Учебное пособие

Язык Pascal

и основы программирования в Delphi

А.Я. Архангельский

Учебное пособие

Язык Pascal
и основы программирования в Delphi

Допущено учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению 654600 "Информатика и вычислительная техника"

Москва
Издательство **БИНОМ**
2004

УДК 004.43
ББК 32.973.26-018.1
А87

А.Я. Архангельский

Язык Pascal и основы программирования в Delphi. Учебное пособие — М.: ООО «Бином-Пресс», 2004 г. — 496 с.: ил.

Книга является учебником по программированию, по языку Pascal и по основам программирования в Delphi. Предназначена для старших классов школ, лицеев, колледжей, младших курсов ВУЗов при изучении информатики и других смежных дисциплин. Может служить основой для самостоятельного обучения программированию и начального изучения Delphi. Отличается от большинства учебников по основам программирования тем, что использует не традиционную среду Turbo Pascal, а объектно-ориентированную версию языка Pascal — Object Pascal. И обучает разработке программ для Windows, а не для DOS.

В книге описаны разнообразные приемы программирования, начиная с простых традиционных алгоритмов обработки массивов и строк. Рассмотрены и более сложные задачи, включая, например, рекурсию, решение нелинейных уравнений, работу со списками, очередями, стеками. Рассмотрено также объектно-ориентированное программирование. Дается методика разработки прикладных программ для Windows. Уделяется внимание программированию баз данных.

Книга рассчитана на начинающих и не требует никаких предварительных сведений ни о языке Pascal, ни о Delphi.

ISBN 5-9518-0091-9

Содержание

От автора

О чем эта книга

Эта книга является учебником по программированию, по языку Pascal и по основам программирования в Delphi. Остановлюсь немного на каждой из этих трех составляющих, хотя их непросто отделить друг от друга.

Изучивший эту книгу освоит самые разнообразные приемы программирования, начиная с простых традиционных алгоритмов обработки массивов и строк. Рассмотрены и более сложные задачи, включая рекурсию и решение нелинейного уравнения, работу со списками, очередями, стеками. И, конечно, рассмотрено объектно-ориентированное проектирование, без которого трудно представить себе современное программирование.

В качестве языка программирования выбран Pascal — традиционный язык, с которого обычно начинается обучение. Но наряду с этим традиционным языком рассмотрена и его современная версия — Object Pascal. В ней присутствуют такие элементы любого современного языка, как классы, наследование, исключения. Без освоения всего этого аппарата обучение программированию не может быть полноценным.

В качестве среды разработки выбрана среда Delphi — творение той же фирмы Borland, которая в свое время разработала среду Turbo Pascal. На том, почему я отказался от традиционного обучения на Turbo Pascal и предпочел Delphi, я остановлюсь чуть позже. А пока несколько слов о тех, для кого предназначается эта книга. На мой взгляд, ее с успехом можно использовать в старших классах школ, лицеев, колледжей и на младших курсах ВУЗов при изучении информатики и других смежных дисциплин. Но с тем же успехом она может служить основой для самостоятельного обучения программированию и начального изучения Delphi. А освоив материал этой книги, каждый может решить для себя, интересно ли это ему и стоит ли продолжать обучение с помощью иной, более углубленной литературы.

Turbo Pascal или Delphi?

Я убежден, что в наше время не стоит начинать обучение программированию с традиционного языка Turbo Pascal. Конечно, при одной оговорке: если мощности компьютеров в дисплейном классе хватает, чтобы поставить Windows (хотя бы Windows 95) и Delphi. Если это возможно, то обучение программированию на Delphi имеет множество неоспоримых преимуществ перед ориентацией на Turbo Pascal. Прежде всего, это интереснее для учащихся. Ну, кто сейчас работает в DOS? Создание программ для DOS вызывает нынче у обучаемых недоумение и скуку. Другое дело создать самим программу для Windows! А ведь интерес — непременное условие успешного обучения.

Итак, интерес обучения — первое достоинство Delphi. Второе — можно существенно сократить изучение тех элементов языка Turbo Pascal, которые связаны

с вводом информации с экрана и выводом на экран. Не говоря уже о реализации в Turbo Pascal графического режима. Учащимся, на мой взгляд, совершенно зря забивают голову этими безнадежно устаревшими сведениями, которые им заведомо никогда не пригодятся. Конечно, есть и в наше время ситуации, когда надо создавать программы именно для DOS. Но создавать такие программы приходится только немногим профессионалам. И ориентируются они при этом не на Turbo Pascal. Кстати, создавать программы для DOS можно и в Delphi. Только вряд ли это стоит рассматривать в курсе основ программирования.

Так что если цель учебного курса — изучение основ программирования (языка Pascal), а не изучение Delphi, то схема организации курса, на мой взгляд, может быть следующей. Учащимся показывают, как создать в Delphi новый проект и сохранить его, как перенести на форму кнопку, окна редактирования Edit (для ввода/вывода простой информации), окно Memo (для ввода/вывода многострочной информации). И показывают, как написать обработчик щелчка на кнопке. Изучение всего этого займет не более часа — столько же, сколько изучение ввода/вывода в Turbo Pascal. А более ничего и не надо, чтобы начать обучение программированию. Далее можно читать тот курс программирования, который обычно читается с ориентацией на Turbo Pascal. А время, сэкономленное на изучении графики в Turbo Pascal, можно потратить на рассмотрение дополнительных компонентов и на графику в компоненте Chart, несоизмеримую по возможностям и красоте с графикой Turbo Pascal (на изучение Chart требуется не более 20 минут, намного меньше, чем на графику в Turbo Pascal).

Как построена эта книга

Последние годы я постоянно пишу книги о Delphi, и все никак не могу остановиться. Очень уж нравится мне эта система, позволяющая даже новичку за несколько минут (или часов) создать программу, которую пользователь не сможет отличить от сделанной профессионалом. А уж профессионалу Delphi позволят делать с компьютером, Windows и пользователем все что угодно.

Так что я с удовольствием изучаю все новые стороны этой необозримой системы проектирования и делюсь приобретаемыми знаниями с читателями. Но это все очень большие книги. А меня давно уговаривали написать еще и сравнительно небольшой учебник, который было бы удобно использовать при изучении Delphi в ВУЗах и старших классах школ, да и как пособие для самообучения. Я долго не поддавался этому нажиму, хотя и читаю несколько таких курсов. Но одно дело — чтение курса в дисплейном классе, и совсем другое дело — написание учебника. Чтение курса — это живое общение с аудиторией, возможность экспромта, если что-то из изложенного непонятно, или, наоборот, вызвало повышенный интерес. А учебник — это что-то сухое, излагающее непререкаемые истины и, увы, обычно скучное.

Так что, согласившись, в конце концов, писать учебник, я решил, что отойду от принятых канонов. Никакого теоретизирования, завершающегося небольшими иллюстративными примерами. Скорее, наоборот — прежде всего примеры, работа с системой, и только для пояснений немного теории. Примеры я старался подбирать интересные, но насколько это получилось — не мне судить.

В изложении материала данной книги естественным образом переплетаются вопросы программирования, описание языка и реализация приложений для

Windows с помощью Delphi. Изложение строится на основе последней на данный момент версии Delphi 7. Но ничего специфического для этой версии в книге не рассматривается. Все изложенное справедливо и для предшествовавших версий, вплоть до Delphi 4, а частично даже для Delphi 1. Так что неважно, какая версия стоит на том компьютере, с которым вы будете работать.

Книга состоит из трех частей. Первая из них представляет собой курс основ программирования и языка Pascal. Тем, кто не стремится к большему, можно ограничиться только этой частью. Но думаю, что вряд ли стоит отказываться от изучения второй части, посвященной методике разработки прикладных программ в Delphi. В этой части книги читатель научится проектировать современный, удобный, красивый интерфейс для своих программ. Без такого интерфейса, без сопутствующей ему графики любая программа остается полуфабрикатом, которым никто, кроме ее автора, пользоваться не будет.

Третья, относительно небольшая часть книги посвящена программированию работы с базами данных. Эта часть обычно отсутствует в традиционных курсах основ программирования. Но почти ни одна современная программа не обходится без баз данных. Та информация, которая в традиционных учебных задачах обычно хранится в файлах, гораздо естественнее и эффективнее может храниться в таблицах баз данных. Так что, на мой взгляд, обучение программированию без освоения хотя бы основ работы с базами данных будет мало эффективным и не позволит создавать программы на современном уровне.

Тем, кто учится

Начинать изучение этой книги можно практически с нуля. Впрочем, сейчас, вероятно, трудно найти человека, который не умеет, хотя бы в общих чертах, работать с Windows. Так что я считал, что такими начальными навыками читатель обладает. Но большего для начала изучения не требуется. А после изучения этой книги, как я надеюсь, читатель сможет грамотно проектировать прикладные программы самого разного назначения для Windows. Но сразу хотел бы предупредить: законченным специалистом читатель не станет. Особенно это относится к освоению такой мощной системы, как Delphi. В данной книге изложена только малая часть возможностей Delphi. Впрочем, я надеюсь, что ее изучение пробудит у многих интерес к программированию и к Delphi. Для таких любознательных читателей в конце книги дается перечень дополнительных источников информации — книг и сайтов в Интернете, которые позволят им углубить свои знания и стать ценными, востребованными специалистами. А потребность в таких специалистах очень велика.

В конце каждой главы книги имеются разделы «Некоторые итоги», «Вопросы для самопроверки» и «Задачи». Хотел бы обратить внимание на то, что в разделах «Некоторые итоги» не столько суммируются сведения, которые изложены в главе, сколько даются некоторые общие рекомендации, связанные с материалом главы. Советовал бы не пропускать их. А разделы «Задачи» содержат весьма немного заданий на разработку примеров с помощью материала, изложенного в соответствующей главе. Это связано с тем, что я не могу заочно сказать, кому какая область применения приемов программирования более интересна. Так что предлагаемые задачи — это скорее намек на то, что вы могли бы сделать. Если хотите действительно освоить программирование, придумывайте сами интересные для вас задачи и ре-

шайте их. Без такого самостоятельного и заинтересованного творчества освоить программирование невозможно.

Тем, кто учит

При написании книги я не ориентировался на какую-то одну общую программу курса, поскольку основы программирования в разных ВУЗах, школах, лицеях изучаются в очень разных объемах. Книга содержит много материала, и если изучать его весь, это займет немало времени. Однако мне кажется, из этого материала нетрудно скомпоновать курсы различного объема и глубины изучения. Если задачи курса ограничиваются только основами программирования и языком Pascal, можно использовать первую часть книги, в которой рассмотрены все эти вопросы. Причем, в зависимости от объема курса и в этой части ряд разделов можно пропустить. В этом случае остальные, нерассмотренные части книги можно использовать для самостоятельной работы учащихся, для проведения факультативных занятий, для организации работы в кружках, клубах и т.п.

Теперь о технической стороне обучения. Выбор версии Delphi для обучения зависит от мощности используемых компьютеров. Желательно, чтобы на компьютерах можно было поставить Windows 98, а лучше, конечно, Windows 2000 или XP. Тогда можно выбирать для обучения любую версию Delphi. Впрочем, стремиться к самым новым версиям не имеет смысла. Имеющиеся в них возможности все равно в рамках данной книги не затрагиваются, а каждая последующая версия предъявляет все большие требования к вычислительной мощности компьютера и, прежде всего, к памяти. Так что я советовал бы ставить версию, не более Delphi 6. Без особых потерь можно поставить и более старую версию, например, 5 или 4. Ну а если с мощностью компьютеров совсем плохо, то можно даже ограничиться Windows 95 и Delphi 1. Правда, в этом случае не все, что изложено в двух последних частях книги, можно будет реализовать. Но изучение основ программирования, составляющих содержание первой части книги, не пострадает.

Часть 1

Программирование на языке Object Pascal

Глава 1

Знакомство с интегрированной средой разработки Delphi

В этой главе

- *вы узнаете, что такое объектно-ориентированное программирование*
- *освоите основные операции, выполняемые в интегрированной среде разработки Delphi*
- *создадите свои первые приложения Delphi для Windows*
- *познакомитесь с несколькими компонентами библиотеки Delphi*

1.1 Что такое Delphi

Delphi — одна из самых мощных систем, позволяющих на самом современном уровне создавать как отдельные прикладные программы Windows, так и разветвленные комплексы, предназначенные для работы в корпоративных сетях и в Интернете. Это с точки зрения потребителя. А с точки зрения разработчика Delphi — это *система визуального объектно-ориентированного программирования*. Давайте разберемся, что это такое.

1.1.1 Объектно-ориентированное программирование

Объектно-ориентированное программирование (сокращенно ООП) — это в наше время совершенно естественный подход к построению сложных (и не очень сложных) программ и систем. Когда вы открываете любую программу Windows, вы видите окно с множеством кнопок, разделов меню, окон редактирования, списков и т.п. Все это *объекты*. Причем сами по себе они ничего не делают. Они ждут каких-то *событий* — нажатия пользователем клавиш или кнопок мыши, перемещения курсора и т.д. Когда происходит подобное событие, объект получает *сообщение* об этом и как-то на него реагирует: выполняет некоторые вычисления, разворачивает список, заносит символ в окно редактирования. Вот такая программа Windows и есть объектно-ориентированное прикладная программа. Кстати, о терминологии. Все, что вы будете создавать, это *прикладные программы*, так как системными программами (операционными системами) мы заниматься не будем. Но для краткости, как это принято в литературе, мы будем далее называть их просто *приложениями* (по-английски — application).

Приложение, построенное по принципам объектной ориентации — это не последовательность каких-то операторов, не некий жесткий алгоритм. Объектно-ориентированная программа — это совокупность объектов и способов их взаимодействия. Отдельным (и главным) объектом при таком подходе во многих случаях можно считать пользователя программы. Он же служит и основным, но не единственным, источником событий, управляющих приложением.

Попробуем разобраться с основным понятием ООП — объектом. Для начала можно определить объект как некую *совокупность данных и способов работы с ними*. Данные — это характеристики объекта. Пользователь и объекты программы должны, конечно, иметь возможность читать эти данные объекта, как-то их обрабатывать и записывать в объект новые значения.

Здесь важнейшее значение имеют *принципы инкапсуляции и скрытия данных*. Принцип скрытия данных заключается в том, что внешним объектам и пользователю прямой доступ к данным, как правило, запрещен. Делается это из двух соображений.

Во-первых, для надежного функционирования объекта надо поддерживать *целостность и непротиворечивость* его данных. Если не позаботиться об этом, то внешний объект или пользователь могут занести в объект такие неверные данные, что он начнет функционировать с ошибками.

Во-вторых, необходимо изолировать внешние характеристики объектов от особенностей внутренней реализации данных. Для внешних потребителей данных должен быть доступен только *пользовательский интерфейс* — описание того, какие имеются данные и функции, и как их использовать. А внутренняя *реализация* — это дело разработчика объекта. При таком подходе разработчик может в любой момент модернизировать объект, изменить структуру хранения и форму представления данных, но если при этом не затронут интерфейс, внешний потребитель этого даже не заметит. И, значит, во внешней программе и в поведении пользователя ничего не придется менять.

Как вы увидите в дальнейшем, разделение интерфейса и реализации используется не только в объектах, но и при построении программ и отдельных программных модулей.

Чтобы выдержать принцип скрытия данных, в объекте обычно определяются процедуры и функции, обеспечивающие все необходимые операции с данными: их чтение, преобразование, запись. Эти функции и процедуры называются *методами*, и через них происходит общение с данными объекта.

Совокупность данных и методов их чтения и записи называется *свойством*. Со свойствами вы будете иметь дело на протяжении всей этой книги. Свойства можно устанавливать в процессе проектирования, их можно изменять программно во время выполнения вашей прикладной программы. Причем внешне это все выглядит так, как будто объект имеет какие-то данные, например, целые числа, которые можно прочитать, использовать в каких-то вычислениях, заложить в объект новые значения данных. В процессе проектирования вашего приложения с помощью Delphi вы можете видеть значения некоторых из этих данных в окне Инспектора Объектов, можете изменять эти значения. В действительности все обстоит иначе. Все общение с данными происходит через методы их чтения и записи. Это происходит и в процессе проектирования, когда среда Delphi запускает в нужный момент эти методы, и в процессе выполнения приложения, поскольку компилятор Delphi незримо для разработчика вставляет в нужных местах программы вызовы этих методов.

Помимо методов, работающих с отдельными данными, в объекте имеются методы, работающие со всей их совокупностью, меняющие их структуру. Таким образом, объект является *совокупностью свойств и методов*. Но это пока нельзя считать законченным определением объекта, поскольку прежде, чем дать полное определение, надо еще рассмотреть взаимодействие объектов друг с другом.

Средой взаимодействия объектов (как бы силовым полем, в котором существуют объекты) являются *сообщения*, генерируемые в результате различных *собы-*

тий. События наступают, прежде всего, вследствие действий пользователя — перемещения курсора мыши, нажатия кнопок мыши или клавиш клавиатуры. Но события могут наступать и в результате работы самих объектов. В каждом объекте определено множество событий, на которые он может реагировать. В конкретных экземплярах объекта могут быть определены *обработчики* каких-то из этих событий, которые и определяют реакцию данного экземпляра объекта. К написанию этих обработчиков, часто весьма простых, и сводится, как будет видно далее, основное программирование при разработке *графического интерфейса пользователя* с помощью Delphi.

Теперь можно окончательно определить объект как *совокупность свойств и методов, а также событий, на которые он может реагировать*. Условно это изображено на рис. 1.1. Внешнее управление объектом осуществляется через обработчики событий. Эти обработчики обращаются к методам и свойствам объекта. Начальные значения данных объекта могут задаваться также в процессе проектирования установкой различных свойств. В результате выполнения методов объекта могут генерироваться новые события, воспринимаемые другими объектами программы или пользователем.

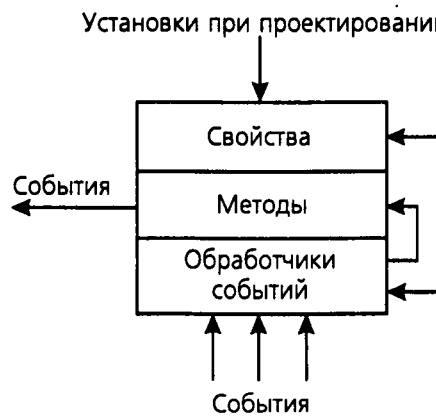


Рис. 1.1
Схема организации объекта

Представление о программе как о некоторой фиксированной совокупности объектов не является полным. Чаще всего сложная программа — это не просто какая-то предопределенная совокупность объектов. В процессе работы объекты могут создаваться и уничтожаться. Таким образом, структура программы является *динамическим образованием*, меняющимся в процессе выполнения. Основная цель создания и уничтожения объектов — экономия ресурсов компьютера и, прежде всего, памяти. Несмотря на бурное развитие вычислительной техники, память, наверное, всегда будет лимитировать возможности сложных приложений. Это связано с тем, что сложность программных проектов растет теми же, если не более быстрыми, темпами, что и техническое обеспечение. Поэтому от объектов, которые не нужны на данной стадии выполнения программы, нужно освобождаться. При этом освобождаются и выделенные им области памяти, которые могут использоваться вновь создаваемыми объектами. Простой пример этого — окно-заставка с логотипом, появляющееся при запуске многих приложений. После начала реального выполнения приложения эта заставка исчезает с экрана и никогда больше не появится

в данном сеансе работы. Было бы варварством не уничтожить этот объект и не освободить занимаемую им память для более продуктивного использования.

С целью организации динамического распределения памяти во все объекты заложены методы их создания — *конструкторы* и уничтожения — *деструкторы*. Конструкторы тех объектов, которые изначально должны присутствовать в приложении (прикладной программе), срабатывают при запуске программы. Деструкторы всех объектов, имеющихся в данный момент в приложении, срабатывают при завершении его работы. Но нередко и в процессе выполнения различные новые объекты (например, новые окна документов) динамически создаются и уничтожаются с помощью их конструкторов и деструкторов.

Включать объекты в свою программу можно двумя способами: *вручную* включать в нее соответствующие операторы (это приходится делать не очень часто) или путем *визуального программирования*, используя заготовки — *компоненты*.

1.1.2 Визуальное программирование интерфейса

Сколько существует программирование, столько существуют в нем и тупики, в которые оно постоянно попадает и из которых, в конце концов, доблестно выходит. Один из таких тупиков или кризисов был в свое время связан с разработкой графического интерфейса пользователя. Программирование вручную всяких привычных пользователю окон, кнопок, меню, обработка событий мыши и клавиатуры, включение в программы изображений и звука требовало все больше и больше времени программиста. В ряде случаев весь этот сервис начинал занимать до 80-90% объема программных кодов. Причем весь этот труд нередко пропадал почти впустую, поскольку через год — другой менялся общепринятый стиль графического интерфейса, и все приходилось начинать заново.

Выход из этой ситуации обозначился благодаря двум подходам. Первый из них — стандартизация многих функций интерфейса, благодаря чему появилась возможность использовать библиотеки, имеющиеся, например, в Windows. В частности, появился *API Windows* — интерфейс, в котором описано множество функций. От версии к версии Windows набор функций расширяется, внутреннее описание функций совершенствуется, но формы вызова функций не изменяются. В итоге, при смене стиля графического интерфейса (например, при переходе от Windows 3.x к Windows 95, а затем к Windows 2000 и XP) приложения смогли автоматически приспосабливаться к новой системе без какого-либо перепрограммирования. На этом пути создались прекрасные условия для решения одной из важнейших задач совершенствования техники программирования — *повторного использования кодов*. Однажды разработанные вами формы, компоненты, функции могли быть впоследствии неоднократно использованы вами или другими программистами для решения их задач. Каждый программист получил доступ к наработкам других программистов и к огромным библиотекам, созданным различными фирмами. Причем была обеспечена совместимость программного обеспечения, разработанного на разных алгоритмических языках.

Вторым революционным шагом, кардинально облегчившим жизнь программистов, явилось появление визуального программирования, возникшего в Visual Basic и нашедшего блестящее воплощение в Delphi и C++Builder фирмы Borland.

Визуальное программирование позволило свести проектирование пользовательского интерфейса к простым и наглядным процедурам, которые дают возможность

за минуты или часы сделать то, на что ранее уходили месяцы работы. В современном виде в Delphi это выглядит так.

Вы работаете в Интегрированной Среде Разработки (ИСР или Integrated development environment — IDE) Delphi. Среда предоставляет вам *формы* (в приложении их может быть несколько), на которых размещаются компоненты. Обычно это оконная форма, хотя могут быть и невидимые формы. На форму с помощью мыши переносятся и размещаются пиктограммы компонентов, имеющихся в библиотеках Delphi. С помощью простых манипуляций вы можете изменять размеры и расположение этих компонентов. При этом вы все время в процессе проектирования видите результат — изображение формы и расположенных на ней компонентов. Вам не надо мучиться, многократно запуская приложение и выбирая наиболее удачные размеры окна и компонентов. Результаты проектирования вы видите, даже не компилируя программу, немедленно после выполнения какой-то операции с помощью мыши.

Но достоинства визуального программирования не сводятся к этому. Самое главное заключается в том, что во время проектирования формы и размещения на ней компонентов Delphi автоматически формирует коды программы, включая в нее соответствующие фрагменты, описывающие данный компонент. А затем в соответствующих диалоговых окнах пользователь может изменить заданные по умолчанию значения каких-то свойств этих компонентов и, при необходимости, написать обработчики каких-то событий. То есть проектирование сводится, фактически, к размещению компонентов на форме, заданию некоторых их свойств и написанию, при необходимости, обработчиков событий.

Компоненты могут быть *визуальные*, видимые при работе приложения, и *невизуальные*, выполняющие те или иные служебные функции. Визуальные компоненты сразу видны на экране в процессе проектирования в таком же виде, в каком их увидит пользователь во время выполнения приложения. Это позволяет очень легко выбрать место их расположения и их дизайн — форму, размер, оформление, текст, цвет и т.д. Невизуальные компоненты видны на форме в процессе проектирования в виде пиктограмм, но пользователю во время выполнения они не видны, хотя и выполняют для него за кадром весьма полезную работу.

В библиотеки визуальных компонентов Delphi включено множество типов компонентов, и их номенклатура очень быстро расширяется от версии к версии. Имеющегося уже сейчас вполне достаточно, чтобы построить практически любое самое замысловатое приложение, не прибегая к созданию новых компонентов. При этом даже неопытный программист, делающий свои первые шаги на этом поприще, может создавать приложения, которые выглядят совершенно профессионально.

Компоненты библиотек Delphi и типы других объектов оформляются в виде *классов*. Классы — это типы, определяемые пользователем. В классах описываются свойства объекта, его методы и события, на которые он может реагировать. Язык программирования Object Pascal, который используется в Delphi, предусматривает только инструментарий создания классов. А сами классы создаются разработчиками программного обеспечения. Создатели Delphi уже разработали для вас множество очень полезных классов и включили их в библиотеки системы. Этими классами вы пользуетесь при работе в Интегрированной Среде Разработки. Впрочем, это нисколько не мешает создавать вам свои новые классы.

Если бы при создании нового класса вам пришлось все начинать с нуля, то эффективность этого занятия была бы под большим сомнением. Да и разработчики Delphi вряд ли создали бы в этом случае такое множество классов. Действительно, представьте себе, что при разработке нового компонента, например, какой-нибудь

новой кнопки, вам пришлась бы создавать все: рисовать ее изображение, описывать все свойства, определяющие ее место расположения, размеры, надписи и картинки на ее поверхности, цвет, шрифты, описывать методы, реализующие ее поведение — изменение размеров, видимость, реакции на сообщения, поступающие от клавиатуры и мыши. Вероятно, представив себе все это, вы отказались бы от разработки новой кнопки.

К счастью, в действительности все обстоит гораздо проще, благодаря одному важному свойству классов — *наследованию*. Новый класс может наследовать свойства, методы, события своего *родительского класса*, т.е. того класса, на основе которого он создается. Например, при создании новой кнопки можно взять за основу один из уже разработанных классов кнопок и только добавить к нему какие-то новые свойства или отменить какие-то свойства и методы родительского класса.

Благодаря визуальному объектно-ориентированному программированию была создана технология, получившая название *быстрая разработка приложений*, по-английски *RAD* — Rapid Application Development. Эта технология характерна для новых поколений систем программирования. Впервые она была реализована в среде Visual Basic. Но, пожалуй, именно в Delphi была создана в свое время наиболее удачная среда RAD, которая повлияла в дальнейшем на аналогичные разработки других фирм.

1.1.3 Задачи, решаемые с помощью Delphi

Коротко перечислим те задачи, которые вы можете решать с помощью Delphi:

- Быстрое создание профессионально выглядящего оконного интерфейса для приложений любой сложности и любого назначения: инженерных, офисных, бухгалтерских, информационно-поисковых. Интерфейс, созданный даже начинающим программистом, автоматически удовлетворяет всем требованиям Windows и настраивается на используемую операционную систему, поскольку использует многие функции, процедуры, библиотеки Windows. Так что, в какой бы области вы далее ни работали, умение создавать такие прикладные программы очень повысит ваш авторитет в глазах руководства.
- Создание современного пользовательского интерфейса для любых ранее разработанных программ DOS и Windows. Нередко в учреждении или фирме существуют и успешно эксплуатируются прикладные программы, разработанные в разное время, разными коллективами, для разных операционных систем. С помощью Delphi эти приложения можно снабдить современным удобным оконным интерфейсом, объединить разрозненные приложения в единую систему, обеспечить их стилистическое единство, наладить обмен информации между приложениями.
- Создание мощных систем работы с локальными и удаленными базами данных любых типов. Базы данных — хранилища информации любого вида используются практически во всех современных прикладных программах. Это могут быть данные о сотрудниках и структуре какой-то организации, сведения о наличии и движении каких-то товаров, сведения о технических характеристиках и производителях комплектующих изделий или приборов, экономическая информация и многое другое. Подход, используемый в Delphi, позволяет получить доступ к базам, созданным на любой платформе: InterBase, Microsoft Access, FoxPro, Paradox, dBase, Sybase, Microsoft SQL, Oracle и др.

- Создание баз данных различных типов с помощью инструментария Delphi.
- Формирование и печать из приложения сложных отчетов самого различного назначения, включающих тексты, таблицы и графики.
- Управление из своего приложения такими продуктами Microsoft, как Word, Excel, почтовые и другие программы, что позволяет использовать все их богатейшие возможности.
- Создание системы помощи (Help), как для своих приложений, так и для любых других, с которыми, в частности, можно работать просто через Windows.
- Использование самых современных технологий для разработки приложений, предназначенных для работы с Интернет.
- и многое другое.

В этом списке не перечислены многие дополнительные возможность Delphi. Список приведен не для того, чтобы перегружать вас излишней информацией. Его цель — показать, что изучать Delphi действительно стоит. Владение даже основами работы с Delphi существенно повысит вашу ценность как специалиста, независимо от сферы вашей деятельности.

1.2 Интегрированная среда разработки Delphi

1.2.1 Общий вид и настройка окна ИСР

Теперь, после изучения, возможно не очень интересного, но необходимого материала в разд. 1.1, вы готовы для работы с Delphi. Запустите Delphi, выбрав пиктограмму Delphi в разделе меню Windows Пуск | Программы. Когда вы щелкнете на пиктограмме Delphi, перед вами откроется основное окно Интегрированной Среды Разработки (рис. 1.2).

Интегрированная Среда Разработки (Integrated Development Environment — IDE, в дальнейшем мы будем использовать для нее аббревиатуру ИСР) — это среда, в которой есть все необходимое для проектирования, запуска и тестирования приложений, и где все нацелено на облегчение процесса создания программ. ИСР интегрирует в себе редактор кодов, отладчик, инструментальные панели, редактор изображений, инструментарий баз данных — все, с чем приходится работать.

В Delphi 7 вид окна ИСР представлен на рис. 1.2. Если вы работаете с другой версией Delphi, окно может несколько отличаться от приведенного на рис. 1.2. Но в общих чертах окна всех версий достаточно сходны.

В верхней части окна ИСР вы видите *полосу главного меню*. Ее состав несколько различается от версии к версии и, кроме того, зависит от варианта Delphi, с которым вы работаете.

Ниже полосы главного меню расположены две *инструментальные панели*. Левая панель (состоящая в свою очередь из нескольких панелей) содержит два ряда *быстрых кнопок*, дублирующих некоторые наиболее часто используемые команды меню. Правая панель содержит *палитру компонентов* библиотеки визуальных компонентов. Палитра состоит из ряда страниц, закладки которых видны в ее верхней части. Набор страниц и их состав зависит от версии и варианта Delphi, но все компоненты, с которыми мы далее будем иметь дело, имеются во всех версиях.

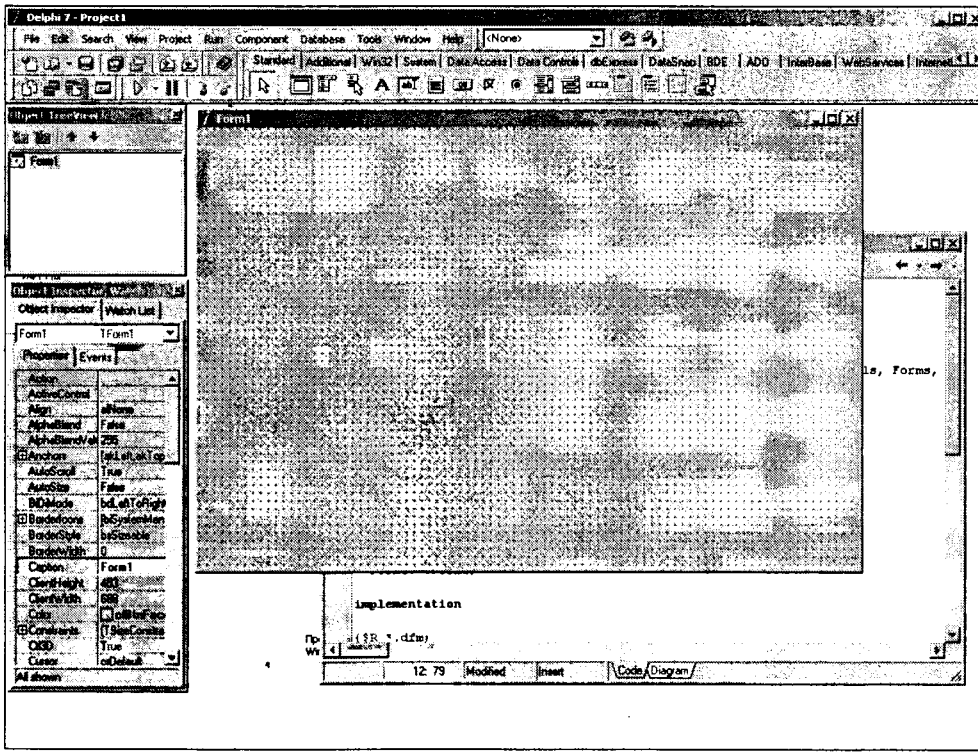


Рис. 1.2
Основное окно Интегрированной Среды Разработки в Delphi 7

Правее полосы главного меню в Delphi 7 и 6 размещена еще одна небольшая инструментальная панель, содержащая выпадающий список и две быстрые кнопки. Это панель сохранения и выбора различных конфигураций окна ИСР, которые вы сами можете создавать и запоминать.

В основном поле окна вы можете видеть слева два окна: вверху — Дерево Объектов (Object TreeView), под ним — Инспектор Объектов (Object Inspector). Окно Дерева Объектов отображает иерархическую связь компонентов и объектов вашего приложения. Мы не будем далее работать с этим окном. Поэтому советую вам закрыть его, чтобы освободить побольше места размещенному под ним окну Инспектора Объектов. Это основной инструмент, с помощью которого вы в дальнейшем будете задавать свойства компонентов и обработчики событий. Правее этих окон вы можете видеть окно пустой формы, готовой для переноса на нее компонентов. Под ним расположено окно Редактора Кодов. Часто оно при первом взгляде на экран невидимо, так как его размер равен размеру формы и окно Редактора Кодов практически полностью перекрывается окном формы. На рис. 1.2 это окно немного сдвинуто и выглядывает из-под окна формы.

Если вы в первый раз открыли окно ИСР Delphi, то советую проверить и, если требуется, произвести некоторые настройки, которые облегчат вашу дальнейшую работу.

Совет

Для удобной работы в ИСР Delphi полезно провести некоторые настройки.

Выполните команду Tools | Environment Options. В открывшемся многостраничном диалоговом окне перейдите на страницу Preferences и включите индикатор Project desktop группы опций автосохранения Autosave options. Тогда при каждом очередном запуске Delphi будет загружаться ваше последнее приложение предыдущего сеанса, и будут открываться все окна, которые были открыты в момент предыдущего выхода из Delphi. Это очень удобно, если вы намерены продолжать работу над тем же проектом.

Откройте в том же окне страницу Explorer и выключите на ней индикатор Automatically show Explorer. Это предотвратит автоматическое появление окна Исследователя Кода, которое вам пока не потребуется и только будет мешать, уменьшая площадь окна Редактора Кода.

Выполните команду Tools | Debugger Options. В открывшемся многостраничном диалоговом окне перейдите на страницу Language Exceptions и выключите индикатор Stop On Delphi Exceptions. Это предотвратит появление при ошибках в ваших программах окна сообщения, которое пока мало что вам даст, но зато легко смутит вашу душу, еще не окрепшую в борьбе с ошибками.

Все эти настройки достаточно выполнить только один раз. Delphi запомнит их, и в последующих сеансах они будут действовать автоматически.

Для облегчения процесса проектирования и отладки ваших будущих приложений полезно провести еще некоторые дополнительные настройки.

Совет

Выполните команду Project | Options. В открывшемся многостраничном окне перейдите на страницу Compiler и выключите индикатор Optimization. Это сделает ваши приложения несколько менее эффективными, но зато вы будете иметь меньше проблем с их отладкой.

В том же окне на странице Packages включите индикатор Built with runtime packages. Это включит так называемую поддержку пакетов времени выполнения. В результате размер выполняемых модулей ваших проектов уменьшится на порядок, что сэкономит немало места на диске.

Проведя эти настройки, включите в этом многостраничном окне индикатор Default, расположенный в левом нижнем углу и видимый на любых страницах окна. Включение этого индикатор обеспечит статус ваших установок как установок по умолчанию для всех ваших будущих проектов.

1.2.2 Создание нового проекта

Проще всего ознакомиться с основными инструментами ИСР Delphi можно в процессе реальной работы. Так что давайте приступим к построению вашего первого приложения. Но сначала нам надо создать для него новый проект.

Если вы работаете с Delphi 7 или 6, то для того, чтобы начать новый проект, выполните команду File | New, и в открывшемся каскадном меню выберите раздел Application. Если вы работаете с Delphi 5 или 4, то для создания нового проекта вам надо выполнит команду File | New Application. Возможно, вам будет задан вопрос, хо-

тите ли вы сохранить изменения в вашем предыдущем проекте. Если ничего стоящего в предыдущем проекте не было, ответьте на этот вопрос отрицательно — «No».

В результате у вас будет создан новый проект и откроется пустая форма.

Предупреждение

Начинающие работать с Delphi очень часто допускают следующую ошибку: вместо описанных выше команд используют для создания нового проекта команду File | New | Form в Delphi 7 и 6 или команду File | New Form в Delphi 4 и 5. Еще чаще выполняют эти же команды соответствующей быстрой кнопкой, о которой будет сказано далее. Эти команды создают не новый проект, а новую форму в существующем проекте. В результате вы испортите прежний проект, а ваша новая форма работать в нем все равно не будет.

Первое действие, которое я рекомендовал бы вам выполнить после создания нового проекта — сохранить его.

Хороший стиль программирования

После того как вы создали приложение с пустой формой, сразу сохраните его в нужном каталоге. И в течение работы над проектом почаще выполняйте сохранение.

Если вы начинаете работу с того, что сохраняете проект, и в дальнейшем регулярно повторяете сохранение, вы можете не опасаться любых неожиданностей типа сбоя компьютера или Delphi, вызванных техническими причинами или недопустимыми действиями вашего собственного еще не отлаженного приложения при его запуске. Но есть и еще аргумент в пользу немедленного сохранения проекта и модулей. В многооконных приложениях, которые будут рассмотрены значительно позднее, это позволяет сразу задать модулям имена, которые будут использоваться в программе для взаимных ссылок модулей друг на друга. Если вы не выполнили сразу сохранение формы, то будете вынуждены сначала ссылаться на ее имя по умолчанию, а в дальнейшем изменять эти ссылки.

Сохранить проект можно командой File | Save All. Удобно также использовать соответствующую быструю кнопку — на рис. 1.2 она четвертая слева на верхней панели быстрых кнопок (см. также приведенную в разд. 1.6 табл. 1.1).

При первом сохранении Delphi спросит у вас имя файла сохраняемого модуля, а затем — имя файла проекта. Дело в том, что каждый проект состоит из нескольких файлов: файл проекта с расширением *.dpr*, файл модуля формы с расширением *.pas*, и несколько других файлов, о которых будет сказано позднее. Имена этих дополнительных файлов совпадают с именем проекта или модуля. Так что при сохранении надо дать имена только файлу проекта и модуля. Но учтите, что Delphi не допускает одинаковых имен модулей и проектов. Поэтому задаваемые вами имена файлов должны быть разными.

Предупреждение

Не задавайте одинаковые имена различным файлам и вообще стремитесь не использовать в проекте повторяющиеся имена.

Хороший стиль программирования

Всегда при сохранении проектов и модулей задавайте осмысленные имена файлов.

Почему не стоит соглашаться при сохранении с именами, предлагаемыми Delphi по умолчанию — *Project1*, *Unit1* и т.п.? Во-первых, имя файла проекта будет в дальнейшем именем вашего выполняемого модуля. Наверное, не очень хорошо будет отдавать заказчику проект с таким странным названием, как *Project1*. Во-вторых, имя файла модуля будет и именем самого модуля. В проекте с одним модулем имя *Unit1*, возможно, не так уж и страшно. Но если у вас в проекте будет несколько модулей, то вряд ли вам будет удобно все время помнить, что такое *Unit1*, а что такое *Unit5*. Кроме того, представьте себе, что вы на некоторое время прекратили работу над своими проектами, а потом опять ее возобновили. Или *сопровождение* ваших проектов (их последующая модификация и совершенствование) осуществляется не вами, а кем-то другим. Вы уверены, что легко будет разобраться в многочисленных *Project1* и *Unit1* непонятного назначения?

Хороший стиль программирования

При разработке простых прототипов проектов удобно задавать имена файлов проектов и модулей одинаковыми, различая их только префиксом «P» или «U» соответственно. Например, файл проекта — *PEditor1*, файл модуля — *UEditor1*. Это упрощает понимание того, какие файлы к какому варианту проекта относятся, поскольку на начальных стадиях проектирования у вас может появиться несколько альтернативных вариантов одного и того же проекта.

Так что в нашем примере можете дать сохраняемым файлам имена, например, *PSimple* и *USimple* (в любом случае пишите имена латинскими символами).

И еще один важный совет.

Хороший стиль программирования

Заведите себе за правило отводить для каждого нового проекта новый подкаталог (папку Windows). Удобная структура каталогов существенно облегчает работу над проектами.

Связан этот совет с тем, что если помещать несколько проектов в один каталог, то и вы, и, возможно, Delphi скоро запутаетесь в том, какие файлы к какому проекту относятся. Размещение проектов в разных каталогах избавит вас в дальнейшем от многих неприятностей.

Если у вас возникает несколько вариантов выполнения проекта, а обычно так и бывает, то желательно внутри каталога проекта создавать подкаталоги для каждого варианта. Удобная структура каталогов существенно облегчает работу над серьезными проектами.

Новый каталог вы можете создать средствами Windows перед началом проекта, или создавать его можно в окне стандартного диалога сохранения файлов с помощью соответствующей быстрой кнопки.

Итак, рекомендации по созданию нового проекта сводятся к следующему.

- Создайте новый каталог для своего нового проекта.
- Создайте новый проект командой File | New | Application или File | New Application.
- Сразу сохраните проект и файл модуля командой File | Save All.

В последующих сеансах работы вы можете открыть сохраненный проект командой File | Open Project. Но если вы работали с проектом недавно, то много удобнее открыть его командой File | Reopen. А еще удобнее воспользоваться стрелочкой рядом с соответствующей этой команде быстрой кнопкой — эта кнопка вторая слева на верхней панели быстрых кнопок (см. рис. 1.2 или приведенную в разд. 1.6 табл. 1.1). В обоих случаях открывается окно, показанное на рис. 1.3. В его верхней части вы легко найдете несколько проектов, с которыми работали в последнее время, а в нижней части — ряд файлов, с которыми вы работали.

Предупреждение

Не путайте: проект можно выбирать только в верхней части окна рис. 1.3. Если вы выберете имя из нижней части окна (очень частая ошибка у начинающих!), то просто откроете файл модуля, который никак не будет связан с вашим проектом. И потом долго будете недоумевать, почему все, что вы делаете с открывшейся формой, не видно, когда вы выполняете ваш проект.

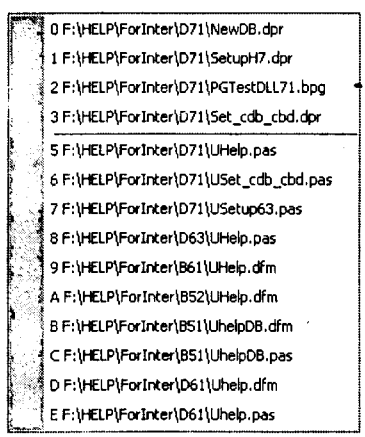


Рис. 1.3
Окно предшествующих проектов и файлов

Повторю, что имеется еще более удобный способ автоматически открывать при загрузке Delphi тот проект, с которым вы работали в предыдущем сеансе. Для этого надо выполнить настройку, описанную в разд. 1.2.1.

Совет

Начинающие работать с Delphi часто закрывают окна, которые им в данный момент мешаются: форму, Редактор Кода, Инспектор Объектов. Не надо этого делать. Но если все-таки по ошибке вы закрыли то или иное окно, его можно сделать видимым командами View | Object Inspector (окно Инспектора Объектов), View | Units (окно Редактора Кода), View | Forms (окно формы).

1.2.3 Компоновка формы и задание свойств компонентов

Итак, вы создали новый проект и сохранили его. Теперь давайте спроектируем в нем приложение. Пусть окно этого приложения должно иметь кнопку, при щелчке на которой будет появляться некоторая надпись. Так что вам следует поместить на форму проекта требуемые компоненты. Для этого, конечно, надо, чтобы форма было видна. Но она может оказаться у вас загороженной окном Редактора Кода. В этом случае щелкните на быстрой кнопке Toggle Form/Unit — она третья слева на нижней панели быстрых кнопок (см. рис. 1.2, а также приведенную в разд. 1.6 табл. 1.1). Каждый щелчок на этой кнопке переключает фокус между формой и окном Редактора Кодов с загруженным в нем текстом модуля, соответствующего форме.

