Name
FROM ' :Школа:school.db'
WHERE
(Class = '10a')
ORDER BY Name, Fam
```

обеспечивает выборку из базы данных **Школа** записей, у которых в поле Class находится текст 10a, т. е. формирует упорядоченный по алфавиту список учеников 10 а класса.

Другой пример. Запрос

```
SELECT Fam, Name
FROM " :DATA:school.db"
```

```
WHERE
(Fam > 'K') and (Fam < 'Л')
ORDER BY Name, Fam
```

обеспечивает выбор информации об учениках, фамилии которых начинаются на букву К.

Запрос может быть сформирован и записан в свойство SQL во время разработки формы или во время работы программы.

Для записи запроса в свойство SQL во время разработки формы используется редактор списка строк (рис. 17.22), окно которого открывается в результате нажатия кнопки с тремя точками в строке свойства SQL окна **Object Inspector**.

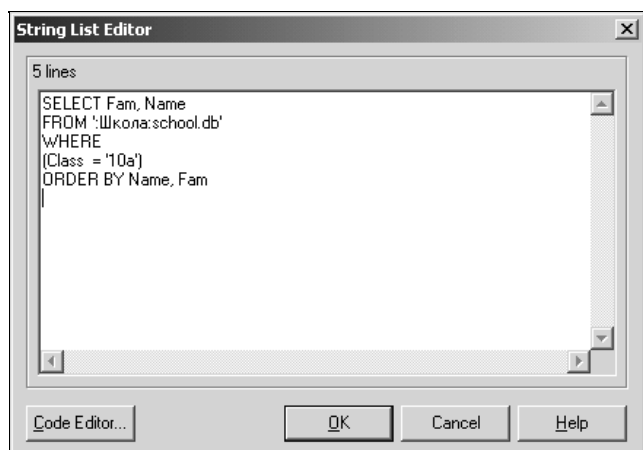


Рис. 17.22. Пример запроса к базе данных **Школа**

Свойство SQL представляет собой список строк. Поэтому, чтобы сформировать запрос во время работы программы, нужно, используя метод Add, добавить строки (SQL-инструкции) в список SQL.

Ниже приведен фрагмент кода, который формирует запрос на поиск информации о конкретном человеке (критерий выбора — содержимое поля Fam должно совпадать со значением переменной fam).

```
with form1.Query1 do
begin
  Close;           // закрыть файл — результат выполнения
                  // предыдущего запроса
  SQL.Clear;      // удалить текст предыдущего запроса
  // записываем новый запрос в свойство SQL
  SQL.Add('SELECT Fam, Name, Class');
  SQL.Add('FROM "Школа:school.db"');
  SQL.Add('WHERE');
  SQL.Add('(Fam = '' + fam + '')');
```

```

SQL.Add('ORDER BY Name, Fam');
Open;           // активизируем выполнение запроса
end;

```

Следующая программа, ее текст приведен в листинге 17.2, а диалоговое окно — на рис. 17.23, демонстрирует возможность изменения запроса, а точнее критерия запроса, во время работы программы. Программа обеспечивает вывод как всего списка учеников, так и его части. Например, посредством выполнения запроса выводится информация только о конкретном ученике.

Для просмотра базы данных и результата выполнения запроса используется компонент `DBGrid1`, который через компонент `DataSource1` взаимодействует с компонентом `Table1` (при просмотре всей базы данных) или с компонентом `Query1` (при просмотре результата выполнения запроса).

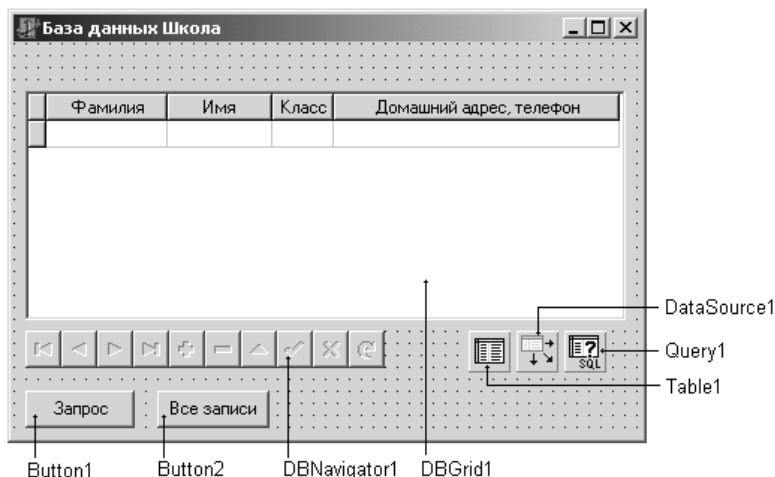


Рис. 17.23. Форма приложения **База данных Школа**

Листинг 17.2. База данных Школа

```

unit school2_;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, Db, DBTables, ExtCtrls, DBCtrls, StdCtrls;

type
  TForm1 = class(TForm)
    Table1: TTable;    // таблица - вся база данных

```

```

Query1: TQuery;      // запрос – записи,
                     // удовлетворяющие критерию выбора
DataSource1: TDataSource; // источник данных – таблица или запрос
DBGrid1: TDBGrid;    // таблица для отображения БД или
                     // результата выполнения запроса
DBNavigator1: TDBNavigator;
DBText1: TDBText;
Button1: TButton;    // кнопка Запрос
Button2: TButton;    // кнопка Все записи
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure FormActivate(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

    // щелчок на кнопке Запрос
procedure TForm1.Button1Click(Sender: TObject);
var
    fam: string[30];
begin
    fam:=InputBox('Выборка информации из БД',
                  'Укажите фамилию и щелкните на ОК.', '');
    if fam <> '' then
        begin // пользователь ввел фамилию
            with form1.Query1 do begin
                Close; // закрыть результат выполнения предыдущего запроса
                SQL.Clear; // удалить текст предыдущего запроса

                // формируем новый запрос
                SQL.Add('SELECT Fam, Name, Class');
                SQL.Add('FROM ":Имя:school.db"');
                SQL.Add('WHERE');
                SQL.Add('(Fam = "' + fam + '")');
                SQL.Add('ORDER BY Name, Fam');
                Open; // активизируем выполнение запроса
            end;

```

```
if Query1.RecordCount <> 0 then
    DataSource1.DataSet:=Query1 // отобразить результат
                                // выполнения запроса
else begin
    ShowMessage('В БД нет записей, удовлетворяющих ' +
                'критерию запроса.');
```

DataSource1.DataSet:=Table1;

```
end;
end;
end;
```

// щелчок на кнопке Все записи

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
```

```
    DataSource1.DataSet:=Table1; // источник данных — таблица
```

```
end;
```

// активизация формы

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

```
begin
```

```
    DataSource1.DataSet := Table1;
```

```
    Table1.Active := True;
```

```
end;
```

```
end.
```

Процедура TForm1.Button1Click запускается нажатием кнопки **Запрос**. Она принимает от пользователя строку (фамилии) и, записывая (добавляя) строки в свойство SQL формирует текст запроса. Затем эта процедура, вызывая метод Open, активизирует выполнение запроса.