Компоненты переносятся на форму из палитры компонентов, которую вы можете видеть вверху справа на рис. 1.2. Палитра позволяет сгруппировать компоненты в соответствии с их смыслом и назначением. Эти группы или *страницы* снабжены закладками.

Поскольку число предопределенных компонентов, конечно, возрастает от версии к версии, то наиболее полной является библиотека Delphi 7. Палитра этой библиотеки приведена на рис. 1.4.

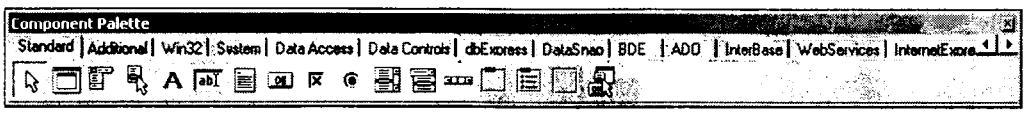


Рис. 1.4
Палитра компонентов в Delphi 7

Поскольку число страниц в Delphi 7 велико и не все закладки видны на экране одновременно, в правой части палитры компонентов имеются две кнопки со стрелками, направленными влево и вправо. Эти кнопки позволяют перемещать отображаемую на экране часть палитры. На некоторых страницах расположено столько компонентов, что они не помещаются в видимой части страницы. В этих случаях на концах страницы появляются дополнительные кнопочки, позволяющие перемещаться вдоль страницы.

Чтобы перенести компонент на форму, надо открыть соответствующую страницу библиотеки и указать курсором мыши необходимый компонент. При этом кнопка-указатель , размещенная в левой части палитры компонентов, приобретет вид нажатой кнопки . Это значит, что вы находитесь в состоянии, когда собираетесь поместить компонент на форму. Поместить выбранный компонент на форму очень просто — надо сделать щелчок мышью в нужном месте формы.

Совет

Начинающие работать с Delphi часто не просто щелкают на форме при переносе компонентов, а обводят мышью рамку, задающую размер компонента. В большинстве случаев не стоит это делать. Задаваемые по умолчанию размеры кнопок, простых окон редактирования и многих других компонентов хорошо продуманы с точки зрения удобства и эстетики. Так что при переносе компонентов на форму лучше ограничиться простым щелчком на форме. Если потребуется, вы всегда сможете потом изменить размеры компонентов.

Есть и другой способ поместить компонент на форму — достаточно сделать двойной щелчок на пиктограмме компонента, и он автоматически разместится в центре вашей формы.

Если вы выбрали компонент, а затем изменили ваше намерение размещать его, вам достаточно нажать кнопку указателя слева на палитре. Это прервет процесс размещения компонента, и программа вернется в нормальный режим, в котором вы можете выбирать другой компонент или выполнять какую-то команду.

Удалить ошибочно перенесенный на форму компонент очень просто: выделите его и нажмите клавишу Delete.

Имена компонентов, соответствующих той или иной пиктограмме, вы можете узнать из ярлычка, появляющегося, если вы задержите курсор мыши над этой пиктограммой в палитре компонентов. Если вы выберете в палитре компонент и нажмете клавишу F1, то вам будет показана справка по типу данного компонента. Конечно, пояснения будут на английском языке, что может вызвать некоторые трудности. Однако если вы установили на компьютере справки [3], то для большинства компонентов появятся темы и русских справок, и вы сможете выбрать по желанию английскую или русскую. Тут надо сразу сделать одно замечание. Имена на ярлычках выглядят, например, так: **MainMenu**, **Button** и т.д. Однако в Delphi все имена классов в действительности начинаются с символа «T», например, **TMainMenu**, **TButton**. Под такими именами вы можете найти описания соответствующих компонентов во встроенной в Delphi справочной системе или в русской справке [3].

В нашем примере вам надо перенести на форму кнопку и метку — это компонент, отображающий заданный в нем текст. Для кнопки выберите компонент **Button** со страницы Standard. Выделите пиктограмму кнопки (она седьмая слева на рис. 1.4 с надписью «OK») и затем щелкните курсором мыши в нужном вам месте формы. На форме появится кнопка, которой Delphi присвоит имя по умолчанию — **Button1**. Аналогичным образом перенесите на форму с той же страницы Standard палитры компонентов метку **Label** (она на странице четвертая слева). В этой метке в процессе выполнения приложения будет появляться текст при нажатии пользователем кнопки. Delphi присвоит ей имя **Label1**.

Разместите компоненты на форме примерно так, как показано на рис. 1.5 а. При этом уменьшите до разумных размеров окно формы, так как в вашем первом приложении никаких других компонентов не будет.

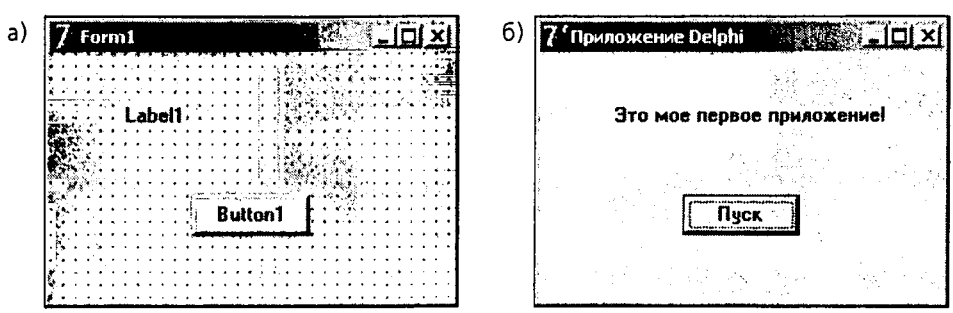


Рис. 1.5
Форма (а) и окно в процессе выполнения (б) вашего приложения

Теперь можно задать некоторые свойства компонентов. Для этого используется Инспектор Объектов (см. разд. 1.2.1). В нем отображаются свойства того объекта, который выделен на форме. Например, щелкните один раз (одинарный, не двойной щелчок!) на форме. Тогда в окне Инспектора Объектов отобразятся свойства формы. Точнее, это окно имеет две страницы, показанные на рис. 1.6: Properties (свойства) и Events (события). В верхней части окна имеется выпадающий список всех компонентов, размещенных на форме, включая саму форму. В нем вы можете выбрать тот компонент, свойства и события которого вас интересуют. Так что можно этот компонент и не выделять предварительно щелчком мыши на форме.

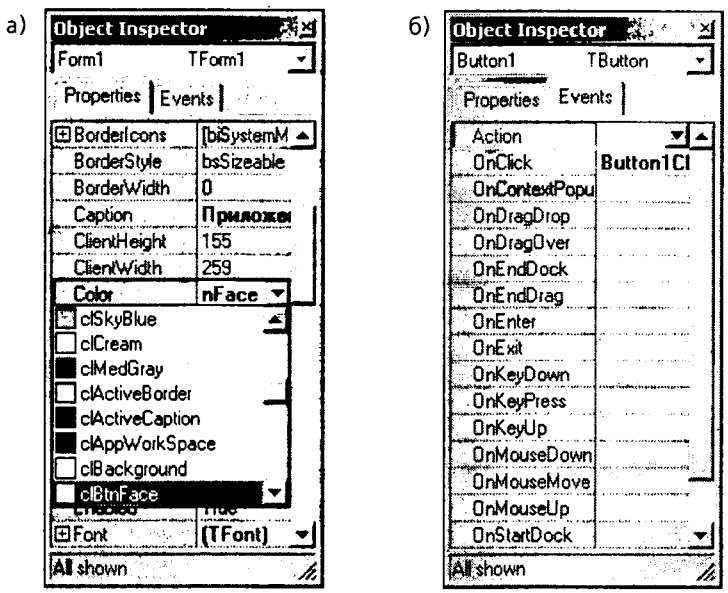


Рис. 1.6
Страница свойств (а) и страница событий (б) Инспектора Объектов

Пока обсудим страницу свойств для формы. Эта страница показана на рис. 1.6 а. Вы можете выделить в окне какое-то свойство и изменить его значение в окошке справа от этого свойства. Например, выделите свойство **Caption**. Это свойство — та надпись, которая видна в окне заголовка вашей формы. По умолчанию значение этого свойства совпадает с именем компонента, к которому свойство относится. В данном случае это «Form1». Измените значение свойства на что-нибудь осмысленное, например: «Приложение Delphi». Вы увидите, что в заголовке формы немедленно появится это значение.

Если щелкнуть на некоторых свойствах, например, на свойстве **Color** (цвет), то справа от имени свойства откроется окно выпадающего списка. Нажав в нем на кнопочку со стрелкой вниз, вы можете увидеть список возможных значений свойства (см. рис. 1.6 а). Например, смените значение свойства **Color** с принятого по умолчанию **clBtnFace** (цвет поверхности кнопок) на **clWindow** (цвет окна). Вы увидите, что поверхность формы изменит свой цвет.

О цветах компонентов имеет смысл поговорить особо. В выпадающем списке свойства **Color** в окне Инспектора Объектов вы можете увидеть большой набор предопределенных констант, обозначающих цвета. Все их можно разбить на две

группы: статические цвета типа **clBlack** — черный, **clGreen** — зеленый и т.д., и системные цвета типа **clWindow** — текущий цвет фона окон, **clMenuText** — текущий цвет текста меню и т.д. Если вы задаете статические цвета, они будут оставаться неизменными при работе приложения на любом компьютере. Это не очень хорошо, поскольку пользователь не сможет адаптировать вид вашего приложения к своим потребностям. Так что в большинстве случаев следует использовать для своего приложения палитру системных цветов. Это те цвета, которые устанавливает пользователь при настройке Windows. Когда вы создаете новую форму или размещаете на ней компоненты, Delphi автоматически присваивает им цвета в соответствии со схемой цветов, установленной в Windows. Конечно, вы можете менять эти установки по умолчанию. Но если при этом вы используете соответствующие константы системных цветов, то, когда пользователь изменит цветовую схему оформления экрана Windows, ваше приложение также будет соответственно изменяться, и не будет выпадать из общего стиля других приложений. Подробнее вопросы цветового решения приложений будут рассмотрены в разд. 5.1.2.

Вернемся к свойствам в окне Инспектора Объектов. Рядом с некоторыми свойствами вы можете видеть знак плюс. Это означает, что данное свойство является объектом, который в свою очередь имеет ряд свойств. Найдите, например, свойство **Font** (шрифт). Рядом с ним вы увидите знак плюс. Щелкните на этом плюсе или сделайте двойной щелчок на свойстве **Font**. Вы увидите, что откроется таблица таких свойств, как **Color** (цвет), **Height** (высота), **Name** (имя шрифта) и др. Среди них вы увидите свойство **Style** (стиль), около которого тоже имеется знак плюс. Щелчок на этом плюсе или двойной щелчок на этом свойстве раскроет дополнительный список подсвойств, в котором вы можете, например, установить в **true** свойство **fsBold** (полужирный). Значения **true** или **false** (истина или ложь) в подобных свойствах означает соответственно, что свойства включены или выключены. Кстати, для смены **true** на **false** и обратно в подобных свойствах не обязательно выбирать значение из выпадающего списка. Достаточно сделать двойной щелчок на значении свойства, и оно изменится. После того как вы просмотрели или изменили подсвойства, вы можете опять сделать двойной щелчок на головном свойстве или щелчок на знаке минус около него, и список подсвойств свернется.

Итак, установите в свойстве **Font** полужирный стиль шрифта. Теперь посмотрите на вашу форму. Вы увидите, что надписи в метке и на кнопке стали полужирными. Почему так? Ведь вы изменили свойство формы, а не кнопки и метки.

У всех компонентов имеется свойство **Parent** — *родительский компонент*. Не ищите это свойство в окне Инспектора Объектов. Это так называемое *свойство времени выполнения*. Его вы сможете изменять программно, когда научитесь программировать в Delphi. А во время проектирования свойства времени выполнения недоступны.

Итак, что такое родительский компонент? Вы поместили компоненты метки и кнопки непосредственно на форме. Как вы увидите далее, часто компоненты размещаются не непосредственно на формах, а на панелях, зрительно объединяющих группы компонентов по их назначению. В этом случае сначала на форме размещаются панели, а потом на них располагаются компоненты. Оконный компонент — форма, панель и т.д., включающий в себя как контейнер другие компоненты, выступает по отношению к ним как родительский компонент. У каждого компонента есть родитель. Им может быть форма или другой оконный компонент.

Что дает понятие родительского компонента? Компонент может наследовать многие свойства своего родителя. Для всех визуальных компонентов вы можете увидеть в Инспекторе Объектов такие свойства, как **ParentColor**, **ParentFont** и **ParentShowHint**, для оконных компонентов имеется еще свойство **ParentCtl3D**. Эти свойства указывают (если их значения установлены в **true**), что дочерний компонент наследует от родительского соответственно цвет, атрибуты шрифта, показа ярлычков, атрибуты своего оформления. Именно поэтому при изменении шрифта формы изменились шрифты надписей кнопки и метки. Если бы для компонентов вы задали значение **ParentFont = false**, этого не произошло бы. Поэкспериментируйте со свойствами **Font** и **ParentFont** формы и компонентов, чтобы до конца понять влияние свойства родительского компонента на свойства дочерних компонентов.

Связь родительского и дочернего компонента не ограничивается сказанным. Значения свойств **Left** и **Top**, которые вы можете видеть в Инспекторе Объектов для любого визуального компонента и которые определяют положение левого верхнего угла компонента, измеряются в системе координат родительского компонента. Таким образом, например, при перемещении границ родительского компонента будут синхронно перемещаться и все его дочерние компоненты.

Имеется еще два важных свойства, которые связывают дочерние компоненты с родительским. Это свойства **Visible** — видимый, и **Enabled** — доступный. Если в процессе выполнения приложения сделать в родительском компоненте **Visible** равным **false**, то станет невидимым не только родительский, но и все его дочерние компоненты. Аналогично, если в процессе выполнения приложения сделать в родительском компоненте **Enabled** равным **false**, то станут недоступными все его дочерние компоненты. Т.е. пользователь не сможет нажимать кнопки и производить любые другие действия в пределах данного родительского компонента. Правда учтите, что все это будет видно только во время выполнения. Во время проектирования вы не почувствуете влияние этих свойств.

Отметим еще одно свойство компонентов, которое часто путается со свойством **Parent**. Это свойство **Owner** — *владелец данного компонента*. Свойство **Owner** устанавливается в момент создания компонента в процессе выполнения приложения. Владелец компонента — этот тот компонент, при уничтожении которого (освобождении занимаемой им памяти) уничтожится и данный компонент. Этим и ограничивается связь между владельцем и компонентами, которыми он владеет, в отличие от множества указанных выше свойств, связывающих родительский и дочерние компоненты.

По умолчанию родителем и владельцем всех компонентов, размещенных на форме, является сама форма. Но если в процессе проектирования компонент размещается не непосредственно на форме, а на другом оконном компоненте, например, на панели, то родителем для него становится эта панель.

Все дочерние компоненты в оконном элементе располагаются в так называемой Z-последовательности. Для перекрывающихся компонентов (располагающихся друг на друге) Z-последовательность определяет, какой из них будет виден. Виден тот, который расположен в этой последовательности выше.

Обычно последовательность компонентов соответствует той, в которой они помещались на форму. Однако неоконные компоненты типа меток всегда лежат в Z-последовательности ниже любых оконных компонентов типа панелей и кнопок.

Теперь остановимся еще на одном свойстве любого компонента — **Name** (имя). По умолчанию Delphi присваивает компонентам имена, состоящие из имени типа

компонента и порядкового номера. Например, «Button1», «Label1» и т.д. А теперь один совет.

Хороший стиль программирования

Приучите себя сразу при переносе компонента на форму изменять его имя Name, принятое по умолчанию. Имя должно быть осмысленным, чтобы потом, разбираясь в коде, вы легко могли бы понять, что означает та или иная процедура, тот или иной компонент. Имена должны записываться только латинскими буквами.

Настоятельно рекомендую следовать этому совету. Тем самым вы сэкономите себе много времени. Представьте себе простенькую форму, где имеется десяток кнопок, десяток меток и пара десятков разделов меню. Уверяю вас, что если вы не даете компонентам осмысленные имена, то очень скоро вы начнете мучительно размышлять, что же это за кнопка **Button7**, щелчок на которой вы обрабатываете в данной процедуре, и почему в нем задается какое-то значение надписи **Caption** неизвестно где расположенной метки **Label3**. А уж на поиски таинственного раздела меню **N18** вы точно потратите немало времени. Так что любите себя, давайте компонентам понятные вам имена.

Рассматривая все эти особенности компонентов, мы отвлеклись от нашего приложения. Итак, задайте значения свойства **Caption** у формы (например, «Приложение Delphi») и у кнопки (например, «Пуск») — см. рис. 1.5. б. Полезно также для тренировки изменить свойства **Name** кнопки и метки, хотя, конечно, в нашем простом приложении делать это не обязательно.

1.2.4 Задание обработчика события, компиляция и выполнение приложения

Теперь все необходимые свойства компонентов заданы и осталось написать один оператор языка Object Pascal, чтобы при щелчке на кнопке в метке отображался какой-нибудь текст. Щелчок на кнопке, это событие, которое кнопка может обработать. Для задания обработчиков событий служит страница Events Инспектора Объектов (см. рис. 1.6 б). Выделив тот или иной компонент или форму, вы увидите на этой странице список тех событий, на которые компонент может реагировать. Практически все компоненты имеют событие **OnClick**, которое возникает, когда пользователь щелкнул мышью на компоненте. Для кнопки это основное событие. Так что выделите на форме вашу кнопку, перейдите в окне Инспектора Объектов на страницу событий, и найдите там строку события **OnClick**. Сделайте двойной щелчок на белом поле рядом с этим событием. Вы попадете в окно Редактора Кода, в котором будет видна заготовка обработчика события. Надо сказать, что для такого компонента, как кнопка, то же самое можно сделать проще: двойным щелчком на кнопке. В обоих случаях вы увидите в окне Редактора Кода текст:

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

end;
```

Курсор будет расположен в пустой строке между ключевыми словами **begin** и **end**. Увиденный вами код — это заготовка обработчика события, которую автоматически сделала Delphi. Вам остается только в промежутке между **begin** и **end** написать необходимые операторы.

Не будем пока разбираться в коде заголовка обработчика. Этот код станет понятен после изучения языка программирования в следующей главе. А пока напишите в строке межу словами **begin** и **end** следующий текст:

```
Label1.Caption:='Это мое первое приложение!';
```

Таким образом, полностью ваш обработчик события должен иметь вид:

```
procedure TForm1.Button1Click(Sender: Tobject);
begin
 Label1.Caption:='Это мое первое приложение!';
end;
```

Оператор, который вы написали, означает следующее. Символы «:=» обозначают в языке Object Pascal операцию присваивания, в которой тому, что написано перед этими символами, присваивается значение того, что написано после символов присваивания. Слева вы написали: **Label1.Caption**. Это значит, что вы присваиваете значение свойству **Caption** компонента **Label1**. Все указания свойств и методов производятся аналогичным образом: пишется имя компонента, затем ставится точка, а затем без пробела пишется имя свойства или метода. В данном случае свойству **Caption** вы присваиваете строку текста «Это мое первое приложение!». В приведенном операторе подразумевается, что имя метки **Label1** вами не изменялось. Если вы изменили его, то вместо «Label1» надо написать введенное вами имя.

Описанный выше способ доступа к свойствам и методам объектов с помощью символа точки используется и в тех случаях, когда свойство само является объектом. Например, вы можете написать оператор:

```
Label1.Font.Color := clRed;
```

Левая часть оператора означает, что вы задаете значение свойству **Color** (цвет) объекта **Font** (шрифт), являющегося свойством метки **Label1**. А константа **clRed** в правой части оператора соответствует красному цвету. Значения констант цветов вы можете найти в окне Инспектора Объектов в списке около свойства **Color** любого компонента, или щелкнув на этом свойстве и нажав клавишу F1 — вызов справки.

Исключением из описанного правила доступа к свойствам и методам с помощью символа точки являются свойства и методы формы. По умолчанию свойства и методы, в которых не указан явно объект, относятся к форме. Так что если вы запишете приведенный выше оператор следующим образом

```
Color := clRed;
```

то он будет изменять цвет поверхности формы, а заодно и цвет поверхности всех расположенных ней дочерних компонентов, в которых свойство **ParentColor** равно **true**. А если вы напишете оператор

```
Font.Color := clRed;
```

то изменится цвет текста всех расположенных на форме дочерних компонентов, в которых свойство **ParentColor** равно **true**.

Впрочем, можно и в этих операторах использовать ссылку на форму. Например, если имя формы **Form1**, то можно написать:

```
Form1.Color := clRed;
Form1.Font.Color := clRed;
```

Но вернемся к нашему обработчику щелчка на кнопке. Если вы написали первый идентификатор его оператора — имя метки (Label1), поставили точку и ненадолго задумались, то вам всплывет подсказка (см. рис. 1.7), содержащая список всех свойств и методов метки. Это начал работать Знаток Кода, который стремится подсказать вам свойства и методы компонентов, аргументы функций и их типы, конструкции операторов. Вы можете выбрать из списка нужное ключевое слово, нажать клавишу Enter и выбранное слово (свойство, метод) окажется вписанным в текст. Можете поступить иначе: начать писать нужное свойство. Тогда Знаток Кода сам найдет по первым введенным символам нужное свойство. Когда вы увидели, что нужное слово найдено, можете его не дописывать, а нажать Enter, и Знаток Кода допишет его за вас.

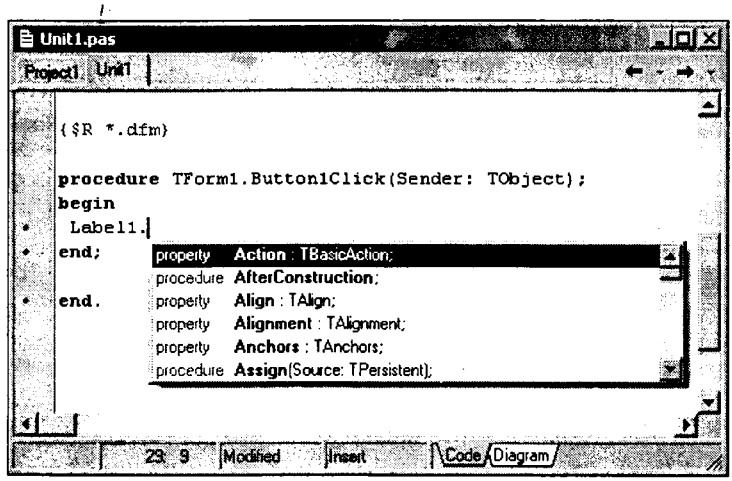


Рис. 1.7
Подсказка Знатока Кода Code Insight

В дальнейшем при работе с библиотечными функциями вы увидите, что подсказки всплывают также и в этом случае, облегчая задание необходимых параметров функций.

Подсказки Знаток Кода по умолчанию упорядочены по областям видимости и категориям, что не очень удобно. Вы можете изменить характер упорядочивания, щелкнув в окне Знатока Кода правой кнопкой мыши и включив во всплывшем меню раздел Sort by Name — сортировка по алфавиту.

Совет

Отсутствие подсказок при работе с объектами и функциями, если только вы не отключили подсказки при настройке ИСР Delphi, означает обычно, что вы ошиблись в имени объекта или функции. Так что не игнорируйте отсутствие подсказки, проверьте правильность написания кода, предшествующего моменту, когда должна бы появиться подсказка.

Несмотря на то, что мы еще не знакомились с языком программирования, обращу ваше внимание на одно обстоятельство. Посмотрите заголовок созданного вами обработчика события:

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

А теперь посмотрите повыше в коде, и вы найдете похожую строку:

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

Эта строка — объявление того обработчика события, который вы создали. Я обращаю внимание на это обстоятельство по следующей причине. Как показывает практика, вы нередко можете выделить в окне Инспектора Объектов не то событие, обработчик которого хотите написать, или при выборе события выделите не тот компонент. Не удаляйте вручную неверный обработчик события. Удалите только, текст обработчика, если успели что-то написать между словами **begin** и **end**. А пустой обработчик удалит, когда придет время, сама среда Delphi. Если же вы все-таки удалили неверный обработчик события, то удалите одновременно и его объявление. Иначе вы не сможете выполнить вашу программу.

Предупреждение

Если вы случайно написали обработчик не того события, которое хотели, или обработчик события не того компонента, который вам нужен, удалите только тело ошибочного обработчика — текст между словами **begin** и **end**. Но не удаляйте эти ключевые слова и заголовок обработчика события. Если же вы все-таки удалили неверный обработчик события, то не забудьте одновременно удалить и его объявление. Иначе вы получите сообщение об ошибке вида: «Unsatisfied forward or external declaration: ...», т.е. о нереализованном объявлении. Если получите такое сообщение об ошибке, найдите это объявление и удалите его.

Итак, ваше приложение готово. Теперь можете выполнить команду Run | Run, чтобы выполнить его. Проще сделать то же самое, нажав быструю клавишу с зеленой стрелкой, направленной вправо (см. рис. 1.2 или приведенную в разд. 1.6 табл. 1.1). Еще проще нажать горячую клавишу F9. Если вы ничего не напутали в том единственном операторе, который написали (если напутали, посмотрите в Приложении сообщения об ошибках и поправьте код), то после недолгой компиляции перед вами появится окно вашего первого приложения. Нажав в нем кнопку Пуск, вы увидите указанную вами строку текста (рис. 1.5 б). Убедитесь, что вы создали полноценное приложение Windows, написав всего один оператор. Можете попробовать различные манипуляции с окном: перемещение его, изменение размеров его рамки курсором мыши, свертывание и развертывание, вызов системного меню при щелчке на пиктограмме в левом верхнем углу. В заключение закройте приложение, щелкнув на кнопке с крестиком в его правом верхнем углу.

Теперь, прежде чем продвигаться далее, имеет смысл обсудить, что же произошло, когда вы вызвали выполнение приложения. Сначала сработал *компилятор* Delphi. Это программа, которая создала на основе вашего проекта *исполняемый файл*, имеющий имя, совпадающее с именем проекта, и расширение *.exe*. Вы можете убедиться средствами Windows, что такой файл действительно создался в том каталоге, в котором вы сохраняли проект.

Исполняемый файл содержит машинные коды и является двоичным, а не текстовым. Так что смотреть его любым текстовым редактором бессмысленно. Но любой компьютер сможет его понять и выполнить. Поэтому вы спокойно можете перенести его в другой каталог или на другой компьютер, выполнить средствами Windows и получить тот же замечательный результат, который видели при запуске из среды Delphi.

Правда, надо учесть одно обстоятельство. Если вы выполнили ту рекомендацию по настройке среды Delphi, которая была дана в разд. 1.2.1 и сводилась к включению индикатора Build with runtime packages на странице Packages окна опций проекта, то ваше приложение может быть выполнено только на компьютере, на котором поставлена Delphi или на котором имеется соответствующий пакет времени выполнения. Пакетами времени выполнения мы заниматься не будем, так как это выходит за рамки основ программирования в Delphi. А если вы хотите сделать свое приложение полностью автономным и доступным любому пользователю, даже не слыхавшему о Delphi, то надо выключить указанный выше индикатор и повторно скомпилировать проект. Правда, размер исполняемого модуля при этом резко увеличится. Это связано с тем, что хотя вы написали всего один оператор, Delphi включает в исполняемый файл огромный код, обеспечивающий отображение окна вашего приложения и реакции на действия пользователя. Так что надо сказать спасибо Delphi за то, что вам пришлось написать менее одной тысячной части полного кода.

Вы использовали для компиляции и выполнения команду Run | Run или соответствующие ей быструю кнопку и горячие клавиши. Но имеется еще несколько команд, осуществляющих компиляцию приложения. В меню Project вы можете увидеть разделы Compile имя проекта, Build имя проекта, Syntax Check имя проекта, Compile All Projects, Build All Projects. Команды, содержащие имя вашего проекта, относятся только к вашему активному проекту. Команды, содержащие слова All Projects, используются в случае одновременной работы с несколькими проектами и означают компиляцию всех открытых проектов. Подобные задачи мы рассмотрим много позже. А в вашем случае одного проекта команды с именем проекта и со словами All Projects идентичны.

Команда Syntax Check только проверяет синтаксис вашего кода и не создает исполняемый файл. Команды Compile и Build осуществляют компиляцию и создают исполняемый файл, но не выполняют его. Различие между этими командами проявляется в более сложных проектах, чем тот, который пока создали вы. В сложных проектах может быть несколько модулей с несколькими формами. Поскольку обычно на каждом этапе проектирования вы вносите изменения в код какого-то одного или нескольких, но не всех модулей, было бы нерационально каждый раз компилировать все модули. Такую выборочную компиляцию осуществляет команда Compile. Она компилирует только те модули, в которые были внесены изменения с момента последней компиляции. Подобная компиляция позволяет в больших проектах экономить немало времени, поскольку модули, которые не изменялись, повторно не компилируются. Команда Build компилирует все файлы проекта независимо от времени их предыдущей компиляции. Это необходимо делать, если, например, вы решили, ничего не изменяя в тексте проекта, откомпилировать проект с другими опциями компилятора. В частности, если, как говорилось выше, вы разрабатывали проект с включенной опцией Build with runtime packages, а теперь хотите получить окончательный вариант автономного приложения с выключенной опцией.

Имеются еще случаи, когда приходится использовать команду Project | Build. Дело в том, что рассмотренная ранее команда Run | Run, которая используется обычно

в процессе разработки и отладки приложения, последовательно выполняет команду Compile — выборочную компиляцию и вызов исполняемого модуля. Но Delphi не всегда точно определяет модули, в которых проведены изменения и которые, следовательно, надо компилировать. В частности, если вы изменили только свойства каких-то компонентов в окне Инспектора Объектов, но не изменяли код приложения, Delphi может это не заметить и при выполнении команды Run | Run не провести новую компиляцию модуля. В этом случае вы с удивлением обнаружите, что вы изменяете значение свойства, но это никак не сказывается на работе приложения.

Предупреждение

Если вы изменяете значение свойства какого-то компонента, но при выполнении команды Run | Run это никак не сказывается на работе приложения, выполните команду Project | Build, а затем выполните команду Run | Run. Аналогичным образом надо компилировать приложение, если вы ничего в нем не изменяли, а только задали какие-то другие значения опций в окне опций проекта, вызываемом командой Project | Options.

1.3 Приложение с компонентами ввода/вывода текстовой информации

Вас можно поздравить с созданием в Delphi вашего первого приложения для Windows. В разд. 1.2 его создание излагалось довольно долго, но это связано с тем, что параллельно рассказывалось о приемах работы с ИСР Delphi. Если вы повторите создание этого приложения, вы поймете, что в действительности для этого требуется не более пяти минут. В этом и заключается прелесть визуального программирования — прототип будущего приложения можно создавать очень просто и быстро. Теперь давайте построим несколько более сложное приложение. Его цель — ознакомиться в общих чертах с парой компонентов, позволяющих вводить и выводить текстовую информацию. Подобное приложение потребуется нам в следующей главе при изучении языка Object Pascal.

Пусть это приложение содержит два окна редактирования, в которые пользователь сможет вводить информацию. И пусть в приложении будет кнопка и многострочное окно редактирования. При щелчке пользователя на кнопке приложение будет выводить в многострочное окно редактирования результат обработки введенной пользователем информации. Например, пользователь вводит в окна фамилию и имя, а при щелчке на кнопке в многострочное окно редактирования выводится шапка характеристики данного лица. Или пользователь вводит в окна два числа, а в многострочное окно выводится результат каких-то вычислений, использующих заданные числа.

При описании подобного приложения я не буду столь многословно, как в разд. 1.2, описывать все необходимые операции. Будем считать, что основными приемами создания приложений вы уже овладели.

Начните новое приложение и сохраните его. При сохранении дайте приложению имя *Charact*. На это имя я буду далее ссылаться.

Перенесите на форму уже знакомую вам кнопку **Button** (седьмая слева на странице Standard). С той же страницы библиотеки перенесите два окна редактирования **Edit** (пятая слева пиктограмма с надписью «ab») и многострочное окно редактирования **Memo** (шестая слева пиктограмма). Расположите эти компоненты примерно так, как показано на рис. 1.8. Можете изменить надписи окна приложения («Характеристики» на рис. 1.8) и кнопки («Выполнить» на рис. 1.8). Можете также установить в форме полужирный шрифт, чтобы надписи были более четкими. Но в окне **Memo** шрифт надо оставить обычным. Неплохо также перенести на форму метки, в которых будут надписи, помогающие пользователю разобраться в назначении различных окон («Фамилия», «Имя и отчество», «Характеристика» на рис. 1.8).

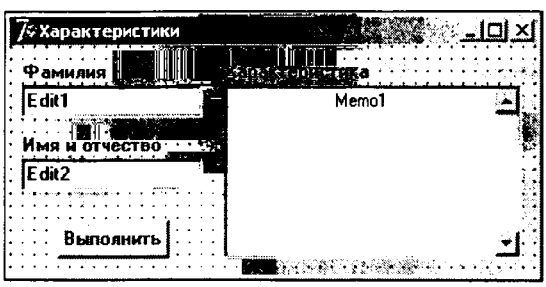


Рис. 1.8
Форма приложения Charact

Выделите на форме одно из окон **Edit** и посмотрите в окне Инспектора Объектов его свойства. Главное свойство — **Text**. Это свойство можно задавать во время проектирования, а во время выполнения в этом свойстве можно прочитать текст, введенный пользователем. В нашем случае, вероятно, во время проектирования текст задавать не надо, а наоборот — следует стереть тексты «Edit1» и «Edit2», заданные в окнах по умолчанию и равные именам соответствующих компонентов. Из других свойств обратите внимание на свойство **ReadOnly**. Если задать его значение равным **true**, то пользователь сможет только читать текст в окне, но не сможет его изменять. В нашем приложении, конечно, надо оставить значение **false**, заданное по умолчанию в этом свойстве.

Теперь выделите компонент **Memo1** и посмотрите его свойства. Основное свойство — **Lines**. Вы можете увидеть в окне Инспектора Объектов около этого свойства его тип — **TStrings**. Запомните этот тип. Он очень часто будет встречаться вам при работе с Delphi. В процессе проектирования значения свойств этого типа можно задавать, нажав кнопку с многоточием около свойства в окне Инспектора Объектов. Откроется окно редактора строк, показанное на рис. 1.9. В нем вы можете записать какой-то текст, который будет показан пользователю в первый момент при запуске приложения. Но в нашем примере никакого текста писать не надо. Наоборот, надо стереть текст «Memo1», заданный по умолчанию.

Во время выполнения приложения для удаления текста из окна надо выполнить процедуру **Clear**. Этот метод относится к самому окну, а не к его свойству **Lines**. Так что оператор

```
Memo1.Clear;
```

удалит текст из окна **Memo1**.

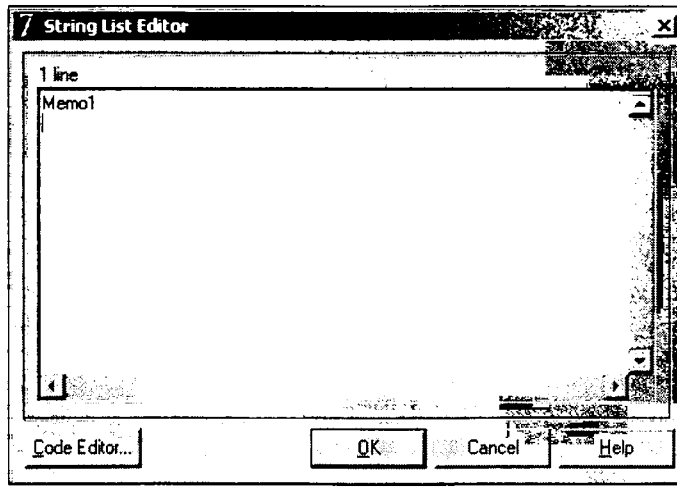


Рис. 1.9
Окно редактора строк

Для занесения новой строки в конец текста окна редактирования можно воспользоваться методами **Add** или **Append** свойства **Lines**. В качестве аргументов в эти методы передается строка текста. Например, оператор

```
Memo1.Lines.Add('Х А Р А К Т Е Р И С Т И К А');
```

занесет в окно строку с тестом «Х А Р А К Т Е Р И С Т И К А». Строки текста, как вы видите в приведенном выше операторе, заключаются в одинарные кавычки.

Окно **Memo** имеет, как и окно **Edit**, свойство **ReadOnly** — только для чтения. Если окно используется только для отображения информации, например, лицензионного соглашения на вашу великолепную программу, то это свойство надо установить в **true**. Но в нашем примере предполагается, что кто-то может написать в этом окне характеристику. Так что значение **ReadOnly** должно быть равно **false**.

Свойство **Alignment** определяет выравнивание текста в окне и может принимать значения **taLeftJustify** — влево, **taRightJustify** — вправо, **taCenter** — по центру. По умолчанию задано значение **taLeftJustify**, но, вероятно, в нашем примере лучше установить значение **taCenter**.

Свойство **ScrollBars** управляет появлением в окне полос прокрутки. Оно может принимать следующие значения: **ssNone** — полос прокрутки нет, **ssHorizontal** или **ssVertical** — включена соответственно горизонтальная или вертикальная полоса, **ssBoth** — включены обе полосы. Если вы хотите дать возможность пользователю писать длинные характеристики, вероятно, имеет смысл задать **ScrollBars** равным **ssVertical**, чтобы включить вертикальную полосу, облегчающую прокрутку текста.

Теперь вы можете написать обработчик щелчка на кнопке (вспомните, как это делается). Пусть такой щелчок будет выводить в окно **Memo1** шапку характеристики того, чьи фамилия, имя и отчество занесены в окна редактирования. В приведенном ниже коде предполагается, что это студент. Но вы, конечно, можете заменить слово «Студент» на «Сотрудник» или «Учащийся» — т.е. на то, что вам больше подходит. Обработчик щелчка на кнопке может иметь следующий вид:

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

```
 Memo1.Clear;
 Memo1.Lines.Add('Х А Р А К Т Е Р И С Т И К А');
 Memo1.Lines.Add('Студент ' + Edit1.Text + ' ' + Edit2.Text);
 Memo1.SetFocus;
end;
```

Первый оператор этого кода очищает окно **Memo1**. Второй — заносит в окно заголовок «Х А Р А К Т Е Р И С Т И К А». Третий оператор комбинирует из текстов, введенных пользователем в окна **Edit1** и **Edit2**, строку вида «Студент ... ...». Как видно, соединение нескольких строк в одну осуществляется просто операцией сложения. К строке «Студент » (она должна кончаться пробелом, чтобы это слово не слилось с фамилией) добавляется текст окна **Edit1**, затем добавляется строка пробела, и к этому добавляется текст окна **Edit2**.

Последний не обязательный оператор методом **SetFocus** переводит фокус на компонент **Memo1**. Так что пользователь немедленно может начать писать текст характеристики.

Сохраните ваше приложение (мы еще к нему вернемся и сделаем намного мощнее) и выполните команду Run | Run. Если вы все сделали аккуратно, в частности, не забыли поставить точки с запятой в конце каждого оператора (обычная ошибка начинающих) и не запутались в кавычках, то после успешной компиляции вы увидите окно вашего выполняющегося приложения и сможете убедиться, что оно работает правильно (см. рис. 1.10).

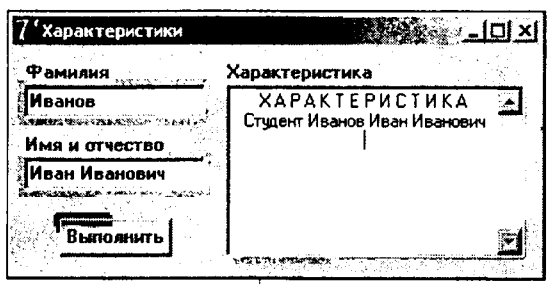


Рис. 1.10
Приложение Charact во время выполнения

Конечно, в подобном приложении надо бы еще сделать очень многое, прежде всего, ввести возможность сохранения характеристики в файле и чтения из файла. Оформление приложения тоже пока очень скромное. Надо бы добавить полосу меню, инструментальную панель, возможность форматирования текста и многое другое. Со временем мы вернемся к этому приложению и усовершенствуем его.

1.4 Набор файлов проекта Delphi

Проект Delphi состоит из форм, модулей, установок параметров проекта, ресурсов и т.д. Вся эта информация размещается в файлах. Многие из этих файлов автоматически создаются Delphi. Ресурсы, такие, как битовые матрицы, пиктограммы и т.д., находятся в файлах, которые вы получаете из других источников или создаете при помощи многочисленных инструментов и редакторов ресурсов, имеющихся в вашем распоряжении. Кроме того, компилятор также создает файлы.

Зачем желательно знать состав файлов проекта? Ведь эти файлы создаются в среде Delphi автоматически! И все-таки, знать назначение различных файлов полезно. Разработка проектов Delphi и обучение работе с Delphi часто проходит на разных компьютерах. Например, в дисплейном классе или на работе, и на домашнем компьютере. Значит, надо знать, какие файлы требуется переносить с компьютера на компьютер, чтобы работа не остановилась. Причем иногда на разных компьютерах могут стоять разные версии Delphi. В этом случае требуется согласование файлов проекта в разных версиях.

Ниже приведена характеристика основных файлов, используемых при создании выполняемого файла приложения.

Головной файл проекта (*.dpr*)	Этот текстовый файл используется для хранения информации о формах и модулях. В нем содержатся операторы инициализации и запуска программы на выполнение.
Файл модуля (*.pas*)	Каждой создаваемой вами форме соответствует текстовый файл модуля, используемый для хранения кода. Иногда вы можете создавать модули, не связанные с формами. Многие из функций и процедур Delphi хранятся в модулях.
Файл формы (*.dfm*)	Это двоичный или текстовый файл, который создается Delphi для хранения информации о ваших формах. Каждому файлу формы соответствует файл модуля (*.pas*).
Файл параметров проекта (*.dfo*)	В этом файле хранятся установки параметров проекта.
Файл ресурсов (*.res*)	Этот бинарный файл содержит используемую проектом пиктограмму и прочие ресурсы.
Файл группы файлов (*.bpg*)	Этот файл, создается, если вы работаете с группой проектов.
Файлы резервных копий (*.~dp, .~df, .~pa*)	Это соответственно файлы резервных копий для файлов проекта, формы и модуля. Если вы что-то безнадежно испортили в своем проекте, можете соответственно изменить расширения этих файлов и таким образом вернуться к предыдущему, не испорченному варианту.
Исполняемый файл (*.exe*)	Это исполняемый файл вашего приложения.
Объектный файл модуля (*.dcu*)	Это откомпилированный объектный файл модуля (*.pas*), который компонуется в окончательный исполняемый файл.

Если вам требуется перенести проект с одного компьютера на другой, чтобы продолжить работу над ним, или перенести проект из одной папки Windows в другую, надо перенести файлы *.dpr*, *.pas*, *.dfm*, *.res*. Если вы работаете с группой проектов, можно перенести также файл *.bpg*. Остальные файлы переносить не имеет смысла: файлы *.exe* и *.dcu* все равно создадутся заново при первой же компиляции проекта, а файлы резервных копий вряд ли вам понадобятся. В разд. 1.5 описан другой способ копирования проекта средствами ИСР Delphi.

Часто вам требуется перенести на другой компьютер или в другой каталог не весь проект, а только один из модулей проекта, связанный с какой-то формой. Например, вы разработали какую-то хорошую форму, которую хотите использовать в различных своих проектах. При переносе формы средствами Windows надо

иметь в виду, что информация о формах Delphi хранится в двух файлах. В файле с расширением *.dfm* хранится информация о внешнем виде формы, ее размерах, местоположении на экране и т.д. В текстовом файле с расширением *.pas* хранится код модуля, соответствующего данной форме. Имена обоих этих файлов одинаковы. Вы задаете это имя, когда в первый раз сохраняете ваш модуль. Так что перенести куда-то модуль — это значит перенести оба эти файла. Впрочем, в разд. 1.5 рассмотрены способы переноса формы из одного проекта в другой, которые избавляют вас от забот о наборе файлов формы.

В версиях Delphi, предшествующих Delphi 5, файл формы *.dfm* был двоичным. Для того чтобы посмотреть его в текстовом виде, надо было щелкнуть на форме правой кнопкой мыши и из всплывшего меню выбрать раздел View as Text. Тогда в окне Редактора Кода появлялся текстовый файл формы. При желании его можно было редактировать. Для возврата к графическому представлению формы надо было на тексте файла щелкнуть правой кнопкой мыши и выбрать команду View As Form. Все эти возможности сохранены и более новых версиях. Но, начиная с Delphi 5, по умолчанию файлы форм хранятся в текстовом виде.

Текстовое представление форм вы можете использовать для переноса форм из одной версии Delphi в другую (особенно при переносе из более поздней версии в более раннюю) или для переноса в Delphi форм, разработанных в C++Builder. Borland обеспечивает обратную совместимость своих версий. Например, проект, созданный в Delphi 5, можно открыть в Delphi 7. Только при этом будет задан вопрос, надо ли обновить проект в формате более новой версии. А вот обратная задача так просто не решается. Например, форму, созданную в Delphi 7, невозможно напрямую перенести в проект Delphi 5. Если отрыть в Delphi 5 проект, созданный в Delphi 7, будут сообщения об ошибках в форме и проект не откроется. Однако если вы сохранили форму Delphi 7 в текстовом виде (например, записали текст в текстовом файле с помощью программы Windows «Блокнот»), то такую форму можно будет перенести в Delphi 5. Для этого откройте в Delphi 5 пустую форму, щелкните на ней правой кнопкой мыши и из всплывшего меню выберите раздел View as Text. Затем замените текст в окне Редактора Кода сохраненным текстом проекта Delphi 7, и выполните с помощью правой кнопки мыши команду View as Form. Форма появится в том же виде, в котором была в Delphi 5. Далее останется только скопировать в проект коды из модулей, разработанных в Delphi 5.

1.5 Повторное использование кодов и форм

Любой специалист, в том числе и специалист, занимающийся программированием, должен научиться любить себя, беречь свои силы и время. Как известно, лень — двигатель прогресса. Но не любая лень, а лень разумно организованная. При работе в Delphi под разумной организацией следует понимать, прежде всего, *повторное использование* кодов и форм. Не стоит, например, жалеть сил на разработку хорошей формы, с удобным сервисом, меню, инструментальными панелями и т.д. (всему этому вы научитесь в последующих главах книги). Но если уж вы создали подобную форму, надо позаботиться, чтобы ее с какими-то дополнениями и изменениями можно было использовать в других ваших проектах. Например, нелепо для разных проектов каждый раз создавать заново форму запроса пароля или форму-заставку с вашей фирменной пиктограммой. Создав один раз подобную форму, надо использовать ее в различных своих проектах.

Повторное использование разработанных вами замечательных процедур и функций решается просто. Вы можете создать модуль, не привязанный к какой-то форме, и поместить в него ваши программные разработки. Чтобы создать подобный модуль, надо выполнить команду File | New | Other (в Delphi 5 и более старых версиях — File | New), и в открывшемся окне New Items на странице New щелкнуть на пиктограмме Unit. Занеся в созданный модуль необходимый код (константы, переменные, процедуры и функции, используемые в различных проектах) сохраните модуль в файле. А затем в любом вашем проекте вы можете подключить этот модуль предложением **uses** (оно будет рассмотрено позднее в разд. 2.3.3) и использовать все богатство реализованных в нем ваших разработок.

Теперь рассмотрим случай, когда вам требуется включить в свой проект форму, разработанную ранее вами или кем-то еще для другого проекта. Подобное повторное использование формы можно реализовать несколькими способами, имеющими различные последствия.

Можно включить готовую форму в проект командой Project | Add to Project или соответствующей быстрой кнопкой. При этом если включаемая форма имеет то же имя, которое имеет одна из уже имеющихся в проекте форм (например, **Form1**, если вы не привыкли изменять имена форм, принимаемые Delphi по умолчанию), то вы получите предупреждение вида: «The project already contains a form or module named Form1» — «Проект уже содержит форму или модуль с именем Form1». В результате форма откроется, но в проект не включится. Аналогичный вариант будет, если вы не следуете в своей работе уже дававшемуся в разд. 1.2.2 совету присваивать модулям уникальные имена. Если в проекте уже имеется модуль *Unit1* и вы пытаетесь включить из другого каталога модуль формы, тоже имеющий имя *Unit1*, то вам будет выдано такое же предупреждение, новый модуль загрузится, но в проект не включится.

Разрешить подобный конфликт можно следующим образом. Перейдите в окно Редактора Кода в открывшийся, но не включенный в проект модуль. Щелкните правой кнопкой мыши и во всплывшем меню выберите команду Close Page. Модуль закроется. После этого переименуйте в вашем проекте форму, вызвавшую конфликт (задайте для нее новое имя в свойстве **Name**). Если конфликт вызван совпадением имен модулей, то сохраните конфликтующий модуль командой File | Save As, дав ему новое имя. После этого можете повторить попытку добавления в проект новой формы.

Предупреждение

Учтите, что форма, содержащаяся в одном приложении и включенная описанным способом в другое приложение, становится общей для обоих приложений. Если вы сделаете в ней какие-то изменения, а потом перекомпилируете оба приложения, то внесенные изменения отразятся на обоих приложениях.

Введенное описанными действиями совместное владение несколькими приложениями одной и той же формой имеет свои плюсы и минусы. Если эти приложения представляют собой некую группу связанных друг с другом приложений, рассчитанных на применение одними и теми же пользователями, то наличие общих форм можно только приветствовать. Какие-то усовершенствования, введенные в подобной форме, согласованно отобразятся во всех использующих ее приложениях (по-

сле их перекомпиляции). Например, если это форма, запрашивающая пароль пользователя, то, конечно, хорошо, если она будет общей (и значит идентичной) в различных приложениях.

Если же приложения, совместно использующие форму, совершенно разные или форма используется в них для разных целей, то, введя изменения в форму в одном приложении, вы рискуете испортить прежнее приложение, если соберетесь его перекомпилировать. Например, изменив имя программы в форме, описывающей новое приложение, вы невольно введете то же имя и в прежнее приложение, что будет, несомненно, ошибкой.

Чтобы избежать совместного владения формой несколькими приложениями, после того, как вы включили в новое приложение форму из другого приложения, перейдите в окно Редактора Кода в модуль этой формы и выполните команду File | Save As, сохранив модуль в каталоге нового приложения и, если хотите, под другим именем (имя изменять не обязательно). В этом случае разные приложения будут использовать совершенно разные копии одной формы, и изменения одной из них не затронут другие приложения.

Еще один способ создания автономной копии формы — использование меню Delphi. Вы можете в любой момент последовательно выполнить команды File | Open, указав файл открываемой формы, и команду File | Save As. Первая из этих команд откроет форму, а вторая сохранит ее в указанном вами каталоге под указанным именем.

Описанные способы копирования форм с помощью Delphi имеют преимущества перед рассмотренным в разд. 1.4 копированием средствами Windows. В данном случае вам не приходится заботиться о том, чтобы не забыть скопировать все связанные с формой файлы — *.pas*, *.dfm*, а иногда и некоторые другие . Все они будут скопированы автоматически.

Иногда у вас может возникнуть потребность в процессе работы над проектом не копировать, а просто посмотреть формы и методы из других проектов. Это могут быть формы примеров, поставляемых с Delphi. Или могут быть ранее разработанные вами формы, из которых вы хотите взять какие-то операторы, решающие задачу близкую к той, которой вы заняты в данных момент.

Открыть некоторый существующий модуль, не включая его в текущий проект, очень легко. Достаточно выполнить команду File | Open, и указанный вами файл модуля окажется в окне Редактора Кода. Вы можете просматривать его и соответствующую ему форму, копировать через буфер обмена Clipboard какие-то операторы в свои модули.

Когда необходимость в открытом модуле отпадет, щелкните на его коде правой кнопкой мыши и из всплывшего меню выберите команду Close Page. Страница Редактора Кода с текстом данного модуля и его форма будут закрыты.

В Delphi имеется еще одна возможность — заимствование включаемой в проект формы из Депозитария. Депозитарий — это хранилище форм, проектов и Мастеров создания форм и проектов, как поставляемых с Delphi, так и ваших собственных. Для заимствования формы из Депозитария надо выполнить команду File | New | Other (в версиях, младше Delphi 6 — команду File | New), после которой откроется окно New Items (новые элементы). На страницах Forms (рис. 1.11) и Dialogs вы найдете ряд форм, которые можете заимствовать для своего проекта.

Форма заимствования определяется тремя радиокнопками, размещенными в нижней части окна: Copy — копировать, Inherit — наследовать, Use — использовать. Если включена кнопка Copy, то файлы формы просто будут скопированы

в ваше приложение. При этом никакой дальнейшей связи между исходной формой и копией не будет. Вы можете спокойно изменять свойства вашей копии, и это никак не отразится на форме, хранящейся в Депозитарии. Именно этот режим надо использовать, если вы хотите перенести в свой проект одну из форм, включенных в Депозитарий создателями Delphi, или создать в проекте автономную копию собственной формы, помещенной в Депозитарий.

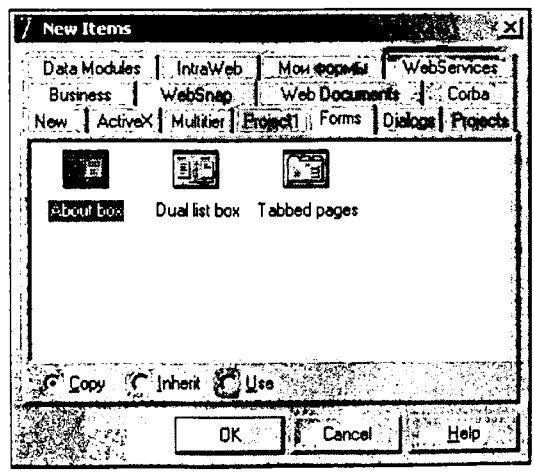


Рис. 1.11
Страница Forms окна Депозитария New Items

При включенной кнопке Inherit вы получите в своем проекте форму, наследующую размещенной в Депозитарии. Это значит, что если вы что-то измените в форме, хранящейся в Депозитарии, то это отразится при перекомпиляции во всех проектах, которые наследуют эту форму. Однако изменения в наследуемых формах никак не скажутся на свойствах формы, хранящейся в Депозитарии.

При включенной кнопке Use вы получите режим использования. В этом случае в ваш проект включится сама форма, хранящаяся в Депозитарии. Значит любое изменение свойств формы, сделанное в вашем проекте, отразится и на хранящейся в Депозитарии форме, и на всех проектах, наследующих или использующих эту форму (при их перекомпиляции).

Таким образом, режим Inherit целесообразно использовать для модулей проектов, использующих какую-то базовую форму, включенную вами в Депозитарий, а режим Use — для изменения этой базовой формы. Тогда усовершенствование вами базовой формы будет синхронно сказываться на всех использующих ее проектах при их перекомпиляции.

Теперь посмотрим, как можно сохранить свою собственную форму в Депозитарии. Эту процедуру вы можете опробовать на любой созданной вами ранее форме, например, на форме приложения *Charact*, созданной в разд. 1.4. Перед занесением формы в Депозитарий ее модуль должен быть обязательно сохранен в файле. Щелкните на вашей форме правой кнопкой мыши и выберите во всплывшем контекстном меню раздел Add To Repository. Откроется диалоговое окно, вид которого приведен на рис. 1.12. В верхнем окне Title вы должны написать название вашей формы — подпись под ее пиктограммой при входе в Депозитарий. В следующем окне — Description можете написать более развернутое пояснение. Его может увидеть пользо-

ватель, войдя в Депозитарий, щелкнув правой кнопкой мыши и выбрав во всплывшем меню форму отображения View Details. В выпадающем списке Page вы можете выбрать страницу Депозитария, на которой хотите разместить пиктограмму своей формы. Впрочем, вы можете указать и новую страницу с новым заголовком. В результате она появится в Депозитарии (страница «Мои формы» на рис. 1.11).

В окне Author вы можете указать сведения о себе как об авторе. Наконец, если стандартная пиктограмма вас не устраивает, вы можете выбрать другую, щелкнув на кнопке Browse. После выполнения всех этих процедур щелкните на кнопке OK и ваша форма окажется включенной в Депозитарий. Теперь вы можете использовать ее в последующих ваших приложениях точно так же, как описано выше для форм, поставляемых с Delphi.

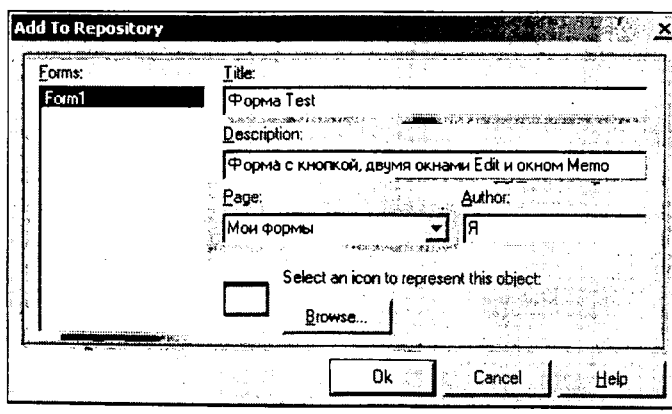


Рис. 1.12
Окно добавления формы в Депозитарий

В Депозитарий можно включать не только формы, но и целые проекты. Если вы хотите включить в него ваш проект, откройте его и выполните команду Project | Add To Repository. Дальнейшие действия аналогичны тем, которые вы выполняете при включении в Депозитарий формы. Отличие проекта от формы при их заимствовании из Депозитария состоит в том, что проект можно взять оттуда только в режиме Copy, т.е. скопировать его и далее сохранить под другим именем. Если вы хотите взять проект, хранящийся в Депозитарии, то начать работу надо не с привычной команды File | New | Application, а с команды File | New | Other. Вы выбираете проект из Депозитария, и вам сразу же предлагается диалоговое окно выбора каталога, в котором вы хотите сохранить копию проекта. После этого вы можете обычным образом работать с этой копией и менять в ней все, что вам захочется.

Вы можете опробовать этот механизм, например, на созданном вами в разд. 1.3 проекте *Charact*, который вы в дальнейшем, возможно, захотите усовершенствовать. Тогда впоследствии вы легко сможете заимствовать этот проект из Депозитария и создавать на его основе новые варианты.

Если есть механизм включения в Депозитарий форм и проектов, то должен быть и механизм их удаления. Удаление объектов, хранящихся в Депозитарии, выполняется командой Tools | Repository. При этом открывается окно, представленное на рис. 1.13. То же окно открывается, если в окне Депозитария New Items щелкнуть правой кнопкой мыши и выбрать из контекстного меню раздел Properties.

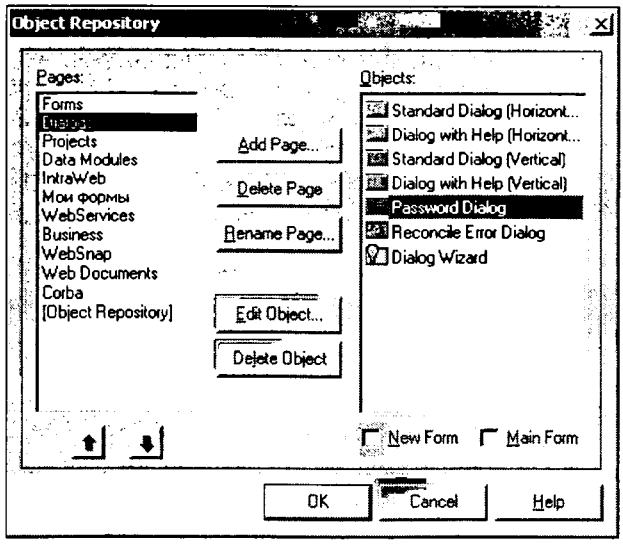


Рис. 1.13
Окно реорганизации Депозитария

В левой панели окна на рис. 1.13 вы можете выбрать одну из страниц и в правой панели увидеть состав этой страницы. Вы можете добавить (Add Page), удалить (Delete Page), переименовать (Rename Page) страницы Депозитария, поменять их последовательность с помощью кнопок со стрелками, выделить один из хранящихся объектов и удалить его (Delete Object), отредактировать (Edit Object) информацию об объекте. Вы можете также выделить в правой панели одну из форм и включить флажок Main Form (главная форма). Тогда при открытии нового проекта будет появляться не обычная пустая форма, а именно эта помеченная в Депозитарии как главная. Если для одной из форм включить флажок New Form (новая форма), то именно эта форма, а не пустая, будет включаться в проект при выполнении команды File | New | Form.

Если вы выделите в правой панели не форму, а проект (включенный вами или один из расположенных на странице Projects), то вместо индикаторов Main Form и New Form появится индикатор New Project. Если его включить, то именно этот проект будет в дальнейшем открываться при создании вами нового проекта: при выполнении команды File | New | Application и даже при щелчке на пиктограмме Application в окне New Items.

В окне рис. 1.13 вы можете и реорганизовывать страницы Депозитария. Для этого достаточно перетащить мышью объект из правого окна на нужную страницу в левом окне. Если же перетащите объект в строку [Object: Repository], то он не будет виден ни на одной странице Депозитария, но храниться будет. И когда потребуется, его можно будет взять из объектов этой строки и перенести на любую страницу.

Часто в процессе разработки вам надо создавать различные варианты вашего проекта, различающиеся только какими-то деталями. Из сравнения этих вариантов вы в конечном итоге выбираете, какой из них лучше, и на его основе создаете итоговый проект. Для создания вариантов вам требуется скопировать в другой каталог или под другим именем существующий проект, чтобы сделать в нем какие-то изменения. В разд. 1.4 было рассмотрено копирование файлов проекта средствами Windows. Но часто это проще делать средствами ИСР Delphi. Откройте проект,

который хотите скопировать, и выполните команду File | Save Project As. В диалоговом окне сохранения файла можете выбрать каталог и имя сохраняемого проекта. Далее выделите в окне Редактора Кода файл модуля и выполните команду File | Save As. При этом вы сохраните в новом каталоге или под новым именем ваш модуль. Если в проекте несколько модулей, надо повторить эту операцию, выделяя поочередно в окне Редактора Кода каждый из них. В заключение выполните команду File | Save All. Все файлы проекта будут сохранены, и вы сможете на их основе создавать новый вариант, не портя прежние файлы.

1.6 Некоторые итоги

В заключение давайте вспомним то, что было рассмотрено в этой главе. В разделе 1.1 были введены основные термины (объект, свойство, метод, сообщение и событие), используемые в ООП — объектно-ориентированном программировании. Рассмотрены принципы инкапсуляции и скрытия данных — основополагающие в ООП. Как вы увидите в следующей главе, вытекающее из этих принципов разделение интерфейса и реализации заложено в структуре любой программы и отдельных программных модулей. Изложены также предварительные сведения о классах и наследовании. Конечно, в дальнейшем эти понятия будут рассмотрены значительно подробнее.

Последующие разделы 1.2 и 1.3 позволили вам ознакомиться с основами работы в ИСР Delphi и создать ваши первые приложения для Windows. При изучении этих разделов вы использовали ряд команд меню Delphi и ряд быстрых кнопок. Имеет смысл просуммировать эту информацию и заодно привести список некоторых других команд и быстрых кнопок, которые будут использоваться в дальнейшем.

Инструментальные панели быстрых кнопок для Delphi 7 представлены на рис. 1.14. Панель Интернет по умолчанию невидима. В Delphi 6 ее состав несколько отличается от приведенного, а в Delphi 5 она вообще отсутствует. В Delphi 4 нет и панели настройки конфигурации (рис. 1.14 б). Назначение размещенных на панелях быстрых кнопок можно узнать из ярлычков, появляющихся, если вы поместите курсор мыши над соответствующей кнопкой и на некоторое время задержите его. В табл. 1.1 приведены пиктограммы этих кнопок, соответствующие им команды меню и горячие клавиши, а также краткие пояснения.

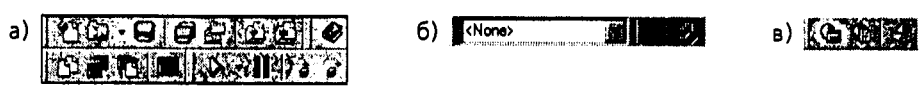


Рис. 1.14
Инструментальные панели в Delphi 7:
основные (а), панель настройки конфигурации (б) и панель Интернет (в)

Таблица 1.1. Быстрые кнопки, команды меню, горячие клавиши

Пиктограмма	Команда меню / горячие клавиши	Пояснение команды
	File | New | Other	Открыть проект или модуль из Депозитария.
	File | Open File | Reopen	Открыть файл проекта, модуля, пакета. Кнопочка со стрелкой справа от основного изображения соответствует команде Reopen, позволяющей открыть проект или файл из списка недавно использовавшихся.
	File | Save As File | Save (Ctrl–S)	Сохранить файл модуля, с которым в данный момент идет работа.
	File | Save All	Сохранить все (все файлы модулей и файл проекта).
	File | Open Project (Ctrl–F11)	Открыть файл проекта.
	Project | Add to Project (Shift–F11)	Добавить файл в проект.
	Project | Remove from Project	Удалить файл из проекта.
	Help | Contents	Вызвать страницу Содержание встроенной справки.
	View | Units (Ctrl–F12)	Переключиться на просмотр текста файла модуля, выбираемого из списка.
	View | Forms (Shift–F12)	Переключиться на просмотр формы, выбираемой из списка.
	View | Toggle Form/Unit (F12)	Переключиться между формой и соответствующим ей файлом модуля.
	File | New | Form	Включить в проект новую форму.
	Run | Run (F9)	Выполнить приложение. Кнопочка со стрелкой справа от основного изображения позволяет выбрать выполняемый файл, если вы работаете с группой приложений.
	Run | Program Pause	Пауза выполнения приложения и просмотр информации CPU. Кнопка и соответствующий раздел меню доступны только во время выполнения приложения.
	Run | Trace Into (F7)	Пошаговое выполнение программы с заходом в функции.
	Run | Step Over (F8)	Пошаговое выполнение программы без захода в функции.
	View | Desktops | Save Desktop	Сохранение текущей конфигурации окна (начиная с Delphi 5).

Пиктограмма	Команда меню / горячие клавиши	Пояснение команды
	View | Desktops | Set Debug Desktop	Установка конфигурации окна при отладке (начиная с Delphi 5).
	File | New | Other | WebSnup | WebSnup Data Module	Создание модуля данных приложения WebSnup для Web (начиная с Delphi 6).
	File | New | Other | WebSnup | WebSnup Page Module	Создание модуля страницы приложения WebSnup для Web (начиная с Delphi 6).
	File | New | Other | WebSnup | WebSnup Application	Создание приложения WebSnup для Web (начиная с Delphi 6).
	–	Вызов внешнего редактора страниц Web, который устанавливается кнопкой Edit на странице Internet окна опций, вызываемого командой Tools | Environment Options (только в Delphi 6).

На рис. 1.14 и в табл. 1.1 приведен стандартный состав инструментальных панелей быстрых кнопок. Однако Delphi предоставляет вам широкие возможности настроить панели по своему усмотрению, добавить в них какие-то быстрые кнопки для часто применяемых вами команд, убрать кнопки, которыми вы редко пользуетесь, сделать некоторые из инструментальных панелей невидимыми.

Конечно, пока вы познакомились далеко не со всеми быстрыми кнопками и командами, приведенными в табл. 1.1. Многие из них вы освоите значительно позднее. Но такие из уже описанных кнопок и команд, как File | Open, File | Reopen, File | Save All, View | Units, View | Forms, View | Toggle Form/Unit, Run | Run, следует запомнить. Так же, как команды Project | Build, View | Object Inspector и Project | Options, не имеющие соответствующих быстрых кнопок. Полезно также запомнить сочетания горячих клавиш, соответствующие различным часто используемым командам. Все это существенно облегчит вам в дальнейшем разработку приложений. А если вы в процессе изучения последующих глав забудете, как выполнять какую-то операцию, вернитесь к этому разделу и табл. 1.1 вам поможет.

В разд. 1.2 и 1.3 вы познакомились также с несколькими компонентами: метками **Label**, кнопками **Button**, окнами редактирования **Edit** и **Memo**. Знакомство это сугубо предварительное. В дальнейшем вы узнаете об этих и других компонентах значительно больше. Но то, что вы уже узнали о них, полезно запомнить, так как далее я буду считать, что с основными свойствами этих компонентов и с методикой создания обработчиков событий компонентов вы уже знакомы.

1.7 Проверьте себя

1.7.1 Вопросы для самопроверки

1. Что такое объектно-ориентированное программирование?
2. Что такое свойство объекта?

3. Зачем и в каких случаях имеет смысл программно создавать и уничтожать объекты во время выполнения?
4. Что такое конструкторы и деструкторы объектов?
5. Что такое принципы инкапсуляции и скрытия данных и зачем они применяются?
6. Что такое API Windows?
7. Что такое наследование классов?
8. Перечислите последовательность операций при создании нового проекта
9. Зачем надо давать осмысленные имена файлам проекта и модуля?
10. Почему нельзя файлам проекта и модуля давать одинаковые имена?
11. Почему в процессе проектирования полезно изменять имена компонентов, и как это можно делать?
12. Что делать, если вы случайно закрыли окно формы, Редактора Кода или Инспектора Объектов?
13. Как переключиться между формой и окном Редактора Кодов с загруженным в нем текстом модуля?
14. Что такое свойство времени выполнения, можно ли задать его значение с помощью Инспектора Объектов?
15. Какой смысл имеет понятие родительского компонента, и что означают свойства ParentColor компонентов?
16. Значение какого свойства надо изменить, чтобы задать надпись в заголовке окна формы, в метке, на кнопке?
17. Как можно в каких-то компонентах, размещенных на форме, задать полужирный шрифт (курсив, подчеркивание), оставив в других компонентах обычный шрифт?
18. Как изменить цвет надписи метки, кнопки?
19. Как изменить цвет фона метки, кнопки?
20. Как удалить текст из окна Memo?
21. Как добавить строку текста в окно Memo?
22. Какие файлы надо скопировать при переносе проекта Delphi с одного компьютера на другой?
23. Как занести в Депозитарий свою собственную форму?
24. В чем различие режимов Copy, Inherit и Use при заимствовании формы из Депозитария?

1.7.2 Задачи

1. Создайте приложение, содержащее кнопки с надписями «Черный», «Белый», «Красный», и пусть по щелчку на них окно формы окрашивается в соответствующий цвет (если забыли, как это делается, посмотрите разд. 1.2.2).
2. Дополните приложение пункта 1 окном Memo, и пусть кнопки окрашивают в соответствующий цвет фон этого окна, не затрагивая цвет формы.
3. Измените приложение пункта 2 так, чтобы окрашивался цвет текста, а не фона, окна Memo.

4. Дополните приложение *Charact* (разд. 1.3) еще одним окном редактирования, в котором пользователь может вводить группу (подразделение, класс), к которому относится студент (сотрудник, учащийся). И пусть по щелчку на кнопке в качестве второй строки в окно Memo заносится текст вида «Студент ... ..., группа ...».
5. Создайте приложение, содержащее кнопку, метку и окно Edit. По щелчку на кнопке в метке должен отображаться текст вида «Я освоил материал первой главы на ...», где вместо многоточия должна быть оценка, вводимая пользователем в окно Edit. Задайте осмысленный русский текст в заголовке окна формы.

Глава 2

Основы программирования на языке Object Pascal

В этой главе:

- *вы ознакомитесь с синтаксисом языка Object Pascal*
- *изучите целые, действительные и булевы типы данных и все операции, применимые к ним*
- *изучите все операторы языка Object Pascal*
- *научитесь создавать разветвленные программы и программы с циклическими фрагментами*
- *освоите методику создания и отладки алгоритмов*
- *познакомитесь со многими реальными алгоритмами, включая алгоритмы шифровки и дешифровки текстов, решения нелинейных уравнений, рекурсивные алгоритмы*

2.1 Немного истории

В 1970 году швейцарский ученый Никлаус Вирт разработал язык программирования, названный им Pascal в честь математика и философа семнадцатого столетия Блеза Паскаля. Язык был разработан для обучения структурному программированию, отличался строгостью, логичностью, и скоро стал основным языком при обучении в школах и университетах всего мира. Внедрение этого языка в практику дало огромный эффект, приучило программистов к дисциплинированному написанию программ, отличающихся ясностью, простотой тестирования и отладки, легкостью модификации. Впрочем, язык использовался не только для обучения, но и для разработки серьезных программ. Пожалуй, наиболее распространенной реализацией языка Pascal стал пакет Turbo Pascal фирмы Borland. Он появился в 1983 году, а год спустя был перенесен в среду MS DOS. Пакет Turbo Pascal предоставлял пользователю очень удобную среду разработки (конечно, с учетом возможностей DOS, поскольку Windows с его развитым сервисом еще не существовал), а компилятор Turbo Pascal отличался быстротой и эффективностью.

Было создано 7 версий пакета Turbo Pascal, после чего фирма Borland завершила совершенствование этого продукта. Связано это было с развитием Windows, падением значения DOS, а главное — с появлением концепции объектно-ориентированного программирования (ООП). Начала развиваться новая линия продуктов фирмы Borland — среда Delphi, реализующая возможности ООП и визуального проектирования. С ее появлением пакет Turbo Pascal и его версия языка Pascal стали достоянием истории. Конечно, они составили веху в истории программирования и заслуживают глубокого уважения. Но сейчас язык Turbo Pascal уже давно не является живым языком программирования и существует в наше время примерно на

тех же правах, как латинский или древнегреческий. Специалистам и историкам программирования полезно иметь о нем представление. Но никто не создает, и не будет создавать реальных программ на этом языке. Те профессионалы, которым приходится решать задачи в DOS (есть такие области программирования), предпочитают, естественно, C++. А рядовым разработчикам и тем более начинающим программистам не имеет смысл его изучать. Практика показывает, что овладеть основами работы с Delphi проще, чем обучиться работе с Turbo Pascal. Так что оправданием для изучения Turbo Pascal может служить только одно — крайне слабые компьютеры, на которых нельзя поставить Windows и Delphi. Можно надеяться, что этот аргумент скоро исчезнет из нашей жизни.

Язык Pascal в Delphi был существенно дополнен по сравнению с версией Turbo Pascal и стал называться языком Object Pascal, что подчеркивает его объектную направленность. А в 2002 году после появления новых сетевых технологий, предъявивших свои требования к языкам программирования, Object Pascal подвергся новой серьезной модификации, и теперь его создатели называют его языком Delphi. Впрочем, изучение этих нововведений выходит далеко за рамки изучения основ программирования. Так что в данной книге мы дальше языка Object Pascal не пойдем.

Язык Object Pascal является универсальным языком программирования, не ориентированным на какие-то специальные области применения. Так что если сравнивать его с другими языками, то в настоящее время можно отметить, пожалуй, только один язык аналогичного назначения — язык C++. Этот язык появился раньше, чем Object Pascal, и объектная ориентация в нем реализована тоже раньше. Так что Object Pascal постоянно вынужден догонять C++. В последних версиях Object Pascal реализовано почти все, что имеется в C++. Пожалуй, отставание наблюдается только в таких элементах, отсутствующих в Object Pascal, как шаблоны и перегрузка операций (последнее реализовано в Object Pascal, но достаточно сложно). Так что, хотя язык C++, конечно, более мощный и обладает более богатыми библиотеками, я рискну утверждать, что на Object Pascal можно реализовать все, что можно реализовать на C++. Причем в 99 случаях из 100 реализация на Object Pascal получится более простой с точки зрения программирования, хотя, иногда, не столь эффективной. Язык Object Pascal, как и его предшественники линии Pascal, подкупает своей простотой, логичностью и наглядностью. Поэтому обучиться программированию на Object Pascal гораздо проще, чем на C++. Похоже, авторы и сторонники C++ тоже начинают осознавать не во всем оправданную усложненность этого языка. В 2000 году корпорация Microsoft анонсировала новый язык C# (произносится как Си шарп). Это язык, уже завоевавший своих сторонников и свою нишу в новых продуктах Microsoft, во многом похож на C++, но намного проще. Кстати, один из руководителей группы, создавшей C#, — Андерс Хейльсберг, создатель языка Turbo Pascal и руководитель коллектива, создавшего Delphi. Так что, хотя это по соображениям конкурентной борьбы не афишируется, в новом языке использованы и некоторые подходы, характерные для Object Pascal.

Будущее языка C# пока не очевидно. Должно пройти время, чтобы понять его перспективы. Так что пока ничего лучше для обучения основам программирования, чем Object Pascal, на мой взгляд, не придумано. Изучив этот язык, вы сможете создавать как простые, так и весьма сложные приложения для любой области применения. А если потом потребуется, то, владея языком Object Pascal, вы легко освоите любой другой язык.

2.2 Синтаксис языка

Основные синтаксические правила записи программ на языке Object Pascal сводятся к следующему:

- Все используемые типы, константы, переменные, функции, процедуры должны быть объявлены или описаны до их первого использования.
- Прописные и строчные буквы идентичны. Например, имена **LABEL1**, **Label1** и **label1** идентичны. В программировании используется понятие *идентификатор* — это имя, которое вы или система присваивает переменной, функции, объекту и т.д. При записи идентификаторов могут использоваться латинские буквы, цифры, символ подчеркивания "_". Идентификатор не может начинаться с цифры и не может содержать пробелов. Длина идентификатора не ограничена, но воспринимается не более 255 первых символов идентификатора. Впрочем, реально лучше использовать короткие, но осмысленные идентификаторы.
- При ссылках на идентичные идентификаторы, описанные в разных местах, например, в разных модулях или в разных объектах, используется нотация с точкой, в которой сначала перечисляются идентификаторы объектов, разделенные символами точки. Например: **Unit2.A** — переменная или объект **A**, объявленный в модуле **Unit2**. Или **Form2.Label1.Caption** — свойство **Caption** метки **Label1**, размещенной на форме **Form2**.
- Каждое предложение языка кончается символом точка с запятой (";"). Немногие исключения из этого правила будут оговорены особо. В частности, точку с запятой можно не ставить (а можно ставить) перед ключевым словом **end**.
- В строке может размещаться несколько операторов. Однако с точки зрения простоты чтения текста этим не надо злоупотреблять. Вообще надо писать программу так, чтобы ее было легко читать и вам, и постороннему человеку, которому, может быть, придется ее сопровождать. Надо выделять объединенные смыслом операторы в группы, широко используя для этого отступы и комментарии.
- Программа или отдельный модуль завершаются оператором "**end.**" (ключевое слово **end** с символом точки).
- Комментарии в тексте заключаются в фигурные скобки: {текст комментария}. Вместо фигурных скобок можно использовать символы круглых скобок с символами звездочки "*": (*текст комментария*). Комментарии, заключенные в фигурные скобки или в круглые скобки со звездочками, могут вводиться в любом месте текста, в частности, внутри операторов, и занимать любое количество строк. Текст комментария в фигурных скобках не может начинаться с символа доллара, поскольку сочетание символов {$ воспринимается как начало директивы компилятора. Еще один способ введение комментария — размещение его после двух символов «слэш» ("//"). Этот комментарий должен занимать конец строки, в котором он введен, и не может переходить на следующую строку. Любой текст в строке, помещенный после символов "//" воспринимается как комментарий. Ниже приведен рассмотренный в разд. 1.3 код обработчика события, снабженный комментариями:

```
{ Занесение в окно Memo1 начала характеристики той личности, чьи
фамилия, имя и отчество записаны в окнах Edit1 и Edit2 }

Memo1.Clear;        // Удаление текста в окне Memo1
Memo1.Lines.Add('Х А Р А К Т Е Р И С Т И К А');
Memo1.Lines.Add('Студент ' + Edit1.Text + ' ' + Edit2.Text);
Memo1.SetFocus;     // Передача фокуса окну Memo1
```

- Операторные скобки **begin...end** выделяют составной оператор. Все операторы, помещенные между ключевыми словами **begin** и **end**, воспринимаются синтаксически как один оператор.

Совет

При записи составного оператора после того, как написали слово begin, сразу перейдите на следующую строку и напишите завершающее слово end. А потом вставляйте между begin и end требуемые операторы. Этот прием обеспечит завершение каждого составного оператора, т.е. равное число слов begin и end. Иначе почти неизбежно в сложных программах со множеством вложенных друг в друга составных операторов где-то не будет хватать завершающего end, и вы потратите много времени на поиск этой ошибки. И еще один совет — в подобных вложенных друг в друга составных операторах помечайте комментариями их начало и конец, чтобы сразу было ясно, к какому слову begin относится каждое слово end. Это необходимо для того, чтобы при последующей модификации программы был ясно, куда надо вставлять дополнительные операторы.

Все это должно выглядеть примерно так:

```
begin // первый begin
  ...
  begin // второй begin
    ...
  end; // конец второго begin
  ...
end; // конец первого begin
```

Помимо кодов языка Object Pascal и комментариев, текст приложения может содержать *директивы компилятора*, точнее директивы *препроцессора*, срабатывающего в начале компиляции. О назначении компилятора (создании из ваших исходных текстов исполняемого файла) уже говорилось в разд. 1.2.4. Компилятор может работать в разных режимах, и с помощью директив компилятора вы можете управлять этими режимами. Многие директивы Delphi включает в проект автоматически. Вам не обязательно в них разбираться. Только будьте осторожны и не удалите случайно эти директивы из текста вашего приложения. Кроме того, многие директивы заданы по умолчанию, и вы их даже не увидите в тексте проекта. Но, как вы увидите далее, в ряде случаев вам надо изменить эти установленные по умолчанию директивы, или задать собственные директивы.

Каждая директива компилятора заключается в фигурные скобки и начинается с символа "$". После этого символа без пробела должно быть написано имя директивы. Например, **{$I+}**, **{$I−}**.

Имеется три вида директив: ключевые директивы, директивы параметров и директивы условной компиляции. Ключевые директивы включают или выключают

те или иные возможности компилятора. Обычно используется сокращенная форма их записи, при которой имя директивы состоит из единственной буквы. После имени без пробела должен следовать символ "+" или "–", что означает включение или выключение соответствующей опции. Например:

```
{$I+} // Включение директивы I - проверки ошибок ввода/вывода
```

и

```
{$I-} // Выключение директивы I - проверки ошибок ввода/вывода
```

Директивы параметров определяют значения различных параметров, используемых при компиляции, например, имена файлов или размеры отводимой памяти. Обычно Delphi указывает их автоматически, так что вам не приходится о них заботиться.

Ключевые директивы и директивы параметров имеют установки по умолчанию. Работая в ИСР Delphi, вы можете установить эти значения по умолчанию установкой соответствующих индикаторов страницы Compiler диалогового окна опций проекта, вызываемого командой Project | Options. Если вы захотите узнать установленные директивы, нажмите клавиши Ctrl-O и затем еще раз нажмите "O". В начало вашего файла в Редакторе кода вставятся явным образом все значения директив, используемые в данный момент.

Директивы условной компиляции позволяют в зависимости от задания тех или иных условий компилировать или исключать из компиляции отдельные фрагменты кода. Эти директивы могут использоваться, в частности, для организации отладки приложения. Но мы не будем на них останавливаться — посмотрите их, если хотите, во встроенной справке Delphi или в справке [3].

2.3 Структура программы

Программа, которую строит Delphi в процессе проектирования вами приложения, основана на *модульном принципе*. Сама головная программа получается предельно простой и короткой. Она состоит из объявления списка используемых модулей и нескольких операторов, которые создают объекты тех форм, которые вы задумали, и запускают выполнение приложения.

Принцип модульности очень важен для создания надежных и относительно легко модифицируемых и сопровождаемых приложений. Четкое соблюдение принципов модульности в сочетании с принципом скрытия информации (см. разд. 1.1.1) позволяет внутри любого модуля проводить какие-то модификации, не затрагивая при этом остальных модулей и головную программу.

Все объекты компонентов размещаются в объектах — формах, с которыми вы уже познакомились в гл. 1. Для каждой формы, которую вы проектируете в своем приложении, Delphi создает отдельный *модуль*. Именно в модулях и осуществляется программирование задачи. В обработчиках событий объектов — форм и компонентов, вы помещаете всё свои алгоритмы. В основном они сводятся к обработке информации, содержащейся в свойствах одних объектов, и задании по результатам обработки свойств других объектов. При этом вы постоянно обращаетесь к методам различных объектов.

2.3.1 Структура файла головной программы

Когда вы проектируете приложение, Delphi автоматически создает код головной программы и отдельных модулей. В модули вы вводите свой код, создавая обработчики различных событий. Но головную программу, как правило, вы не трогаете и даже не видите ее текст. Только в исключительных случаях вам надо что-то изменять в тексте головной программы, сгенерированном Delphi. Тем не менее, хотя бы ради этих исключительных случаев, надо все-таки представлять вид головной программы и понимать, что означают ее операторы.

Чтобы увидеть текст головной программы вашего проекта, модуль которого открыт в окне Редактора Кода, надо выполнить команду Project | View Source. Головной файл загрузится в окно Редактора Кода. Он имеет следующую структуру:

```
Program <имя>;
<Объявления подключаемых модулей, а также объявления локальных
 для головного файла типов, классов, констант, переменных,
 описания локальных функций и переменных>

begin
<операторы тела программы>

end.
```

В Delphi головной файл обычно не содержит ничего, кроме операторов инициализации приложения, создания форм и запуска приложения.

Типичный головной файл в Delphi имеет вид:

```
program Project1;

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

{$R *.RES}

{Здесь можно поместить описания каких-то констант, переменных,
 функций, процедур. Все это будет доступно только в пределах
 данного файла.}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Имя программы совпадает с именем файла, в котором сохранен проект. Это же имя присваивается выполняемому файлу приложения.

После заголовка в тексте программы располагается предложение **uses** (см. разд. 2.3.3). В этом предложении перечисляются модули, загружаемые программой — системные и модули приложения (в частности, модули всех форм). В приведенном примере подразумевает, что в проекте созданы две формы с именами **Form1** и **Form2** в модулях с именами *Unit1* и *Unit2*. Названия форм включаются в текст в виде комментариев.

Следующая строка текста — **{$R *.RES}** представляет собой директиву компилятора, которую мы рассматривать не будем. Затем после ключевого слова **begin** и до последнего завершающего программу оператора **end** с точкой (**end.**) записано тело программы.

Первый выполняемый оператор в теле программы инициализирует приложение, два следующих — создают объекты форм **Form1** и **Form2**, последний — начинает выполнение приложения.

В принципе, вы можете ввести какой-то свой дополнительный код в головную программу. Например, вам может потребоваться при запуске приложения на выполнение провести какие-то его настройки. Или сделать какой-то запрос пользователю и в зависимости от ответа создавать или не создавать те или иные формы. В подобных случаях вы можете ввести в код головного файла соответствующие процедуры, обратиться к ним после выполнения инициализации приложения, но до создания объектов форм, а затем, например, создавать или не создавать отдельные формы исходя из результатов работы ваших процедур. Но подобный подход нарушает принцип модульности. Так что я не рекомендую в подавляющем большинстве случаев что-то менять в головном модуле. Все необходимые действия можно реализовать в модулях.

2.3.2 Структура модуля

Теперь рассмотрим структуру модулей, из которых состоит программа на языке Object Pascal. Каждый модуль в общем случае имеет следующую структуру:

```
unit <имя модуля>;

interface       // Открытый интерфейс модуля

{Сюда могут помещаться списки подключаемых модулей, объявления типов,
констант, переменных, функций и процедур, к которым будет доступ из
других модулей.}

implementation // Реализация модуля

{Сюда могут помещаться списки подключаемых модулей, объявления типов,
констант, переменных, к которым не будет доступа из других модулей.
Тут же должны быть реализации всех объявленных в разделе interface
функций и процедур, а также могут быть реализации любых дополнитель-
ных, не объявленных ранее функций и процедур.}

initialization // не обязательный
<Операторы, выполняемые один раз при первом обращении к модулю>

finalization   // не обязательный
<Операторы, выполняемые при любом завершении работы модуля>

end.
```

Раздел **interface** представляет собой внешний интерфейс модуля. Поэтому подключаемые в нем модули, объявленные типы, классы, константы, переменные, функции и процедуры доступны внешним модулям, обращающимся к данному модулю.

Раздел **implementation** представляет собой реализацию модуля. Все подключаемые в нем модули, объявленные типы, классы, константы, переменные, функции и процедуры доступны только в пределах данного модуля. Основное тело модуля составляют коды, реализующие объявленные функции и процедуры.

Раздел **initialization** включает в себя операторы, которые выполняются только один раз при первом обращении программы к данному модулю. Этот раздел не является обязательным. В нем могут помещаться какие-то операторы, производящие начальную настройку модуля.