Примечание

Обратите внимание на то, что перед изменением свойства SQL запрос должен быть закрыт при помощи метода Close (здесь надо вспомнить, что результат выполнения запроса — это файл данных (таблица), который создается в результате выполнения запроса).

Процедура TForm1.Button2Click, которая запускается нажатием кнопки **Все записи**, устанавливает в качестве источника данных для компонента DataSource1 компонент Table1, тем самым обеспечивая переход в режим просмотра всей базы данных.

Если запрос записан в свойство SQL во время разработки формы приложения, то во время работы программы критерий запроса можно изменить простой заменой соответствующей строки текста запроса.

Например, для запроса:

```
SELECT DISTINCT Fam, Name, Class
FROM "Школа:school.db"
WHERE
(Class= '10a')
ORDER BY Name, Fam
```

инструкция замены критерия запроса может быть такой:

```
form1.Query1.SQL[3]:='(Fam="' + fam+ '")'
```

Примечание

Обратите внимание на то, что свойство SQL является структурой типа TStrings, в которой строки нумеруются с нуля.

Динамически создаваемые псевдонимы

Использование псевдонима для доступа к базе данных обеспечивает независимость программы от размещения данных в системе, позволяет размещать программу работы с данными и базу данных на разных дисках компьютера, в том числе и на сетевом. Вместе с тем для простых баз данных типичным решением является размещение базы данных в отдельном подкаталоге того каталога, в котором находится программа работы с базой данных. Таким образом, программа работы с базой данных всегда "знает", где находятся данные. При таком подходе можно отказаться от создания псевдонима при помощи BDE Administrator и возложить задачу создания псевдонима на программу работы с базой данных. Причем, псевдоним будет создаваться автоматически при запуске программы и уничтожаться при завершении ее работы. Очевидно, что такой подход облегчает администрирование базы данных.

В качестве иллюстрации сказанного в листинге 17.3 приведен вариант программы работы с базой данных **Школа**, которая для доступа к базе данных использует динамически создаваемый псевдоним.

Листинг 17.3. База данных Школа (псевдоним базы данных создается динамически)

```
{ Для доступа к файлу таблицы программа использует псевдоним Школа,
  который создается автоматически при запуске программы (процедура
  FormActivate). Предполагается, что файлы данных находятся
  в подкаталоге DATA, того каталога, в котором находится программа. }
unit school3_;

interface
```

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Grids, DBGrids, Db, DBTables, ExtCtrls, DBCtrls, StdCtrls;

type

```
TForm1 = class(TForm)
    Table1: TTable;      // таблица (вся база данных)
    Query1: TQuery;      // записи БД, удовлетворяющие критерию выбора)
    DataSource1: TDataSource; // источник данных – таблица или запрос
    DBGrid1: TDBGrid;    // таблица для отображения БД или
                        // результата выполнения запроса
    DBNavigator1: TDBNavigator;
    DBText1: TDBText;
    Button1: TButton;    // кнопка запрос
    Button2: TButton;    // кнопка Все записи
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
```

private

```
{ Private declarations }
```

public

```
{ Public declarations }
```

```
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
// щелчок на кнопке Запрос
```

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

var

```
fam: string[30];
```

begin

```
fam:=InputBox('Выборка информации из БД',
               'Укажите фамилию и щелкните на ОК.', '');
```

```
if fam <> '' // пользователь ввел фамилию
```

```
then
```

begin

```
    with form1.Query1 do begin
```

```
        Close;      // закрыть результат выполнения
```

```
                    // предыдущего запроса
```

```
        SQL.Clear; // удалить текст предыдущего запроса
```

```

    // записываем новый запрос в свойство SQL
    SQL.Add('SELECT Fam, Name, Class');
    SQL.Add('FROM ":Школа:school.db");
    SQL.Add('WHERE');
    SQL.Add('(Fam = "' + fam + '")');
    SQL.Add('ORDER BY Name, Fam');
    Open;           // активизируем выполнение запроса
end;

if Query1.RecordCount <> 0 then
    DataSource1.DataSet:=Query1 // отобразить результат
                                // выполнения запроса

```

```

else begin

```

```

    ShowMessage('В БД нет записей, удовлетворяющих ' +
                'критерию запроса.');
```

```

    DataSource1.DataSet:=Table1;

```

```

end;

```

```

end;

```

```

end;

```

```

// щелчок на кнопке Все записи

```

```

procedure TForm1.Button2Click(Sender: TObject);

```

```

begin

```

```

    DataSource1.DataSet:=Table1; // источник данных — таблица

```

```

end;

```

```

// активизация формы

```

```

procedure TForm1.FormActivate(Sender: TObject);

```

```

begin

```

```

    with Session do

```

```

        begin

```

```

            ConfigMode := cmSession;

```

```

        try

```

```

            { Если файл данных находится в том же каталоге,
              что и выполняемый файл программы, то в программе
              путь к файлу данных может быть получен из командной
              строки при помощи функции ExtractFilePath(ParamStr(0)).

```

```

              В приведенном примере файл данных находится в подкаталоге
              DATA каталога программы. }

```

```

// создадим временный псевдоним для базы данных

```

```

AddStandardAlias( 'Школа',
                  ExtractFilePath(ParamStr(0))+'DATA\',
                  'PARADOX');

```

```

Table1.Active:=True; // откроем базу данных

```



```
finally
    ConfigMode := cmAll;
end;

end;

end;

end.
```

В рассматриваемом варианте программы предполагается, что база данных находится в подкаталоге `DATA` того каталога, в котором находится исполняемый файл программы. Псевдоним создается процедурой `TForm1.FormActivate`. Непосредственное создание псевдонима выполняет процедура `AddStandardAlias`, которой в качестве параметра передается имя псевдонима и соответствующее ему имя каталога. Так как во время разработки программы нельзя знать, в каком каталоге будет размещена программа работы с базой данных и, следовательно, подкаталог базы данных — `DATA`, имя каталога определяется во время работы программы путем обращения к функциям `ParamStr` и `EtractFilePatch`. Значение первой — полное имя файла исполняемой программы, второй — путь к файлу исполняемой программы. Таким образом, процедуре `AddStandardAlias` передается полное имя каталога базы данных.