При наличии в программе нескольких модулей, содержащих разделы **initialization**, последовательность выполнения операторов этих разделов определяется последовательностью указания соответствующих модулей в операторах **uses** (см. разд. 2.3.3). Например, если в головной программе имеется оператор

```
uses unit1, unit2, ...
```

то сначала будет выполняться (если он есть) раздел **initialization** модуля *unit1*, а затем раздел **initialization** модуля *unit2*. Другой пример. Если в головной программе имеется оператор

```
uses unit1, unit2, unit3...
```

и при этом в модуле *unit1* имеется оператор

```
uses unit3, ...
```

то последовательность выполнения разделов **initialization** модулей будет следующей: *unit3*, *unit1*, *unit2*. Это объясняется следующим образом. Первым загружается модуль *unit1*, поскольку он расположен первым в списке **uses** головного файла программы. Но в его операторе **uses** указан модуль *unit3*. Значит, загружается этот модуль. Затем выполняются операторы раздела **initialization** этого модуля. После этого выполняются операторы раздела **initialization** загруженного модуля *unit1*. Затем загружается модуль *unit2*, расположенный вторым в списке **uses** головного файла программы, и выполняются операторы его раздела **initialization**. Третьим в списке **uses** головного файла следует *unit3*. Но, поскольку он уже был загружен в момент загрузки *unit1*, то повторная его загрузка не производится.

Раздел **finalization** включает в себя операторы, которые выполняются только один раз при любом завершении работы программы: нормальном или аварийном. Этот раздел не является обязательным. В нем могут помещаться какие-то операторы, производящие зачистку «мусора» — удаление временных файлов, освобождение ресурсов памяти и т.п. Введение раздела **finalization** не разрешается, если в модуле нет раздела **initialization**. Последовательность выполнения операторов разделов **finalization** различных модулей обратная той, которая была описана выше для разделов **initialization**.

2.3.3 Предложение uses — подключение модулей

Предложение, определяющее подключаемые к проекту или используемые в данном модуле другие модули, начинается с ключевого слова **uses**, после которого следует список имен модулей, разделяемых запятыми. Например:

```
uses Windows, Messages, SysUtils, Classes, Unit2, MyUnit;
```

Предложение **uses** в головном файле программы может также содержать после имени модуля ключевое слово **in**, после которого указывается имя файла, содержащего модуль (см. разд. 2.3.1). Например:

```
uses Unit1 in 'Unit1.pas', Unit2 in 'c:\examples\Unit2.pas';
```

Форма с **in** используется в случаях, если Delphi неясно, где надо искать соответствующие файлы.

При взаимных ссылках модулей друг на друга не разрешаются циклические ссылки с помощью предложений **uses**, размещенных в разделах **interface**. Такие циклы надо разрывать, перенося одно или все предложения **uses** из разделов **interface** в разделы **implementation** модулей. Например, недопустимыми циклическими ссылками являются следующие предложения **uses** в модулях *Unit1* и *Unit2*:

```
unit Unit1
...
interface
uses ... Unit2;
implementation
...


unit Unit2
...
interface
uses ... Unit1;
implementation
...
```

В этом случае компилятор выдаст сообщение об ошибке:

```
[Fatal Error] Unit1.pas(3): Circular unit reference to 'Unit1'
```

и приложение не будет скомпилировано. Выходом из положения является вариант, когда в одном модуле ссылка **uses** помещена в раздел **interface**, а в другом в **implementation**, или в обоих модулях соответствующие ссылки помещены в разделах **implementation**:

```
unit Unit1
...
interface
...
implementation
uses ... Unit2;
...


unit Unit2
...
interface
...
implementation
uses ... Unit1;

...
```

Работая в Интегрированной Среде Разработки Delphi, ссылки между модулями вашего приложения проще всего вводить не вручную, а с помощью команды File | Use Unit. Соответствующий оператор **uses** автоматически заносится этой командой в раздел **implementation**.

Нередко требуется ввести вручную ссылку на библиотечный модуль, в котором объявлены те или иные функции и процедуры, используемые в приложении. Если вызовы этих функций и процедур содержатся только в разделе **implementation**, целесообразно записывать оператор **uses** в этом же разделе. Но если имена функций, процедур, констант или переменных, объявленных в библиотечном модуле, используются в интерфейсе вашего модуля, соответствующий оператор **uses** должен размещаться в интерфейсе.

2.4 Математические выражения

Программы пишутся для того, чтобы что-то вычислять. Поэтому практически в каждой программе используются те или иные математические выражения. Математические выражения могут содержать арифметические операции, круглые скобки, константы, переменные, вызовы функций. Рассмотрим все эти составляющие выражений.

2.4.1 Переменные, арифметические типы данных, числовые константы

Понятие переменной в программировании имеет тот же смысл, что и в математике. Если в математике вы вводите в выражение некоторую переменную **A**, вы подразумеваете, что она может принимать различные значения. И в зависимости от этого значения будет изменяться результат записанного вами математического выражения. В программировании смысл переменной аналогичен. Переменная — это некоторая область в памяти компьютера, в которую программа может записывать различные значения. Чтобы в выражениях можно было ссылаться на эту область, ей присваивается имя — идентификатор. Правила записи идентификаторов см. в разд. 2.2.

Переменные могут быть разных *типов*. Тип определяет то множество значений, которые может принимать переменная. Типы арифметических переменных делятся на две основные группы: целые типы и действительные, часто называемые типами с плавающей запятой. Но внутри каждой из этих групп есть по нескольку типов, характеризующих различную точность и допустимые диапазоны чисел. Дело в том, что для чисел с большим числом разрядов требуется выделять больше памяти. Так что тип переменной надо выбирать исходя из возможных ее значений.

В табл. 2.1 перечисляются целые типы для Delphi 7, даются диапазоны их изменений, указываются затраты памяти под их хранение и указывается, может ли тип хранить отрицательные числа. Единицей измерения памяти является *байт*, который часто называют также *машинным словом*. Эта единица содержит 8 двоичных разрядов — *битов*. Бит — это элемент, который может принимать одно из двух значений: 0 или 1.

Таблица 2.1. Типы целых чисел

Тип	Диапазон значений	Требования к памяти в байтах	Знаковый (может ли хранить отрицательные числа)
Byte	0 ÷ 255	1	нет
Word	0 ÷ 65535	2	нет
Longword	0 ÷ 4294967295	4	нет
ShortInt	–128 ÷ 127	1	да
SmallInt	–32768 ÷ 32767	2	да
Cardinal	0 ÷ 4294967295	4	нет
Integer	–2147483648 ÷ 2147483647	4	да
LongInt	–2147483648 ÷ 2147483647	4	да
Int64	$-2^{63} \div 2^{63} - 1$	8	да

Родовыми типами (т.е. обеспечивающими максимальную производительность) среди перечисленных в таблице являются **Integer** и **Cardinal**. В настоящий момент тип **Integer** эквивалентен типу **LongInt**, но в последующих версиях это может быть изменено.

В большинстве случаев имеет смысл использовать тип **Integer**. Если значения переменных могут превышать допустимые для этого типа, надо использовать тип **Int64**. Если известно, что значения будут заведомо положительные и не превышающие 65535, можно использовать тип **Word**. Наименьшие затраты памяти обеспечивают типы **Byte** и **ShortInt**. Тип **Byte** нередко используется при работе с клавиатурой, так как число возможных стандартных символов компьютера соответствует этому типу.

В табл. 2.2 представлены данные о типах действительных данных, предназначенных для хранения чисел, имеющих дробную часть.

Таблица 2.2. Типы действительных чисел

Тип	Диапазон значений	Число значащих разрядов	Требования к памяти в байтах
Real48	$\pm 2.9 \cdot 10^{-39} \div \pm 1.7 \cdot 10^{38}$	11–12	6
Real	$\pm 5.0 \cdot 10^{-324} \div \pm 1.7 \cdot 10^{308}$	15–16	8
Single	$\pm 1.5 \cdot 10^{-45} \div \pm 3.4 \cdot 10^{38}$	7–8	4
Double	$\pm 5.0 \cdot 10^{-324} \div \pm 1.7 \cdot 10^{308}$	15–16	8
Extended	$\pm 3.6 \cdot 10^{-4932} \div \pm 1.1 \cdot 10^{4392}$	19–20	10
Comp	$-2^{63} + 1 \div 2^{62} - 1$	19–20	8
Currency	–922337203685477.5808 ÷ 922337203685477.5807	19–20	8

Родовым (т.е. обеспечивающим максимальную производительность) является тип **Real**, который в настоящий момент эквивалентен типу **Double**, но в последующих версиях это может быть изменено.

Тип **Currency** используется для представления денежных величин. В памяти он хранится как масштабированное в 10000 раз 8-байтовое целое. Благодаря этому при операциях с величинами типа **Currency** минимизируются ошибки округления, что очень важно для денежных расчетов. В выражениях, в которых смешаны величины типа **Currency** с величинами других действительных типов, значения **Currency** автоматически умножаются или делятся на 10000.

Поскольку затраты памяти под переменную определяются ее типом, то прежде, чем использовать переменную в выражениях, ее надо *объявить*.

Объявление переменной имеет вид:

```
var <список идентификаторов переменных> : <тип>;
```

После ключевого слова **var** может следовать не одно, а множество объявлений переменных. Каждое объявление может содержать список идентификаторов переменных, разделяемых запятыми, или один идентификатор.

Приведем примеры объявления переменных:

```
var I: integer;
    X, Y, Z: real;
```

После того как переменная объявлена, ее можно использовать в выражениях. Например:

```
I := 2;
X := 10.5;
Y := X + I;
```

В этих операторах использован уже знакомый вам по материалам гл. 1 оператор присваивания ":=". Он записывается в виде:

```
<переменная> := <выражение>;
```

где <переменная> — переменная любого типа, а <выражение> — любое допустимое выражение, совместимое по типу с переменной в левой части оператора. Оператор вычисляет значение выражения, записанного как правый операнд операции присваивания **:=**, и присваивает полученное значение переменной в левой части оператора.

Например, оператор

```
I := 2;
```

присваивает переменной **I** значение 2. Оператор

```
I := I + 1;
```

увеличивает значение переменной **I** на 1.

Переменные можно разделить на *локальные* и *глобальные*. Переменные, объявляемые в процедурах и функциях, являются локальными. Они существуют только во время выполнения соответствующей процедуры или функции. Т.е. память для них выделяется только при вызове соответствующей процедуры или функции и освобождается при возврате в вызвавшую процедуру. Переменные, объявленные вне процедур или функций, являются глобальными.

Начиная с Delphi 3, глобальные переменные можно инициализировать в их объявлениях, т.е. задавать им начальные значения. Это делается с помощью предложения:

```
<идентификатор переменной> : <тип> = <константное выражение>;
```

Приведем примеры инициализации переменных (во втором операторе использована операция деления — символ "/"):

```
var I: integer = 7;
    A: real = 5 / 100;
```

Локальные переменные инициализировать нельзя. В Delphi 1 нельзя инициализировать и глобальные переменные. В этих случаях выходом из положения может являться применение типизированных констант (см. разд. 2.4.9). Другой подход — программно задавать значение переменной перед первым ее использованием. Учтите, что если этого не сделать, то значение переменной непредсказуемо. Оно зависит от случайного состояния памяти в момент, когда под переменную выделяется соответствующая область памяти. Так что если вы написали код:

```
var I, J: integer;
...
J := I + 1;
```

то значения переменных **I** и **J** будут абсолютно непредсказуемы. Правда, компилятор Delphi выдаст в подобном случае замечание вида: «Variable 'I' might not have been initialized» — «Возможно, переменная 'I' не инициализирована». Но приложение все-таки будет скомпилировано и выполнено. А уж что оно будет вычислять — это никому неизвестно.

В приведенных примерах уже встречались константы: числа. Константы, являющиеся целыми числами, обычно записываются в десятичной форме. Например, 7, 22, –5. Константы, являющиеся действительными числами, могут записываться в десятичном виде с точкой, отделяющей целую часть от дробной, или с указанием порядка, после символа "e". Например, 6.2 (6,2), 7e5 ($7 \cdot 10^5$), –3.1e–4 ($-3{,}1 \cdot 10^{-4}$).

Для представления целых чисел могут использоваться также константы в шестнадцатеричной форме. Все мы привыкли к десятичной системе счисления. Для записи чисел в ней используются цифры от 0 до 9, а «цена» каждого разряда (цифры) числа в 10 раз выше «цены» предыдущего разряда. Компьютер использует другую — двоичную систему счисления. В ней только две цифры: 0 и 1, а цена каждого следующего разряда в 2 раза выше цены предшествующего разряда. Четыре двоичных разряда могут содержать числа от 0 до 15. Иногда удобно при программировании оперировать четверками двоичных разрядов, задавая целые чйсла в шестнадцатеричной системе счисления. Для представления числа, хранящегося в четырех двоичных разрядах (т.е. в одном шестнадцатеричном), используются шестнадцатеричные цифры: 0, 1, ..., 9, A, B, C, D, E, F. Как видите, приходится вводить для отображения цифр, отсутствующих в десятичной системе (от 10 до 15), латинские буквы. Цена каждого разряда шестнадцатеричного числа в 16 раз больше цены предыдущего разряда. Константа, представляющая целое число в шестнадцатеричном виде, начинается с символа доллара "$", после которого записывается до 8 шестнадцатеричных цифр. Диапазон значений шестнадцатеричных констант от $00000000 до $FFFFFFFF. Впрочем, шестнадцатеричные константы используются достаточно редко, так что не будем уделять им особое внимание.

2.4.2 Арифметические операции

В выражениях, приведенных в предыдущем разделе, уже использовались арифметические операции сложения (символ "+") и деления (символ "/"). Эти и другие операции применяются к выражениям — *операндам*. Большинство операций имеют два операнда, один из которых помещается перед знаком операции, а другой — после. Например, операция сложения "+" имеет два операнда: **X** и **Y** и складывает их. Такие операции называются бинарными. Существуют и унарные операции, имеющие только один операнд, помещаемый после знака операции. Например, запись **−X** означает применение к операнду **X** операции унарного минуса "−".

Арифметические операции применяются к действительным и целым числам. Определены следующие бинарные арифметические операции:

Обозначение	Операция	Типы операндов	Тип результата	Пример
+	сложение	integer, real	integer, real	`X + Y`
−	вычитание	integer, real	integer, real	`X - Y`
*	умножение	integer, real	integer, real	`X * Y`
/	деление действительных чисел	integer, real	real	`X / Y`
div	целочисленное деление	integer	integer	`I div 2`
mod	остаток целочисленного деления	integer	integer	`I mod 6`

Определены следующие унарные арифметические операции:

Обозначение	Операция	Типы операндов	Тип результата	Пример
+	Унарный плюс (подтверждение знака)	integer, real	integer, real	+7
−	Унарный минус (изменение знака)	integer, real	integer, real	−X

Почти все приведенные операции достаточно очевидны. Заслуживают отдельного рассмотрения только операции над целыми числами **div** и **mod**. Результат целочисленного деления **X div Y** равен результату деления **X/Y**, округленному в сторону нуля до ближайшего целого. Например, выражения **8 div 3** и **8 div 4** вернут 2.

Операция **mod** возвращает остаток от целочисленного деления своих операндов, т.е. выражение **X mod Y** эквивалентно выражению **X − (X div Y) * Y**. Например, **8 mod 4** вернет 0, **9 mod 4** вернет 1 и т.д. Это часто используется в тех случаях, когда надо обеспечить циклическое изменение какой-то переменной при изменении другой переменной. Например, если переменная **i** пробегает значения от 0, увеличиваясь каждый раз на 1, то переменная **j**, заданная выражением

```
j := i mod 3;
```

будет пробегать значения 0, 1, 2, 0, 1, 2, и т.д.

Последовательность выполнения операций в сложных выражениях может задаваться круглыми скобками. Например, выражение **(X + Y) * Z** обеспечит сначала сложение значений переменных **X** и **Y**, а затем умножение полученной суммы на **Z**. Если же скобки не ставятся, то считается, что операции "*", "/", **div** и **mod** имеют приоритет перед операциями "+", "–". А операции одного уровня приоритета выполняются слева направо. Так что арифметические операции выполняются в той же последовательности, как это принято в математике.

Ниже в таблице приведены примеры: в первом столбце — выражения, во втором — указанная скобками последовательность их вычисления согласно приведенным правилам, в последнем столбце — пояснения.

Выражение	Последовательность вычислений	Пояснения
`X + Y - Z`	`((X + Y) - Z)`	Все операции одного уровня приоритета
`X * Y / Z`	`((X * Y) / Z)`	Все операции одного уровня приоритета
`X * Y + Z / A`	`(X * Y) + (Z / A)`	Операции "*" и "/" имеют более высокий приоритет, чем операция "+"

Если в операции участвуют операнды разных типов, например, **integer** и **real**, то при вычислениях применяется автоматическое *приведение типов*. Оно сводится к тому, что тип операнда «младшего» типа (с меньшей точностью и затратами памяти) приводится к типу операнда «старшего» типа (с большей точностью и затратами памяти). Так что потери точности не происходит, и результат операции имеет тип, соответствующий «старшему» типу. Пусть, например, имеется следующий код:

```
var i: integer;
    a: real;
...
a := i + a;
```

При вычислении выражения, записанного в последнем операторе, сначала значение переменной **i** будет приведено к типу **real**, затем оно сложится со значением переменной **a**, и результат вычислений будет иметь тип **real**. Так что этот результат без дополнительного приведения типа присвоится переменной **a**.

Оператор присваивания также осуществляет при необходимости автоматическое приведение типа своей правой части к типу левой части. Правила приведения типов в этом операторе не отличаются от рассмотренных выше. Например, при выполнении оператора

```
a := i;
```

значение целой переменной **i** будет приведено к типу **real**. Но вот обратное преобразование типов невозможно. Например, оператор

```
i := a;
```

вызовет сообщение компилятора об ошибке: «Incompatible types: 'Integer' and 'Real'» — «Несовместимые типы 'Integer' и 'Real'». И справедливо, поскольку непонятно, как привести к целому числу значение переменной, которое может содержать дробную часть.

Помимо описанного автоматического приведения типов иногда может использоваться явное приведение типов с помощью следующей конструкции:

```
<идентификатор типа>(<выражение>)
```

Явное *приведение типа переменной* можно проводить для любых типов, имеющих одинаковый размер. В разд. 2.7.2 и 2.10 вы увидите примеры такого приведения. Но преобразовать подобным образом действительное число в целое невозможно. Если по смыслу кода все-таки требуется привести действительное число к целому, то надо пояснить, как это следует делать: то ли выделить целую часть числа, то ли округлить до ближайшего целого. Для этого применяются библиотечные функции **Ceil**, **Floor**, **Trunc**, **Round**. Функция **Ceil** округляет число до ближайшего целого в сторону увеличения, **Floor** — в сторону уменьшения, **Trunc** — в сторону нуля. Функция **Round** округляет до ближайшего целого. Ниже приведены примеры, поясняющие характер округления.

	a = 3.5	a = –3.5	a = 3
Ceil(a)	4	–3	3
Floor(a)	3	–4	3
Trunc(a)	3	–3	3
Round(a)	4	–4	3

Функции **Trunc** и **Round** объявлены в модуле *System*, который обычно косвенно, через другие модули подключен к проекту. А если вы используете функции **Ceil** и **Floor**, то учтите, что они объявлены в модуле *Math*. Этот модуль автоматически к приложению не подключается. Так что если вы его не подключите сами, то, записав, например, оператор

```
i := Ceil(R1);
```

получите сообщение компилятора об ошибке: «Undeclared identifier: 'Ceil'» — «Необъявленный идентификатор 'Ceil'». Надо включить в имеющееся предложение **uses** (см. разд. 2.3.3) ссылку на модуль *Math*, или лучше включить в раздел **implementation** предложение:

```
uses Math;
```

2.4.3 Тестовый пример

Давайте попробуем применить арифметические операции на практике. Создайте форму, содержащую два окна редактирования **Edit1** и **Edit2** и кнопку. Пусть в окна редактирования **Edit1** и **Edit2** пользователь будет вводить какие-то числа, а по щелчку пользователю будет показано окно сообщения с результатом применения к этим числам каких-то арифметических операций.

Вроде бы сделать это несложно, но надо сначала обсудить одну проблему. Текст, вводимый пользователем в окно редактирования, представляет собой строку типа **string**. А для арифметических операций этот текст надо преобразовать в число. Для такого преобразования строки в действительное число используется функция **StrToFloat**. При вызове функции в языке Object Pascal после имени функции ставятся круглые скобки, внутри которых записываются аргументы, пре-

дусмотренные описанием соответствующей функции. Для функции **StrToFloat** предусмотрен один аргумент — строка преобразуемого текста. Так что, например, выражение **StrToFloat(Edit1.Text)** возвращает действительное число, полученное в результате преобразования текста в окне **Edit1**. Описанная функция возвращает действительное число. Если пользователь должен писать в окне не действительное, а целое число, то преобразование теста в целое число осуществляется аналогичной функцией **StrToInt**.

Мы выяснили, как можно перевести текст, вводимый пользователем, в число. Далее эти числа можно использовать в любых математических выражениях. Но если численные результаты вычислений надо показать пользователю в какой-то метке или в окне, необходимо решить обратную задачу — перевести числа в текст. Это осуществляется функциями **FloatToStr** или **FloatToInt** в зависимости от того, действительное или целое число надо перевести в строку. В качестве аргумента в эти функции передается преобразуемое число, а возвращаемое функциями значение — строка текста.

Для отображения пользователю результатов расчета мы воспользуемся простейшей функцией **ShowMessage**, показывающей пользователю диалоговое окно с сообщением. В качестве аргумента в эту функцию передается строка сообщения.

Итак, давайте построим приложение, иллюстрирующее применение рассмотренных функций и арифметических операций. Пусть пользователь вводит в окна **Edit1** и **Edit2** длины сторон прямоугольника, а по щелчку на кнопке функцией **ShowMessage** пользователю отображается окно, пример которого показан на рис. 2.1, содержащее текст: "Площадь прямоугольника = ...".

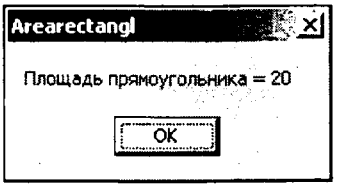


Рис. 2.1
Пример окна, отображающего результаты расчета

Обработчик щелчка на кнопке подобного приложения может иметь вид:

```
procedure TForm1.Button1Click(Sender: TObject);
var R1, R2, R3: real;
begin
 R1 := StrToFloat(Edit1.Text); // Первое число
 R2 := StrToFloat(Edit2.Text); // Второе число
 R3 := R1 * R2;                // Произведение
 ShowMessage('Площадь прямоугольника = ' + FloatToStr(R3));
end;
```

В этом коде объявлено три действительных переменных **R1**, **R2** и **R3**. Далее в переменную **R1** с помощью функции **StrToFloat** заносится число, введенное пользователем в окно **Edit1**, а в переменную **R2** аналогичным образом заносится число из окна **Edit1**. Следующий оператор вычисляет произведение этих чисел и сохраняет его в переменной **R3**. Затем вызывается диалоговое окно, и в него заносится строка вида "Площадь прямоугольника = ...", где вместо многоточия заносится значение переменной **R3**, преобразованное функцией **FloatToStr** в текст.

Выполните этот пример, и убедитесь, что он работает правильно. Если вы допустили в коде какие-то ошибки, и что-то работает неправильно, обратитесь к разд. 2.4.5, в котором рассказано о методике отладки приложений. А если вы допустили при написании кода какие-то синтаксические ошибки, получили сообщения компилятора об ошибках, и не знаете, что делать, посмотрите в конце книги Приложение, в котором описаны некоторые наиболее часто встречающиеся сообщения компилятора. Вероятно, вы найдете среди них то, которое выдал вам компилятор, поймете его смысл и сможете исправить ошибку.

Честно говоря, в приведенном примере можно было бы обойтись без объявления и использования переменных. Например, заведомо не нужна переменная **R3**. Она используется для хранения результата вычисления, но этот результат нужен только для того, чтобы показать его пользователю. Так что можно не отводить для него переменную, а прямо передать соответствующее выражение в качестве аргумента в функцию **FloatToStr**. Попробуйте удалить (можно просто закомментировать, поместив в начале строки символы "//") оператор, задающий значение **R3**. А последний оператор, запишите в виде:

```
ShowMessage('Площадь прямоугольника = ' + FloatToStr(R1 * R2));
```

Можете выполнить этот вариант приложения и убедиться, что он работает так же, как предыдущий. Правда, при компиляции этого варианта вы увидите замечание компилятора: «Variable 'R3' is declared but never used in 'TForm1.Button1Click'» — «Переменная 'R3' объявлена, но нигде не используется в процедуре 'TForm1.Button1Click'». Подобные сообщения можно игнорировать, если указанная переменная введена для какого-то будущего применения. Если же переменная действительно не нужна, лучше убрать из кода ее объявление. Действительно, зачем занимать под ненужную переменную лишнюю память. Хотя, конечно, это несерьезные затраты памяти. Другое дело, если бы вы объявили ненужный вам массив, хранящий несколько тысяч чисел. Вот такое ненужное объявление действительно надо было бы удалять.

В данном примере можно было бы обойтись и без переменных **R1**, **R2**. В них запоминаются числа, введенные пользователем. Но можно было бы их не запоминать, а сразу использовать для вычислений. Иначе говоря, можно было бы ограничиться оператором:

```
ShowMessage('Площадь прямоугольника = ' +
                  FloatToStr(StrToFloat(Edit1.Text) *
                             StrToFloat(Edit2.Text)));
```

Так что в данном примере объявление и использование переменных не является необходимым. Впрочем, введение переменных сделало код более прозрачным — т.е. более понятным. Если вы сравните первоначальный вариант кода с последним приведенным оператором, то, несомненно, первый вариант понятнее, особенно с учетом введенных в нем комментариев. А это значит, что вернувшись к этому проекту когда-то в будущем, чтобы его усовершенствовать, вы легко вспомните, как он работает, и без труда сможете его модернизировать. Это очень важно! Дело в том, что любая настоящая программа нуждается в *сопровождении*. После какого-то периода ее эксплуатации возникает необходимость добавить в нее новые возможности, изменить форму представления данных, усовершенствовать интерфейс пользователя. Значит, вам или кому-то из ваших коллег придется модернизировать программу. И в этом случае надо, чтобы код программы был понятен. В плохо написанной программе через некоторое время после ее создания часто настолько

трудно разобраться даже ее автору, что проще сделать ее заново, чем что-то в ней менять.

Хороший стиль программирования

Стремитесь к тому, чтобы код вашего приложения был понятным и легко модифицируемым. Не скупитесь на подробные комментарии. Хорошо документированный код — признак профессионализма его автора.

Помимо понятности кода, переменные, конечно, имеют и более важное назначение — запоминание данных и результатов вычислений для их последующего использования. Например, если вы хотите дополнить ваше тестовое приложение еще вычислением периметра прямоугольника, то при наличии переменных **R1** и **R2** достаточно выполнить оператор:

```
ShowMessage('Площадь прямоугольника = ' +
            FloatToStr(R3) + ', периметр прямоугольника = ' +
            FloatToStr(2.*(R1 + R2)));
```

При отсутствии переменных **R1** и **R2** в этом операторе пришлось бы по два раза вызывать функции **StrToFloat** для преобразования текстов окон редактирования в числа. А это было бы ничем не оправданным снижением эффективности кода, так как каждый вызов функции требует затрат времени. Конечно, в данном простеньком примере эти затраты заметить было бы невозможно. Но все-таки надо сразу приучаться заботиться об эффективности кода.

Кстати, задумавшись об эффективности вычислений, обратите внимание на то, что в приведенном выше примере константа задана числом с точкой: "2.". Можно было бы точку не записывать. Но тогда константа рассматривалась бы как целая, и затратилось бы время на ее приведение (см. разд. 2.4.2) к действительному типу, так как правый операнд операции умножения — действительное число.

2.4.4 Области видимости и время жизни

Переменные, а также такие рассмотренные далее элементы программы, как именованные константы, функции и процедуры, могут использоваться только в их *области видимости*, которая зависит от того места, в котором определен элемент. При определении области видимости используется понятие *блок*. Блок состоит из разделов объявлений и следующего за ними составного оператора:

```
объявления
begin
  операторы
end;
```

Область видимости переменной, константы, функции и других элементов, объявленных в головном файле программы, в функции или процедуре, ограничивается тем блоком, в котором элемент объявлен. Такие элементы называются *локальными*, в отличие от *глобальных* элементов, объявленных в разделе **interface** или **implementation** модуля. Элементы, объявленные в разделе **implementation**, видны от точки объявления до конца этого раздела, включая все расположенные в нем функции и процедуры. Элементы, объявленные в разделе **interface**, видимы не только в данном модуле, но и во всех модулях, которые подключили данный модуль своим предложением **uses**.

Если во вложенном блоке какой-то идентификатор внешнего блока переопределен (т.е. объявлен заново), то в этом вложенном блоке виден только переопределенный идентификатор. Например, если во внешнем блоке объявлена переменная **In**, а затем во вложенном блоке переменная **In** объявлена повторно, то эта новая локальная переменная не имеет ничего общего с внешней переменной. Во вложенном блоке, где переменная объявлена повторно, видна только эта новая переменная. А во внешнем блоке видна только внешняя переменная.

Использование в другом модуле предложения **uses**, включающего модуль, в интерфейсной части которого объявлен некоторый элемент, включает тем самым новую область его видимости уже в другом модуле. Когда в предложении **uses** перечисляется несколько модулей, то области видимости определяются так же, как в случае вложенных блоков. При этом первый из перечисленных модулей аналогичен самому внешнему блоку, а последний — самому внутреннему.

Переменные, объявленные в блоке, не только видны именно в нем, но и существуют только в нем. Они создаются (для них отводится область памяти) только при передаче управления в данный блок, и они уничтожаются, когда управление покидает этот блок. Так что их *время жизни* определяется временем между передачей управления в блок и выходом из блока. В таких локальных переменных нельзя сохранять какую-либо информацию в промежутках между передачами управления в блок.

Вернемся к тестовому примеру, рассмотренному в разд. 2.4.3. В нем переменные **R1**, **R2** и **R3** объявлены в процедуре обработчика щелчка на кнопке **Button1-Click**. Следовательно, память для них отводится только в тот момент, когда начинает выполняться эта процедура, т.е. когда пользователь щелкнул на кнопке. Пока выполняются операторы процедуры, переменные существуют. Но как только выполнился последний оператор, память, отведенная под эти переменные, освобождается, т.е. переменные уничтожаются.

Можно было бы вынести объявление переменных из процедуры и сделать переменные глобальными. Попробуйте, например, вынести соответствующее предложение **var** из процедуры и поместить его перед ее заголовком, т.е. после ключевого слова **implementation** (см. разд. 2.3.2). Переменные станут глобальными, но в работе приложения ничего не изменится. Кроме того, что теперь переменные будут существовать и после окончания работы процедуры, т.е. в промежутках между щелчками пользователя на кнопке. А зачем они нужны в это время? Только лишние непроизводительные затраты памяти.

Если вы перенесете предложение **var** в раздел интерфейса **interface**, то переменные тоже будут глобальными. Но в этом случае они будут доступны и внешним модулям (если такие есть — пока мы не рассматривали приложения с несколькими модулями), в которых в предложениях **uses** имеется ссылка на данный модуль.

Приложения с несколькими модулями будут рассмотрены в других главах. А пока посмотрим, зачем могут потребоваться глобальные переменные в рамках одного модуля. Пусть, например, мы хотим считать, сколько раз пользователь щелкнул на кнопке, работая с нашим приложением. С помощью локальных переменных этого сделать нельзя. Так что для решения этой задачи введите в раздел **implementation** вне процедуры обработки щелчка глобальную переменную. Например:

```
var N: word = 0;
```

Оператор создает переменную **N** и задает ей начальное значение 0. Поскольку число щелчков будет целым и неотрицательным, тип переменной задается равным

word (если считаете, что найдется пользователь, который щелкнет на кнопке более 65 тысяч раз, можете задать тип **Cardinal** — см. табл. 2.1 в разд. 2.4.1).

После введения этой глобальной переменной для подсчета и отображения числа щелчков достаточно добавить в созданный ранее обработчик щелчка на кнопке оператор, подсчитывающие число обращений к нему, и изменить оператор отображения результатов:

```
N := N + 1;
ShowMessage('Площадь прямоугольника = ' +
            FloatToStr(R3) + ', периметр прямоугольника = ' +
            FloatToStr(2.*(R1 + R2)) +
            ', число обращений = ' + IntToStr(N));
```

2.4.5 Отладка приложений

Мастерство программиста — разработчика приложения, определяется вовсе не его умением писать безошибочные программы (написать сложную программу без ошибок не может никто). Мастерство определяется умением разработчика быстро, эффективно и надежно отлаживать и тестировать свое приложение. Поскольку в разд. 2.4.3 вы создали свое первое приложение, содержащее переменные и операции с ними, мы можем теперь рассмотреть методику отладки приложений в ИСР Delphi.

Откройте проект, разработанный в разд. 2.4.3 и 2.4.4. Поскольку было создано несколько вариантов приложения, уточню — я подразумеваю следующий вариант:

```
var N: word = 0;

procedure TForm1.Button1Click(Sender: TObject);
var R1, R2, R3: real;
begin
 R1 := StrToFloat(Edit1.Text); // Первое число
 R2 := StrToFloat(Edit2.Text); // Второе число
 R3 := R1 * R2;                // Произведение
N := N + 1;
ShowMessage('Площадь прямоугольника = ' +
            FloatToStr(R3) + ', периметр прямоугольника = ' +
            FloatToStr(2.*(R1 + R2)) +
            ', число обращений = ' + IntToStr(N));
end;
```

Я уверен, что ваше приложение работает верно. Оно слишком простое, чтобы можно было что-то в нем напутать. Разве что перепутать знаки операций и вместо умножения написать, например, сложение. Уверяю вас, что в сложных приложениях у вас могут случаться и такие глупые ошибки (впрочем, умных ошибок вообще не бывает). Тогда приложение будет нормально компилироваться, выполняться, но будет выдавать неверные результаты. В подобных случаях бессмысленно сидеть, глядя на заведомо неверный результат на экране монитора, и думать, почему же он такой странный. Это достаточно непродуктивные размышления. Не ломайте голову, пытаясь решить подобную задачу. Голова вам еще пригодится, а для решения возникшей загадки у вас еще нет необходимой информации. Чтобы получить эту информацию, надо заняться отладкой, которая позволит вам заглянуть внутрь программы и увидеть, как выполняются операторы.

Для того чтобы отследить выполнение программы, вам надо остановить выполнение на каком-то операторе. Для этого есть несколько способов. Один из них — команда Run | Run To Cursor (горячая клавиша F4). Она обеспечивает выполнение приложения вплоть то того момента, когда должен выполняться оператор, на котором находится курсор в окне Редактора Кода. Другая возможность — щелкнуть в окне Редактора Кода на полосе слева от того оператора, перед выполнением которого вы хотите остановиться. В полосе появится красная точка, а сама строка окрасится красным цветом. Это значит, что вы ввели на этом операторе так называемую *точку прерывания*.

Реализуйте тот или иной вариант. Можете, например, поместите курсор в строку оператора, присваивающего значение переменной **R2**, и нажать F4. Или ввести точку прерывания на этой строке и нажать F9, т.е. обычным образом инициировать выполнение приложения. Впрочем, сразу надо оговориться, что применение точек прерывания предпочтительнее. Вы можете установить сразу несколько точек прерывания в разных местах программы. Наше приложение слишком простое, чтобы делать это. Но в сложной разветвленной программе не всегда ясно, по каким ветвям алгоритма проходит ее выполнение. Тогда легче это понять, установив одновременно точки прерывания в разных ветвях. Кроме того, в точки прерывания можно, как будет показано далее, вводить условия останова, что очень полезно в циклических программах, которые будут рассмотрены в разд. 2.8.7.

После того как приложение начало выполняться, щелкните в нем на кнопке Выполнить. Выполнение прервется, и вы увидите окно Редактора Кода с выделенной строкой, на которой остановилось выполнение (см. рис. 2.2). Подведя курсор, например, к переменной **R1**, вы увидите всплывающее окно, в котором будет написано текущее значение этой переменной. Это работает один из инструментов ИСР Delphi — Мастер оценки выражений (ToolTip Expression Evaluation). Вы можете выделить в тексте какое-то выражение, например, «Edit2.Text», или «R1 + R2», и также увидите его значение. Только учтите, что в данный момент значение переменной **R1** уже присвоено, а переменным **R2** и **R3** еще нет, так как соответствующие операторы не выполнялись. Так что значения этих переменных могут быть любыми.

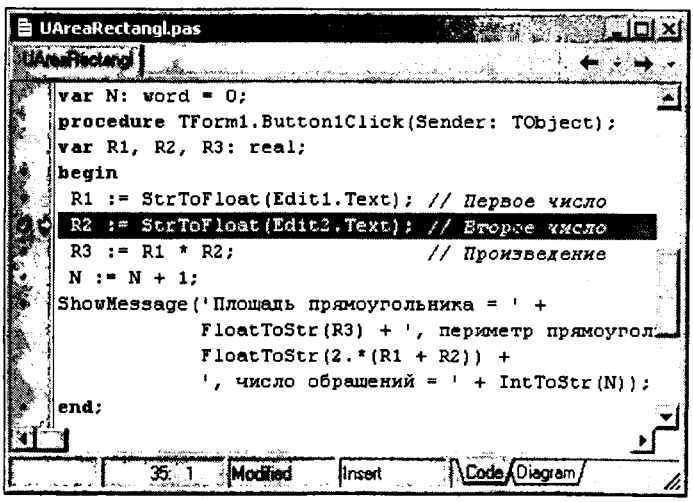


Рис. 2.2
Окно Редактора Кода с точкой прерывания

Мастер оценки выражений, конечно, хороший инструмент, но он дает значения только отдельных переменных, а в сложных приложениях вам надо бы иметь перед глазами значения сразу нескольких переменных, чтобы из их сравнения понять причины неправильной работы. Такую возможность предоставляет вам окно наблюдения Watches. Сделать его видимым можно командой View | Debug Windows | Watches. Соответствующую команду можно также выбрать из контекстного меню, всплывающего при щелчке правой кнопкой в окне Редактора Кода. Но всего этого даже не надо делать. Достаточно подвести курсор в коде к интересующей вас переменной и нажать Ctrl-F5. При этом окно наблюдения автоматически откроется и в нем появится имя переменной и ее значение. Затем вы можете подвести курсор к другой переменной, опять нажать Ctrl-F5 и в окне наблюдения появится новая строка. Более того, вы можете выделить курсором какое-то выражение, нажать Ctrl-F5 и в окне наблюдения увидеть значение этого выражения (рис. 2.3). Если окно Watches уже открыто, то можно поступить еще проще: выделить в окне Редактора Кода интересующее вас выражение и перетащить его мышью в окно Watches.

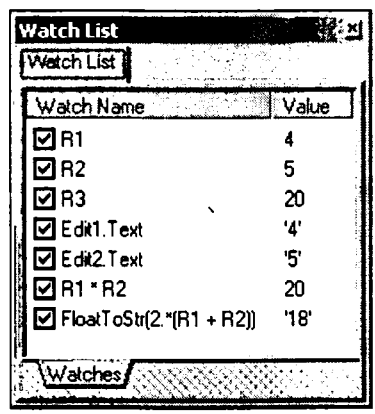


Рис. 2.3
Окно наблюдения

Индикаторы около наблюдаемых величин в окне Watches (они введены только в Delphi 7) позволяют отключать вывод в окно наблюдения соответствующего выражения во время выполнения приложения. Это повышает производительность выполнения. А после того, как приложение остановлено, и вам надо все-таки посмотреть данное выражение в окне наблюдения, можете включить индикатор и увидеть значение наблюдаемой величины.

Иногда вы можете не увидеть в окне значения каких-то заказанных вами переменных, а получить сообщение: «Variable ... inaccessible here due to optimization». Это следствие работы оптимизирующего компилятора, который может по своему усмотрению распоряжаться некоторыми переменными программы и хранить их не в памяти, а в быстрых регистрах системы. Таким образом обеспечивается более быстрое выполнение приложения. Спорить с компилятором не стоит, так как он знает, как создавать эффективные программы. В подобных случаях надо просто на время отладки отключить оптимизацию. В разд. 1.1 уже давался совет сделать это. Для отключения оптимизации надо выключить флажок Optimization страницы Compiler окна Project Options, которое вы можете вызвать командой Project | Options.

Другая ошибка наблюдения может появиться в случае, если вы хотите наблюдать выражение, содержащее какие-то функции. Перенесите, например, в окно наблюдения выражение «FloatToStr(2.*(R1 + R2))». Вы получите сообщение: «Inaccessible value» — «Недоступная величина». Дело в том, что это выражение содержит функцию **FloatToStr**, а по умолчанию вызов функций из окна наблюдения запрещен. Но это запрещение можно отменить. Перейдите в окно наблюдения, выделите в нем нужную строку и нажмите клавиши Ctrl-F5. Вы попадете в окно задания списка наблюдения Watch Properties (рис. 2.4). В то же окно выводит двойной щелчок на строке окна наблюдения или команда Run | Add Watch.

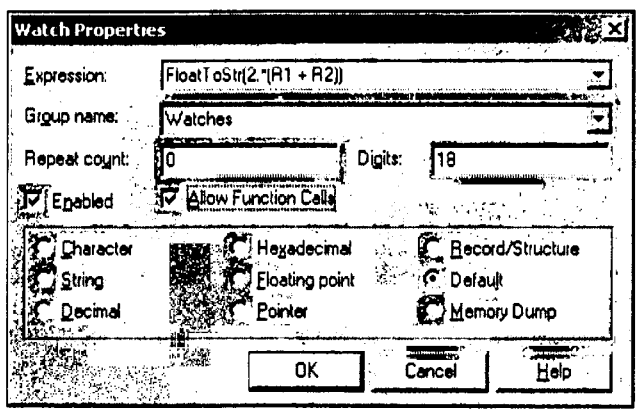


Рис. 2.4
Окно задания списка наблюдения Watch Properties

В окне Watch Properties вы можете написать в окошке Expression любое выражение, содержащее переменные, константы, функции. Индикатор Enabled позволяет отключить вывод в окно наблюдения соответствующего выражения во время выполнения приложения, т.е. в Delphi 7 дублирует индикаторы, которые вы видите на рис. 2.3. В предшествующих версиях Delphi это единственный механизм отключения и включения вывода.

Индикатор Allow Function Calls разрешает включать в окно наблюдения выражения, содержащие вызовы функций. Установите этот индикатор, и сообщение об ошибке в окне наблюдения исчезнет. Можно поступить иначе: выполнить команду Tools | Debugger Options и в появившемся многостраничном диалоговом окне на странице General установить опцию Allow function calls in new watches. Тогда всегда все новые выражения, добавляемые в окно наблюдения, будут допускать вызовы функций.

Выпадающий список вверху позволяет выбрать выражение из тех, которые использовались ранее, и при необходимости отредактировать его. Это удобно, если надо выводить в окно наблюдений похожие выражения. Например, если вам надо вывести значение **Label1.Caption**, **Label2.Caption** и **Label3.Caption**, то достаточно один раз написать это выражение, а в дальнейшем брать его из выпадающего списка и только менять в нем цифру.

После того как вы остановились и собрали с помощью окна наблюдения всю необходимую информацию, вы можете выполнить приложение по шагам, отслеживая изменение этой информации при выполнении каждого оператора. Это можно делать клавишей F8 (команда Step Over) или клавишей F7 (команда Trace Into).

Клавиша F7 обеспечивает пошаговое выполнение программы с заходом в вызываемые функции и процедуры (не библиотечные, а разработанные вами), а клавиша F8 вход в функции и процедуры не производит. В нашем примере вы не заметите различия между этими двумя вариантами, так как создание собственных функций мы будем рассматривать позднее, в разд. 2.7.

Пройдя по шагам какие-то операторы, вы можете в любой момент продолжить нормальное выполнение приложения, нажав горячую клавишу F9. Если же вы уже нашли ошибку и хотите исправить ее, вы можете завершить отладку и выполнение приложения, нажав горячие клавиши Ctrl-F2. Эти же клавиши Ctrl-F2 можно использовать, если при выполнении приложения оно по непонятным для вас причинам перестало вас слушаться. Обычно это связано с какими-то ошибками кода, вызвавшими «зацикливание» — бесконечное повторение какого-то фрагмента кода. В подобных случаях надо вернуться в ИСР Delphi и нажать клавиши Ctrl-F2. Выполнение приложения прекратится, и вы сможете наметить план дальнейшей отладки.

Совет

Во всех случаях, когда вы потеряли управление выполняющимся приложением, или когда приложение выдало непонятное вам сообщение об ошибке, перейдите в ИСР Delphi, прервите выполнение приложения клавишами Ctrl-F2, и продумайте план дальнейшей отладки.

Теперь рассмотрим подробнее работу с точками отладки. Вы уже знаете, как их вводить. Правда, учтите, что точки прерывания можно устанавливать только на выполняемых операторах. Если вы, например, попробуете установить точку прерывания на строке, содержащей объявление переменной, то в момент запуска приложения в красной точке, выделяющей строку прерывания, появится крестик. Тем самым Delphi предупреждает, что прерывания не будет, поскольку оператор невыполняемый. Аналогичный крестик может появиться и в том случае, если компилятор в процессе оптимизации кода убрал какой-то написанный вами оператор (вспомните, как можно отключить оптимизацию). Для того чтобы убрать ранее введенную точку прерывания, достаточно щелкнуть мышью на красной точке левее кода соответствующей строки.

Delphi дает возможность задавать условия прерывания. Для этого надо щелкнуть правой кнопкой мыши на точке в строке, в которой вы ввели прерывание, и выбрать во всплывшем меню раздел Breakpoint properties. Перед вами откроется окно свойств точки прерывания, представленное на рис. 2.5.

Два верхних окошка редактирования — Filename (имя файла) и Line Number (номер его строки) в данном режиме недоступны. Они автоматически заполнились в момент, когда вы установили в своем коде точку прерывания.

Окошко Condition (условие) позволяет вам ввести некоторое условное выражение. Прерывание будет происходить только в случае, если это условие выполняется. Окошко Pass Count позволяет указать, при котором по счету выполнении записанного условия произойдет останов. Значение 0 означает, что прерывание будет происходить при любом проходе.

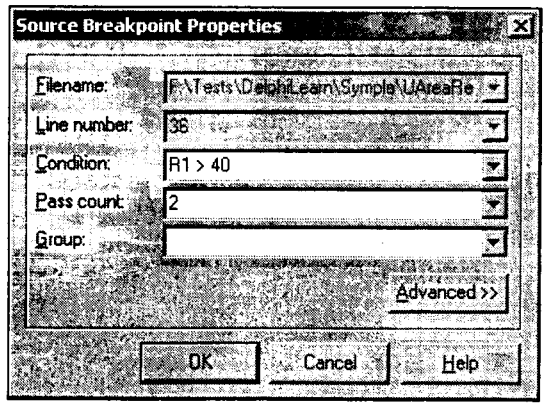


Рис. 2.5
Окно задания свойств прерывания в указанной строке файла

Конечно, все это очень полезно в сложных задачах, до которых мы еще не добрались. В частности, задавать значение Pass Count имеет смысл при отладке циклических кодов, которые будут рассматриваться в разд. 2.8.7. Но вы можете проверить все это и на нашем примере. Задайте, например, условие «R1 > 40» и число проходов 2. Вы сможете убедиться, что во время выполнения прерывание наступит, только если вы второй раз щелкните на кнопке при числе в первом окне, превышающем 40.

Команда View | Debug Windows | Breakpoints позволяет открыть окно списка точек прерывания, которое показано на рис. 2.6. В этом окне можно выделить уже введенную точку прерывания, щелкнуть на ней правой кнопкой мыши и из контекстного меню выбрать команду Properties, которая покажет окно задания свойств прерывания (рис. 2.5) и позволит отредактировать информацию. Отличие от рис. 2.5 будет в том, что станут доступны окошки задания файла и строки, а также в окне появится индикатор Keep existing Breakpoint. Если включить его, то прежняя точка прерывания сохранится, а новая информация будет относиться к новой точке.

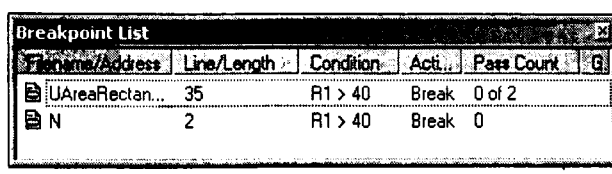


Рис. 2.6
Окно списка точек прерывания в Delphi 7

Если в окне списка точек прерывания щелкнуть правой кнопкой мыши не на одной из уже введенных точек прерывания, то во всплывшем меню появится раздел Add — добавление новых точек с подразделами: Source Breakpoint — уже рассмотренная точка прерывания в указанной строке файла приложения, Address Breakpoint — прерывание при переходе управления по заданному адресу, и Data Breakpoint — прерывание при изменении данных. Последний вид прерываний очень полезен, если какая-то переменная принимает неправильное значение, но не ясно, при выполнении какой команды кода это происходит. Оба последних вида прерывания в вашем меню будут недоступны, поскольку, пока приложение не загружено

в память, Delphi не знает адресов команд и переменных. Чтобы задать эти виды прерывания, надо запустить приложение на выполнение и после этого, ничего в нем не делая, вернуться в среду Delphi и открыть окно точек прерывания. После этого выберите из контекстного меню команду Add | Data Breakpoint.

При выполнении этой команды вы попадете в диалоговое окно свойств прерывания при изменении данных (рис. 2.7). Оно похоже на рассмотренное ранее (рис. 2.5), но содержит окошко адреса (Address), в котором вы можете указать идентификатор переменной, и окошко длины переменной (Length), которое обычно заполняется автоматически. Правда, к сожалению, в окошке Address вы можете указать только имя глобальной переменной, так как только такие переменные имеют постоянный адрес. Если вы попытаетесь указать локальную переменную, вам будет выдано сообщение об ошибке: «Invalid address» — «Ошибочный адрес». В нашем примере только одна глобальная переменная — **N**. Укажите ее и задайте, если хотите, условие прерывания и число проходов. Нажмите кнопку OK. После этого вернитесь в выполняемое приложение и начните с ним работать. Когда условия прерывания будут выполнены, выполнение остановится, и в окне Редактора Кода будет выделена строка, расположенная после того оператора, который изменил значение указанной вами переменной. В нашем примере останов будет после оператора, задающего значение переменной **N**.

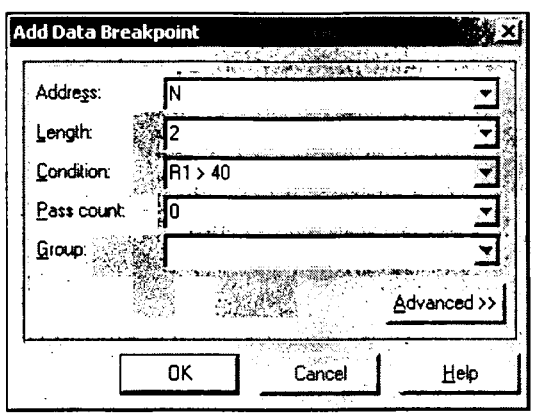


Рис. 2.7
Окно задания свойств прерывания, возникающего при изменении данных

Учтите, что доступность точки прерывания по изменению переменной сохраняется только в течение данного сеанса выполнения приложения. По завершении выполнения точка становится недоступной и при следующем запуске ее надо повторно активизировать, устанавливая с помощью контекстного меню ее индикатор Enabled.

2.4.6 Математические функции

Помимо арифметических операций в математических выражениях могут использоваться вызовы функций, как библиотечных, так и созданных самим разработчиком программы. Вопросы создания собственных функций мы обсудим в разд. 2.7. А библиотечные функции имеет смысл рассмотреть сейчас. Некоторые из них, связанные с округлением действительных чисел, уже были рассмотрены в разд. 2.4.2. В табл. 2.3 приведен список некоторых других часто используемых

функций (полный список вы можете найти во встроенной английской справке Delphi, или в русской справке [3], если она у вас поставлена).

Таблица 2.3 Некоторые математические функции

Функция	Описание	Аргумент
Abs(X)	абсолютное значение	целое или действительное выражение
ArcCos(X)	арккосинус	константное выражение **Extended**
ArcSin(X)	арксинус	константное выражение **Extended**
ArcTan(X)	арктангенс	константное выражение **Extended**
Cos(X)	косинус	выражение **Extended**
Exp(X)	экспонента	действительное выражение
Frac(X)	дробная часть числа	выражение **Extended**
Int(X)	целая часть аргумента	действительное выражение
Ln(X)	натуральный логарифм от X	действительное выражение
Log2(X)	логарифм по основанию 2 от X	выражение **Extended**
Log10(X)	десятичный логарифм от X	выражение **Extended**
LogN(Base, X)	логарифм по основанию Base от X	выражения **Extended**
Max(A,B)	максимум двух чисел	выражения **Integer**, **Int64**, **Single**, **Double**, **Extended**
Min(A,B)	минимум двух чисел	выражения **Integer**, **Int64**, **Single**, **Double**, **Extended**
Pi	число Пи: 3.1415926535897932385	–
Power(X, E)	возведение X в произвольную степень E: X^E	выражения **Extended**
Sin(X)	синус	выражение **Extended**
Sqr(X)	квадрат аргумента: X*X	выражение **Extended**
Sqrt(X)	квадратный корень	выражение **Extended**
Tan(X)	тангенс	выражение **Extended**

Функции **ArcCos**, **ArcSin**, **Log10**, **Max**, **Min**, **Power**, **Tan** объявлены в модуле Math. Этот модуль автоматически не подключается к проекту, так что его надо подключить вручную оператором **uses**.

Обратите внимание на то, что в языке Pascal, как и во многих других, нет операции возведения в степень. Для возведения числа в произвольную степень используется функция **Power**. Если возводимое в степень число **X** отрицательно, то показатель степени **E** должен быть целым. Функцию **Power** можно, в частности, использовать и для извлечения квадратного корня, и для возведения в квадрат. Однако для этих часто выполняемых операций имеются функции соответственно **Sqrt** и **Sqr**, которые выполняются быстрее.

Функция **Int** возвращает целую часть числа, т.е. производит округление в сторону нуля. В этом отношении она подобна функции **Floor**, рассмотренной

в разд. 2.4.2. Но функция **Int** возвращает значение типа **Extended**, т.е. действительное число, хотя и без дробной части. А функция **Floor** возвращает целое значение типа **Integer**.

В тригонометрических функциях аргумент X — угол в радианах. Обратные тригонометрические функции возвращают главное значение угла в радианах. В функциях **ArcSin** и **ArcCos** аргумент должен лежать в пределах от −1 до 1. Функции **ArcSin** и **ArcTan** возвращают результат в пределах $[-\pi/2 \ldots \pi/2]$, **ArcCos** — в пределах $[0 \ldots \pi]$.

2.4.7 Пример — калькулятор

Чтобы протестировать применение библиотечных функций, давайте создадим программу калькулятора. Пока реализуем в ней только ввод данных, вычисление функций и работу с памятью. Реализацией арифметических операций займемся позднее. Возможный вид приложения показан на рис. 2.8.

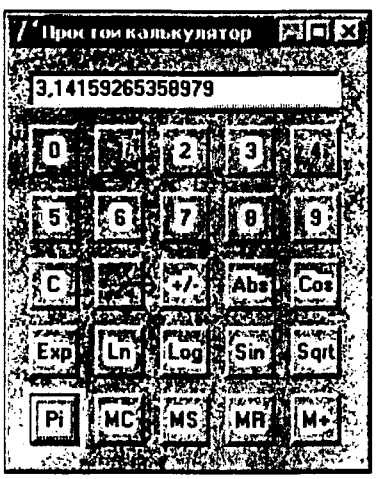


Рис. 2.8
Приложение простого калькулятора

Начните новое приложение и поместите на форме одно окно редактирования **Edit** и 25 кнопок **Button**. Кстати, обилие кнопок в этом приложении дает хороший повод рассмотреть работу в ИСР Delphi с группами компонентов. Переносить одинаковые компоненты из библиотеки на форму можно не по одному, а сразу все. Для этого надо в момент выделения компонента в палитре библиотеки держать нажатой клавишу Shift. Затем можете отпустить эту клавишу и щелкать мышью в тех местах формы, где хотите располагать компоненты. Каждый щелчок перенесет на форму новый экземпляр компонента. Когда все перенесено, нажмите кнопку со стрелкой в левой части палитры, и перенос компонентов завершится. При размещении компонентов, в нашем случае — кнопок, не тратьте время на поиск точного места для каждого компонента. В ИСР имеется инструментарий, облегчающий выравнивание компонентов на форме.

Задайте на кнопках соответствующие надписи (свойство **Caption**), поясняющие их назначение. Чтобы не запутаться в многочисленных кнопках нашего приложения, вспомните совет, который давался в разд. 1.2.3 — задавать компонентам ос-

мысленные имена (свойство **Name**). Давайте зададим кнопкам, на которых изображены цифры, имена **Button0**, **Button1**, ..., **Button9** в соответствии с этими цифрами. Остальным кнопкам задайте имена, содержащие идентификаторы соответствующих функций. Например, **BAbs**, **BExp** и т.п. Но имена компонентов не должны совпадать с именами функций! Иначе компилятор не сможет понять, обращаетесь ли вы к компоненту или функции.

Предупреждение

Нельзя, чтобы имена компонентов или иных объектов совпадали с именами переменных, функций, процедур и т.п. Точно так же нельзя, чтобы имена переменных совпадали с именами библиотечных функций. Любые совпадения идентификаторов недопустимы. При таких совпадениях компилятор не сможет понять, что скрывается за тем или иным идентификатором, и будет выдавать сообщения о различных синтаксических ошибках.

На форме можно выделить группу компонентов, к которым применяется та или иная операция. Выделение группы возможно двумя способами. Если компоненты расположены непосредственно на форме и рядом друг с другом, то для выделения группы достаточно обвести курсором рамку вокруг них, и все они окажутся выделенными. Если компоненты группы расположены не непосредственно на форме, а, например, на панели, то выделить их рамкой невозможно. Так же невозможно выделить рамкой группу компонентов, расположенных в разных местах формы. В этих случаях выделение производится иначе. Выделяйте нужные компоненты курсором, нажав и не отпуская при этом клавишу Shift. Все компоненты, выделенные таким образом, войдут в группу.

С выделенной группой компонентов можно производить следующие операции:

- Перемещать их одновременно, потянув курсором за один из выделенных компонентов. Это очень удобно, когда надо переместить группу компонентов на форме, не изменяя их взаимного расположения.
- Задавать в Инспекторе Объектов общие для всей группы свойства. При выделении группы на странице свойств в Инспекторе Объектов будут видны только их общие свойства. А индивидуальные свойства, такие, например, как имя компонента **Name** или надпись **Caption**, исчезнут. Вы можете задать особенности шрифта, оформления, цвет и т.п., которые будут присущи всем компонентам выделенной группы.
- Задать общий для всех компонентов группы обработчик какого-то события.
- Скопировать всю группу в буфер обмена Clipboard командой Edit | Copy или горячими клавишами Ctrl-C. После этого вы можете, например, открыть какую-то другую форму и перенести на нее группу из Clipboard командой Edit | Paste или горячими клавишами Ctrl-V. Это простой способ переносить фрагменты с одной формы на другую или с одной панели формы на другую панель.
- Выравнивать компоненты группы по размеру и взаимному расположению.

В нашем примере вы можете объединить в группу все кнопки и задать им, например, одинаковый размер (свойства **Height** — высота и **Width** — ширина). При надписях, показанных на рис. 2.8, и при полужирном шрифте достаточно размера

30 x 30. Впрочем, задание и выравнивание размеров компонентов группы еще проще осуществить командой Edit | Size — выравнивание размеров. Ту же команду можно выбрать не из меню Edit, а из раздела Position контекстного меню, которое всплывает, если вы щелкнете правой кнопкой мыши на одном из компонентов группы (именно на компоненте, так как если вы щелкнете в стороне, то выделение группы снимется).

При выполнении этой команды вам открывается окно, представленное на рис. 2.9. Левая часть окна — Width устанавливает ширину компонентов. Вы можете выбрать варианты: No change — не изменять, Shrink to smallest — уменьшить до размера минимального из компонентов группы, Grow to largest — увеличить до размера максимального из компонентов группы, Width — задать в окне рядом с этой радиокнопкой ширину компонента в пикселах. Аналогичные варианты предлагаются в правой части окна для Height — высоты компонентов.

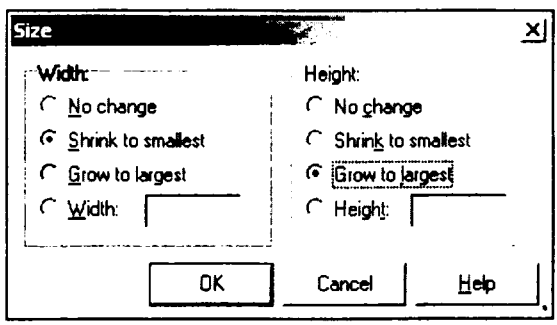


Рис. 2.9
Окно выравнивания размеров группы компонентов

В нашем примере выровнять размеры кнопок с помощью этого окна можно двумя путями. Можно сначала задать размеры одной кнопки и убедиться, что такие размеры вас устраивают. А затем выделить все кнопки как группу и в окне рис. 2.9 установить ширину радиокнпокой Shrink to smallest (так как ширина 30 меньше стандартной ширины остальных кнопок), и установить высоту радиокнпокой Grow to largest (так как высота 30 больше стандартной высоты кнопок). А можно поступить иначе: сразу выделить все кнопки и в окне рис. 2.9 задать в окошках Width и Height числа 30. Все кнопки группы станут одинаковыми квадратами.

Когда вы размещаете компоненты на форме, трудно бывает добиться аккуратного вида окна и симметричного расположения компонентов. Решение этой задачи облегчает команда Edit | Align — выравнивание размещения. Ту же команду можно выбрать не из меню Edit, а из раздела Position контекстного меню, которое всплывает, если вы щелкнете правой кнопкой мыши на одном из компонентов группы.

При выполнении этой команды открывается окно, представленное на рис. 2.10. Левая часть окна — Horizontal устанавливает выравнивание компонентов по горизонтали. Вы можете выбрать варианты: No change — не изменять, Left sides — выровнять компоненты по их левым сторонам (т.е. левые стороны компонентов будут расположены друг под другом), Center — выровнять компоненты по их центрам, Right sides — выровнять компоненты по их правым сторонам, Space equally — разместить с равными интервалами между компонентами, Center in window — расположить в центре окна.

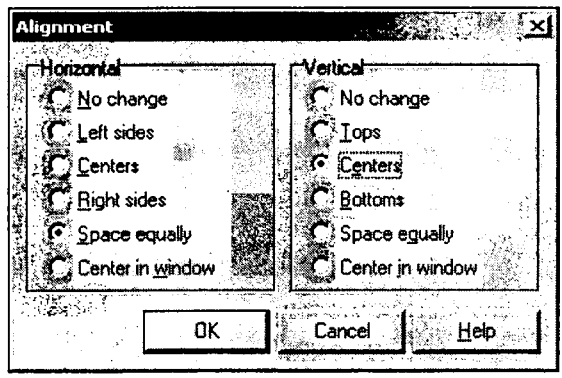


Рис. 2.10
Окно выравнивания размещения группы компонентов

Правая часть окна — Vertical устанавливает выравнивание компонентов по вертикали. Тут имеются аналогичные варианты. Вы можете выбрать: No change — не изменять, Tops — выровнять компоненты по их верхним сторонам, Center — выровнять компоненты по их центрам, Bottoms — выровнять компоненты по их нижним сторонам, Space equally — разместить с равными интервалами между компонентами, Center in window — расположить в центре окна.

Необходимо сделать некоторые уточнения по различным режимам выравнивания.

При выравнивании по границам компонентов (Left sides, Right sides, Tops, Bottoms, Center) на месте остается самый левый (при выравнивании по горизонтали) или самый нижний (при выравнивании по вертикали) компонент, а местоположение остальных подгоняется под него.

В режиме Space equally крайние компоненты (левый и правый при выравнивании по горизонтали и верхний и нижний при выравнивании по вертикали) не перемещаются. Получающиеся интервалы определяются положением этих крайних компонентов и числом и размерами промежуточных. Так что, пользуясь этим режимом, разместите сначала крайние компоненты так, как вам нужно, а уж затем используйте выравнивание. Выравниваются расстояния между верхними (при выравнивании по вертикали) или левыми (при выравнивании по горизонтали) границами компонентов. Поэтому, если выравниваемые компоненты имеют разные размеры, то расстояния между их примыкающими друг к другу краями будут не одинаковы. Компоненты могут даже наложиться друг на друга. Так что выравнивание по равным расстояниям имеет смысл только для компонентов равных размеров.

Режим Center in window не означает, что каждый компонент расположится в горизонтальном или вертикальном направлении в центре окна. В центре расположится только центр выделенной группы, а относительные сдвиги компонентов сохранятся неизменными. Кроме того, учтите, что речь в данном случае идет не об окне формы, а о родительском окне компонентов группы. Если компоненты расположены, например, на панели, то подразумевается центр этой панели.

Думаю, что после некоторых экспериментов с окном рис. 2.10 вы сможете установить все кнопки стройными рядами с равными интервалами между рядами и кнопками в рядах. В заключение полезно выделить все компоненты формы (горячие клавиши Ctrl-A) и выбрать в окне рис. 2.10 варианты Center in window по горизонтали и вертикали. Вся группа компонентов переместится в центр окна формы.

Теперь можно программировать это приложение. Ниже приведен код модуля, с некоторыми несущественными сокращениями.

```
unit UCalc;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    BAbs: TButton;
    BExp: TButton;
    BLn: TButton;
    BLog: TButton;
    BSin: TButton;
    BCos: TButton;
    BSqrt: TButton;
    Button1: TButton;
    ...
    Button0: TButton;
    BPoint: TButton;
    BClear: TButton;
    BSign: TButton;
    BPi: TButton;
    BMC: TButton;
    BMS: TButton;
    BMR: TButton;
    BMPlus: TButton;
    procedure BClearClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure BSignClick(Sender: TObject);
    procedure BAbsClick(Sender: TObject);
    procedure BCosClick(Sender: TObject);
    procedure BExpClick(Sender: TObject);
    procedure BLnClick(Sender: TObject);
    procedure BLogClick(Sender: TObject);
    procedure BSinClick(Sender: TObject);
    procedure BSqrtClick(Sender: TObject);
    procedure BPiClick(Sender: TObject);
    procedure BMCClick(Sender: TObject);
    procedure BMSClick(Sender: TObject);
    procedure BMRClick(Sender: TObject);
    procedure BMPlusClick(Sender: TObject);
  private
{ Private declarations }
  public
{ Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.dfm}

uses Math;
var M: real = 0.;

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

procedure TForm1.Button1Click(Sender: TObject);
begin
 Edit1.Text := Edit1.Text + (Sender as TButton).Caption;
end;

procedure TForm1.BSignClick(Sender: TObject);
begin
 Edit1.Text := FloatToStr(-(StrToFloat(Edit1.Text)));
end;

procedure TForm1.BAbsClick(Sender: TObject);
begin
 Edit1.Text := FloatToStr(Abs(StrToFloat(Edit1.Text)));
end;

...

procedure TForm1.BPiClick(Sender: TObject);
begin
 Edit1.Text := FloatToStr(Pi);
end;

procedure TForm1.BMCClick(Sender: TObject);
begin
 M := 0.;
end;

procedure TForm1.BMSClick(Sender: TObject);
begin
 M := StrToFloat(Edit1.Text);
end;

procedure TForm1.BMPlusClick(Sender: TObject);
begin
 M := M + StrToFloat(Edit1.Text);
end;

procedure TForm1.BMRClick(Sender: TObject);
begin
 Edit1.Text := FloatToStr(M);
end;

end.
```

Рассмотрим приведенный код. Интерфейсную часть модуля Delphi формирует без вашего участия, так что рассматривать ее не будем. Я привел ее только для того, чтобы были ясны имена, присвоенные кнопкам. Ваш собственный текст начинается с оператора

```
uses Math;
```

подключающего модуль *Math*, в котором объявлены многие функции, используемые в приложении. Следующий оператор объявляет действительную переменную **M**, которая будет выполнять роль памяти калькулятора, и задает ее начальное значение 0. Процедура **BClearClick** является обработчиком щелчка на кнопке C — очистка. В ней методом **Clear** стирается текст в окне **Edit1**.

Следующая процедура **Button1Click** является общим обработчиком щелчка на любой кнопке с цифрами и кнопке с десятичной запятой. Суть его сводится к тому, что к тексту окна **Edit1** добавляется справа соответствующая цифра или запятая. Можно было бы, конечно, написать 11 обработчиков щелчков на 11-ти кнопках, каждый из которых добавлял бы цифру, соответствующую этой кнопке, или запятую. Но можно существенно сократить код, написав для всех этих кнопок один обработчик щелчка. Посмотрим, как это делается.

Сначала о технике составления одного обработчика для нескольких событий. В нашем случае это можно сделать двумя путями. Можно написать обработчик для одной кнопки, например, для **Button1**. А затем поочередно выделять на форме кнопки, в окне Инспектора Объектов на странице событий нажимать кнопку выпадающего списка около события **OnClick**, и выбирать из списка идентификатор уже написанного обработчика **Button1Click**. Но можно поступить проще: выделить все кнопки с цифрами и кнопку с запятой в группу, и в окне Инспектора Объектов на странице событий сделать двойной щелчок около события **OnClick**. Обработчик, который вы при этом напишете, будет срабатывать при щелчке на любой кнопке выделенной группы.

Теперь, зная, как создать общий обработчик, давайте подумаем, как должен выглядеть его код. Нам надо знать, во-первых, на какой все-таки кнопке щелкнул пользователь, и, во-вторых — какую цифру (или запятую) добавлять в текст окна **Edit1**. Вторая задача решается просто. Поскольку мы записали соответствующие цифры и запятую в свойство **Caption** кнопок, то в окно **Edit1** надо добавлять свойство **Caption** нажатой кнопки. Так что остается только первая задача: понять, на какой кнопке щелкнул пользователь.

В этом может помочь параметр **Sender**, который передается во все обработчики событий. Пока мы на него не обращали внимания, так как его имеет смысл использовать только в случае, подобном нашему: когда один обработчик используется для разных событий. Через параметр **Sender** в обработчик передается объект, в котором произошло событие. Но посмотрите на тип параметра **Sender**. Это объект класса **TObject**, который является базовым для всех объектов и компонентов Delphi. Классы мы будем рассматривать позднее в разд. 3.5. А о наследовании уже упоминали в разд. 1.1.2. Производные классы наследуют от базового класса все его свойства и методы. Но в классе **TObject** не объявлено никаких свойств, которыми мы могли бы воспользоваться. Впрочем, имеется несколько методов, позволяющих определить класс объекта. Метод **ClassName** возвращает строку с именем класса. Так что при щелчке на кнопке выражение **Sender.ClassName** вернет текст «TButton». Но нас это не устроит, так как нам надо определить не класс кнопки, а конкретную кнопку. Точно так же нам не годятся методы **ClassNameIs** и **In-**

heritsFrom (выражения **Sender.ClassNameIs('TButton')** и **Sender.Inherits-From(TWinControl)**) возвращающие истину (**true**), если объект соответственно является объектом указанного класса или наследует указанному классу. Подобные методы мы неоднократно будем использовать в дальнейшем для определения класса объекта. Но в данном примере они нам не подходят. Нам надо рассматривать объект класса **TObject** как объект класса кнопок **TButton**. Вот тогда мы сможем ссылаться на любые свойства и методы кнопок. Это делается с помощью операции **as**. Выражение **(Sender as TButton)**, которое вы можете видеть в обработчике **Button1Click**, рассматривает параметр **Sender** как кнопку класса **TButton**. Так что выражение **(Sender as TButton).Caption** — это символ, занесенный в свойство **Caption** той кнопки, на которой щелкнул пользователь. Этот символ добавляется справа к тексту окна **Edit1**.

Если рассуждения, приведшие нас к очень компактному оператору обработчика события, показались вам несколько туманными, попробую провести некоторую аналогию. Класс **TObject** можно рассматривать как очень общее понятие, например, «материальный объект». Конечно, из такого общего понятия мало что можно извлечь. Через ряд промежуточных понятий можно придти к понятию «средство передвижения», затем к понятию «легковой автомобиль», и, наконец, к конкретной марке автомобиля — «Мерседес» или «Ока». Примененная выше операция **as** делает именно это. Она как бы говорит: «я рассматриваю этот объект как автомобиль марки ...». А после этого уже можно обращаться к любым свойствам автомобиля данной марки.

Итак, пожалуй наиболее сложный для понимания, хотя и очень простой по тексту обработчик события мы рассмотрели. Остальные процедуры кода, вероятно, сложностей не вызывают. Процедура **BSignClick** является обработчиком щелчка на кнопке, изменяющей знак числа, отображаемого в окне **Edit1**. Текст окна переводится функцией **StrToFloat** в число, к этому числу применяется унарная операция минуса, и полученное в результате число отображается функцией **FloatToStr**. Из обработчиков щелчков на функциональных клавишах в качестве примеров приведено только два: **BAbsClick**, вычисляющий с помощью функции **Abs** модуль числа в окне **Edit1**, и **BPiClick**, заносящий в окно число π. Остальные обработчики не приведены, так как они отличаются только вызываемыми функциями.

Процедура **BMCClick** — обработчик щелчка на кнопке MC, очищающей память — переменную **M**. Процедура **BMSClick** — обработчик щелчка на кнопке MS, заносящей в память число, записанное в окне **Edit1**. Процедура **BMPlusClick** не заносит, а суммирует число из окна **Edit1** с числом, хранящимся в памяти, т.е. накапливает в памяти сумму чисел. А процедура **BMRClick** извлекает число из памяти и заносит его в окно **Edit1**.

Проверьте созданное приложение в работе, и убедитесь, что оно функционирует верно. В нем пока имеется ряд недостатков: не реализованы арифметические операции, пользователь может написать в окне **Edit1** ошибочный символ, и возникнет ошибка, он может поставить ошибочно несколько десятичных запятых в числе и т.п. Некоторые из этих усовершенствований мы скоро введем в наше приложение. А остальные вы попробуете сделать самостоятельно. Так что сохраните этот проект — он еще нам понадобится.

2.4.8 Ошибки выполнения арифметических операций и библиотечных функций

В этом разделе мы рассмотрим ошибки, которые могут возникать при выполнении арифметических операций и вызове библиотечных функций. Правда, мы еще не изучали некоторых аспектов языка программирования, которые требуются для обработки ошибок вычисления. Но все-таки, мне кажется, что некоторые приемы борьбы с ошибками надо рассмотреть сейчас.

Начнем с операций с целыми числами. Если результат выполнения какой-либо операции превышает максимальное значение для данного типа, то полученное значение будет неправильным. Например, если **i**, **i1**, **i2** — переменные типа **integer** и **i** = 2147483646, **i1** = 1, **i2** = 2, сумма **i** + **i1** будет равна 2147483647, т.е. вычислится правильно, а сумма **i** + **i2** будет равна −2147483648, т.е. совершенно неверному отрицательному числу (минимально возможному значению **integer**). Это произойдет в силу того, что максимально допустимое значение **integer** равно 2147483647. Подобное поведение целых чисел надо учитывать и отслеживать программно. Иначе могут возникнуть трудно отлавливаемые ошибки выполнения. При проверках допустимости тех или иных значений можно использовать объявленные в модуле *System* константы **MaxInt** и **MaxLongint** — соответственно максимальные значения типов **Integer** и **Longint** (см. допустимый диапазон значений в табл. 2.1 в разд. 2.4.1).

При выполнении операций с действительными числами могут возникать ошибки переполнения (значение превышает максимально допустимое для данного типа число), потери порядка (значение меньше минимально допустимого), деления на нуль. Способ обработки этих ошибок зависит от так называемой маски FPU (floating-point unit) — слова, управляющего в Windows обработкой ошибок при операциях с плавающей запятой. В Delphi в модуле Math имеется функция **SetExceptionMask**, которая позволяет управлять маской FPU. Не вдаваясь в детали, надо учесть следующее. Если выполнить в программе оператор

```
SetExceptionMask([]);
```

то при всех ошибках выполнения операций с плавающей запятой будут генерироваться так называемые исключения. А если выполнить оператор

```
SetExceptionMask([exInvalidOp, exDenormalized, exZeroDivide,
                  exOverflow, exUnderflow, exPrecision]);
```

то никаких исключений генерироваться не будет. В этих случаях ошибка выполнения операции приведет к тому, что в качестве результата будет выдано значение одной из предопределенных в Delphi констант: **Infinity** (положительная бесконечность), **NegInfinity** (отрицательная бесконечность), **NaN** (нецифровое значение). Если преобразовывать эти значения в строку обычной функцией **FloatToStr**, то будут выданы соответственно тексты "INF", "−INF", "NAN". Последующее использование полученных значений в арифметических операциях приведет к выдаче в качестве результата аналогичных значений.

Константы **Infinity**, **NegInfinity** и **NaN** не могут использоваться в операциях сравнения. Для проверки результата в модуле *Math* введены функции **IsInfinite** и **IsNan**, в которые в качестве аргумента передается проверяемое значение. Первая из этих функций возвращает **true**, если значение **AValue** равно **Infinity** или **NegInfinity**, а вторая — если значение **AValue** равно **NaN**. Если функцией **IsInfinite** обнаружено, что значение бесконечное, то знак бесконечности можно оп-

ределить функцией **Sign**. В качестве аргумента в нее передается проверяемое значение. Функция возвращает 1, если значение положительное, возвращает −1, если значение отрицательное, и возвращает 0 для нулевого значения.

Организовывать в программе многочисленные проверки результатов вычислений достаточно хлопотно. Поэтому проще и надежнее применять аппарат исключений. Исключение — это некоторый особый объект, генерируемый в случае возникновения ошибки. Если не принять мер для его перехвата, то в результате приложение выдаст грозное сообщение на английском языке, которое вряд ли порадует пользователя и пояснит ему, в чем дело. Поэтому надо организовывать перехват исключений, который предохранит пользователей от многих неприятностей. Делается это следующим образом.

Если в каких-то операторах возможны ошибки, связанные с недопустимыми значениями переменных или с недопустимыми значениями аргументов библиотечных функций, поместите эти операторы в конструкцию:

```
try
 <Исполняемый код, в котором могут случиться ошибки>
except
 <Код, исполняемый в случае ошибки>
end;
```

Между ключевыми словами **try** и **except** может помещаться группа операторов, при выполнении которых возможны ошибки. Если ошибки не возникнут, то операторы, помещенные вами между ключевыми словами **except** и **end**, выполняться не будут, и далее будут выполняться операторы, которые расположены после всей этой конструкции, т.е. после **end**. А если возникнет ошибка, будет сгенерирован объект исключения, выполнение прервется, и выполнятся операторы между **except** и **end**. Эти операторы называются перехватчиками исключений. Затем выполнение процедуры, в которой помещена вся эта конструкция, прервется.

В перехватчиках исключений могут помещаться специальные операторы, избирательно реагирующие на различные классы исключений. Их синтаксис:

```
on тип_исключения do оператор;
```

Подробнее об этих операторах вы можете узнать, посмотрев соответствующие темы во встроенной справе Delphi или в справке [3]. Но вместо этих специальных операторов можно написать обычные операторы, например, информирующие пользователя о возникшей ошибке и советующие, как ее исправить.

Вы уже не раз, начиная с разд. 2.4.3, применяли для сообщений функцию **ShowMessage**. Так что вы можете поместить в перехватчиках исключений, например, такой оператор:

```
ShowMessage('Вы ошиблись, исправьте исходные данные!');
```

Он покажет пользователю окно, представленное на рис. 2.11.

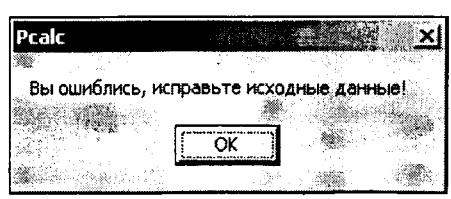


Рис. 2.11
Окно сообщения об ошибке

Можете проверить рассмотренные способы обработки ошибок на примере калькулятора, описанного в разд. 2.4.7. Если использовать перехват исключений, то все процедуры, вызывающие библиотечные функции, а также процедуры, использующие функцию **StrToFloat** надо оформить таким образом:

```
try
  Edit1.Text := FloatToStr(Sqrt(StrToFloat(Edit1.Text)));
 except
  ShowMessage('Вы ошиблись, исправьте исходные данные!');
 end;
```

В данном случае приведен пример процедуры, вычисляющей квадратный корень. В случае ошибки любого рода — неверных символов числа в окне **Edit1** (не сработает функция **StrToFloat**) или в случае отрицательного числа (не сработает функция **Sqrt**) — пользователю будет показано окно рис. 2.11. И не появится никаких сообщений об ошибке на английском языке, которые могут травмировать не очень опытного пользователя.

Впрочем, такую картину вы увидите, если выполнили рекомендацию по настройке ИСР Delphi, содержащуюся в разд. 1.2.1: выполнить команду Tools | Debugger Options, и на странице Language Exceptions диалогового окна выключить индикатор Stop On Delphi Exceptions. Если же вы это не сделали, то в случае ошибок увидите окно, показанное на рис. 2.12. Это сообщение отладчика о сгенерированном исключении. Правда, это окно появляется только при выполнении приложения из ИСР Delphi. Если вы выполните приложение средствами Windows, то этого окна не будет. Так что пользователь вашего приложения подобными сообщениями не будет травмирован. Его увидит только разработчик — то есть вы. Но я подозреваю, что вам оно тоже не доставит удовольствия, во всяком случае, пока вы не очень разбираетесь в исключениях. Именно поэтому я советовал вам настроить ИСР Delphi так, чтобы подобные сообщения не возникали.

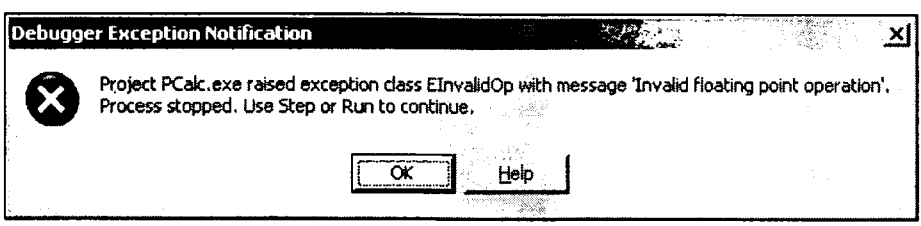


Рис. 2.12
Окно сообщения отладчика Delphi о генерации исключения

2.4.9 Именованные и типизированные константы

Помимо переменных и числовых констант, в математические выражения могут входить *именованные константы*. Это константы, которым присвоен идентификатор. Объявляются именованные константы с помощью ключевого слова **const**. Это слово начинает раздел объявления констант. После него может следовать ряд объявлений констант в форме:

```
<идентификатор константы> = <константное выражение>;
```

Под константным выражением понимается математическое выражение, в которое не входят переменные, но могут входить числовые константы и ранее объяв-

ленные именованные константы. Допускаются также вызовы некоторых (не всех) библиотечных функций.

Ниже приведены примеры объявления именованных констант:

```
const NMax = 55;
      Pi2 = 2 * Pi; // удвоенное число Пи
      Kgr = Pi/180; // коэффициент пересчета градусов в радианы
      Krg = 1./Kgr; // коэффициент пересчета радиан в градусы
```

Объявление именованной константы является указателем для компилятора заменить во всем тексте идентификатор константы его значением. Такая замена производится только в процессе компиляции и не отражается на исходном тексте. Константные выражения вычисляются компилятором в процессе компиляции и заменяются в выполняемом файле вычисленными значениями. Это сокращает время выполнения и экономит память, так как под константы и промежуточные результаты вычисления константных выражений память отводить не надо.

Цель объявления именованной константы — сделать текст более осмысленным и облегчить при необходимости изменение значения константы во всем тексте. Например, если в тексте многократно используется число 55, означающее максимально допустимое значение каких-то переменных, то применение в кодах идентификатора объявленной выше константы **NMax** делает текст более понятным, чем непосредственное использование числа 55. К тому же, через некоторое время вы можете захотеть расширить возможности вашей программы и изменить записанное в ней допустимое значение переменных, например, на 100. Вам будет много проще изменить это число в одном месте программы — в объявлении константы **NMax**, чем искать по всему тексту числа 55, которые, к тому же, в разных частях программы могут иметь разный смысл. Возможность подобной настройки приложения характеризует его *масштабируемость*.

Хороший стиль программирования

Не используйте в кодах конкретные числовые и иные константы, если в дальнейшем может потребоваться их изменить. Объявляйте соответствующие именованные константы и снабжайте их объявление комментариями. Это повысит масштабируемость приложения (возможность его настройки), сделает код более понятным и облегчит дальнейшее сопровождение программы.

Теперь рассмотрим еще один вид констант — *типизированные константы*. Несмотря на такое неудачное название и несмотря на то, что они объявляются в разделе констант **const**, в действительности они являются не константами, а переменными, инициализируемыми при их объявлении, т.е. получающими начальное значение. Обычные локальные переменные, как указывалось в разд. 2.4.1, нельзя инициализировать. А типизированные константы могут использоваться для задания начальных значений локальным переменным, впрочем, и глобальным тоже. В частности, в Delphi 1 это единственный способ задания начальных значений при объявлении любых переменных. Но начиная с Delphi 3 глобальные переменные можно инициализировать при их объявлении, не прибегая к типизированным константам.

Типизированная константа объявляется выражением:

```
const <идентификатор> : <тип> = <константное выражение>;
```

Например:

```
const In: integer = 7;
      Angl: double = 35*Pi/180;
```

В дальнейшем тексте программы с типизированными константами можно обращаться как с обычными переменными, изменяя, когда требуется, их значения. Впрочем, это можно делать только при включенной директиве компилятора **{$J+}** (см. о записи директив компилятора в разд. 2.2). До Delphi 7 эта директива компилятора была включена по умолчанию. Начиная с Delphi 7, она по умолчанию выключена (включена директива **{$J−}**). При выключенной директиве **{$J+}** изменять во время выполнения значения типизированных констант невозможно. В этом случае они становятся просто именованными константами. Так что если вы компилируете в Delphi 7 проекты, созданные в более ранних версиях и использующие типизированные константы, надо или включить директиву **{$J+}**, или убрать из проекта типизированные константы, переведя их в переменные, т.е. заменив **const** на **var**. А в новых проектах, вероятно, использовать типизированные константы не стоит — это явно устаревший элемент языка Object Pascal.

2.4.10 Логические поразрядные операции

Иногда целая переменная рассматривается не как число, а как собрание двоичных разрядов (битов), каждый из которых может принимать значение 0 или 1. И каждый бит или каждая их комбинация имеет свой смысл. Такие значащие биты или их комбинации называются *флагами*. А само число, объединяющее флаги, в этом случае является множеством флагов.

С флагами вы часто будете иметь дело на протяжении этой книги. Например, в каком-то диалоговом окне флаги могут означать доступные пользователю кнопки: младший (самый правый) разряд — кнопка OK, второй разряд — кнопка Отменить и т.д. Тогда число 1 будет соответствовать наличию в окне только кнопки OK, число 2 — наличию только кнопки Отменить, число 3 — присутствию обеих этих кнопок.

Использование флагов очень выгодно с точки зрения экономии памяти. Одна переменная даже самого компактного типа **Byte** (см. табл. 2.1 в разд. 2.4.1), занимающая всего один байт — 8 битов (двоичных разрядов), способна хранить 255 комбинаций восьми флагов. В переменной типа **Word** может храниться 65535 комбинаций 16 флагов. Переменная типа **Longword** способна хранить 4294967295 комбинаций 32 флагов.

Для использования флагов в языке предусмотрены *логические поразрядные операции*, которые работают с целыми операндами как с собраниями двоичных разрядов:

Обозначение	Операция	Пример
not	поразрядное отрицание	`not X`
and	поразрядное И	`X and Y`
or	поразрядное ИЛИ	`X or Y`
xor	поразрядное исключающее ИЛИ	`X xor Y`
shl	поразрядный сдвиг влево	`X shl 2`
shr	поразрядный сдвиг вправо	`Y shl I`

Все операции выполняются поразрядно, т.е. применяются к одинаковым разрядам операндов. Операция **not** изменяет значения всех разрядов своего единственного операнда на противоположные: 0 заменяется на 1, а 1 — на 0.

Операция **and** заносит в разряд результата 1 только в том случае, если в этом разряде и в первом, и во втором операнде записаны 1. В противном случае в разряд результата записывается 0.

Операция **or** заносит в разряд результата 1 в случае, если в этом разряде или в первом, или во втором операнде записана 1. В разряд результата записывается 0 только в том случае, если в этом разряде и в первом, и во втором операнде записаны 0.

Операция **xor** заносит в разряд результата 1 в случае, если значения этого разряда в первом и втором операндах различны, и заносит 0, если они одинаковы, независимо от того, что в них записано: нули или единицы. Ниже приведен пример записи в двоичном виде разрядов значений двух операндов **X** и **Y**, и результатов применения к ним различных операций.