Перенос программы управления базой данных на другой компьютер

Довольно часто возникает необходимость переноса созданной программы управления базой данных на другой компьютер, например, для того, чтобы продемонстрировать ее своим друзьям или знакомым. В отличие от процесса переноса обычной программы, когда, как правило, достаточно скопировать только исполняемый файл (exe-файл), при переносе программы управления базой данных необходимо выполнить перенос `BDE`.

Здесь следует вспомнить, что `BDE` представляет собой набор программ, библиотек и драйверов, обеспечивающих работу прикладной программы с базой данных. Выполнить перенос `BDE` на другой компьютер вручную практически невозможно.

Поэтому Borland рекомендует создавать установочную программу, которая выполнит копирование всех необходимых файлов, в том числе и компонентов `BDE`. В качестве средства создания установочной программы Borland настоятельно рекомендует использовать утилиту `InstallShield Express`, которая входит в состав всех наборов Delphi. Поставляемая с Delphi версия этой утилиты специально адаптирована к задаче переноса и настройки `BDE`.

Можно попытаться установить BDE вручную. Ниже перечислены файлы (их имена определены опытным путем), необходимые для работы с базой данных Paradox.

BLW32	DLL
IDAPI32	DLL
IDBAT32	DLL
IDPDX32	DLL
IDR20009	DLL
USA	BLL
CHARSET	BLL

Эти файлы нужно установить на компьютер пользователя, затем убедиться, что в реестре Windows есть перечисленные ниже разделы и параметры. Если эти разделы и параметры отсутствуют, их следует создать вручную.

Раздел: HKEY_LOCAL_MACHINE\Software\Borland\Database engine

Параметр: DLLPATH должен содержать путь к DLL файлам BDE

Раздел: HKEY_LOCAL_MACHINE\Software\Borland\BLW32

Параметр: BLAPIPATH должен содержать путь к BLL файлам BDE

Заключение

В одной книге, тем более ориентированной на начинающих программистов, нельзя рассмотреть все вопросы, связанные с программированием на Object Pascal, все компоненты и возможности Delphi 6. Многие интересные темы остались за рамками книги. Вместе с тем, в книге рассмотрены фундаментальные понятия программирования, базовые структуры данных и методы работы с ними, основные возможности среды разработки и методы работы в ней — все то, что должен знать и уметь начинающий программист, стремящийся стать профессионалом.

Работая над книгой, я старался выстроить материал таким образом, чтобы в тот момент, когда читатель перевернет последнюю страницу, он был бы в состоянии написать вполне приличную программу, которая бы не "ломалась" от неверно введенных данных и которая могла бы произвести впечатление на коллег-программистов, пусть тоже начинающих, например, наличием справочной системы. Согласитесь, что когда программа не "ломается", это не производит особого впечатления (так и должно быть). Но когда у программы есть справочная система, то она выглядит вполне профессионально!

Научиться программировать, не владея общими вопросами теории и практики программирования, практически невозможно. Книги, заглавия которых начинаются фразами "Практическое руководство...", "Наиболее полное описание...", как правило, рассматривают узкие специфические вопросы конкретного языка или среды программирования. Они, безусловно, полезны. Однако следует обращать внимание и на книги, посвященные общим вопросам теории и практики программирования. Авторами этих книг, чаще всего, являются ведущие специалисты в области программирования. Здесь хочется порекомендовать работы Вирта, Страуструпа, Зелковица, Гринзоу (см. приложение 4 *"Рекомендуемая дополнительная литература"*).

В заключение еще раз повторю, что научиться программировать можно только программируя, только решая конкретные задачи. Поэтому ищите задачи и программируйте!

ПРИЛОЖЕНИЕ 1

Язык Pascal (краткий справочник)

Зарезервированные слова и директивы

Таблица П1.1. Зарезервированные слова

and	file	not	then
array	for	object	to
asm	function	of	type
begin	goto	or	unit
case	if	packed	until
const	implementation	procedure	uses
constructor	in	program	var
destructor	inherited	record	while
div	inline	repeat	with
do	interface	set	xor
downto	label	shl	
else	mod	shr	
end	nil	string	

Таблица П1.2. Директивы

absolute	far	near	virtual
assembler	forward	private	
external	interrupt	public	

Структура модуля

Модуль состоит из последовательности разделов. Каждый раздел начинается ключевым словом и продолжается до начала следующего раздела.

unit ИмяМодуля;

interface // раздел интерфейса

{ Здесь находятся описания процедур и функций модуля,
которые могут использоваться другими модулями. }

const // раздел объявления констант

{ Здесь находятся объявления глобальных констант модуля,
которые могут использоваться процедурами и функциями модуля. }

type // раздел объявления типов

{ Здесь находятся объявления глобальных типов модуля,
которые могут использоваться процедурами и функциями модуля }

var // раздел объявления переменных

{ Здесь находятся объявления глобальных переменных модуля,
которые могут использоваться процедурами и функциями модуля }

implementation // раздел реализации

{ Здесь находятся описания (текст) процедур и функций модуля }

end.

Основные типы данных

К основным типам данных Pascal относятся:

☐ Целые числа (integer)

☐ Дробные числа (real)

☐ Символы (char)

☐ Строки (string)

☐ Логический тип (boolean)

☐ Целые числа и числа с плавающей точкой могут быть представлены в различных форматах.

Таблица П1.3. Целые числа.

Формат	Диапазон
Shortint	−128..127
Integer	−32 768..32 767
Longint	−2 147 483 648..2 147 483 647
Byte	0..255
Word	0..65 535

Таблица П1.4. Числа с плавающей точкой

Формат	Диапазон	Значащих цифр
Real	2,9e-39 .. 1,7e38	11—12
Single	1,5e-45 .. 3,4e38	7—8
Double	5,0e-324 .. 1,7e308	15—16
Extended	3,4e-4932 .. 1,1e4932	19—20

Строки

Объявление переменной-строки длиной 255 символов:

```
Имя:string;
```

Объявление переменной-строки указанной длины.

```
Имя:string[ ДлинаСтроки ];
```

Массив

Объявление одномерного массива:

```
ИмяМассива: array [НижнийИндекс...ВерхнийИндекс] of ТипЭлементов;
```

Объявление двумерного массива:

```
ИмяМассива: array [НижнийИндекс1..ВерхнийИндекс1,  
                    НижнийИндекс2..ВерхнийИндекс2]  
of ТипЭлементов;
```

Запись

Вариант 1.