X	001101
Y	100001
not X	110010
X or Y	101101
X and Y	000001
X xor Y	101100
(X xor Y) xor Y	001101

Обратите внимание на последнюю строку таблицы. В ней приведен пример, в котором операция **xor** применена к операндам **X** и **Y**, а затем повторно применена к полученному результату и к тому же операнду **Y**. В итоге получено значение операнда **X**. Такая особенность операции **xor** широко применяется в программировании для шифровки и дешифровки (см. разд. 2.8.7.2), для занесения и стирания изображений и в других случаях. Первое применение **xor** к некоторому числу с заданным вторым операндом (ключом) изменяет число (шифрует). А повторное применение **xor** с тем же ключом восстанавливает первоначальное значение числа (дешифрует его).

Теперь рассмотрим применение логических поразрядных операций для работы с флагами на следующем примере:

```
// Объявление флагов
const f1 = $1;
      f2 = $2;
      f3 = $4;
var f, r: byte;
begin
// Формирование множества всех флагов
 f := f1 or f2 or f3;
// Стирание всех флагов, кроме f1
 f := f1;
// Добавление флага f2
 f := f or f2;
// Добавление флага f3
 f := f or f3;
```

```
// Стирание флага f1
 f := f and not f1;
// Проверка наличия флага f1
 r := f and f1;  // 0, так как флаг отсутствует
// Проверка наличия флага f2
 r := f and f2;  // не 0, так как флаг присутствует
// Проверка наличия хотя бы одного из флагов f1 и f2
 r := f and (f1 or f2);  // не 0, так как присутствует флаг f2
end;
```

Флаги можно объединять операцией **or**. Эта же операция заносит флаг в множество. Действительно, независимо от того, какие значения соответствующих разрядов были в множестве до этого, операция **or** занесет единицы в те разряды, в которых содержатся единицы во флаге. А остальные разряды множества останутся неизменными. Для удаления флага из множества можно использовать операцию **or**, примененную к отрицанию (операции **not**) флага. Применение операции **not** инвертирует значения всех разрядов флага. Так что в тех разрядах, где во флаге были записаны единицы, оказываются нули. А все остальные разряды получают значение 1. Если применить операцию **and** к этому результату и множеству флагов, то в множестве обнулятся те разряды, в которых во флаге были единицы, а все остальные разряды останутся неизменными.

Проверка наличия флага во множестве осуществляется операцией **and**. Результат равен 0, если флаг отсутствует, и отличен от нуля, если флаг имеется.

В выражениях, содержащих несколько операций (как в нескольких из приведенных операторов), надо учитывать приоритет операций. Среди логических поразрядных операций наивысший приоритет имеет операция **not**. Несколько ниже приоритет операций **and**, **shl** и **shr**. Их приоритет на уровне арифметических операций умножения и деления. Низший приоритет имеют операции **or** и **xor**. Их приоритет на уровне приоритета арифметических операций сложения и вычитания.

С учетом приоритета операций, выражение

```
f and f1 or f2
```

выполняется как

```
(f and f1) or f2
```

и в приведенном выше примере всегда давало бы ненулевой результат. Именно поэтому в приведенном примере использованы скобки, которые обеспечивают требуемую последовательность выполнения операций.

Теперь рассмотрим операции сдвига. Выражения **X shl Y** и **X shr Y** сдвигают значение **X** влево или вправо на **Y** битов. Это эквивалентно умножению или делению **X** на 2^{Y}. Действительно, сдвиг влево на одну позицию в 2 раза повышает цену каждого двоичного разряда, а сдвиг вправо уменьшает цену каждого разряда. Это аналогично тому, как в десятичной системе счисления, сдвинув на n позиций влево все цифры и дополнив нулями освободившиеся правые позиции, мы получим число, умноженное на 10^{n}. Операции сдвига выполняются намного быстрее операций умножения и деления. Так что если вам надо умножить или разделить число на 2^{Y}, лучше применять сдвиг. Например, следующий оператор умножает целую переменную X на 4:

```
X := X shl 2;
```

2.5 Порядковые и ограниченные типы данных

Порядковыми (ordinal) типами называются те, в которых значения упорядочены, и для каждого из них можно указать предшествующее и последующее значения. К порядковым относятся, в частности, уже рассмотренные целые типы (см. разд. 2.4.1), и рассмотренные в разд. 2.6 символьные типы.

Для порядковых типов предопределен ряд функций:

Функция	Параметр	Возвращаемое значение
Ord	Выражение порядкового типа	Порядковый номер значения данного выражения
Pred	Выражение порядкового типа	Величина, предшествующая значению данного выражения
Succ	Выражение порядкового типа	Величина, следующая для значения данного выражения
High	Идентификатор порядкового типа или переменная порядкового типа	Максимально возможное значение
Low	Идентификатор порядкового типа или переменная порядкового типа	Минимально возможное значение

Функция **Ord** имеет смысл, например, для символьных типов и будет подробнее рассмотрена в разд. 2.6. Для целых типов она просто возвращает значение того же числа, которое задано в качестве ее аргумента. Функции **Pred** и **Succ** для целых чисел возвращают соответственно предыдущее и последующее значения. Например, выражение **Pred(4)** вернет 3, а выражение **Succ(4)** вернет 5.

Функции **High** и **Low** позволяют определить пределы, в которых могут лежать значения переменной. Например, если переменная **I** имеет тип **integer**, то выражение **High(I)** вернет 2147483647, а выражение **Low(I)** вернет −2147483648, т.е. соответственно максимальное и минимальное значения переменных типа **integer** (см. табл. 2.1 в разд. 2.4.1). Впрочем, в качестве аргумента в эти функции может передаваться не идентификатор переменной, а тип. Например: **High(integer)** или **Low(integer)**.

Для порядковых типов определены также процедуры инкремента **Inc** и декремента **Dec**. Эти процедуры соответственно увеличивают или уменьшают на единицу порядковый номер своего аргумента. Таким образом, оператор

```
Inc(I);
```

эквивалентен оператору

```
I := I + 1;
```

и оператору

```
I := Succ(I);
```

А оператор

```
Dec(I);
```

эквивалентен оператору

```
I := Pred(I);
```

и оператору

```
I := I - 1;
```

Надо учитывать, что процедуры **Inc** и **Dec** выполняются быстрее, чем эквивалентные им сложение и вычитание. Так что всегда предпочтительнее использовать именно эти процедуры.

В более общем случае в процедуры **Inc** и **Dec** можно передать еще второй аргумент — целое число, на которое уменьшается или увеличивается переменная. Так что, например, оператор

```
Inc(I, 5);
```

увеличит значение переменной **I** на 5, а оператор

```
Dec(I, 5);
```

уменьшит **I** на 5.

Для порядковых типов можно задать поддиапазон их возможных значений для вводимой вами переменной — это и будет *ограниченный тип*. Задается диапазон значений ограниченного типа выражением вида

```
<минимальное значение>..<максимальное значение>
```

Например:

```
var  Letter: 'a'..'z';
     Num:    1..12;
```

В этих примерах переменная **Letter** может принимать только символы латинских букв в нижнем регистре, а переменная **Num** — только целые числа в диапазоне 1 ÷ 12 (это могут быть, например, номера месяцев).

В приведенных примерах задаются ограниченные типы конкретным переменным. Но вы можете объявить собственный ограниченный тип данных, указать для него идентификатор, и затем использовать его при объявлении переменных. Объявляется собственный тип с помощью ключевого слова **type**. Это слово начинает раздел объявления типов. После него может следовать ряд объявлений типов в форме:

```
<идентификатор типа> = <описание типа>;
```

Например, объявления

```
type TCyrLetter = 'а'..'я';
var Sym1, Sym2 : TCyrLetter;
```

задают имя нового типа — **TCyrLetter** (строчные символы кириллицы) и определяют две переменные введенного пользователем типа — **Sym1** и **Sym2**.

Ограниченные типы используются, как вы увидите далее, при объявлениях размеров массивов (см. разд. 3.1), в операторе **case** (см. разд. 2.8.5) и в ряде других случаев.

2.6 Символьные типы данных

Символьные типы предназначены для хранения одного символа. Они относятся к целым порядковым типам (см. разд. 2.5). Основной символьный тип — **Char**. Он занимает один байт памяти. Значения символов располагаются в последовательности, которая принята в стандарте ANSI и используется в программах Windows. Индексы этой последовательности лежат в пределах от 0 до 255.

В табл. 2.4 приведены индексы основных символов. Как видно из этой таблицы, символы цифр от 0 до 9 расположены подряд в порядке возрастания цифр. Аналогично в алфавитном порядке расположены латинские буквы и буквы кириллицы (кроме букв "Ё" и "ё"). Это облегчает задание ограниченных типов (см. разд. 2.5).

Таблица 2.4. Основные символы и соответствующие им индексы

Символ	Индекс
нулевой символ	0
BackSpace	8
Tab (табуляция	9
Enter (новая строка)	13
пробел	32
0	48
1	49
...	
9	57
A (латинская буква)	65
B	66
...	
Z	90
a (латинская буква)	97
b	98
...	
z	122
Ё	168
ё	184
А (буква кириллицы)	192
Б	193
...	
Я	223
а (буква кириллицы)	224
б	225
...	
я	255

Поскольку символьный тип **char** относится к порядковым, для него определены такие функции и процедуры, как **Ord**, **Pred**, **Succ**, **Inc**, **Dec** и др. Функция **Ord**, возвращает номер символа. Например, выражение **Ord('я')** вернет 255. Для символьного типа определена также функция **Chr**, возвращающая символ любого целого значения. Например, **Chr(255)** возвращает букву "я". Функция **Chr** противоположна по смыслу функции **Ord**. Т.е. **Ord(Chr(255))** вернет 255, а **Chr(Ord('я'))** вернет символ "я".

Символьным переменным можно присваивать значения символьных констант. Символьные константы записываются как соответствующий символ, заключенный в одинарные кавычки. Например, 'A'. Но подобную форму записи можно использовать только для печатаемых символов. Например, символ табуляции или символ перехода на новую строку так записать невозможно. Поэтому используется еще одна форма записи символьных констант: символ "#", после которого указывается десятичное или шестнадцатеричное число от 0 до 255, соответствующее коду ASCII нужного символа (см. табл. 2.4). Например, обозначение #13 соответствует символу перевода строки.

2.7 Процедуры и функции

2.7.1 Объявление и описание функций и процедур

Процедуры и функции представляют собой программные блоки, которые могут вызываться из разных частей программы. При вызове в них передаются некоторые переменные, константы, выражения, являющиеся аргументами, которые в самих процедурах и функциях воспринимаются как формальные параметры. При этом функции возвращают значение определенного типа, которое замещает в вызвавшем выражении имя вызванной функции.

Например, оператор

```
I := 5 * F(X);
```

вызывает функцию **F** с аргументом **X**, умножает возвращенное ею значение на 5 и присваивает результат переменной **I**.

Допускается также вызов функции, не использующий возвращаемого ею значения. Например:

```
F(X);
```

В этом случае возвращаемое функцией значение игнорируется. Такой вызов может иметь смысл, если функция вызывается для того, чтобы изменить какие-то значения глобальных переменных, чтобы показать пользователю какое-то сообщение и т.д.

Функция описывается следующим образом:

```
function <имя функции>(<список параметров>):
                                        <тип возвращаемого значения>;
<объявления локальных переменных, типов, констант, описания вложенных
функций и процедур>
begin
 <операторы тела функции>
end;
```

Первая строка, содержащая имя функции, список параметров и тип возвращаемого значения, называется заголовком или объявлением функции. Если функция, описываемая в данном модуле, должна быть доступна из других модулей, то эта первая строка должна быть продублирована в разделе **interface** в качестве объявления функции.

Список параметров и объявления локальных элементов и вложенных процедур не обязательны. Если список параметров отсутствует, то скобки после имени функции не ставятся.

Формы описания списка параметров будут рассмотрены позднее. В простом случае это список имен формальных параметров с указанием их типов. Например, объявление:

```
function FSum(X1, X2: real; A: integer): real;
```

объявляет функцию с именем **FSum**, с тремя параметрами **X1**, **X2** и **A**, из которых первые два типа **real**, а последний — **integer**. Тип возвращаемого результата — **real**. Имена параметров **X1**, **X2** и **A** — локальные, т.е. они имеют значение только внутри данной функции и никак не связаны с именами аргументов, переданных при вызове функции. Значения этих параметров в начале выполнения функции равны значениям аргументов на момент вызова функции. Подробнее эти вопросы будут рассмотрены в разд. 2.7.2.

Тело функции пишется по тем же правилам, что и любой код программы. При этом надо учитывать области видимости различных элементов программы. В теле функции видны объявленные в ней локальные элементы (переменные, функции и т.п.) и глобальные элементы. Подробнее об этом см. в разд. 2.4.4.

Возвращаемое значение в теле функции может присваиваться или имени функции, или специальной предопределенной переменной **Result**. Например, тело приведенной выше функции **FSum** может иметь вид:

```
begin
 Fsum := A * (X1 + X2);
end;
```

или

```
begin
 Result := A * (X1 + X2);
end;
```

Результат работы обеих приведенных выше функций будет одинаковым. Но переменная **Result** имеет одно принципиальное отличие от имени функции. Имя функции — это не переменная. Оно может фигурировать только в левой части оператора присваивания, и не может входить ни в какие выражения. А **Result** — это обычная переменная. Поэтому, например, приведенный выше код можно было бы переписать следующим образом:

```
begin
 Result := X1 + X2;
 Result := Result * A;
end;
```

С именем функции это сделать невозможно, так как оно не может встречаться в правой части оператора присваивания.

В Object Pascal существует предопределенная процедура **exit**, которая обеспечивает выход из функции или процедуры в любом месте ее тела. Например, приведенный выше текст мог бы быть записан так:

```
begin
 Result := X1 + X2;
 if (A = 1) then exit;
 Result := Result * A;
end;
```

В этом коде использован оператор **if**, который будет рассмотрен позднее в разд. 2.8.4.4. Но смысл его достаточно прост: если **A** равняется 1, то вызывается процедура **exit**. Сумма **X1** + **X2** к этому моменту уже занесена в результат **Result**. А умножать этот результат на 1 не имеет смысла.

Прервать выполнение функции или процедуры можно также генерацией какого-то исключения (см. разд. 2.4.8). Наиболее часто в этих целях используется процедура **Abort**, генерирующая «молчаливое» исключение **EAbort**, не связанное с каким-то сообщением об ошибке. Если в программе не предусмотрен перехват этого исключения, то применение функции **Abort** выводит управление сразу наверх из всех вложенных друг в друга процедур и функций.

Объявление процедуры практически ничем не отличается от объявления функции, кроме того, что для процедуры не указывается возвращаемое значение:

```
procedure <имя процедуры>(<список параметров>);
<объявления локальных переменных, типов, констант, описания вложенных
функций и процедур>
begin
 <операторы тела процедуры>
end;
```

Например:

```
procedure Pr1(S: string);
begin
 Form1.Label1.Caption:=S;
end;
```

Вызов этой процедуры может иметь вид:

```
Pr1('Привет !');
```

Впрочем, с процедурами вы уже неоднократно имели дело: все обработчики событий, которые вы писали, являются процедурами.

Теперь посмотрим, где можно размещать в модуле вводимые вами функции и процедуры. Их можно делать глобальными, помещая ее вне каких-либо других процедур и функций. Тогда вы сможете вызывать их из любых процедур и функций, расположенных в коде после их описания. Пусть, например, вы написали функцию вычисления площади круга. Если вы поместите ее реализацию в начале раздела **implementation** вашего модуля, то сможете вызывать ее из любых других функций, процедур и обработчиков событий. Например:

```
function Area(R: Real): Real;
// Площадь круга с радиусом R
begin
```

```
  Result := Pi *  sqr(R);
end;

procedure TForm1.Button1Click(Sender: TObject);
var A: real;
begin
 A := Area(StrToFloat(Edit1.Text));
 ...
end;

procedure TForm1.Button2Click(Sender: TObject);
var B: real;
begin
 B := Area(StrToFloat(Edit2.Text));
 ...
end;
```

Но если вы поместите реализацию функции в конце модуля, то в операторах вызова этой функции получите сообщение об ошибке: «Undeclared identifier: 'Area'» — «Неопределенный идентификатор: 'Area'». Вспомните первое правило синтаксиса языка Object Pascal (см. разд. 2.2), гласящее, что все идентификаторы должны быть объявлены до первого их использования. Это правило вы нарушили, и компиляция модуля невозможна. В данном случае все просто: надо поместить реализацию функции раньше функций и процедур, вызывающих ее. Но иногда это невозможно. Например, вы можете написать функцию **F1**, вызывающую в каких-то ситуациях другую вашу функцию **F2**, а та в свою очередь может вызывать функцию **F1**. Получается известная неразрешимая дилемма: что было сначала — курица или куриное яйцо? Выход из подобного положения: включить в раздел интерфейса модуля **interface** объявление функции, которая вызывается до ее реализации. В приведенном примере вы можете включить в интерфейс (например, перед ключевым словом **implementation**) объявление:

```
function Area(R: Real): Real;
```

Тогда все будет нормально, даже если реализацию функции **Area** вы поместите в самом конце модуля. Кстати, в этом случае функцию смогут вызывать даже другие модули, ссылающиеся на данный в своем предложении **uses**.

Рассмотренные варианты относились к глобальным функциям и процедурам, которые должны вызываться из разных функций и процедур программы. Если же вы используете реализованную функцию только в какой-то одной процедуре, то лучше сделать ее локальной. Например:

```
procedure TForm1.Button1Click(Sender: TObject);
{ Переменные A и B могут использоваться и в Area,
  и в основной процедуре }
var A, B: real;
// Локальная функция
function Area(R: Real): Real;
// Площадь круга с радиусом R
begin
  Result := Pi *  sqr(R);
end;
// Основная процедура
begin
 ...
```

```
 A := Area(StrToFloat(Edit1.Text));
 ...
 B := Area(StrToFloat(Edit2.Text));
 ...
end;
```

Еще один вариант размещения ваших функций и процедур в модуле: включение их в класс формы. Классы мы будем рассматривать много позднее (см. разд. 3.5), но включение своих функций в класс формы можно обсудить сейчас. Если вы посмотрите начало вашего модуля, то сможете найти фрагмент следующего вида:

```
type
  TForm1 = class(TForm)
    ...
  private
{ Private declarations }
    {Тут могут включаться объявления элементов,
     доступных в данном модуле}
  public
{ Public declarations }
    {Тут могут включаться объявления элементов, доступных
    в данном модуле и во внешних модулях, ссылающихся на данный}
  end;
```

Это объявление класса вашей формы **TForm1**. Вы можете включить в его разделы **private** или **public** объявление вашей функции. Например:

```
private
 function Area(R: Real): Real;
```

Но тогда вы должны несколько изменить заголовок реализации этой функции в разделе модуля **implementation**, указав в нем, что вы реализуете функцию класса **TForm1**:

```
function TForm1.Area(R: Real): Real;
begin
  Result := Pi *  sqr(R);
end;
```

В нашем простом примере вы не заметите разницы между предыдущими вариантами и включением функции в класс. Только не надо будет думать о том, в каком месте раздела **implementation** модуля поместить реализацию функции. Но если вы захотите обратиться из вашей функции к компонентам, имеющимся на форме, то разница будет. Например, в функции, включенной в класс формы, вы можете написать оператор

```
Label1.Caption := FloatToStr(Result);
```

Но если вы хотите включить подобный оператор в функцию, не относящуюся к классу формы, то должны будете заменить его следующим:

```
Form1.Label1.Caption := FloatToStr(Result);
```

Функция или процедура, не являющаяся элементом класса формы, может получить доступ к компонентам формы только через указание имени объекта этой формы — в данном примере **Form1**. До сих пор вам не приходилось ссылаться на компоненты таким образом только потому, что все обработчики событий, с которыми вы имели дело, автоматически объявлялись Delphi элементами класса формы.

Процедуры и функции могут вызывать не только друг друга, но и сами себя. Такой вызов, называемый рекурсивным, будет рассмотрен в разд. 2.9.

2.7.2 Различные варианты передачи параметров в функции и процедуры

Список параметров, передаваемый в процедуры и функции, как было показано в предыдущем разделе, состоит из имен параметров и указаний их типа. Например, в объявлении

```
procedure Pr(X1, X2: real; A: integer);
```

указано три параметра **X1**, **X2**, **A** и определены их типы. Вызов такой процедуры может иметь вид:

```
Pr(Y, X2, 5);
```

Это только один из способов передачи параметров в процедуру, называемый *передачей по значению*. Работает он так. В момент вызова функции в памяти создаются временные переменные с именами **X1**, **X2**, **A**, и в них копируются значения аргументов **Y**, **X2** и константы 5. На этом связь между аргументами и переменными **X1**, **X2**, **A** разрывается. Вы можете изменять внутри процедуры значения **X1**, **X2** и **A**, но это никак не отразится на значениях аргументов. Аргументы при этом надежно защищены от непреднамеренного изменения своих значений вызванной процедурой или функцией.

К недостаткам такой передачи параметров по значению относятся затраты времени на копирование значений и затраты памяти для хранения копии. Если речь идет о какой-то переменной простого типа, это, конечно, не существенно. Но если, например, аргумент — массив из тысяч элементов, то соображения затрат времени и памяти могут стать существенными.

Еще одним недостатком передачи параметров по значению является невозможность из функций и процедур изменять значения некоторых аргументов, что во многих случаях очень желательно.

Возможны и другие способы передачи параметров: как переменных (иногда это называется передачей по ссылке), как констант и как выходных величин. Эти способы осуществляются указанием перед именем соответствующего параметра спецификатора **var**, **const** или **out**.

При передаче параметра *по ссылке* перед его именем в заголовке процедуры должно быть указано ключевое слово **var**. Например:

```
procedure Pr(var X1: real; X2: real; A: integer);
```

В этом случае не происходит копирования значения аргумента в локальную, временную переменную в процедуре. Процедура реально работает не с параметром, а со ссылкой — указателем на место хранения аргумента в памяти. Соответственно, в приведенном примере любые изменения параметра **X1**, произведенные в процедуре, в действительности относятся не к этому параметру, а к тому аргументу **Y**, который передан при вызове процедуры. Таким образом, ключевое слово **var** позволяет возвращать информацию из процедуры в вызвавшую его внешнюю процедуру.

Передача параметра *как константы* осуществляется заданием перед его именем ключевого слова **const**. Например:

```
procedure Prc(const X1:real; X2: real; A: integer);
```

Такая передача параметра почти идентична по своим последствиям обычной передаче по значению. Как и в том случае, значение аргумента невозможно изменить, изменяя параметр процедуры. Но есть одно существенное отличие: если при передаче параметра по значению вы можете в тексте процедуры изменять значение соответствующей ему локальной переменной, то при передаче параметра как константы это невозможно. При попытке в теле процедуры изменить значение параметра, переданного как константа, компилятор выдаст сообщение об ошибке.

Передача параметра как константы позволяет сделать код более эффективным, так как при этом компилятору заведомо известно, что никакие изменения параметра невозможны.

Передача параметра *как выходного* осуществляется заданием перед его именем спецификатора **out**. Например:

```
procedure Prc(out X1:real; X2: real; A: integer);
```

Передача параметра как выходного очень похожа на передачу по ссылке. Отличие в том, что при этом не гарантируется передача в процедуру начального значения этого параметра. Часто такая передача применяется для аргументов, являющихся структурами. При этом память, занимаемая аргументом, в момент обращения к процедуре очищается. Таким образом, параметр, переданный как выходной, указывает на некоторое место в памяти, являющееся просто контейнером, куда процедура должна занести соответствующее выходное значение.

Варианты передачи параметра по ссылке, как константы и как выходного обладают одним интересным свойством: они допускают объявление этого параметра в списке процедуры без указания его типа. Это так называемый *нетипизированный параметр*. В этом случае предполагается, что процедура или функция может принимать параметр любого типа. Но при этом в качестве аргумента нельзя передавать в процедуру или функцию число или нетипизированную числовую константу.

Пример объявления процедуры с нетипизированными параметрами:

```
procedure Pr(var X1; X2: real; var A);
```

В этом примере параметры **X1** и **A** объявлены нетипизированными.

В тексте процедуры или функции для нетипизированного параметра должна применяться операция явного приведения типов (см. разд. 2.4.2), определяющая тип этого параметра. Например:

```
real(X1):=real(X1)*10;
```

Реальный тип аргумента должен соответствовать типу, принятому в процедуре, но не обязательно точно. Например, если параметр — массив, то размер массива, определенный в процедуре, может превышать размер аргумента. Если же получается несоответствие типов (например, аргумент — **real**, а в процедуре он описан как **integer**), то никаких сообщений об ошибках и никаких исключений генерироваться не будет, но результаты вычислений будут неверными.

2.7.3 Параметры со значениями по умолчанию

Для параметров в объявлениях функций и процедур могут задаваться значения по умолчанию. Значение по умолчанию — это значение параметра в случае, если он не передан в вызове функции или процедуры. Таким образом, введение параметров со значениями по умолчанию равносильно разрешению передавать при вызове функции не все необходимые аргументы.

Значения по умолчанию задаются добавлением в конце объявления параметра знака равенства "=", после которого записывается константное выражение. Пусть, например, вы хотите написать функцию, которая рассчитывает суммарную силу, действующую на тело объемом **V** с плотностью **P**, погруженное в жидкость (например, воду) с плотностью **PH2O**. Как известно, формула, выражающая эту суммарную силу, направленную вверх (если ответ будет отрицательным, значит сила направлена вниз — тело тонет), следующая: **F = G * V * (P – PH2O)**, где **G** — ускорение свободного падения.

Функцию, определяющую эту силу, можно описать следующим образом:

```
function Arh(V:double = 1; P:double = 0.5;
             PH2O:double = 1; G:double = 9.81): double;
begin
 Arh := G * V * (PH2O - P);
end;
```

Здесь всем параметрам даны значения по умолчанию. Объем **V** по умолчанию принят равным 1 м^3, плотность тела **P** по умолчанию равна 0.5 т/м^3 (плотность некоторых пород дерева), плотность воды **PH2O** принята по умолчанию равной 1 т/м^3, а ускорение свободного падения **G** принято равным 9,81 м/с^2.

Если при вызове функции параметр по умолчанию не указан, то в функцию автоматически передается его значение по умолчанию. Например, если вызвать приведенную функцию оператором

```
F := Arh();
```

то значение **F** будет равно силе при значениях всех параметров по умолчанию.

Аргументы по умолчанию должны быть самыми правыми (последними) аргументами в списке параметров функции. Если вызывается функция с двумя или более параметрами по умолчанию и если пропущенный параметр не является самым правым в списке, то все параметры справа от пропущенного тоже пропускаются.

Например, вызов той же функции оператором

```
F := Arh(2);
```

позволяет рассчитать силу, действующую на тело объемом 2 м^3 при значениях всех остальных параметров по умолчанию. Вызов функции оператором

```
F := Arh(2,2.6);
```

позволяет рассчитать силу, действующую на алюминиевое (плотность 2.6 т/м^3) тело объемом 2 м^3 при значениях остальных параметров по умолчанию. Аналогично, задав при вызове три параметра, можно рассчитать силу, действующую на тело, погруженное в жидкость другой плотности, а задав все четыре параметра можно определить силу, действующую на тело при эксперименте, проводящемся не на уровне моря (при этом изменится ускорение свободного падения).

Этот пример показывает, что последними в списке параметров со значениями по умолчанию надо указывать те параметры, значения которых в реальных задачах чаще всего остаются равными заданным по умолчанию.

Пропускать при вызове можно только некоторое число последних параметров в списке. Например, нельзя вызвать функцию таким образом:

```
F := Arh(2,,1.1);  // Ошибочный вызов
```

Значения по умолчанию могут задаваться не всем, а только некоторым параметрам функции или процедуры. В этом случае действует правило, согласно которому все параметры со значениями по умолчанию должны размещаться в конце списка в объявлении функции или процедуры. Иначе говоря, если какой-то параметр имеет значение по умолчанию, то и все последующие параметры в списке должны иметь значения по умолчанию.

2.7.4 Перегрузка функций

Вы можете определить в одной и той же области видимости (см. разд. 2.4.4) несколько процедур или функций с одинаковыми именами, но различающихся по числу или типу параметров. Если после соответствующих объявлений поставить ключевое слово **overload**, то при вызове процедуры или функции с этим именем компилятор проанализирует передаваемые параметры, их число и тип и вызовет тот метод, который подходит данным параметрам. Например, вы можете определить следующие функции:

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X / Y;
end;

function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

Обе функции объявлены как перегруженные с именем **Divide**, но первая из них получает действительные аргументы, а вторая — целые. Значит, если будет записан вызов **Divide(5, 3)**, будет вызываться первая функция, а при вызове **Divide(5.0, 3.0)** — вторая.

Еще один пример. Вы можете написать функции с одинаковым именем **Area**, вычисляющие площади различных геометрических фигур. Например:

```
function Area(X, Y: Real): Real; overload;
// Площадь прямоугольника со сторонами X и Y
begin
  Result := X * Y;
end;

function Area(R: Real): Real; overload;
// Площадь круга с радиусом R
begin
  Result := Pi *  sqr(R);
end;
```

2.7.5 Объявление и использование процедурных типов

Иногда бывает полезно объявить переменную, соответствующую типу процедуры или функции — переменную так называемого процедурного типа. Тогда присваивая ей в процессе выполнения значения тех или иных функций того же типа, можно одним и тем же неизменным оператором вызывать различные функции. Процедурный тип объявляется с помощью знакомых вам ключевых слов **function**

и **procedure**, после которого указывается список аргументов и тип результата. Например, вы можете ввести следующее объявление:

```
var F: function(const X: Extended): Extended = ArcCos;
```

Тогда переменной **F** можно присвоить значение любой функции, список параметров и тип которой совпадает с указанными в объявлении. Например, под это объявление подходят обратные тригонометрические функции (см. разд. 2.4.6). Одна из них — **ArcCos** задана в качестве начального значения. Отметим, что имена аргументов в объявлении процедурной переменной и в функции не обязательно должны совпадать. Но малейшее несовпадение типов недопустимо. Например, в приведенном объявлении нельзя исключить спецификатор **const**, который имеется в объявлениях обратных тригонометрических функций.

Если вы объявили указанным образом переменную **F**, то далее вы можете вызвать соответствующую функцию обычным оператором:

```
A := F(X);
```

Тогда переменной **A** будет присвоено значение арккосинуса переменной **X**. Но если вы перед вызовом функции выполните оператор

```
F := ArcSin;
```

то тот же оператор присвоит **A** значение арксинуса. Только не забудьте, если захотите проверить все это на практике, подключить оператором **uses** модуль *Math*. А то компилятор не поймет идентификаторы обратных тригонометрических функций.

Можно объявлять не только переменные процедурных типов, но и собственные типы функций или процедур. Тогда функции или процедуры такого типа можно передавать в другие функции и процедуры в качестве аргументов. Рассмотрим простой пример. Пусть вы хотите реализовать функцию, которая рассчитывала бы арксинус или арккосинус, но возвращала бы угол не в радианах, как это делают библиотечные функции, а в градусах. Это можно сделать следующим образом:

```
uses math;

type TFDeg = function(const X: Extended): Extended;

function TrigDeg(X: Extended; F: TFDeg): Extended;
const K = 180. / Pi;
begin
 Result := K * F(X);
end;
```

Предложением **type** объявляется тип функции **TFDeg**, соответствующий функциям **ArcSin** и **ArcCos**. Далее реализуется функция **TrigDeg**, которая принимает в качестве второго параметра функцию типа **TFDeg**. Такой функцией может быть **ArcSin** или **ArcCos**. В теле функции **TrigDeg** вызывается переданная в нее функция, и возвращается результат ее вычисления, умноженный на коэффициент, переводящий радианы в градусы.

Вызов функции **TrigDeg** в программе может иметь вид:

```
A := TrigDeg(X, ArcSin);
```

или

```
A := TrigDeg(X, ArcCos);
```

Еще один пример применения процедурных типов вы найдете в разд. 2.8.7.4.

2.8 Операторы

2.8.1 Оператор присваивания и его соотношение с методом Assign

Оператор присваивания вы уже знаете и не раз использовали. Отметим только, что применительно к объектам надо четко представлять различие между оператором присваивания и методом копирования **Assign**, свойственным многим классам объектов. Метод **Assign** объявлен следующим образом:

```
<объект-приемник>.Assign(<объект-источник>);
```

Например:

```
A.Assign(B);
```

Этот оператор копирует содержание объекта **A** (все его свойства) в объект **B**.

Для тех же самых объектов **A** и **B** можно записать оператор присваивания:

```
A := B;
```

Различие между двумя приведенными операторами следующее. Метод **Assign** копирует содержимое одного объекта в другой. Таким образом в памяти будет иметься два объекта **A** и **B** одинакового содержания. А оператор присваивания, примененный к указателям (имя объекта — это указатель на объект), присваивает указателю **A** значение указателя **B**. Таким образом, и **A**, и **B** будут указывать на один и тот же объект в памяти. А тот объект, на который до выполнения этого оператора указывал **A**, может быть вообще потерян, если в программе где-то не хранится другой указатель на него.

2.8.2 Оператор передачи управления goto

Оператор **goto** позволяет прервать обычный поток управления — последовательное выполнение операторов, и передать управление в произвольную точку кода, помеченную специальной меткой. Метки, на которые может передаваться управление, объявляются ключевым словом **label**, после которого следует список меток. Каждая метка может обозначаться допустимым идентификатором или числом от 0 до 9999. Например, следующее объявление

```
label Lbegin, 1, second;
```

объявляет три метки: **Lbegin**, **1** и **second**. Объявление меток допускается в тех же местах кода, где делаются другие объявления: переменных и констант.

В тексте программы метка отмечает точку, в которую передается управление оператором **goto**. Метка может располагаться в любом месте блока, как после оператора **goto**, передающего на нее управление, так и до этого оператора. Не разрешается передавать управление на метку, расположенную в другом блоке.

Точка, в которую может передаваться управление, помечается именем метки, после которого следует двоеточие ":". Например:

```
Lbegin:
```

Затем может следовать оператор, на который передается управление.

Сам оператор **goto** имеет форму:

```
goto <метка>;
```

Все три элемента этой конструкции: объявление меток, сами метки и операторы **goto**, передающие на них управление, должны размещаться в пределах одного блока (см. разд. 2.4.4). Таким образом, организация работы с операторами **goto** может выглядеть, например, так:

```
label Lbegin, 1, second;
...
begin
 ...
 Lbegin: ...
 ...
 goto 1;
 ...
 second: ...
 ...
 1: ...
 ...
 if ... then goto 1 else goto second;
 ...
end;
```

В свое время при появлении концепции структурного программирования на оператор **goto** обрушился поток критики, и его применение стало рассматриваться как дурной тон. Действительно, чрезмерно широкое применение **goto** делает структуру программы крайне запутанной и затрудняет ее сопровождение. Однако во многих случаях стремление обойтись без оператора **goto** не только не упрощает код, а еще более его запутывает. Так что этот оператор, безусловно, имеет право на существование.

Хороший стиль программирования

Увлечение оператором goto делает структуру программы крайне запутанной и затрудняет ее сопровождение, так как трудно отыскать, куда передается управление в том или ином случае. Но принципиальный отказ от оператора goto в ряде случаев не только не упрощает код, а еще более его запутывает. Так что использование оператора goto надо минимизировать, но ни в коем случае не отказываться от него, если этот оператор позволяет сделать программу более простой и наглядной.

2.8.3 Оператор with

Оператор **with** используется для сокращения записи при обращении к свойствам и методам объекта, а также при обращении к полям записей (это тип данных, который будет рассмотрен в разд. 3.3.1). В этих случаях применение **with** позволяет избежать повторных ссылок на объект в последующих операторах. Оператор **with** может записываться следующим образом:

```
with <объект> do <оператор>;
```

В операторе, следующем за ключевым словом **do**, можно для полей, свойств и методов объекта, указанного как <объект>, не включать ссылки на этот объект. При этом каждый идентификатор в операторе, который совпадает с именем поля, свойства, метода объекта, трактуется как относящийся к этому объекту, и к нему неявно добавляется ссылка на этот объект.

Например, группу операторов

```
Form1.Label1.Left := Form1.Label1.Left + 10;
Form1.Label1.Caption := Label2.Caption;
Form1.Label1.Font.Color := clRed;
```

с помощью **with** можно записать короче:

```
with Form1.Label1 do begin
 Left := Left + 10;
 Caption := Label2.Caption;
 Font.Color := clRed;
end;
```

Возможна другая форма оператора **with**:

```
with <объект 1>, <объект 2>, ..., <объект n> do <оператор>;
```

Она соответствует множеству вложенных друг в друга конструкций **with**:

```
with <объект 1> do
 with <объект 2> do
  ...
   with <объект n> do
    <оператор>;
```

2.8.4 Операторы выбора if

2.8.4.1 Логические выражения, операции отношения и булевы операции

Оператор **if** предназначен для выполнения тех или иных действий в зависимости от истинности или ложности некоторого условия. Условие задается выражением, имеющим результат булева типа. Переменная или выражение булева типа (логическое выражение) может принимать одно из двух значений: **true** — истина, **false** — ложь. Для объявления булевой переменной используется тип **Boolean**. Например:

```
var L: Boolean;
```

В языке предопределены две булевы константы: **true** и **false**.

В логических выражениях может использоваться ряд операций отношения, производящих сравнение двух операндов. Операции возвращают **true**, если указанное соотношение операндов выполняется, и **false**, если соотношение не выполняется. Определены следующие операции отношения:

Обозначение	Операция	Пример
=	Равно	I = MMax
<>	Не равно	X <> Y
<	Меньше чем	X < Y

Обозначение	Операция	Пример
>	Больше чем	`Len > 0`
<=	Меньше или равно	`Cnt <= I`
>=	Больше или равно	`I >= 1`

Операнды должны иметь совместимые типы, за исключением типов **real** и **integer**, которые могут сравниваться друг с другом. Например, выражение

```
10. * sin(A) > 1
```

вернет **true**, если синус от переменной **A** окажется больше 0,1.

Применение только операций отношения не позволяет сформулировать сложные условия, например, выяснить, выполняется ли одновременно несколько условий. Для формирования подобных сложных логических выражений используются булевы операции. Они принимают операнды булева типа, так что могут принимать результаты операций отношения, и возвращают результат тоже булева типа. Определены следующие булевы операции:

Обозначение	Операция	Пример
not	Отрицание, возвращает **true**, если операнд = **false**.	`not (C <> A)`
and	Логическое И, возвращает **true**, если оба операнда = **true**.	`Done and (Total > 0)`
or	Логическое ИЛИ, возвращает **true**, если хотя бы один операнд = **true**.	`A or B`
xor	Логическое исключающее ИЛИ, возвращает **true**, если значения операндов не равны друг другу.	`A xor B`

Операции имеют те же обозначения, что и рассмотренные в разд. 2.4.10 логические поразрядные операции. И логика этих операций та же. Но те операции применялись к целым операндам, работали с их отдельными разрядами и возвращали целый результат. А булевы операции работают с булевыми операндами и возвращают булев результат: **true** или **false**.

Рассмотрим примеры совместного использования операций отношения и булевых операций. Выражение

```
(A > B) and (A < C)
```

возвращает **true**, если **B < A < C**. Выражение

```
(C > B) and ((A > C) or (A < B))
```

возвращает **true**, если **C > B** и **A** лежит вне интервала [B, C]. В выражениях, подобных этому, надо учитывать приоритет булевых операций. Он подобен приоритету логических поразрядных операций. Наивысшим приоритетом обладает операция **not**. Ниже приоритет операции **and**. Самый низкий приоритет имеют операции **or** и **xor**. Так что если бы в приведенном выражении не были поставлены соответствующие скобки, т.е. если бы оно имел вид

```
(C > B) and (A > C) or (A < B)
```

то выражение возвращало бы **true**, если **A > C > B** или **A < B**, т.е. смысл выражения был бы совсем иной.

Компилятор Delphi поддерживает два режима вычисления операций **and** и **or**: полный и сокращенный. В режиме полного вычисления все логическое выражение вычисляется до конца, даже если после вычисления первого операнда результат ясен. В режиме сокращенного вычисления расчет прерывается, как только результат ясен. Например, операция **or** дает результат **true**, если хотя бы один операнд равен **true**. Значит, если первый операнд равен **true**, то результат ясен, и в режиме сокращенного вычисления расчет прервется без вычисления второго операнда. Аналогично если первый операнд операции **and** равен **false**, то результат операции ясен, и второй операнд не вычисляется.

Директива компилятора **{$B−}** (см. о директивах в разд. 2.2), работающая по умолчанию, обеспечивает сокращенный режим вычисления, а директива **{$B+}** — полный. Режим полного вычисления можно также установить опцией Complete Boolean Evaluation на странице опций компилятора в окне опций проекта.

2.8.4.2 Операторы if

Теперь рассмотрим оператор выбора **if**. Он имеет две формы: **if...then** и **if...then...else**. Форма **if...then** имеет вид:

```
if <условие> then <оператор>;
```

Если условие возвращает **true**, то указанный в конструкции **if** оператор выполняется. В противном случае управление сразу передается оператору, следующему за конструкцией **if**. Например, в результате выполнения операторов

```
C := A;
if B > A then C := B;
```

переменная **C** станет равна максимальному из чисел **A** и **B**, поскольку оператор **C := B** будет выполнен только при **B > A**.

Форма конструкции **if...then...else** имеет вид:

```
if <условие> then <оператор1> else <оператор2>;
```

Если условие возвращает **true**, то выполняется первый из указанных операторов, в противном случае выполняется второй оператор. Обратите внимание, что в конце первого оператора перед ключевым словом **else** точка с запятой не ставится.

Приведем примеры.

```
if J = 0
 then
  ShowMessage('Деление на нуль')
 else
  Result := I/J;
```

В качестве и первого, и второго оператора могут, конечно, использоваться составные операторы:

```
if J = 0
 then begin
  ShowMessage('Деление на нуль');
  Result := 0;
 end
```

```
else
 Result := I/J;
```

Опять обратите внимание, что перед **else** точка с запятой не ставится.

При вложенных конструкциях **if** могут возникнуть неоднозначности в понимании того, к какой из конструкций **if** относится элемент **else**. Компилятор всегда считает, что **else** относится к последней из конструкций **if**, в которой не было раздела **else**.

Например, в конструкции

```
if <условие1> then if <условие2> then <оператор1>
                  else <оператор2>;
```

else будет отнесено компилятором ко второй конструкции **if**, т.е. <оператор2> будет выполняться в случае, если первое условие истинно, а второе ложно. Иначе говоря, вся конструкция будет прочитана как

```
if <условие1>
 then begin
  if <условие2> then <оператор1> else <оператор2>
 end;
```

Если же вы хотите отнести **else** к первому **if**, это надо записать в явном виде с помощью операторных скобок **begin...end**:

```
if <условие1>
 then begin
   if <условие2> then <оператор1>
  end
 else <оператор2>;
```

2.8.4.3 Пример — усовершенствованный калькулятор

В качестве примера применения оператора **if** и ряда рассмотренных операций давайте вернемся к приложению калькулятора, созданному в разд. 2.4.7, и усовершенствуем его. В том приложении имеется пока множество недостатков. Прежде всего — в нем отсутствуют кнопки, соответствующие арифметическим операциям. Кроме того, пользователь не гарантирован от возможности ввести по ошибке несколько десятичных запятых в одном числе или ввести нули, предшествующие значащим цифрам. И в целом алгоритм работы нашего калькулятора отличается от общепринятого. Попробуйте выполнить стандартную программу Windows *Калькулятор*, или поработать с любым имеющимся у вас калькулятором и составить алгоритм его работы. Очень полезная задача для развития алгоритмического мышления. К сожалению, в рамках книги я не могу попросить вас сделать это самостоятельно, выслушать ваши версии и обсудить их. Кто хочет, попробуйте решить эту задачу, не заглядывая сначала в приведенную далее мою версию (это тоже только версия, полученная в результате экспериментов, так как я не знаю истинного алгоритма). Моя версия алгоритма выглядит следующим образом (она несколько упрощена, так как мы еще не изучали работу со строками, которая необходима для реализации некоторых возможностей).

При щелчках на кнопках с функциями с одним аргументом (у нас пока реализованы только такие функции) выполняется соответствующая функция с аргументом, заданным в окне редактирования, и задается признак ввода нового числа, чтобы последующие щелчки на кнопках с цифрами начинали новое число. Для дальнейшего изложения обозначим этот признак как **LBegin**. Так что в данном случае задается **LBegin = true**.

При щелчке на кнопках с цифрами или на кнопке с запятой алгоритм довольно сложный:

1. Если **LBegin = true**, то признак ввода запятой (обозначим его как **LPoint**) устанавливается в **false** — это будет означать, что запятая еще не вводилась.
2. Если щелчок на кнопке с символом "0", а текст в окне редактирования уже равен "0", то ничего делать не надо. Если щелчок на кнопке с запятой, а **LPoint = true**, то тоже ничего делать не надо.
3. Если условия пункта 2 не выполнены, надо заносить текст в окно редактирования.
   - **3.1** Если щелчок на кнопке с запятой, то значение **LPoint** задается равным **true**, чтобы в дальнейшем запятые не заносить. Если **LBegin = true**, то в окно редактирования заносится символ "0".
   - **3.2** Если **LBegin = false** или щелчок на кнопке с запятой, то к тексту в окне редактирования добавляется справа символ кнопки, т.е. ввод числа продолжается. В противном случае символ кнопки заносится в окно редактирования вместо имеющегося там текста, т.е. ввод числа начинается.
   - **3.3** Задается **LBegin = false**.

При щелчке на кнопке арифметической операции число, введенное в окно редактирования, запоминается как первый операнд, запоминается операция, соответствующая нажатой кнопке. Устанавливается в **true** признак того, что была введена операция (для этого признака введем переменную **LOp**). Устанавливается в **true** **LBegin** — признак начала ввода числа.

При щелчке на кнопке с символом "=" выполняются следующие действия:

1. Если **LOp = true** (была задана операция), то число, записанное в окне редактирования, запоминается как второй операнд.
2. Выполняется запомненная операция (если она запомнена) с запомненными первым и вторым операндами.
3. Результат вычислений запоминается как первый операнд. Устанавливается в **true** **LBegin** — признак начала ввода числа. Устанавливается в **false** значение **LOp** (указывает, что новая операция не задана).

Приведенный алгоритм обеспечивает последовательное применение запомненной операции к результату вычисления и запомненному второму операнду при последовательном щелчке несколько раз на кнопке с символом "=".

Теперь рассмотрим реализацию рассмотренных алгоритмов. Добавьте в приложение, созданное в разд. 2.4.7, кнопки с символами "+", "–", "*", "/", "=" (рис. 2.13).

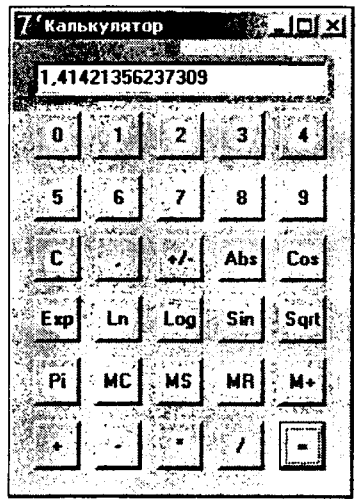


Рис. 2.13
Усовершенствованное приложение калькулятора

Ниже приведен код раздела **implementation** усовершенствованного калькулятора. Операторы, отсутствовавшие в предыдущей реализации приложения, отмечены в нем жирным шрифтом.

```
uses Math;
var M: real = 0.;
    LBegin: Boolean = true;  // фиксация начала ввода
    LPoint: Boolean = false; // фиксация ввода запятой
    LOp: Boolean = false; // фиксация щелчка на кнопке операции
    Num1, Num2: real;         // первый и второй операнды
Op: char = '0';
const SErr = 'Вы ошиблись, исправьте исходные данные!';

procedure TForm1.BClearClick(Sender: TObject);
// щелчок на кнопке очистки
begin
 Edit1.Clear;
 LBegin := true;
 Op := ' ';
 LOp := false;
end;

procedure TForm1.Button1Click(Sender: TObject);
// щелчок на кнопках с цифрами и кнопке с запятой
begin
 // Если ввод не начат, LPoint = false
 if LBegin then LPoint := false;
 {
   Если нажата кнопка с символом "0" (Button0)
   и в окне Edit1 только символ "0",
   или нажата кнопка с запятой (BPoint),
   и запятая уже вводилась (LPoint = true),
   то заносить символ в Edit1 не надо.
   Так что при этих условиях
   оператор структуры if не выполняется
```

```
 )
if not (
        ((Sender = Button0) and (Edit1.Text = '0')) or
        ((Sender = BPoint) and LPoint)
       )
 then begin
  if (Sender = BPoint)
   then begin
    LPoint := true;
    if LBegin
     then Edit1.Text := '0,'
     else Edit1.Text := Edit1.Text + ',';
   end   // Sender = BPoint
   else  // Sender <> BPoint
    if (not LBegin) and (Edit1.Text <> '0')
     then Edit1.Text := Edit1.Text +(Sender as TButton).Caption
     else Edit1.Text := (Sender as TButton).Caption;
 LBegin := false;
 end;
end;

procedure TForm1.BSignClick(Sender: TObject);
begin
 try
  Edit1.Text := FloatToStr(-(StrToFloat(Edit1.Text)));
 except
  ShowMessage(Serr);
 end;
end;

procedure TForm1.BAbsClick(Sender: TObject);
begin
 try
  Edit1.Text := FloatToStr(Abs(StrToFloat(Edit1.Text)));
  LBegin := true;
 except
  ShowMessage(Serr);
 end;
end;

...

procedure TForm1.BPiClick(Sender: TObject);
begin
 Edit1.Text := FloatToStr(Pi);
 LBegin := true;
end;

procedure TForm1.BMCClick(Sender: TObject);
begin
 M := 0.;
end;

procedure TForm1.BMSClick(Sender: TObject);
begin
 try
  M := StrToFloat(Edit1.Text);
```

```
  LBegin := true;
 except
  ShowMessage(Serr);
 end;
end;

procedure TForm1.BMRClick(Sender: TObject);
begin
 try
  Edit1.Text := FloatToStr(M);
  LBegin := true;
 except
  ShowMessage(Serr);
 end;
end;

procedure TForm1.BMPlusClick(Sender: TObject);
begin
 try
  M := M + StrToFloat(Edit1.Text);
  LBegin := true;
 except
  ShowMessage(Serr);
 end;
end;

procedure TForm1.BPlusClick(Sender: TObject);
begin
 try
  Num1 := StrToFloat(Edit1.Text);
  Op := (Sender as TButton).Caption[1];
  LBegin := true;
  LOp := true;
 except
  ShowMessage(Serr);
 end;
end;

procedure TForm1.BEqClick(Sender: TObject);
begin
 try
  if LOp then Num2 := StrToFloat(Edit1.Text);
  if Op = '+'
   then Edit1.Text := FloatToStr(Num1 + Num2)
   else if Op = '-'
    then Edit1.Text := FloatToStr(Num1 - Num2)
    else if Op = '*'
     then Edit1.Text := FloatToStr(Num1 * Num2)
     else if Op = '/'
      then Edit1.Text := FloatToStr(Num1 / Num2);
  Num1 := StrToFloat(Edit1.Text);
  LBegin := true;
  LOp := false;
 except
  ShowMessage(Serr);
 end;
end;
```

По сравнению с ранее рассмотренным вариантом в приложение добавлены три булевы переменные **LBegin**, **LPoint** и **LOp**, смысл которых пояснялся в приведенном ранее описании алгоритмов. Введены также две действительные переменные **Num1** и **Num2**, хранящие значения первого и второго операндов арифметических операций. Переменная **Op** предназначена для хранения символа операции. Она имеет тип **char** — символ (см. разд. 2.6). Начальное значение переменной задано равным пустому символу. Константа **SErr** хранит строку, которая используется во многих процедурах для информации пользователя об ошибочных данных.

Обработчик **BClearClick** щелчка на кнопке очистки отличается от предыдущего варианта только операторами задания очистки соответствующих глобальных переменных. А вот процедура **Button1Click** — совместный обработчик щелчков на кнопках с цифрами и кнопке с запятой, написана, фактически, заново.

Ее первый оператор **if** проверяет, установлен ли признак начала ввода числа **LBegin**. Если установлен, то в переменную **LPoint** заносится значение **false**, поскольку никакие символы еще не вводились, а значит, не вводился и символ запятой. Второй оператор **if** содержит сложное условие. Операция **not** отрицает условие, заключенное в скобки. А это условие проверяет те ситуации, в которых символ кнопки не должен добавляться к тексту окна **Edit1**. Он не должен добавляться, если нажата кнопка с символом "0" (**Button0**) и строка в окне **Edit1** равна "0" (не надо добавлять нули, предшествующие значащим цифрам), или если нажата кнопка с запятой (**BPoint**), а запятая уже вводилась (**LPoint = true**). В обоих случаях выражение в скобках вернет **true**, его отрицание вернет **false**, и операторы, расположенные между **begin** ... **end** в структуре **if**, выполняться не будут. Во всех остальных случаях они выполнятся. Первый из этих операторов представляет собой еще одну структуру **if**. Первый составной оператор этой структуры срабатывает, если нажата кнопка с запятой. Тогда задается **LPoint = true** — это свидетельствует о том, что запятая введена. А в окно **Edit1** заносится 0 с запятой, если ввод числа начинается (**LBegin = true**), или запятая добавляется к тексту, если ввод числа продолжается. Для всех остальных кнопок (раздел **else** структуры **if**) при продолжении числа и тексте, не равном "0", символ кнопки добавляется справа к тексту окна **Edit1**, а в иных случаях просто заносится в текст окна **Edit1**. В заключение в любых случаях задается **LBegin = false** — это свидетельствует о том, что формирование числа начато.

Процедуры **BSignClick**, **BAbsClick** и прочие пропущенные обработчики щелчков на кнопках функций, а также обработчики щелчков на кнопках, связанных с памятью, отличаются от рассмотренных в разд. 2.4.7 только обработкой возможных исключений, о которой рассказано в том же разд. 2.4.7, и заданием **LBegin = true**, т.е. подготовкой к вводу нового числа.

Процедура **BPlusClick** является обработчиком щелчков на кнопках, соответствующих арифметическим функциям. Первый оператор процедуры запоминает в переменной **Num1** значение числа, введенного в окне **Edit1**, в качестве первого операнда операции. Следующий оператор запоминает в переменной **Op** символ операции. Остальные операторы задают признак начала ввода нового числа (**LBegin**) и признак того, что задана арифметическая операция (**LOp**). Кроме того, как и в предыдущих процедурах обеспечивается перехват ошибок, которые могут возникнуть при чтении числа из окна редактирования.

Процедура **BEqClick** является обработчиком щелчка на кнопке с символом "=". Ее первый оператор запоминает число, введенное в окне **Edit1**, в качестве второго операнда операции, если **LOp = true**, т.е. если перед этим была задана арифмети-

ческая операция. Следующий многоступенчатый оператор **if** ... **then** ... **else if** ... **then** ... выполняет запомненную арифметическую операцию и заносит результат в окно **Edit1**. Следующий оператор запоминает этот результат в качестве первого операнда **Num1** последующих операций. Далее осуществляется подготовка ввода нового числа и задается **LOp = false** — это означает, что запомненная операция уже один раз использована.

Вглядитесь в приведенный код, чтобы четко представлять роль многочисленных операторов **if**, использованных в нем. Это позволит вам в дальнейшем без труда реализовывать любые сложные алгоритмы с проверкой различных условий. Выполните приложение, и убедитесь, что оно работает почти так же, как стандартная программа Windows *Калькулятор*, или как любой другой имеющийся у вас калькулятор. Некоторые отличия в работе вы сможете ликвидировать потом, когда освоите работу с текстами и с приложениями, содержащими несколько форм.

2.8.5 Условный оператор множественного выбора case

Рассмотренный в разд. 2.8.4 оператор **if** позволяет выбирать только одно из двух возможных действий в зависимости от того, выполняется или нет условие. В случаях, когда вариантов различных действий много, как в рассмотренной процедуре **BEqClick**, приходится строить многоуровневую и довольно неуклюжую последовательность операторов **if**. В подобных случаях выходом из положения является оператор **case**, который позволяет провести анализ значения некоторого выражения и в зависимости от его значения выполнять те или иные действия. В общем случае формат записи оператора **case** следующий:

```
case <выражение> of
 <список значений 1>: <оператор 1>;
  ...
 <список значений n>: <оператор n>;
else
 <оператор>
end;
```

В этой конструкции <выражение> — это математическое выражение, возвращающее результат так называемого порядкового типа. Порядковым называется тип данных, в котором значения упорядочены, и для каждого из них можно указать предшествующее и последующее значения. К порядковым типам (см. о них подробнее в разд. 2.5) относятся, целые типы, символы и ряд других. Поэтому в данном операторе нельзя, например, использовать выражения, возвращающие действительные числа или строки, так как они не относятся к порядковым типам.

Списки значений в структуре **case** могут содержать одно или несколько разделенных запятыми возможных значений константных выражений. После списка ставится двоеточие ":", а затем пишется оператор (это может быть составной оператор), который должен выполняться, если выражение приняло одно из перечисленных в списке значений. После выполнения этого оператора работа структуры **case** завершается, и управление передается следующему за этой конструкцией оператору. Обратите внимание на это обстоятельство, поскольку в некоторых языках, например, в C, аналогичная **case** структура **switch** выполняется иначе, и после выполнения оператора соответствующего раздела структуры продолжается выполнение последующих операторов, если только не использован оператор **break** языка C. В языке Pascal иначе: операторы всех последующих разделов не выполняются.

Если значение выражения не соответствует ни одному из перечисленных во всех списках, то выполняется оператор, следующий после ключевого слова **else**. Впрочем, раздел **else** не обязательно должен включаться в структуру **case**. В этом случае, если в списках не нашлось соответствующего значения выражения, то ни один оператор структуры не будет выполнен.

Списки могут содержать константы и константные выражения, которые совместимы по типу с объявленным выражением и которые компилятор может вычислить заранее, до выполнения программы. Недопустимо использование переменных и многих функций. В списках не допускается повторение одних и тех же значений, поскольку в этом случае выбор был бы неоднозначным.

В качестве примера примените оператор **case** в разработанном вами приложении калькулятора. В процедуре **BEqClick** (см. разд. 2.8.4.3) многоступенчатый оператор **if** можно заменить следующим:

```
case Op of
  '+': Edit1.Text := FloatToStr(Num1 + Num2);
  '-': Edit1.Text := FloatToStr(Num1 - Num2);
  '*': Edit1.Text := FloatToStr(Num1 * Num2);
  '/': Edit1.Text := FloatToStr(Num1 / Num2);
end;
```

Несомненно, такой оператор компактнее и, главное, нагляднее приведенного в разд. 2.8.4.3 оператора **if**.

Еще более ярко преимущества оператора **case** проявятся, если вы попробуете внести в приложение калькулятора одно усовершенствование, о котором мы пока не говорили. В этом приложении пользователь может вводить в окно редактирования числа, не пользуясь соответствующими кнопками на форме. Это удобно, так как не требует многочисленных щелчков на кнопках с цифрами. Но зато пользователь не гарантирован от случайных ошибок: он может промахнуться и нажать на клавиатуре нецифровой символ. Правда, мы во всех процедурах перехватываем подобные ошибки. Но еще лучше было бы, если бы пользователь просто не мог ввести ошибочный символ.

Контроль ввода можно осуществлять в обработчике события **OnKeyPress** окна **Edit**. Это событие наступает при нажатии пользователем любой клавиши. Выделите на форме компонент **Edit1**, перейдите в окне Инспектора Объектов на страницу событий, и сделайте двойной щелчок на белом поле около идентификатора этого события. Вы увидите в окне Редактора Кода заголовок обработчика события, который будет иметь вид:

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

Параметр **Key** этого обработчика — это тот символ, который нажал пользователь на клавиатуре. Он передан в функцию, как вы видите, со спецификатором **var**. Как вы уже знаете (см. разд. 2.7.2), это передача параметра по ссылке, и значит, вы можете его изменять. В частности, если вы не хотите, чтобы пользователь мог ввести в окно редактирования некоторый символ, вы можете заменить его нулевым символом, и в окно не будет ничего записано.

Поскольку мы хотим, чтобы пользователь не мог ввести в окно ничего, кроме цифр и символа запятой, мы можем записать в обработчике события **OnKeyPress** следующий оператор:

```
case Key of
  '0', '1', '2', '3', '4', '5',
```

```
  '6', '7', '8', '9', ',', #8: ;
else
  Key := #0;
end;
```

В этом операторе задан список символов, после которого стоит точка с запятой. Эта точка с запятой означает пустой оператор. Так что соответствующее условие словами можно расшифровать так: «Для символов из этого списка ничего делать не надо». А раздел **else** данного оператора говорит, что для всех остальных значений параметра **Key** надо заменить этот параметр нулевым символом. Так что все прочие символы в окне редактирования не отобразятся.

Список в этом операторе содержит все цифровые символы, символ запятой, и символ BackSpace, соответствующий клавише, стирающей предыдущий символ. Об этом символе будет сказано немного позднее. О форме представления подобных непечатаемых символов было сказано в разд. 2.6. Там же приведена табл. 2.4, из которой следует, что код символа BackSpace равен 8. Зачем этот символ включен в множество допустимых в приведенном операторе? Если бы этого не было сделано, нажатие пользователем клавиши BackSpace для того, чтобы стереть предыдущий символ, не воспринималось бы. А это неприятно удивило бы пользователя и сократило его возможности редактирования.

Приведенный оператор **case** можно сократить, если использовать ограниченный тип (см. разд. 2.5), содержащий все цифры:

```
case Key of
  '0'..'9', ',', #8: ;
else
  Key := #0;
end;
```

Выполните приложение вашего калькулятора с этим дополнением, и убедитесь, что пользователь не может ввести в окно редактирования запрещенные символы. А теперь сравните приведенный оператор **case** с реализацией той же логики с помощью рассмотренного в разд. 2.8.4 оператора **if**:

```
if (Key <> '0') and (Key <> '1') and (Key <> '2') and
   (Key <> '3') and (Key <> '4') and (Key <> '5') and
   (Key <> '6') and (Key <> '7') and (Key <> '8') and
   (Key <> '9') and (Key <> ',') and (Key <> #8)
then Key := #0;
```

Естественно, оператор **case** более компактен, обозрим и понятен, чем этот монстр.

Рассмотренный прием ограничения множества символов, вводимых в окна редактирования, полезен во многих случаях. Например, в разд. 1.3 было построено приложение *Charact*, содержащее окна **Edit**, в которые пользователь должен был вводить фамилию, имя и отчество. Очевидно, что допустимыми символами в этих окнах являются только символы кириллицы. Так что, вернувшись к этому приложению, вы можете написать общий обработчик событий **OnKeyPress** этих окон в виде:

```
case Key of
  'а'..'я', 'А'..'Я', 'ё', 'Ё', #8: ;
else
  Key := #0;
end;
```

Тем самым вы избавите пользователя от возможных случайных ошибок ввода латинских символов, цифр, знаков препинания и т.п. А то, почему в данном операторе отдельно выделены символы "ё" и "Ё", вы поймете, если посмотрите табл. 2.4 в разд. 2.6.

2.8.6 Применение в операторах выбора множеств и перечислимых типов

Множество — это группа элементов, которая ассоциируется с ее именем и с которой можно сравнивать другие величины, чтобы определить, принадлежат ли они этому множеству. Один и тот же элемент не может входить во множество более одного раза. Как частный случай, множество может быть пустым.

Множество определяется перечислением его элементов, заключенным в прямоугольные скобки. Такая форма определения называется *конструктором множества*. Например, если множество возможных единичных символов, которые могут быть получены в ответ на вопрос программы "Yes/No", содержит символы "y", "Y", "n" и "N", то это множество можно описать таким конструктором:

```
['y','Y','n','N']
```

Для определения, принадлежит ли переменная множеству, служит операция **in**. В предыдущем примере проверить, дал ли пользователь один из допустимых ответов, можно оператором:

```
if (Key in ['y','Y','n','N'])
     then  <оператор, выполняемый при допустимом ответе>
```

Множества могут содержать не только отдельные значения, но и ограниченные типы. Например, в разд. 2.8.5 рассматривались обработчики события **OnKeyPress**, ограничивающие множество символов, которые пользователь может вводить в окно редактирования. Приводились операторы **case**, позволяющие пользователю вводить в окно только цифры и символ запятой, если предполагается ввод числа, или только русские буквы, если предполагается, например, ввод фамилии. Используя множества, те же задачи можно решить компактнее с помощью оператора **if**:

```
// Ввод в окно только чисел
if not(Key in ['0'..'9', ',', #8]) then Key:=#0;

// Ввод в окно только русского текста, например, фамилии
if not(Key in ['а'..'я', 'А'..'Я', 'ё', 'Ё', #8]) then Key:=#0;
```

В приведенных операторах множество заранее не объявлялось в виде типа. Но если, например, в приложении в ряде мест надо проводить проверки, аналогичные приведенным выше, то целесообразнее объявить переменную или типизированную константу типа множества, или тип множества и несколько переменных этого типа. Объявление типа множества делается в форме

```
set of <базовый тип>
```

Приведем примеры.

Объявление глобальной переменной с инициализацией (в Delphi 1 это не допускается):

```
var K: set of Char=['0'..'9', ',', #8];
...
if (Key in K) then ...
```

Объявление типизированной константы:

```
const K: set of Char=['0'..'9', ',', #8];
```

Объявление типа множества и переменных этого типа:

```
type TDigit = set of '0'..'9';
var D1, D2: TDigit;
...
D1:=['0','1'];
D2:=['2'..'9'];
```

Помимо операции **in** для множеств определен еще ряд операций:

Обозначение	Операция	Типы операндов	Тип результата	Пример
+	объединение	множество	множество	`Set1 + Set2`
-	разность	множество	множество	`S - T`
*	пересечение	множество	множество	`S * T`
<=	подмножество	множество	**Boolean**	`Q <= MySet`
>=	включающее множество	множество	**Boolean**	`S1 >= S2`
=	эквивалентность	множество	**Boolean**	`S2 = MySet`
<>	неэквивалентность	множество	**Boolean**	`MySet <> S1`

В этих операциях действуют следующие правила.

Z является элементом **X + Y**, если он является элементом **X**, или **Y**, или и **X**, и **Y**.

Z является элементом **X − Y**, если он является элементом **X**, но не является элементом **Y**.

Z является элементом **X * Y**, если он является элементом и **X**, и **Y**.

Выражение **X <= Y** возвращает **true**, если каждый элемент **X** является элементом **Y**.

Выражение **X >= Y** эквивалентно выражению **Y <= X**.

Выражение **X = Y** возвращает **true**, если **X** и **Y** содержат точно одни и те же элементы. В противном случае выражение **X <> Y** возвращает **true**.

Приведем примеры. Включить элемент **Z** во множество **S** можно оператором:

```
S := S + [Z];
```

В результате во множестве **S** будет присутствовать элемент **Z**, независимо от того, был ли он там до этого.

Удалить элемент **Z** из множества **S** можно оператором:

```
S := S - [Z];
```

В результате элемент **Z** будет отсутствовать во множестве **S**, независимо от того, был ли он там до этого.

Например, подсвойство **Style** свойства **Font** является множеством, которое может содержать элементы **fsBold** — полужирный шрифт, **fsItalic** — курсив, **fsUnderline** — подчеркивание, **fsStrikeOut** — зачеркивание. Так что если во время выполнения вы хотите обеспечить, например, полужирный шрифт в окне **Memo1**, то можете выполнить оператор:

```
Memo1.Font.Style :=  Memo1.Font.Style + [fsBold];
```

А если, наоборот, хотите перейти к обычному шрифту, это делается оператором:

```
Memo1.Font.Style :=  Memo1.Font.Style - [fsBold];
```

Теперь рассмотрим еще один тип данных, часто используемый в операторах выбора — *перечислимый тип*. Перечислимые типы определяют упорядоченное множество идентификаторов, представляющих собой возможные значения переменных этого типа. Вводятся эти типы для того, чтобы сделать код более понятным. В частности, многие типы Delphi являются перечислимыми, что упрощает работу с ними, поскольку дает возможность работать не с абстрактными числами, а с осмысленными значениями.

Приведем пример, который покажет смысл введения пользователем своего перечислимого типа. Пусть, например, в программе должна быть переменная **Mode**, в которой зафиксирован один из возможных режимов работы приложения: чтение данных, их редактирование, запись данных. Можно, конечно, дать переменной **Mode** тип **Word** и присваивать этой переменной в нужные моменты времени одно из трех условных чисел: 0 — режим чтения, 1 — режим редактирования, 2 — режим записи. Тогда программа будет содержать операторы вида

```
if (Mode = 1) then ...
```

Через некоторое время уже забудется, что означает значение **Mode**, равное 1, и разбираться в таком коде будет очень сложно. А можно поступить иначе: определить переменную **Mode** как переменную перечислимого типа и обозначить ее возможные значения как **mRead**, **mEdit**, **mWrite**. Тогда приведенный выше оператор изменится следующим образом:

```
if (Mode = mEdit) then ...
```

Конечно, такой оператор понятнее, чем предыдущий.

Переменная перечислимого типа может определяться предложением вида:

```
var <имя переменной> : (<значение 1>, ..., < значение n>);
```

Например

```
var Mode : (mRead, mEdit, mWrite);
```

Если переменная определена так, то ей можно присваивать указанные значения, можно проверять ее величину, сравнивая с возможными значениями. Кроме того, надо учитывать, что перечислимые типы относятся к целым порядковым типам (см. разд. 2.5), и к ним применимы любые операции сравнения (>, < и т.п.), а также процедуры и функции, определенные для порядковых типов.

Если возможные значения типа определены, как указано выше, в переменной при ее объявлении, то нельзя будет ввести другую переменную с теми же значениями. Если такая потребность имеется, то перечислимый тип надо определять именно как тип:

```
type <имя типа> = (<значение 1>, ..., < значение n>);
```

Тогда можно ввести в программе несколько переменных этого типа. Например:

```
type TMode = (mRead, mEdit, mWrite);
var Mode1, Mode2 : TMode;
```

2.8.7 Операторы циклов

2.8.7.1 Предварительные сведения о массивах

Очень часто в программах присутствуют фрагменты, которые надо многократно повторять, что-то меняя в условиях их операторов. Обычно это связано с перебором каких-то символов в строках, каких-то объектов и т.п. Чаще всего подобные циклические вычисления связаны с обработкой массивов. *Массив* представляет собой структуру, позволяющую хранить под одним именем совокупность данных какого-то типа. Массивы — очень важные элементы любого языка программирования. Они будут подробно рассмотрены в разд. 3.1. Но некоторые предварительные сведения о массивах надо дать сейчас, так как без них трудно будет рассматривать операторы циклов и некоторые другие вопросы.

Массив характеризуется своим именем, типом хранимых элементов, размером (числом хранимых элементов), нумерацией элементов и размерностью.

Объявление переменной как одномерного массива имеет вид:

```
var <имя массива>: array <ограниченный тип> of <тип элементов>;
```

Например:

```
var A: array [1..10] of integer;
```

объявляет массив с именем **A**, содержащий 10 целых чисел. Доступ к элементам этого массива осуществляется выражением **A[i]**, где **i** — *индекс*, являющийся в данном примере, как видно из объявления, целым числом в диапазоне 1÷10. Например, **A[1]** — значение первого элемента, **A[2]** — второго, **A[10]** — последнего.

Элементы массива могут иметь любой тип. Например, предложение

```
var S: array [0..10] of char;
```

определяет массив символов. Впрочем, любая текстовая строка, например, хорошо знакомое вам свойство **Text** окон **Edit** тоже является массивом символов. Например, **Edit1.Text[1]** — первый (левый) символ текста в окне **Edit1**, **Edit1.Text[2]** — второй символ и т.д. Число символов в таком массиве может быть определено функцией **Length**. Например, выражение **Length(Edit1.Text)** вернет число символов, записанных в окне **Edit1**.

Можно объявлять и многомерные массивы, т.е. массивы, элементами которых являются массивы. Например, двумерный массив можно объявить таким образом:

```
Var A2: array[1..10,1..3] of integer;
```

Этот оператор описывает двумерный массив, который можно представить себе как таблицу, состоящую из 10 строк и 3 столбцов.

Доступ к значениям элементов многомерного массива обеспечивается через индексы, перечисляемые через запятую. Например, **A2[4,3]** — значение элемента, лежащего на пересечении четвертой строки и третьего столбца.

Впоследствии в разд. 3.1 мы рассмотрим подробнее различные варианты массивов. Но пока нам хватит и этих кратких сведений.

2.8.7.2 Оператор цикла for

Циклическое выполнение некоторой группы операторов можно, конечно, организовать уже рассмотренными ранее операторами, например, операторами **if** и **goto**:

```
var i, n: integer;
label 1;
...
n := 10;
i := 1;
1:
<группа повторяющихся операторов>
Inc(i);
if (i < n) then goto 1;
```

Группа операторов, расположенная в этом фрагменте между меткой 1 (см. о метках в разд. 2.8.2) и вызовом процедуры **Inc** (см. разд. 2.5), будет выполняться **n** раз. Но поскольку необходимость в подобных фрагментах встречается очень часто, а излишнее применение оператора **goto** не приветствуется, в языке имеется несколько специальных операторов, облегчающих построение циклов.

Один из них — оператор **for**, обеспечивающий циклическое повторение некоторого оператора (в частности, составного оператора) заданное число раз. Повторяемый оператор называется *телом цикла*. Повторение цикла определяется некоторой *управляющей переменной (счетчиком)*, которая увеличивается или уменьшается на единицу при каждом выполнении тела цикла. Повторение завершается, когда управляющая переменная достигает заданного значения.

Оператор **for** записывается в одной из следующих форм:

```
for <счетчик>:=<начальное значение> to <конечное значение>
   do <оператор>;
```

или

```
for <счетчик>:=<начальное значение> downto <конечное значение>
   do <оператор>;
```

<счетчик> — локальная управляющая переменная порядкового типа (см. разд. 2.5). В начале выполнения оператора **for** ей присваивается <начальное значение>. После каждого очередного выполнения тела цикла <оператор> ее значение увеличивается (в первой форме с **to**) или уменьшается (во второй форме с **downto**) на единицу. Когда значение управляющей переменной достигает значения <конечное значение>, тело цикла выполняется последний раз, после чего управление передается оператору, следующему за структурой **for**. <начальное значение> и <конечное значение> являются выражениями, совместимыми по типу с управляющей переменной.

Если заданные начальное и конечное значения равны друг другу, тело цикла выполняется только один раз. Если в форме с **to** начальное значение больше конечного или в форме с **downto** начальное значение меньше конечного, то тело цикла не выполняется ни разу.

Внутри цикла значение управляющей переменной может использоваться в выражениях. Однако после окончания выполнения структуры **for** значение управляющей переменной не определено.

Приведем примеры использования оператора **for**. Рассмотрим следующие операторы:

```
var Data: array[1..50] of real;
    I: integer;
...
for I := 1 to 50 do
```

```
 Data[I] := Random;

Memo1.Clear;
for I := 1 to 50 do
 Memo1.Lines.Add(FloatToStr(Data[I]));
```

Они объявляют массив действительных чисел **Data**, содержащий 50 элементов, и управляющую переменную цикла **I**. В первом цикле **for** массив заполняется случайными действительными числами, лежащими в диапазоне от 0 до 1. Для этого используется библиотечная функция **Random**, которая генерирует подобные числа (подробнее об этой функции рассказано в разд. 3.1.5.2). Второй цикл **for** отображает значения элементов массива в окне **Memo1**.

Конечно, оба цикла **for** можно было бы объединить в один:

```
for I := 1 to 50 do
 begin
  Data[I] := Random;
  Memo1.Lines.Add(FloatToStr(Data[I]));
 end;
```

Экспериментируя с этим кодом, попробуйте два варианта: объявление переменной **I** как локальной, т.е. внутри обработчика щелчка на кнопке, и объявление ее как глобальной переменной вне обработчика. В последнем случае вы получите предупреждение компилятора: «For loop control variable must be simple local variable» — «В качестве управляющие переменной цикла for надо использовать простую локальную переменную». В принципе, можно, конечно, проигнорировать это предупреждение: приложение откомпилируется, и будет работать правильно. Но использование локальной управляющей переменной позволяет компилятору создать более эффективный код. Кроме того, глобальная управляющая переменная, используемая в различных иногда вложенных друг в друга функциях, может стать причиной трудно отлавливаемых ошибок. Так что лучше прислушаться к рекомендации компилятора и использовать в качестве управляющих локальные переменные.

Следующие операторы вычисляют максимальное значение элементов, расположенных в массиве **Data**:

```
var ...
    AMax: real;
...

AMax:=Data[1];
for I := 2 to 50 do
  if Data[I] > AMax then AMax := Data[I];
```

Сначала переменной **AMax** присваивается значение первого элемента массива **Data**. А затем в цикле просматриваются остальные элементы массива, и если очередной элемент больше **AMax**, то значение **AMax** заменяется значением этого элемента. В результате **AMax** оказывается равным максимальному значению в массиве.

Следующий пример — функция, осуществляющая шифровку и дешифровку строки текста. Шифровка осуществляется поразрядным сложением по операции **xor** (исключающее ИЛИ — см. разд. 2.4.10) каждого символа строки **S** с ключом

Key. Повторное применение той же функции к зашифрованной строке **S** при том же значении ключа осуществит дешифровку строки.

```
function CodeDecode(S: string; Key: word): string;
var i, L: word;
begin
L := Length(S);
for i:=1 to L do
  S[i]:= Chr(Ord(S[i]) xor Key);
 Result := S;
end;
```

В этом коде, помимо рассмотренной в разд. 2.4.10 способности операции **xor** изменять, а при повторном применении — восстанавливать значение своего операнда, применяются функции **Ord** (см. разд. 2.5) и **Chr** (см. разд. 2.6). Сначала очередной символ строки **S[i]** переводится в соответствующее ему целое число функцией **Ord**. Затем к этому числу и ключу **Key** применяется операция **xor**. Результат переводится функцией **Chr** в символ, который заносится в строку.

Обратите внимание на локальную переменную **L**, в которую заносится функцией **Length** длина строки. Можно было бы обойтись без нее, записав оператор **for** в виде:

```
for i:=1 to Length(S) do
...
```

В чем недостаток подобного варианта? Конечное значение, которое в данном случае определяется функцией **Length**, будет вычисляться при каждом проходе цикла. А ведь оно постоянно, так что затраты времени на многократное вычисление одного и того же значения бессмысленны. Следите в своих программах за подобными ситуациями и приучайтесь думать о повышении эффективности ваших кодов.

Хороший стиль программирования

При организации циклов следите за тем, чтобы ни в операторе цикла, ни в теле цикла не содержалось вычислений, которые в каждом цикле дают одинаковый результат. Все подобные вычисления, особенно, связанные с вызовами функций, должны быть выполнены до начала циклической части кода, и их результаты надо присвоить локальным переменным, используемым в циклах. Это может существенно повысить эффективность вашей программы.

Создайте приложение, содержащее кнопку и окно редактирования. Запишите в коде модуля приведенную выше функцию. А в обработчик щелчка на кнопке внесите оператор:

```
Edit1.Text := CodeDecode(Edit1.Text, 10);
```

Выполните приложение, и убедитесь, что оно шифрует и дешифрует (при повторном щелчке на кнопке) строки текста.

Приведенный код обеспечивает простейший вариант шифрования с ключом в виде одного целого числа. Несложно (попробуйте) написать программу, которая перебирала бы все возможные значения ключа (их всего 255), и отображала бы результаты применения каждого ключа к зашифрованной строке, например, в окне **Memo**. Проанализировав эти результаты, можно найти среди них осмысленный, и, следовательно, подобрать ключ. Более надежные алгоритмы шифровки использу-

ют ключи, состоящие из нескольких цифр, которые циклически применяются к символам текста. Уже при пяти числах количество возможных сочетаний достигает одного триллиона, так что описанный выше подбор ключей оказывается мало реальным. Желающие могут попробовать написать подобную программу шифровки и дешифровки.

Следующие операторы содержат три вложенных цикла **for**, осуществляющих вычисление матрицы **Mat**, равной произведению двух квадратных матриц **Mat1** и **Mat2** размером **M** на **M**. Все матрицы представлены двумерными массивами (см. разд. 2.8.7.1). Формула для вычисления:

$$Mat[I,J] = \sum_{K=1}^{M} Mat1[I,K] \cdot Mat2[K,J].$$

```
for I := 1 to M do
  for J := 1 to M do
  begin
    X := 0;
    for K := 1 to M do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
end;
```

2.8.7.3 Оператор цикла repeat

Структура **repeat...until** используется для организации циклического выполнения совокупности операторов, называемой телом цикла, до тех пор, пока не выполнится некоторое условие. Синтаксис управляющей структуры **repeat...until**:

```
repeat
  <операторы тела цикла>
until <условие>;
```

Точка с запятой после последнего оператора тела цикла (перед ключевым словом **until**) может пропускаться.

Структура работает следующим образом. Выполняются операторы тела цикла. Затем вычисляется <условие>, которое должно возвращать результат булева типа. Если выражение условия возвращает **false**, то повторяется выполнение операторов тела цикла, и после этого снова вычисляется условие. Такое циклическое повторение цикла продолжается до тех пор, пока проверяемое условие не вернет **true**. После этого цикл завершается и управление передается оператору, следующему за структурой **repeat...until**.

Поскольку проверка условия осуществляется после выполнения операторов тела цикла, то эти операторы заведомо будут выполнены хотя бы один раз, даже если условие сразу истинно. С другой стороны, программист должен быть уверен, что условие рано или поздно вернет **true**. Если этого не произойдет, то программа «зациклится», т.е. цикл будет выполняться бесконечно. Иногда такие бесконечные циклы используются. Но в этом случае внутри тела цикла должно быть предусмотрено его прерывание в какой-то момент (см. разд. 2.8.7.6), например, оператором **break**, прерывающим цикл, или функциями **Exit** или **Abort**, вызывающими прерывание не только цикла, но и функции или процедуры, внутри которой выполняется данный цикл.

Обычно оператор **repeat** целесообразно использовать для организации поиска среди множества объектов такого, который обладает каким-то определенным свойством. Причем заранее должно быть известно, что множество объектов не пустое, т.е. хотя бы один объект в нем имеется. К тому же должен быть критерий, позволяющий проверить, не является ли текущий объект последним. Тогда тело цикла включает операторы перехода к новому объекту и какой-то его обработки, а условие **until** включает проверку, является ли объект последним или текущий объект обладает искомым свойством. И в том, и в другом случае выполнение цикла прерывается. Если же ни одно из этих условий не выполнено, осуществляется переход к следующему объекту.

Если множество проверяемых объектов может быть пустым, следует использовать другой оператор цикла — **while**, который будет рассмотрен в разд. 2.8.7.5). А если число повторений циклов заранее известно, лучше применять рассмотренный в разд. 2.8.7.2 оператор **for**.

Повторим с использованием цикла **repeat** пример кодировки текста, рассмотренный в разд. 2.8.7.2:

```
S := Edit1.Text;
i := 0;
repeat
 Inc(i);
 S[i]:= Chr(Ord(S[i]) xor Key);
until i = length(S);
Edit1.Text := S;
```

В этом примере перед началом цикла переменной **i** присваивается значение 0. В теле цикла это значение при каждом проходе увеличивается на 1 функцией **Inc**. А условие в предложении **until** прерывает цикл, когда преобразован последний символ.

Сравнение с вариантом, рассмотренным в разд. 2.8.7.2, подтверждает данную выше рекомендацию: если число повторений циклов заранее известно, лучше применять оператор **for**. Код при использовании цикла **repeat** получился более громоздким. Но важнее даже другое: при определенных условиях он может давать ошибки. Сотрите текст в окне **Edit1** и примените этот код. Вы получите сообщение об ошибке. Дело в том, что тело цикла в операторе **repeat** повторяется не менее одного раза. А если текст в окне **Edit1** отсутствует, то при первом же проходе следует обращение к символу **S[1]**, которого в строке нет. Это и вызывает сообщение об ошибке во время выполнения.

А теперь рассмотрим пример, в котором число требуемых проходов цикла заранее неизвестно и применение оператора **repeat** оправдано. Пусть нам требуется найти первое число Фибоначчи, превышающее заданное значение M. Числа Фибоначчи определены так: нулевое число равно 0, первое равно 1, а каждое последующее равно сумме двух предыдущих. Такие числа используются, в частности, в методе оптимизации, который вы, возможно, знаете.

Ниже приведен код обработчика щелчка на кнопке, определяющий первое число Фибоначчи, превышающее значение, указанное пользователем в окне **Edit1**.

```
procedure TForm1.Button1Click(Sender: TObject);
var A1, A2, A, M: integer;
begin
 M := StrToInt(Edit1.Text);
 A1 := 0;
```

```
 A2 := 1;
 repeat
  A := A1 + A2;
  A1 := A2;
  A2 := A;
 until A > M;
 Label1.Caption := 'Найдено число ' + IntToStr(A);
end;
```

В цикле определяется очередное число **A** как сумма двух предыдущих чисел **A1** и **A2**. Затем предыдущее число **A2** запоминается как **A1**, а новое число запоминается как **A2**. Тем самым подготавливается расчет следующего числа Фибоначчи. Число необходимых проходов цикла в данном случае неизвестно заранее, так что оператор **for** применить было бы невозможно.

2.8.7.4 Решение уравнений методом дихотомии, создание собственной библиотеки функций

В данном примере структура **repeat...until** используется при реализации одного из методов решения нелинейных уравнений — метода *дихотомии*. Он применяется для решения уравнений вида

```
f(x) = 0.
```

Функция f(x) предполагается непрерывной. Корень ищется с заданной погрешностью по аргументу на заданном интервале [a, b] изменения x. Предполагается, что на интервале [a, b] имеется один и только один корень. Назовем для дальнейшего изложения интервал, на котором имеется один и только один корень, интервалом неопределенности. Очевидно, что знаки функции f(x) на концах интервала неопределенности различны.

Суть метода дихотомии сводится к тому, чтобы уменьшить начальный интервал неопределенности [a, b] до значения, меньшего заданной величины погрешности по аргументу. В начале поиска рассчитывается значение функции **f** на одном из концов начального интервала неопределенности, например, f(a). Дальнейший процесс строится итерационно. На каждой итерации точка $\mathbf{x_т}$ текущего эксперимента задается в середине интервала [a, b]: $x_т = (a + b) / 2$. Вычисляется значение $f(x_т)$. Если оказывается (см. рис. 2.14), что знаки функций в точках **a** и $\mathbf{x_т}$ совпадают (случай 1 на рис. 2.14), то очевидно, что корень лежит правее $\mathbf{x_т}$. Это следует из того, что на границах интервала неопределенности функция должна иметь разные знаки. Значит, для дальнейшего рассмотрения можно оставить интервал $[x_т, b]$. Иначе говоря, на следующей итерации в качестве текущего значения нижней границы интервала **a** можно взять значение $\mathbf{x_т}$.

Если оказывается, что знаки функций в точках **a** и $\mathbf{x_т}$ различны (случай 2 на рис. 2.14), то очевидно, что корень лежит левее $\mathbf{x_т}$, так как функция уже перешла через нуль. В этом случае для дальнейшего рассмотрения можно оставить интервал $[a, x_т]$. Иначе говоря, на следующей итерации в качестве текущего значения верхней границы интервала **b** можно взять значение $\mathbf{x_т}$.

Таким образом, в результате одного эксперимента в точке $\mathbf{x_т}$ в любом случае интервал неопределенности сокращается в 2 раза. На следующей итерации очередной эксперимент опять проводится в середине оставшегося интервала неопределенности. В результате анализа результатов эксперимента соответственно изменяется одна из границ **a** или **b** и интервал сокращается еще в два раза. Этот процесс про-

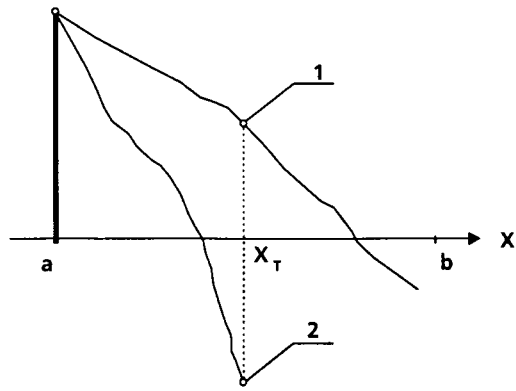


Рис. 2.14
Метод дихотомии

должается до тех пор, пока интервал неопределенности не уменьшится до величины, определяемой заданными допустимыми погрешностями. Если полагать, что очередное выбранное значение $\mathbf{x_т}$ является приближенным значением корня, то погрешность этого приближения не превышает половины интервала неопределенности. Таким образом, вычисления можно заканчивать, когда значение (x – a) станет не больше заданной погрешности.

Постройте приложение, реализующее этот метод. Поместите на форме кнопку, метку и три окна редактирования **Edit**. Пусть в первом из них пользователь вводит нижнюю, а во втором — верхнюю границы начального интервала неопределенности. В третьем окне пусть вводится допустимая погрешность по аргументу. По щелчку на кнопке приложение должно найти с заданной точностью корень уравнения в заданном интервале неопределенности, и выдать результат в метку. Вычисление левой части уравнения мы оформим как функцию, код которой можно будет менять, изменяя тем самым заданное уравнение. Для тестирования метода в приведенном ниже тексте реализуется уравнение вида: X * exp(X) – 10 = 0. Но можете, конечно, записать любое другое уравнение. Корень этого уравнения лежит в пределах от 0 до 10, так что такие значения целесообразно задать по умолчанию в соответствующих окнах **Edit**.