ПеременнаяЗапись: **record**

Поле1: Тип1;

Поле2: Тип2;

ПолеJ: ТипJ;

end;

Вариант 2.

type

ИмяТипаЗаписи = **record**

Поле1: Тип1;

Поле2: Тип2;

...

ПолеJ: ТипJ;

end;

var

ПеременнаяЗапись: ИмяТипаЗаписи;

Сначала объявляется тип записи, затем — переменная-запись.

Инструкции выбора

Инструкция *if*

Вариант 1.

if Условие

then

begin

{ Инструкции, которые выполняются, }

{ если условие истинно. }

end

else

begin

{ Инструкции, которые выполняются, }

{ если условие ложно }

end;

Вариант 2.

```
if Условие
then
begin
    { Инструкции, которые выполняются, }
    { если условие истинно. }
end;
```

Примечание

Если между `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать.

Инструкция *case*

Вариант 1.

```
case Выражение of
    Список1_Констант: begin
        { инструкции }
    end;
    Список2_Констант: begin
        { инструкции }
    end;
    СписокJ_Констант: begin
        { инструкции }
    end;
end;
```

Вариант 2.

```
case Выражение of
    Список1_Констант: begin
        { инструкции }
    end;
    Список2_Констант: begin
        { инструкции }
    end;
    СписокJ_Констант: begin
        { инструкции J }
    end;
```



```
        else
            begin
                { инструкции }
            end;
end;
```

Инструкции между `begin` и `end` выполняются, если значение выражения, записанного после `case`, совпадает с константой из соответствующего списка. Если это не так, то выполняются инструкции, находящиеся после `else`, между `begin` и `end`.

Примечание

Если между `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать.

Циклы

Инструкция *for*

Вариант 1 (с увеличением счетчика).

```
for Счетчик:=НачальноеЗначение to КонечноеЗначение do
begin
    { здесь инструкции }
end;
```

Инструкции между `begin` и `end` выполняются (*КонечноеЗначение – НачальноеЗначение*) + 1 раз.

Если *НачальноеЗначение* > *КонечноеЗначение*, то инструкции между `begin` и `end` не выполняются.

Примечание

Если между `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать.

Вариант 2 (с уменьшением счетчика).

```
for Счетчик:=НачальноеЗначение downto КонечноеЗначение do
begin
    { здесь инструкции }
end;
```

Инструкции между `begin` и `end` выполняются (*НачальноеЗначение – КонечноеЗначение*) + 1 раз.

Если *НачальноеЗначение* < *КонечноеЗначение*, то инструкции между `begin` и `end` не выполняются.

Примечание

Если между `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать.

Инструкция *repeat*

```
repeat  
    { инструкции }  
until Условие;
```

Сначала выполняются инструкции цикла, которые расположены между `repeat` и `until`. Затем вычисляется значение выражения *Условие*, и если оно равно `False`, то инструкции цикла выполняются еще раз. И так до тех пор, пока значение выражения *Условие* не станет равным `True`.

Инструкция *while*

```
while Условие do  
begin  
    { инструкции }  
end;
```

Сначала проверяется *Условие*, если оно истинно, то выполняются инструкции между `begin` и `end`. Затем снова проверяется *Условие*. Если оно выполняется, то инструкции цикла выполняются еще раз. И так до тех пор, пока *Условие* не станет ложным.

Примечание

Если между `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать.

Безусловный переход

Инструкция *GoTo*

```
GoTo Метка;
```

Инструкция осуществляет переход к инструкции, перед которой стоит *Метка*. Метка должна быть объявлена в разделе `label`.

Объявление функции

```
function ИмяФункции(var Параметр1:Тип1;  
                    var Параметр2:Тип2;  
                    . . .  
                    var ПараметрJ:ТипJ) : Тип;  
  
const  
    { описание констант }  
  
var  
    { описание переменных }  
  
begin  
    { инструкции функции }  
  
    Result:=Значение;  
  
end;
```

Примечание

Слово **var** ставится перед именем параметра в том случае, если параметр используется для возврата значения из функции в вызывающую ее программу.

Объявление процедуры

```
procedure ИмяПроцедуры(var Параметр1:Тип1;  
                        var Параметр2:Тип2;  
                        . . .  
                        var ПараметрJ:ТипJ) ;  
  
const  
    { описание констант }  
  
var  
    { описание переменных }  
  
begin  
    { инструкции процедуры }  
  
end;
```

Примечание

Слово **var** ставится перед именем параметра в том случае, если параметр используется для возврата значения из функции в вызывающую ее программу.

Стандартные функции и процедуры

- При описании функций и процедур приняты следующие обозначения:
- ❑ Имена функций и процедур выделены полужирным.
 - ❑ Формальные параметры изображены курсивом. В качестве параметра могут использоваться константы, переменные или выражения соответствующего типа. Если параметром обязательно должна быть переменная, то перед ним поставлено слово `var`. После параметра через двоеточие указывается его тип.
 - ❑ После списка параметров функций через двоеточие указан тип результата, возвращаемого функцией.

Таблица П1.5. Математические функции

Abs (<i>Выражение</i>)	Абсолютное значение аргумента (целый или вещественный тип)
Sqr (<i>Выражение</i>)	Квадрат аргумента (целый или вещественный тип)
Sqrt (<i>Выражение</i> : real): real	Квадратный корень аргумента
Sin (<i>Выражение</i> : real): real	Синус
Cos (<i>Выражение</i> : real): real	Косинус
Arctan (<i>Выражение</i> : real): real	Арктангенс
Exp (<i>Выражение</i> : real): real	Экспонента
Ln (<i>Выражение</i> : real): real	Натуральный логарифм

Таблица П1.6. Преобразования

Int (<i>Выражение</i> : real): real	Целая часть
Trunc (<i>Выражение</i> : real): longint	Целая часть
Round (<i>Выражение</i> : real): longint	Округление к ближайшему целому
IntToStr (<i>Выражение</i>)	Преобразование числового выражения целого типа в строку
FloatToStr (<i>Выражение</i>)	Преобразование вещественного числа в его изображение
FloatToStrF (<i>Выражение</i> , <i>Формат</i> , <i>Точность</i> , <i>КоличествоЦифр</i>)	Преобразование вещественного числа в его изображение с возможностью выбора способа изображения

Таблица П1.6 (окончание)

StrToInt (<i>Строка</i> :string)	Преобразование строки, изображающей целое или вещественное число, в число
StrToFloat (<i>Строка</i> :string)	Преобразование строки, изображающей вещественное число, в число

Таблица П1.7. Работа со строками и символами

Concat (<i>Строка1</i> :string, ..., <i>СтрокаN</i> :string):string	Объединение нескольких строк в одну
Copy (<i>Строка</i> :string, <i>НомерСимвола</i> :integer, <i>Длина</i> :integer):string	Выделение подстроки
Delete (var <i>Строка</i> :string, <i>НомерСимвола</i> :integer, <i>Кол-во</i> :integer)	Удаление части строки
Length (<i>Строка</i> :string):integer	Длина строки
Pos (<i>Строка</i> :string, <i>Подстрока</i> :string):byte	Позиция подстроки в строке
Chr (<i>КодСимвола</i> :byte)	Символ с указанным кодом

ПРИЛОЖЕНИЕ 2

Кодировка символов в Windows

В Windows в основном используется кодировка, которая называется ANSI. Разновидность набора ANSI, содержащая символы русского алфавита, именуется Windows-1251.

Таблица П2.1. Некоторые служебные символы

Код символа	Символ
9	Табуляция
11	Новая строка
13	Конец параграфа
32	Пробел

Таблица П2.2. Символы с кодами 32—127

32		39	'	46	.	53	5
33	!	40	(	47	/	54	6
34	"	41	)	48	0	55	7
35	#	42	*	49	1	56	8
36	\$	43	+	50	2	57	9
37	%	44	,	51	3	58	:
38	&	45	-	52	4	59	;

Таблица П2.2 (окончание)

60	<	77	M	94	^	111	o
61	=	78	N	95	_	112	p
62	>	79	O	96	'	113	q
63	?	80	P	97	a	114	r
64	@	81	Q	98	b	115	s
65	A	82	R	99	c	116	t
66	B	83	S	100	d	117	u
67	C	84	T	101	e	118	v
68	D	85	U	102	f	119	w
69	E	86	V	103	g	120	x
70	F	87	W	104	h	121	y
71	G	88	X	105	i	122	z
72	H	89	Y	106	j	123	[
73	I	90	Z	107	k	124	
74	J	91	[	108	l	125	]
75	K	92	\	109	m	126	~
76	L	93	]	110	n	127	

Таблица П2.3. Символы с кодами 192—255

192	A	199	З	206	О	213	Х
193	Б	200	И	207	П	214	Ц
194	В	201	Й	208	Р	215	Ч
195	Г	202	К	209	С	216	Ш
196	Д	203	Л	210	Т	217	Щ
197	Е	204	М	211	У	218	Ъ
198	Ж	205	Н	212	Ф	219	Ы

Таблица П2.3 (окончание)

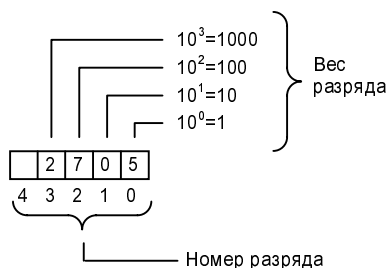
220	Ь	229	Е	238	О	247	Ч
221	Э	230	Ж	239	П	248	Ш
222	Ю	231	З	240	Р	249	Щ
223	Я	232	И	241	С	250	Ъ
224	А	233	Й	242	Т	251	Ы
225	Б	234	К	243	У	252	Ь
226	В	235	Л	244	Ф	253	Э
227	Г	236	М	245	Х	254	Ю
228	Д	237	Н	246	Ц	255	Я

ПРИЛОЖЕНИЕ 3

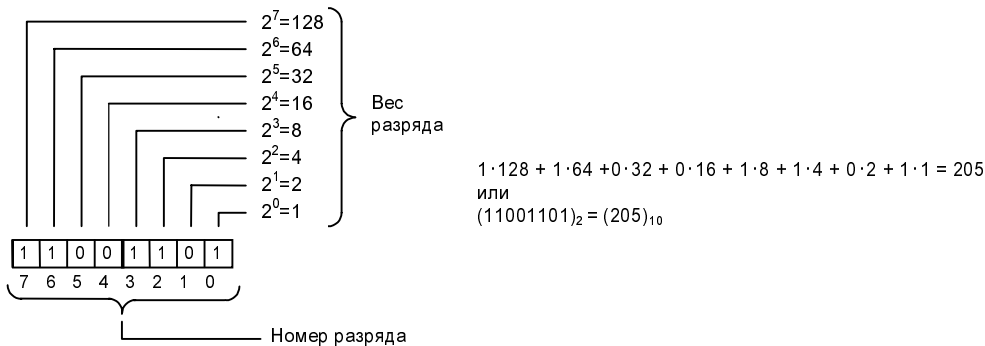
Представление информации в компьютере

Десятичные и двоичные числа

1. В обыденной жизни человек имеет дело с десятичными числами. В *десятичной системе* счисления для представления чисел используются цифры от 0 до 9. Значение числа определяется как сумма произведений цифр числа на весовой коэффициент, определяемый местом цифры в числе. Весовой коэффициент самой правой цифры равен единице, цифры перед ней — десятки, затем ста и т. д. Например, число 2703 равно $2 \cdot 1000 + 7 \cdot 100 + 0 \cdot 10 + 3 \cdot 1$.
2. Если места цифр пронумеровать справа налево и самой правой позиции присвоить номер "ноль", то можно заметить, что вес i -го разряда равен i -й степени десяти.



3. Для внутреннего представления чисел компьютер использует *двоичную систему* счисления. Двоичные числа записываются при помощи двух цифр — нуля и единицы. Как и десятичная, двоичная система позиционная. Весовой коэффициент i -го разряда равен двум в i -й степени.



Память компьютера

1. Память компьютера состоит из ячеек (битов). Каждый бит может хранить одну двоичную цифру. Следовательно, значением бита может быть ноль или единица. Восемь битов объединены в байт. Максимальное число, которое можно записать при помощи восьми двоичных цифр, — это 11111111, что соответствует десятичному числу 255, минимальное — ноль. Поэтому значением байта может быть число от нуля до 255.
2. Память используется для хранения переменных. Так как переменные различных типов могут принимать различные значения, то для их хранения нужно разное количество памяти. Память под переменные выделяется целым числом байтов. Например, значением переменной типа `char` может быть любой из 256 символов. Поэтому для хранения переменной этого типа достаточно одного байта. Значением переменной типа `integer` может быть число от $-32\,768$ до $32\,767$ (65 535 значений), для хранения переменной этого типа требуется два байта. Очевидно, что чем больше диапазон значений типа, тем больше байтов нужно для хранения переменной этого типа (см. табл.).

Тип переменной	Занимаемая память (количество байтов)	Диапазон значений
Char	1	Любой символ
String	256	Строка до 256 символов
String[n]	1×n	Строка до n символов
Byte	1	0..255
Word	2	0..65 535

(окончание)

Тип переменной	Занимаемая память (количество байтов)	Диапазон значений
Integer	2	−32 768..32 767
Longint	4	−2 147 483 648..2 147 483 647
Real	6	2,9e-39 .. 1,7e38
Single	4	1,5e-45 .. 3,4e38
Double	8	5,0e-324 .. 1,7e308
Extended	8	3,4e-4932 .. 1,1e4932

- В программе для хранения одного и того же значения можно использовать переменные разных типов (при этом будет применяться разное количество памяти). Например, если в программе используется переменная `Day`, содержащая число месяца, то для нее можно задать тип `byte`, `integer` или `longint`. В первом случае будет занят один байт памяти, во втором — два, в третьем — четыре. Но реально будет использоваться только один байт, а остальные будут только заняты. Поэтому, выбирая тип для переменной, следует подбирать наиболее подходящий тип для каждой конкретной ситуации. Особо необходимо обращать внимание на описание строковых переменных и массивов.
- Выделяя память для строковых переменных, следует помнить, что если не указана предельная длина строки, то переменной выделяется 256 байтов. Объявляя переменную, предназначенную, например, для хранения имени человека, следует писать `name:string[30]`, а не `name:string`.
- Каждому массиву программы выделяется память, объем которой определяется как типом элементов массива, так и их количеством. Для хранения двумерного массива, например, 20×20 , вещественных чисел нужно более трех килобайт памяти ($20 \times 20 \times 8 = 3200$).
- Память компьютера кажется неограниченной, но если ее использовать нерационально, то в некоторый момент может возникнуть ситуация, связанная с нехваткой памяти.

ПРИЛОЖЕНИЕ 4

Рекомендуемая дополнительная литература

1. Вирт Н. Алгоритмы и структуры данных. Пер. с англ. — М.: Мир, 1989. — 360 с.: ил.
2. Зелковиц М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. Пер. с англ. — М.: Мир, 1982. — 386 с.: ил.
3. Практическое руководство по программированию. Пер. с англ. Б. Мик, П. Хит, Н. Рашби и др.; под ред. Б. Мика, П. Хит, Н. Рашби. — М.: Радио и связь, 1986. — 168 с.: ил.
4. Фокс Дж. Программное обеспечение и его разработка. Пер. с англ. — М.: Мир, 1985. — 368 с.: ил.
5. Язык компьютера. Пер. с англ. под ред. и с предисл. В. М. Курочкина. — М.: Мир, 1989. — 240 с.: ил.
6. Гринзоу Лу. Философия программирования для Windows 95/NT. Пер. с англ. — СПб.: Символ-Плюс, 1997. — 640 с.: ил.

ПРИЛОЖЕНИЕ 5

Описание дискеты

На сопроводительной дискете находится файл Проекты.zip, который содержит проекты, приведенные в книге в качестве примеров.

Скопируйте файл архива во временный каталог или в каталог проектов Delphi и, используя один из стандартных архиваторов, например WinZip, выполните разархивацию. При разархивации файла при помощи WinZip необходимо указать имя диска, на который будет проводиться разархивация. В процессе разархивации на указанном диске будет создан каталог **Проекты**, в подкаталогах которого (табл. П5.1) будут помещены файлы проектов. Если кроме имени диска будет указан еще какой-либо каталог (папка), то каталог **Проекты** будет создан внутри каталога, указанного при разархивации файла. Для работы с проектом надо запустить Delphi и открыть файл проекта (dpr-файл).

Таблица П5.1. Проекты

Проект (каталог)	Краткое описание	В книге
База данных Школа	База данных Школа . Проект school — демонстрирует работу с базой данных в режиме таблицы, проект school2 — выборку информации из базы данных, проект school3 — использование динамического псевдонима. Подкаталог DATA содержит файл данных	Глава 17
Бинарный поиск в массиве	Демонстрация применения алгоритма бинарного поиска, использование компонента CheckBox	Глава 5
Ввод из Мемо	Демонстрация использования компонента Мемо для ввода массива строк	Глава 5

Таблица П5.1 (продолжение)

Проект (каталог)	Краткое описание	В книге
Ввод массива	Демонстрирует ввод и обработку массивов целых (getar.dpr) и дробных (getar1.dpr) чисел, использование компонента <code>StringGrid</code>	Глава 5
Вывод массива	Демонстрирует вывод массива в виде пронумерованного списка	Глава 5
График	Вычерчивает график функции. Демонстрирует использование свойства <code>Pixels</code> , обработку событий <code>OnPaint</code> и <code>OnFormResize</code>	Глава 10
Дача	Вычисляет стоимость поездки на дачу. Демонстрирует использование функции программиста	Глава 6
Два самолета	Демонстрирует использование битовых образов для вывода иллюстраций, свойства <code>TransparentColor</code>	Глава 10
Движ. окр.	Демонстрирует принципы реализации простой мультипликации и использования компонента <code>Timer</code> для задания временных интервалов	Глава 10
Динамический список 1	Демонстрирует создание и вывод неупорядоченного динамического списка	Глава 8
Динамический список 2	Демонстрирует создание и вывод упорядоченного динамического списка	Глава 8
Динамический список 3	Демонстрирует операции добавления и удаления элементов динамического упорядоченного списка	Глава 8
Добавление записи в файл	Демонстрирует процесс добавления записи в файл, использование компонентов <code>ComboBox</code> , <code>RadioButton</code> и <code>RadioGroup</code>	Глава 8
Запись-добавление в файл	Демонстрирует процессы создания нового файла и добавления информации в существующий файл, использование компонента <code>Мемо</code>	Глава 7
Звезды	Рисует на поверхности формы, в точке, в которой пользователь нажал кнопку мыши, контур звезды. Демонстрирует использование процедуры <code>PolyLine</code> , использование процедуры обработки события <code>OnMouseDown</code> для получения координаты точки, в которой нажата кнопка мыши	Глава 10

Таблица П5.1 (продолжение)

Проект (каталог)	Краткое описание	В книге
Звуки Windows	Демонстрация использования компонента <code>MediaPlayer</code> для воспроизведения звукового (WAV) файла	Глава 11
Использование <code>Animate</code>	Демонстрация использования компонента <code>Animate</code> для вывода анимации пользователя, находящейся в AVI-файле	Глава 11
Использование <code>Nhopen</code>	Демонстрация использования <code>ActiveX</code> компонента <code>Nhopen</code> для вывода справочной информации, находящейся в СНМ-файле	Глава 14
Использование <code>MediaPlayer</code>	Демонстрация использования компонента <code>MediaPlayer</code> для воспроизведения сопровождаемой звуком анимации (AVI-файла)	Глава 11
Использование <code>TRY</code>	Демонстрация обработки исключения (ошибки времени выполнения программы) при помощи инструкции <code>try ... except</code>	Глава 13
Квадратное уравнение	Решение квадратного уравнения. Демонстрирует использование процедуры программиста	Глава 6
Кисть	Демонстрирует стили закраски областей	Глава 10
Компонент	Пример компонента программиста (<code>nkedit.pas</code>), программа тестирования компонента (<code>tstNkEdit.dpr</code>) и использующая компонент <code>NkEdit</code> программа (<code>fazenda.dpr</code>)	Глава 16
Консоль	Пример консольного (DOS) приложения (пересчет веса из фунтов в килограммы). Демонстрирует работу со строками, преобразование кодировки символов	Глава 4
Контроль веса	Вычисление оптимального веса. Пример реализации множественного выбора с использованием вложенных инструкций <code>if</code>	Глава 2
Кривая Гильберта	Строит рекурсивную кривую Гильберта	Глава 12
Модуль	Пример модуля программиста. Модуль <code>my_unit</code> содержит функции <code>IsInt</code> и <code>IsFloat</code>	Глава 6
Мультик	Демонстрация создания кадрового мультипликации	Глава 10
Олимпиада	Пример использования (ввод, сортировка, вывод) двумерного массива и компонента <code>StringGrid</code>	Глава 5

Таблица П5.1 (продолжение)

Проект (каталог)	Краткое описание	В книге
Петербург	База данных "Архитектурные памятники Санкт-Петербурга". Подкаталог <code>DATA</code> содержит файл данных (<code>monuments.db</code>) и файлы иллюстраций	Глава 17
Погода	Простая база данных "Погода". Демонстрация обработки ошибок, возникающих при работе с файлами	Глава 7
Поиск в массиве (перебором)	Демонстрирует алгоритм поиска в массиве методом перебора	Глава 5
Поиск маршрута	Демонстрирует использование рекурсивной функции для поиска пути между двумя точками графа	Глава 12
Поиск минимального маршрута	Демонстрирует использование рекурсивной функции для поиска минимального (кратчайшего) пути между двумя точками графа	Глава 12
Поиск минимального элемента массива	Пример программы. Поиск минимального элемента массива чисел	Глава 5
Поиск файла (рекурсия)	Демонстрирует использование механизма рекурсии для поиска файла на диске, использование функции <code>SelectDirectory</code> для выбора каталога и работу со строками <code>WideChar</code>	Глава 12
Покупка	Вычисляет стоимость покупки. Демонстрирует использование компонентов <code>Edit</code> , <code>Label</code> , <code>Button</code> ; применение процедуры обработки события <code>OnKeyPress</code> для фильтрации символов, вводимых в поле <code>Edit</code>	Глава 1
Полиморфизм	Иллюстрирует работу с объектами программиста и понятие "Полиморфизм"	Глава 9
Просмотр AVI	Демонстрирует покадровый и непрерывный просмотр AVI-анимации, использование компонента <code>Animate</code>	Глава 11
Просмотр иллюстраций	Обеспечивает просмотр <code>bmp</code> -иллюстраций, использование функций <code>FindFirst</code> и <code>FindNext</code>	Глава 10
Простое число	Пример программы. Проверяет, является ли число простым. Демонстрирует использование инструкции <code>repeat</code>	Глава 2
Разговор	Пример программы. Вычисление стоимости телефонного разговора. Пример использования инструкции <code>if</code>	Глава 2

Таблица П5.1 (продолжение)

Проект (каталог)	Краткое описание	В книге
Рубль	Дописывает слово рубль после числа. Демонстрирует использование инструкции <code>case</code> для реализации множественного выбора; компонента <code>ListBox</code>	Глава 2
Самолет	Демонстрирует использование битовых образов для создания сложной мультипликации (летающий над городом самолет). <code>arplane.dpr</code> — загрузка битового образа из файла, <code>arplane1.dpr</code> — загрузка битового образа из ресурса	Глава 10
Сапер	Игра Сапер 2001. Демонстрирует работу с массивами, использование графики, рекурсии, ActiveX-компонента <code>hlopen</code>	Глава 15
Сетка	Выводит на поверхность формы координатные оси и оцифрованную сетку. Демонстрирует процесс вычерчивания различных по стилю линий, использование функции <code>TextOut</code>	Глава 10
Скорость бега	Вычисляет скорость, с которой спортсмен пробежал дистанцию. Демонстрирует использование компонентов <code>Edit</code> , <code>Label</code> , <code>Button</code> ; применение процедуры обработки события <code>OnKeyPress</code> для фильтрации символов, вводимых в поле <code>Edit</code>	Введение
Сортировка массива выбором	Демонстрирует алгоритм сортировки массива по возрастанию путем выбора наименьшего элемента	Глава 5
Сортировка массива обменом	Демонстрирует алгоритм сортировки массива методом обмена (пузырька)	Глава 5
Справочная система	Пример справочной системы для программы Квадратное уравнение . Каталог содержит исходный файл документа справочной системы (<code>rtf</code> -файл), файл проекта справочной системы (<code>hprj</code> -файл) и файл справочной системы (<code>hlp</code> -файл)	Глава 14
Таблица символов	Выводит таблицу кодировки символов русского алфавита. Демонстрирует работу с символами, использование вложенных циклов <code>for</code>	Глава 3
Тест, версии 1 и 2	Пример программы. Проверка знаний. Версия 2 демонстрирует динамическое создание компонентов	Глава 15

Таблица П5.1 (окончание)

Проект (каталог)	Краткое описание	В книге
Титаник	Демонстрация использования метода базовой точки для построения и перемещения сложного изображения	Глава 10
Факториал	Пример рекурсивной функции "Факториал"	Глава 12
Фунты	Пересчет веса из фунтов в килограммы. Демонстрирует использование: инструкции <code>case</code> для реализации множественного выбора; компонента <code>ListBox</code>	Глава 2
Фунты-килограммы	Пример программы. Пересчет веса из фунтов в килограммы	Глава 1
Фунты-килограммы 1	Демонстрация использования компонента <code>MediaPlayer</code> для воспроизведения звукового (WAV) файла без участия пользователя	Глава 11
Число ПИ	Вычисление числа "Пи" с заданной точностью. Пример использования инструкции <code>while</code>	Глава 2
Чтение записей из файла	Демонстрация процесса чтения из файла и вывода в поле <code>Мемо</code> записей, удовлетворяющих заданному условию. <u>Замечание.</u> Файл данных (<code>medals.db</code>) создается программой Добавление записи в файл	Глава 8
Чтение из файла	Демонстрирует использование функции <code>EOF</code> в процессе чтения строк из файла	Глава 7

Предметный указатель

D

DecimalSeparator 444

F

FindNext 312

H

Help-компилятор 360

I

Image Editor 291, 447

M

Macromedia Flash 319

Microsoft Help Workshop 360

S

SQL-запрос 490

W

Windows Help (winhlp32) 367

A

Алиас (Alias) 463

Б

База данных:

◊ локальная 459

◊ удаленная 460

Бинарный поиск 150

Битовый образ 279

В

Ввод:

◊ записи 213

◊ из файла 200

◊ с клавиатуры 125

Виртуальный метод 245

Вывод:

◊ записи 213

◊ на экран 123

Г

Графический примитив 259

Д

Деструктор 239

З

Запись 210, 461

И

Идентификатор раздела справки 365

Иллюстрация:

- ♦ добавление к форме 392

- ♦ загрузка из файла 393

Инкапсуляция 240

Инструкция:

- ♦ Goto 113

- ♦ Read, Readln 125

- ♦ Repeat ... Until 110

- ♦ try 347

- ♦ While 107

- ♦ with 212

- ♦ write, writeln 123, 192

- ♦ безусловного перехода 113

- ♦ ввода 125

- ♦ выбора 84

Исключение:

- ♦ обработка 346

К

Класс 237

Компонент:

- ♦ ActiveX 380

- ♦ Animate 301

- ♦ CheckBox 152

- ♦ Database 475

- ♦ Datasource 475

- ♦ DBEdit 478

- ♦ DBGGrid 486

- ♦ DBMemo 478

- ♦ DBNavigator 479

- ♦ DBText 478

- ♦ Hhopen 380

- ♦ Image 392

- ♦ ListBox 95

- ♦ MainMenu 419

- ♦ MediaPlayer 307

- ♦ Memo 141

- ♦ Query 489

- ♦ Table 475

- ♦ Timer 283

- ♦ динамический 406

- ♦ программиста 439

Конструктор 238

Л

Линия:

- ♦ стиль 253

- ♦ толщина 253

- ♦ цвет 253

М

Массив:

- ♦ ввод 134

- ♦ вывод 133

- ♦ компонентов 406

- ♦ многомерный 162

- ♦ объявление 131

- ♦ ошибки при использовании 169

- ♦ поиск максимального элемента 145

- ♦ поиск минимального элемента 145

- ♦ поиск элемента 148

- ♦ сортировка 157

Метод:

- ♦ Arc 266

- ♦ CopyRect 297

- ♦ Create 238

- ♦ Draw 279

- ♦ Ellipse 265

- ♦ FillRect 268

- ♦ Free 239

- ♦ LineTo 259

- ♦ LoadFromFile 275, 393

- ♦ LoadFromResourceName 294

- ♦ MoveTo 260

- ♦ Pie 269

- ♦ Polygon 268

- ♦ PolyLine 262

- ♦ Rectangle 267

- ♦ TextOut 257

- ♦ класса 240

О

Область:

- ♦ тип закрашки 255

Объект 238

Объявление:

- ♦ записи 211

- ♦ файла 191

Оператор:

- ♦ # 116

- ♦ AND 83

- ♦ NOT 83

продолжение рубрики см. на с. 526

Оператор (*прод.*):

- ◊ OR 83
- ◊ логический 83
- Отладка по шагам 350
- Отладчик 349
- Ошибка открытия файла 195
- Ошибка времени выполнения:
- ◊ локализация 345
- ◊ устранение 345

П

Пакет компонентов (DPK) 447

Переменная:

- ◊ динамическая 224
- ◊ контроль значения 353
- ◊ указатель 223

Поле 462

Программирование
объектно-ориентированное 237

Процедура:

- ◊ Append 193
- ◊ AssignFile 192
- ◊ Close 197
- ◊ Rewrite 193

Псевдоним 463

Р

Редактор:

- ◊ образов 447
- ◊ пакета компонентов 455
- ◊ файла ресурсов 291

Рекурсия 325

Ресурс 291, 447

С

Свойство 240

- ◊ Brush 254
- ◊ Canvas 251
- ◊ HelpContext 367
- ◊ HelpFile 367
- ◊ Pen 252
- ◊ Pixels 270
- ◊ Stretch 393
- ◊ наследование 243
- ◊ объекта 240

Символы 115

Событие OnTimer 283

Событийная процедура динамического
компонента 408

Список:

- ◊ добавление в упорядоченный список 230
 - ◊ добавление элемента 226
 - ◊ односвязный 226
 - ◊ удаление элемента 233
- Ссылка на раздел справки 359

Т

Тип:

- ◊ char 115
- ◊ TBitMap 279
- ◊ TextFile 192
- ◊ TPoint 268
- ◊ TRect 268
- ◊ интервальный 209

Точка останова 351

◊ характеристики 352

Транзакция 460

Трассировка 350

Ф

Файл:

- ◊ rtf 357
- ◊ проекта справочной системы 361
- ◊ ресурсов компонента 447

Функция:

- ◊ Bounds 298
- ◊ Chr 116
- ◊ ExtractFilePatch 498
- ◊ FindFirst 312
- ◊ IOResult 195
- ◊ Ord 117
- ◊ ParamStr 498
- ◊ Rect 268
- ◊ Trunc 154

Ц

Цикл:

- ◊ Repeat ... Until 110
- ◊ While 107

Я

Язык:

- ◊ SQL 460
- ◊ структурированных запросов 460