Ниже приведен вариант кода этого приложения.

```
function F(X: real): real;
// Функция левой части уравнения
begin
 Result := X * exp(X) - 10.;
end;

procedure TForm1.Button1Click(Sender: TObject);
// Поиск методом дихотомии корня уравнения X * exp(X)- 10 = 0
var a,  // нижняя граница интервала неопределенности
    b,  // верхняя граница интервала неопределенности
    Dx, // удвоенная допустимая погрешность
    Fa, // значение функции в начале интервала неопределенности
    X,  // текущее приближение корня
    Fx: // значение функции в текущей точке
      real;
begin
```

```
 // исходные данные для расчета
 a := StrToFloat(Edit1.Text);
 b := StrToFloat(Edit2.Text);
 Dx := 2. * StrToFloat(Edit3.Text);
 Fa := F(a);
 // цикл вычислений
 repeat
  // новый эксперимент и функция в новой точке
  X := (a + b) / 2;
  Fx := F(X);
  // изменение границ интервала неопределенности
  if (Fa * Fx > 0)
   then a := X
   else b := X;
   // проверка критерия окончания
 until (b - a <= DX);
 // отображение результата
 Label1.Caption := 'Корень равен ' + FloatToStr((a + b) / 2);
end;
```

Думаю, что подробных комментариев в этом тексте хватит, чтобы все понять. В теле цикла **repeat...until** делается очередной эксперимент в середине текущего интервала неопределенности, и по его результатам корректируются границы интервала. А условие **until** обеспечивает повторение цикла до тех пор, пока не будет достигнута заданная точность.

В данном примере использован оператор **repeat**, поскольку число экспериментов заранее определить трудно, и, значит, трудно использовать оператор **for**. Впрочем, в рассмотренной постановке задачи число требуемых проходов цикла все-таки можно рассчитать. Каждый проход уменьшает интервал неопределенности вдвое. Значит через n проходов интервал неопределенности станет равен $(b - a) / 2^n$. И этот интервал должен быть меньше, чем удвоенная допустимая погрешность. Так что число необходимых проходов цикла можно заранее рассчитать выражением: **Ceil(Log2((b − a) / DX))**. Обратите внимание, что для округления тут надо использовать именно функцию **Ceil** (см. эту функцию и функцию **Log2** в разд. 2.4.6), округляющую число до ближайшего целого в сторону увеличения.

Хотя число проходов цикла в методе дихотомии определить можно, и значит можно использовать оператор **for**, применение оператора **repeat** в данном случае проще и более естественно. А если немного изменить постановку задачи и устранить один из недостатков приведенного алгоритма, то возможность использования оператора **for** вообще отпадет. Речь идет о том, что рассмотренная постановка задачи предполагает задание только абсолютной допустимой погрешности по аргументу. А обычно в подобных задачах поиска корней или оптимизации надо задавать и абсолютную, и относительную погрешности. Дело в том, что мы заранее не знаем, какой величины будет корень. Может быть, он окажется очень небольшим, и тогда абсолютную погрешность надо задавать, исходя из этого минимально ожидаемого значения. Иначе корень будет найден очень приближенно. Но если корень окажется большим, то при малой абсолютной погрешности он будет считаться неоправданно точно. Кроме излишних затрат времени на поиск, возможны и более серьезные проблемы: абсолютная погрешность может оказаться меньше неизбежных вычислительных погрешностей, связанных с конечной разрядной сеткой и ошибками округления. Так что ограничиваться заданием только абсолютной погрешности в общем случае нельзя. Точно так же нельзя ограничиваться и заданием только относительной погрешно-

сти, так как при малых значениях корня, близких к нулю, любая вычислительная погрешность будет соответствовать бесконечной относительной погрешности. Так что следует задавать обе допустимые погрешности, а критерий окончания алгоритма должен ориентироваться на максимальную из них. Если обозначить заданную относительную погрешность через dxrel, а абсолютную через dxabs, то критерий окончания расчета должен выглядеть так: $(b - a) / 2 < \max\{dxabs, dxrel \cdot |b + a| / 2\}$. Второе выражение в скобках — это погрешность, рассчитанная из относительной в предположении, что модуль корня равен $|b + a| / 2$. Как видно из приведенного выражения, при больших текущих значениях **a** и **b** будет учитываться относительная погрешность, а при малых — абсолютная.

Попробуйте самостоятельно реализовать подобный модифицированный алгоритм. И обратите внимание на то, что эта модификация делает невозможной использование цикла **for**, так как число проходов цикла, зависящее от текущих значений границ интервала неопределенности, заранее неизвестно.

Созданное вами приложение может быть полезным при решении различных задач, связанных с поиском корня уравнения. Так что давайте на этом примере научимся грамотно оформлять разрабатываемые алгоритмы. Хотелось бы иметь реализацию метода дихотомии в виде функции, не привязанной к конкретным компонентам и конкретной функции левой части уравнения. Тогда подобную функцию можно было бы включать в различные приложения. Очевидно, что аргументами такой функции должны быть границы начального интервала неопределенности, допустимая погрешность и функция, рассчитывающая левую часть уравнения. Все это может быть оформлено следующим образом:

```
type TFunc = function(X: real): real;

function F(X: real): real;
begin
 Result := X * exp(X) - 10.;
end;

function Dihot(a, b, Dx: real; Func: TFunc): real;
var X, Fa, Fx: real;
begin
 Dx := 2. * Dx;
 Fa := Func(a);
 repeat
  X := (a + b) / 2;
  Fx := Func(X);
  if (Fa*Fx > 0)
  then a := X
  else b := X;
 until (b - a <= DX);
 Result := (a + b) / 2.;
end;
```

Предложение **type** объявляет тип функции, принимающей один действительный аргумент и возвращающей действительно значение. Функция **F**, рассчитывающая в нашем примере левую часть уравнения, является функцией такого типа. А функция **Dihot** принимает в качестве параметров границы начального интервала неопределенности, допустимую погрешность и указатель на функцию, рассчиты-

вающую левую часть уравнения. Вызов метода дихотомии может состоять всего из одного оператора:

```
Label1.Caption := 'Корень равен ' +
                  FloatToStr(Dihot(StrToFloat(Edit1.Text),
                                   StrToFloat(Edit2.Text),
                                   StrToFloat(Edit3.Text), F));
```

Если имеется несколько функций, рассчитывающих левые части уравнений, легко переключаться межу решениями различных уравнений, передавая в **Dihot** имена соответствующих функций. Сделайте приложение, в котором было бы предусмотрено решение нескольких различных уравнений, и пользователь мог бы выбирать необходимое ему уравнение, щелкая на той или иной кнопке.

Итак, вы создали полезную функцию, которую могли бы использовать в различных проектах. Например, в разд. 2.13.2 предлагается решить ряд задач, использующих метод дихотомии. Да и в последующих главах этот метод нам еще понадобится. И вашим друзьям и коллегам такая функция может пригодиться. Можно, конечно, в каждый проект, использующий метод дихотомии, копировать текст разработанной вами функции. Но согласитесь, что это не очень красивый вариант. А более существенно другое. Я надеюсь, что вы усовершенствуете эту функцию (некоторые предложения в этом направлении изложены в разд. 2.13.2). И что тогда делать? Изменять коды во всех использующих ее приложениях?

Разумное решение — начать формировать собственную библиотеку функций, в которую и внести вашего первенца. Такая библиотека может состоять из одного или нескольких модулей **unit**, содержащих ваши функции, процедуры, объявления типов, констант и т.д.

Создать такой модуль можно следующим образом. Выполните команду File | New | Unit (в некоторых более старых версиях Delphi — File | New и на странице New выберите пиктограмму Unit). В окно Редактора Кода загрузится заготовка модуля вида:

```
unit Unit2;

interface

implementation

end.
```

На имя модуля не обращайте пока внимания — оно изменится автоматически при сохранении модуля в файле. Раздел **interface**, как вы уже знаете (см. разд. 2.3.2) является открытым интерфейсом модуля. В нашем случае в него, очевидно, надо занести объявление типа **TFunc** и объявление функции **Dihot**. Снабдите объявления подробным комментарием, чтобы и вам, и другим пользователям было понятно назначение объявляемых величин и аргументов функции. А в раздел **implementation** внесите реализацию функции. Сохраните файл в каком-то каталоге (команда File | Save As). Разумно создать отдельный каталог, например, *MyLib*, для хранения ваших библиотечных модулей. Укажите при сохранении модуля имя файла, например, *MyLib1*. Остальные файлы проекта в каталоге *MyLib* сохранять не надо, чтобы не загромождать библиотеку ненужными файлами.

Текст модуля после сохранения может иметь следующий вид:

```
unit MyLib1;

interface
// Тип функции, описывающей левую часть уравнения,
// в методе дихотомии (в функции Dihot)
type TFunc = function(X: real): real;

// Метод дихотомии для решения уравнения вида f(x) = 0
// a, b - границы начального интервала неопределенности (b > a)
// Dx - допустимая абсолютная погрешность по аргументу
// Func - функция, описывающая левую часть уравнения
// Dihot возвращает найденное значение корня
function Dihot(a, b, Dx: real; Func: TFunc): real;

implementation

function Dihot(a, b, Dx: real; Func: TFunc): real;
// Метод дихотомии для решения уравнения вида f(x) = 0
// a, b - границы начального интервала неопределенности (b > a)
// Dx - допустимая абсолютная погрешность по аргументу
// Func - функция, описывающая левую часть уравнения
// Dihot возвращает найденное значение корня
var X, Fa, Fx: real;
begin
 Dx := 2. * Dx;
 Fa := Func(a);
 repeat
  X := (a + b) / 2;
  Fx := Func(X);
  if (Fa*Fx > 0)
  then a := X
  else b := X;
 until (b - a <= DX);
 Result := (a + b) / 2.;
end;

end.
```

Проведите компиляцию приложения. В каталоге *MyLib* при этом будет создан файл *MyLib1.dcu*. Комбинация файлов *MyLib1.pas* и *MyLib1.dcu* является полным описанием вашего библиотечного модуля.

Теперь посмотрим, как вы или другие пользователи могут использовать библиотечные файлы. Можете опробовать это на приложении, аналогичном рассмотренному ранее и содержащем кнопку, три окна редактирования, метку и единственный оператор в обработчике щелчка на кнопке. Плюс к этому, в приложении должна быть реализована функция, описывающая левую часть уравнения. Для того чтобы эксперимент был чистым, создайте такое приложение заново (это вам теперь очень легко). Только не вводите в него описание функции **Dihot** и объявление типа **TFunc**, так как все это должно браться теперь из библиотеки. Так что код приложения может быть таким:

```
function F(X: real): real;
begin
 Result := X * exp(X) - 10.;
```

```
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
Label1.Caption := 'Корень равен ' +
                  FloatToStr(Dihot(StrToFloat(Edit1.Text),
                                   StrToFloat(Edit2.Text),
                                   StrToFloat(Edit3.Text), F));
end;
```

Если вы попытаетесь компилировать это приложение, то получите сообщение об ошибке, так как компилятор не поймет функцию **Dihot**. Это естественно, так как вы еще не подключили свою библиотеку. Подключить ваш модуль можно несколькими способами. Один из них (наименее удачный) — выполнить команду Project | Add to Project или нажать соответствующую быструю кнопку (см. разд. 1.6) и в открывшемся диалоге добавить в проект файл *MyLib1*. Но этого будет мало, так как надо еще сослаться из основного модуля программы *Unit1* на этот файл. Для этого можно вручную включить в *Unit1* оператор:

```
uses MyLib1;
```

То же самое можно сделать, перейдя в окне Редактора Кода в модуль и выполнив команду File | Use Unit. Откроется диалоговое окно, в котором вы должны указать, что хотите подключить модуль *MyLib1*. Тогда приведенный выше оператор **uses** включится в код автоматически.

Я назвал этот вариант подключения библиотеки не очень удачным, так как включать в проект библиотечный файл не требуется и опасно: вы или какой-то другой пользователь может его случайно испортить, что-то стереть в нем или изменить. Если он сохранит этот испорченный файл, то тем самым испортит библиотеку.

Более правильное подключение библиотеки — включить ссылку на *MyLib1* в предложение **uses**, имеющееся в самом начале модуля вашего проекта. Но чтобы это сработало, надо помочь компилятору отыскать ваш библиотечный файл. Иначе будет получено сообщение, что файл не найден. Компилятор может отыскать файл в двух случаях: если он находится в каталоге проекта, или если он находится в одном из каталогов, которые указаны как библиотечные. Можно использовать любой из этих вариантов. Первый, конечно, менее удобный. Для его реализации надо скопировать средствами Windows файл *MyLib1.dcu* в каталог, где вы сохранили проект. Неудобен этот вариант тем, что надо помнить, где лежит этот файл, чтобы его скопировать, и тем, что, не сохранив проект, вы не сможете его выполнить. К тому же, дополнительные копии файла в ряде проектов будут занимать на диске лишнее место. А главное — если вы усовершенствуете свою функцию, то проекты, в каталогах которых созданы копии файлов, этого не почувствуют. Так что единственно правильный вариант подключения своей библиотеки — указать компилятору, где ее искать.

Прежде всего, вы можете скопировать файлы *MyLib1.pas* и *MyLib1.dcu* в каталог *Lib* вашей версии Delphi. Это каталог, в котором компилятор ищет включаемые в проект файлы. Тогда ни у вас, ни у других пользователей вашего компьютера никаких проблем в использовании вашей библиотеки не будет. Если вы в дальнейшем что-то измените в реализации вашей функции (изменять интерфейс вы не имеете права, можете его только пополнять) и перенесете новые модули в каталог *Lib*, то все проекты при их перекомпиляции автоматически воспримут эти изменения. Возможен и иной путь — указать компилятору ваш каталог *MyLib* как биб-

лиотечный. Для этого надо выполнить команду Tools | Environmet Options и перейти в открывшемся диалоговом окне на страницу Library. На ней надо нажать кнопку с многоточием около окошка Library Path. В открывшемся при этом диалоговом окне надо опять нажать кнопку с многоточием и в стандартном диалоге Windows указать папку вашей библиотеки *MyLib*. Вы вернетесь в предыдущий диалог, в котором станет доступна кнопка Add — добавить. Щелкните на этой кнопке, а затем щелчками на кнопках OK завершите работу со всеми открытыми диалоговыми окнами. Теперь компилятор Delphi будет искать включаемые файлы и в каталоге вашей библиотеки, так что никакие библиотечные файлы копировать не придется.

В заключение посмотрим, какие файлы надо перенести, если вы хотите передать свою библиотеку на другой компьютер другим пользователям. Как минимум, вы должны передать файлы *.dcu* ваших библиотечных модулей. Это обеспечит нормальную работу проекта, если они будут помещены в каталоги, доступные компилятору. Но при отладке пользователь не сможет в пошаговом режиме (см. разд. 2.4.5) войти внутрь ваших функций, и не сможет увидеть даже интерфейс вашего модуля. Правда, зато и не сможет что-то испортить в библиотеке или воспользоваться вашими блестящими идеями по реализации алгоритмов. Если же вы перенесете на другой компьютер и файлы *.pas* ваших модулей, то перенесенная библиотека будет полноценной.

2.8.7.5 Оператор цикла while

Структура **while...do** используется для организации циклического выполнения оператора, называемого телом цикла, пока выполняется некоторое условие. Синтаксис управляющей структуры **while...do**:

```
while <условие> do <оператор>;
```

Структура работает следующим образом. Сначала вычисляется <условие>, которое должно возвращать результат булева типа. Если условие возвращает **true**, то выполняется оператор тела цикла, после чего опять вычисляется выражение, определяющее условие. Такое циклическое повторение выполнения оператора и проверки условия продолжается до тех пор, пока условие не вернет **false**. После этого цикл завершается, и управление передается оператору, следующему за структурой **while...do**.

Поскольку проверка выражения осуществляется перед выполнением оператора тела цикла, то, если условие сразу ложно, оператор не будет выполнен ни одного раза. В этом основное отличие структуры **while...do** от структуры **repeat...until** (см. разд. 2.8.7.3), в котором тело цикла заведомо выполняется хотя бы один раз.

Программист должен быть уверен, что условие рано или поздно вернет **false**. Если этого не произойдет, то программа «зациклится», т.е. цикл будет выполняться бесконечно. Иногда такие бесконечные циклы используются. Но в этом случае внутри тела цикла должно быть предусмотрено его прерывание в какой-то момент.

Обычно оператор **while** целесообразно использовать для организации поиска среди множества объектов того объекта, который обладает каким-то определенным свойством. Причем не исключается, что множество объектов может быть пустым, т.е. не содержащим ни одного объекта. К тому же должен быть критерий, позволяющий проверить, не является ли текущий объект последним. Тогда тело цикла включает операторы перехода к новому объекту и какой-то его обработки, а условие **while** включает проверку, является ли объект не последним и отсутствует ли

в нем искомое свойство. Если одно из этих условий нарушается (объект последний или имеет искомое свойство), выполнение цикла прерывается.

Если множество проверяемых объектов не может быть пустым, то можно использовать уже рассмотренный оператор цикла — **repeat**. Если число повторений циклов заранее известно, лучше применять оператор **for**.

Ниже приведен фрагмент кода решения уравнения методом дихотомии, рассмотренного в разд. 2.8.7.4. Но в данном случае цикл реализован структурой **while...do**, а не **repeat...until**.

```
// цикл вычислений
while (b - a > DX) // проверка критерия продолжения
 do begin
  // новый эксперимент и функция в новой точке
  X := (a + b) / 2;
  Fx := F(X);
  // изменение границ интервала неопределенности
  if (Fa * Fx > 0)
   then a := X
   else b := X;
 end;
```

Если сравнить этот код с приведенным в разд. 2.8.7.4, можно отметить следующие различия. В структуре **while...do** записывается условие продолжения цикла (в нашем примере **b – a > DX**), а в структуре **repeat...until** — условие окончания цикла (в нашем примере **b – a <= DX**). Но главное отличие: цикл **while** может не выполняться ни одного раза, если условие продолжения сразу нарушено. Так если в нашем примере пользователь задаст начальный интервал неопределенности меньший, чем удвоенная допустимая погрешность, то программа сразу выдаст результат без единого прохода цикла. А в цикле **repeat** в этом случае будут сделаны лишние вычисления. Так что в данном случае применение **while** более оправдано.

2.8.7.6 Прерывание цикла: оператор break, процедуры Continue, Exit и Abort

В некоторых случаях желательно прервать повторение цикла, проанализировав какие-то условия внутри него. Это может потребоваться в тех случаях, когда проверки условия окончания цикла громоздкие, требуют многоэтапного сравнения и сопоставления каких-то данных и все эти проверки просто невозможно разместить в выражении условия операторов **for**, **repeat** или **while**.

Один из возможных вариантов решения этой задачи — ввести в код какой-то флаг окончания (переменную). При выполнении всех условий окончания этой переменной присваивается некоторое условное значение. Тогда условие в операторах **for**, **repeat** или **while** сводится к проверке, не равно ли значение этого флага принятому условному значению. Это может выглядеть примерно так:

```
var Lend: boolean;
...
Lend := false;
repeat
  ...
if <какое-то условие> then Lend := true;
if <какое-то другое условие> then Lend := true;
  ...
until Lend;
```

Другой способ решения задачи — использование оператора **break**. Этот оператор прерывает выполнение тела любого цикла **for**, **repeat** или **while** и передает управление следующему за циклом выполняемому оператору. Организация подобного цикла может иметь такой вид:

```
repeat
  ...
  if <какое-то условие> then break;
  <операторы, которые не выполняются при последнем проходе>
until false;
```

или такой:

```
while true
do begin
  ...
  if <какое-то условие> then break;
  <операторы, которые не выполняются при последнем проходе>
end;
```

В обоих примерах реализованы бесконечные циклы, так как условия **until** и **while** не обеспечивают их прерывание. Так что выход из циклов осуществляется только оператором **break**. Одним из преимуществ подобного выхода из цикла является возможность выйти из середины тела цикла, а не только из его начала или конца. Это часто позволяет избежать излишних вычислений, которые приходится делать, если в последний проход выполняются все операторы тела цикла, хотя уже ясно, что они не нужны.

Например, в разд. 2.8.7.3 приводился цикл поиска первого числа Фибоначчи, превышающего заданную величину **M**. Этот цикл лучше было бы организовать так:

```
repeat
 A := A1 + A2;
 if A > M then break;
 A1 := A2;
 A2 := A;
until false;
```

Тогда на последнем проходе тела цикла не срабатывают два его последних оператора, которые подготавливают данные для следующего прохода. Конечно, в данном примере повышение эффективности кода ничтожное. Но ведь бывают задачи, когда последние операторы цикла обращаются к каким-то очень сложным процедурам, требующим большого времени я расчета. Тогда разумная организация цикла становится очень важной.

Еще один способ прерывания, по своим свойствам похожий на использование оператора **break** — применение оператора **goto**, передающего управление какому-то оператору, расположенному вне тела цикла.

Для прерывания циклов, размещенных в процедурах или функциях, можно воспользоваться процедурой **exit**. В отличие от оператора **break**, процедура **exit** прервет не только выполнение цикла, но и выполнение той процедуры или функции, в которой расположен цикл.

Прервать выполнение цикла, а заодно — и блока, в котором расположен цикл, можно также процедурой **Abort**. Она генерирует так называемое «молчаливое» исключение, не связанное с каким-то сообщением об ошибке.

Описанные способы прерывали выполнение цикла. Имеется еще процедура **continue**, которая прерывает выполнение только текущего прохода тела цикла

и передает управление на следующую итерацию. Схемы организации подобных циклов следующие:

```
repeat
  ...
  if <условие> then continue;
  <операторы выполняются, только если условие возвращает false>
until ...;

while ...
do begin
  ...
  if <условие> then continue;
  <операторы выполняются, только если условие возвращает false>
end;
```

2.9 Рекурсия

Рекурсивная функция — это функция, которая вызывает сама себя либо непосредственно, либо косвенно с помощью другой функции. В некоторых задачах применение подобных функций обеспечивает компактное и красивое решение. Но для этого сама постановка задачи должна допускать рекурсию.

Рекурсивная задача в общем случае разбивается на ряд этапов. Для решения задачи вызывается рекурсивная функция. Эта функция знает, как решать только простейшую, так называемую *базовую задачу*. Если эта функция вызывается для решения базовой задачи, она просто возвращает результат. Если функция вызывается для решения более сложной задачи, она делит эту задачу на две части: одну часть, которую функция умеет решать, и другую, которую функция решать не умеет. Эта другая часть должна быть подобна исходной, но несколько проще ее. Для ее решения функция вызывает новую копию самой себя. Такая последовательность вызовов должна в итоге за конечное число шагов сойтись к базовой задаче, которую функция умеет решать. Тогда функция решает ее, возвращает результат предыдущей копии функции, и последовательность возвратов повторяет весь путь назад, пока не дойдет до первоначального вызова и не возвратит конечный результат.

Классический пример рекурсивной задачи — расчет факториала. Формулу для расчета факториала можно представить в виде:

```
n! = 1 · 2 · 3 · … · n.
```

Эта форма представления соответствует циклическому итерационному расчету. Ее программная реализация в виде функции может выглядеть так:

```
function factor1(n: integer): Int64;
var i: integer;
begin
 Result := 1;
 for i := 2 to n do
  Result := Result * i;
end;
```

Обратите внимание на то, что тип функции задан **Int64**. Факториал очень быстро увеличивается с ростом **n**. Так что даже такой тип данных, обеспечивающий максимально допустимое значение 9223372036854775807, позволит вам считать не

более, чем 20!. А уже 21! будет посчитан неверно (см. разд. 2.4.8). Выходом из положения может быть объявление типа функции **real**. Тогда вы сможете считать факториалы больших чисел. Но учтите, что разрядная сетка и в этом случае всего 8 байт. Так что вычисления будут приближенными с точностью до 15–16 десятичных разрядов. Если задать тип **Extended**, число достоверных разрядов возрастет до 19–20.

Вызов рассмотренной функции может иметь вид:

```
Label1.Caption := IntToStr(factor1(StrToInt(Edit1.Text)));
```

В этом операторе предполагается, что число n вводится из окна **Edit1**, а результат заносится в метку **Label1**. Сделайте такое приложение, убедитесь, что оно работает правильно до n = 20. Убедитесь также, что уже при n = 21 будет выдан неверный, отрицательный результат. Посмотрите, как ведет себя аналогичная функция, тип которой объявлен как **real**.

Теперь посмотрим, как организовать тот же расчет рекурсивно, Выражение для факториала можно записать в виде:

```
n! = (n-1)! · n.
```

Тут факториал числа n сводится к вычислению факториала числа n – 1, т.е. к более просто задаче. Такая рекуррентная формула, понижающая размер задачи, обеспечивает возможность рекурсии. Только надо дополнить ее решением базовой, простейшей задачи: 0! = 1, 1! = 1.

Ниже приведена рекурсивная функция, соответствующая рассмотренной рекурсивной задаче:

```
function factor2(n: integer): Int64;
begin
 if(n < 2)
  then Result := 1
  else Result := n * factor2(n - 1);
end;
```

Вызов такой функции может иметь вид:

```
Label1.Caption := IntToStr(factor2(StrToInt(Edit1.Text)));
```

Можете проверить, что рекурсия даст то же самое решение, что и рассмотренная ранее циклическая реализация.

Как видите, одинакового результата можно доиться двумя совершенно разными путями: итерационным и рекурсивным алгоритмами. Можно показать, что любая задача, допускающая рекурсивное решение, может быть решена и итерационно. Эти два подхода идут к решению как бы с разных сторон. Итерационный алгоритм идет от простой задачи к сложной. А рекурсивный алгоритм идет в обратном направлении: от сложной к простой, а затем возвращается к сложной.

В приведенном примере сложность реализации итерационного и рекурсивного решений примерно одинаковая. Но учтите, что рекурсивное решение создает в памяти n копий функции. Так что при n = 20 затраты памяти для рекурсии примерно в 20 раз больше, чем при итерационном решении. К тому же, каждый вызов каждой копии функции требует затрат времени. Так что эффективность рекурсивного решения ниже, чем итерационного.

В данном примере потерю эффективности заметить практически невозможно, так как оба варианта работают очень быстро. Приведем теперь другой пример: расчет числа Фибоначчи. Подобная задача уже решалась в разд. 2.8.7.3, но в несколько

иной формулировке. Напомню, что числа Фибоначчи определены так: первые 2 числа равны 1, а каждое последующее равно сумме двух предыдущих. Иначе говоря, n-ое число равно $F_n = F_{n-1} + F_{n-2}$. Эта формула так и просится на рекурсию, так как делит задачу на две задачи меньшего размера. А решение базовой задачи рекурсии очевидно: $F_0 = 0$, $F_1 = 1$. Но начнем мы с итерационного решения. Мы уже рассматривали его в разд. 2.8.7.3. Реализация его в виде функции может иметь вид:

```
function fib1(n: integer): Int64;
var A1, A2: Int64;
    i: integer;
begin
 A1 := 0;
 A2 := 1;
 if n = 0
  then Result := 0
  else Result := 1;
 for i := 2 to n do
 begin
  Result := A1 + A2;
  A1 := A2;
  A2 := Result;
 end;
end;
```

При **n** = 0 результат выдается равным 0, и цикл не выполняется ни разу. При **n** = 1 результат выдается равным 1, и цикл тоже не выполняется. А при иных значениях **n** цикл выполняется **n** − 1 раз. При каждом проходе результат делается равным сумме предыдущих чисел **A1** (F_{i-2}) и **A2** (F_{i-1}), затем в **A1** запоминается значение **A2**, а в **A2** — значение очередного рассчитанного числа Фибоначчи.

Теперь рассмотрим аналогичную рекурсивную функцию:

```
function fib2(n: integer): Int64;
begin
 if n < 2
  then Result := n
  else Result := fib2(n-1) + fib2(n-2);
end;
```

При **n** < 2 она возвращает значение **n**: 0 или 1. А при других значениях **n** функция дважды рекурсивно вызывает саму себя.

Создайте приложение, тестирующее эти функции. Оно должно содержать окно **Edit**, в котором можно задавать число n, две метки и две кнопки. Запишите в модуле функции двух рассмотренных вариантов расчета чисел Фибоначчи, в обработчик щелчка на одной кнопке введите оператор:

```
Label1.Caption := IntToStr(fib1(StrToInt(Edit1.Text)));
```

а в другой — оператор:

```
Label2.Caption := IntToStr(fib2(StrToInt(Edit1.Text)));
```

Для небольших чисел Фибоначчи обе функции будут работать внешне одинаково. Но уже при n = 35 − 40 вы заметите существенную разницу. Итерационная функция по-прежнему будет срабатывать практически мгновенно. А результатов рекурсивной функции вам придется ждать достаточно долго. Это легко объяснимо. Каждое выполнение функции, кроме последнего, вызывает рекурсивно две копии

функции, каждая из них в свою очередь вызывает две копии и т.д. Так что число рекурсивных вызовов при больших n равно примерно 2^n. В подобных случаях говорят, что метод обладает *экспоненциальной сложностью*. Уже при n = 30 число вызовов порядка миллиарда, при n = 40 — порядка триллиона и т.п. Немудрено, что задержки вычислений видны на глаз. Учтите, что и затраты памяти растут соответственно, так как все копии функции хранятся в памяти. А для итерационной функции затраты памяти постоянны, а время (число проходов цикла) пропорционально n. Это означает, что итерационный метод в данном случае обладает *линейной сложностью*. Так что, несмотря на компактность и красоту приведенной рекурсивной функции, с точки зрения эффективности она не выдерживает никакой критики при больших n.

Рекурсию очень любят авторы учебников по программированию из-за красоты получающихся решений. Но программисты вполне обоснованно относятся к ней настороженно и избегают ее, если в конкретной задаче она обладает экспоненциальной сложностью. Тем не менее, есть задачи, где рекурсия очень естественна и эффективна. Простой пример — поиск файлов в указанном каталоге и во всех вложенных в него каталогах. С помощью рекурсии эта задача решается легко и является примером, когда рекурсивный подход естественнее, проще и эффективнее, чем итерационный. Так что в каждом конкретном случае надо искать наиболее простой и эффективный алгоритм.

Рассмотрение зависимости объема вычислений от размера задачи позволяет оценить теоретическую сложность того или иного алгоритма. Теперь посмотрим, как можно измерить ее экспериментально. Для этого надо научиться измерять интервал времени межу началом и окончанием расчета.

Введите в обработчики щелчков на кнопках приложения, тестирующего расчет чисел Фибоначчи, следующие переменные:

```
var // Переменные для измерения времени расчета
    tbeg, tend: TDateTime;
    Hour, Min, Sec, MSec: word;
```

Первые две переменные **tbeg** и **tend** будут использоваться для запоминания времени начала и конца расчета. Они имеют тип **TDateTime**. Этот тип используется в Delphi для хранения дат и времени. Значение типа **TDateTime** представляет собой действительное число, целая часть которого содержит количество дней, отсчитанное от некоторого начала календаря, а дробная часть равна части 24-часового дня, т.е. характеризует время и не относится к дате. Для 32-разрядных версий Delphi за начало календаря принята дата 00 часов 30 декабря 1899 года. В Delphi 1 за начало отсчета принят год 1, т.е. для перевода даты Delphi 1 в дату последующих версий Delphi надо вычесть из даты число 693594.

Прибавление к значению типа **TDateTime** целого числа **D** равносильно увеличению даты на **D** дней. Разность двух значений типа **TDateTime** дает разность двух моментов времени.

Переменные **Hour**, **Min**, **Sec** и **MSec** мы будем использовать для выделения соответственно часов, минут, секунд и миллисекунд.

Схема измерения интервал времени, необходимого для вычислений, следующая:

```
// Запоминание начала вычислений
tbeg := Time;

< Операторы, время выполнения которых надо измерить >
```

```
// Запоминание конца вычислений
tend := Time;
// Разбиение интервала времени на часы, минуты и т.д.
DecodeTime(tend-tbeg, Hour, Min, Sec, MSec);
// Отображение интервала времени в миллисекундах
Label2.Caption := IntToStr(
   Msec + Sec * 1000 + Min * 60000 + Hour * 3600000) + ' мсек';
```

Первый оператор запоминает в переменной **tbeg** текущий момент времени. Для этого вызывается функция **Time**, которая возвращает текущее время. После первого оператора размещается фрагмент кода, время выполнения которого надо измерить. В нашем примере это оператор, содержащий вызов функции **fib1** или **fib2**. После окончания вычислений в исследуемом фрагменте, в переменной **tend** запоминается текущий момент времени. Далее к разности запомненных моментов времени **tend-tbeg** применяется функция **DecodeTime**, которая выделяет из времени, переданного в нее первым аргументом, часы **Hour**, минуты **Min**, секунда **Sec** и миллисекунды **MSec**. Последний оператор кода отображает в метке значение интервала времени в миллисекундах.

Следует отметить несколько особенностей приведенного кода. Прежде всего, надо учесть, что Windows — многозадачная система. Во время решения вашей задачи она отвлекается на различные фоновые операции, т.е. проводит какую-то работу, хотя вы считаете, что решается только ваша задача. За счет этого результаты повторных измерений интервалов времени могут различаться, по крайней мере, на десяток миллисекунд. Чтобы это не сильно сказывалось на результатах, надо, чтобы измеряемый временной интервал составлял не менее сотен миллисекунд. В нашем примере рекурсивной функции при n = 30 – 40 это условие соблюдается. Но для итерационной функции вы вообще можете получить результирующее время равным нулю, т.е. меньше одной миллисекунды. В подобных случаях, а они чаще всего встречаются на практике, надо включать фрагмент кода, время выполнения которого рассматривается, в цикл, и выполнять его не один, а тысячу, миллион, миллиард раз — столько, сколько надо для получения результирующего интервала не менее сотен миллисекунд. Только, конечно, не забудьте разделить полученный результат на число использованных циклов. И не включайте в цикл посторонние операции. Например, вывод результата в метку или в окно редактирования и чтение из окна редактирования — длинные операции, не имеющие отношения к исследуемому методу. Так что эти накладные расходы должны быть вынесены за пределы тела цикла.

Рассмотренный прием позволяет определить время расчета при заданной сложности задачи. Чтобы определить зависимость времени от сложности, надо в цикле изменять сложность задачи (в нашем примере — n) и выдавать результаты, например, в виде строк в окно **Memo**. Конечно, нагляднее выводить их на график. Но это вы сможете сделать много позднее — после изучения возможностей графики в Delphi.

2.10 Указатели

Указатель является величиной, указывающей на некоторый адрес в памяти, где хранятся какие-то данные. Указатели бывают типизированные, указывающие на данные определенного типа, и нетипизированные (типа **pointer**), которые могут указывать на данные произвольного типа. Чаще всего указатели используются для ра-

боты с объектами в динамически распределяемой области памяти (см. разд. 2.11), особенно при работе с записями (см. разд. 3.3.2).

Объявление типизированного указателя на любой тип имеет вид:

```
type <имя типа указателя> = ^<тип данных>;
```

Например, предложения:

```
type Pint = ^integer;
var P1, P2: Pint;
```

объявляют тип **Pint** указателя на величину типа **integer** и две переменные **P1** и **P2**, являющиеся указателями на значения типа **integer**. Однако надо понимать, что объявление переменных **P1** и **P2** не создает самих величин, на которые они указывают. Выделяется только память под хранение указателей, но сами эти указатели ни на что не указывают. Имеется предопределенная константа **nil**, которая обычно присваивается указателям, которые в данный момент ни на что не указывают. Полезно задавать это значение в качестве начального при объявлении указателей. Это предотвратит в дальнейшем возможность ошибок, связанных с использованием указателей, которые указывают неизвестно куда. Так что приведенное выше объявление указателей **P1** и **P2** лучше оформить так:

```
var P1: Pint = nil;
    P2: Pint = nil;
```

Тогда перед использованием указателя, если нет уверенности, что ему присвоено нужное значение, можно осуществлять проверку:

```
if (P1 <> nil) then ...
```

Хороший стиль программирования

Инициируйте всегда указатели при их объявлении значением nil. А перед их использованием, если нет уверенности, что им присвоены нужные значения, осуществляйте проверку их неравенства значению nil.

Чтобы получить доступ к данным, на которые указывает типизированный указатель, надо применить операцию его *разыменования*. Она записывается с помощью символа ^, помещаемого после указателя. Например, если переменная **P1** является указателем приведенного выше типа **Pint**, то выражение **P1^** — это та целая величина, на которую указывает указатель **P1**. Если **I** — переменная целого типа, то после выполнения оператора

```
P1^ := I;
```

P1 начнет указывать на переменную **I**, и выражение **P1^** будет возвращать значение этой переменной. Того же результата можно добиться операцией адресации. Например, приведенный выше оператор можно заменить эквивалентным ему оператором

```
P1 := @I;
```

Этот оператор присваивает указателю **P1** адрес переменной **I**.

Операция разыменования не применима к указателям типа **pointer**. Чтобы разыменовать указатель **pointer**, надо сначала привести его к другому типу указателя (см. о приведении типов в разд. 2.4.2). Например, если указатель **P** объявлен как

указатель типа **pointer**, то выражение **Pint(P)** приведет его тип к объявленному выше типу **Pint**, после чего его можно будет разыменовывать. Таким образом, с помощью операции приведения типа можно записать такие операторы:

```
Pint(P):=P1;
I := Pint(P)^ + P2^;
```

Указатели широко используются в Object Pascal и Delphi, причем часто неявно для пользователя. Например, передача параметров по ссылке в процедуры и функции осуществляется именно через указатели. В дальнейшем мы не раз будем работать с указателями (например, см. разд. 3.3.3).

2.11 Динамическое распределение памяти

Динамическое распределение памяти широко используется для экономии вычислительных ресурсов. Те переменные или объекты, которые становятся ненужными, уничтожаются, а освобожденное место используется для новых переменных или объектов. Это особенно эффективно в задачах, в которых число необходимых объектов зависит от обрабатываемых данных или от действий пользователя, т.е. заранее неизвестно. В этих ситуациях остается только два выхода: заранее с запасом отвести место под множество объектов, или использовать динамическое распределение памяти, создавая новые объекты по мере надобности. Первый путь, конечно, неудовлетворительный, поскольку связан с излишними затратами памяти и в то же время накладывает на размерность задачи необоснованные ограничения.

Для динамического распределения выделяется специальная область памяти — heap (куча). Динамическое распределение памяти в этой области может производиться двумя различными способами: с помощью процедур **New** и **Dispose** и процедурами **GetMem** и **FreeMem**.

При первом способе выделение памяти производится процедурой

```
procedure New(<имя указателя>);
```

где <имя указателя> — имя переменной, являющейся типизированным указателем (см. разд. 2.10). Этой переменной при успешном завершении процедуры передается адрес начала выделенной области памяти. Размер выделяемой области определяется размером памяти, необходимым для размещения того типа данных, который указан при объявлении указателя.

Рассмотрим примеры динамического выделения памяти. Операторы

```
var P: ^real;
...
New(P);
P^:= 5.5;
```

объявляют переменную **P**, являющуюся указателем на действительное значение. Процедура **New** выделяет память для этого значения. А следующий оператор заносит в эту область число 5.5.

Освобождение памяти, динамически выделенной процедурой **New**, осуществляется процедурой **Dispose**:

```
procedure Dispose(<имя указателя>);
```

В эту процедуру должен быть передан тот указатель, в котором хранится адрес области памяти, выделенной ранее процедурой **New**. Для приведенного выше примера соответствующая процедура **Dispose** имеет вид:

```
Dispose(P);
```

Надо отметить, что применение процедуры **Dispose** освобождает память, но не изменяет значения указателя, не делает его равным **nil**, хотя теперь указатель не указывает ни на что конкретное. Так что обычно при освобождении памяти желательно явным образом присваивать указателям значение **nil**:

```
P := nil;
```

Второй способ динамического выделения памяти связан с применением процедур **GetMem** для выделения памяти и **FreeMem** для ее освобождения. Они имеют следующий синтаксис:

```
procedure GetMem(<имя указателя>,<объем памяти в байтах>);
procedure FreeMem(<имя указателя>,<объем памяти в байтах>);
```

В отличие от процедур **New** и **Dispose** здесь задается не только указатель, в котором устанавливается процедурой **GetMem** адрес выделенной области памяти, но и указывается объем памяти в байтах. Благодаря этому в процедурах могут использоваться не только типизированные, но и нетипизированные указатели (см. разд. 2.10). Если же используется типизированный указатель, то объем необходимой памяти лучше всего определять функцией **SizeOf**, так как размеры памяти, отводимой под тот или иной тип данных, могут изменяться в различных версиях компилятора. В качестве аргумента в функцию **SizeOf** передается тип или имя переменной соответствующего типа. Таким образом, в приведенном выше примере выделение памяти может осуществляться так:

```
GetMem(P, SizeOf(real);
...
FreeMem(P, SizeOf(real);
...
```

Надо иметь в виду, что два рассмотренных метода нельзя смешивать. Например, нельзя освободить методом **FreeMem** память, выделенную ранее методом **New**, и нельзя освободить методом **Dispose** память, выделенную методом **GetMem**.

Вместо функции **GetMem** может использоваться функция **AllocMem**:

```
AllocMem(Size: Cardinal): Pointer
```

Она динамически выделяет область памяти размером **Size** байтов и возвращает указатель на выделенную область. Эта область в дальнейшем, как и в случае функции **GetMem**, может быть освобождена процедурой **FreeMem**.

При выделении памяти возможна генерация исключения, связанного с отсутствием достаточных резервов памяти. В Delphi 1 для предотвращения такой ситуации можно перед выделением памяти вызвать функцию **MaxAvail**, возвращающую максимальный свободный объем динамически распределяемой памяти, и сравнить возвращенное значение с требуемым вам объемом. В старших версиях Delphi эта функция недоступна, и для обработки случаев нехватки памяти надо использовать перехват и обработку исключения **EOutOfMemory** (см. разд. 2.4.8).

2.12 Некоторые итоги

В данной главе вы изучили огромный массив материала. Фактически, вы освоили основную часть языка Object Pascal. Осталось изучить не так уж много — только наиболее сложные его разделы. Правда, именно эти разделы, которые мы рассмотрим в следующей главе, позволяют называть язык Object Pascal. А то, что вы уже изучили — это, скорее, язык Pascal. Впрочем, вы уже готовы писать весьма сложные и эффективные программы в рамках этого языка.

Помимо изучения языка, вы освоились с рядом типичных приемов программирования, включая такие сложные, как рекурсия. Так что, надеюсь, при дальнейшем изучении материала данной книги вы не будете испытывать особых сложностей, разбираясь в многочисленных примерах программ.

Перечислять все, что вы изучили, не имеет смысла. Но мне хочется остановиться на некоторых моментах, которые я пытался подчеркивать в своем изложении.

Прежде всего, обратите внимание на необходимость писать программы так, чтобы они были понятны и вам, и другим разработчикам даже через большой отрезок времени. Каждая реальная программа нуждается в *сопровождении*. Какую бы прекрасную программу вы ни написали и как бы ни были довольны ее пользователи, через некоторое время захочется ее как-то улучшить, добавить новые возможности, а может и устранить какие-то ошибки. Не обязательно модернизацией программы будете заниматься именно вы. Это могут быть другие люди. И представьте себе, сколько нелестных слов они выскажут в ваш адрес, если программа написана непонятно и плохо документирована. Кстати, в данной книге мы еще не раз вернемся практически к каждому приложению, созданному вами при изучении этой главы. Вы могли это видеть и в данной главе, когда в разных разделах мы вводили какие-то усовершенствования в прежние приложения. А когда вам надо будет вернуться к приложению через несколько глав, чтобы добавить в него, например, графику, работу с базой данных, обработку исключений, вы почувствуете на собственном опыте, что значит сопровождение программы.

Хороший стиль программирования

Грамотно оформленная программа, доступная для сопровождения, должна удовлетворять, по крайней мере, следующим требованиям:

1. Модули программы, если их несколько, должны иметь осмысленные имена и располагаться в одном каталоге.
2. Если на форме размещено несколько компонентов одного типа, каждому из них надо давать осмысленное имя, поясняющее его назначение.
3. Каждая функция и процедура должна иметь комментарий, поясняющий ее назначение и смысл ее параметров. Комментариями должны также сопровождаться объявления переменных, констант, функций.
4. Фрагменты кода, имеющие законченный смысл, должны сопровождаться комментариями, поясняющими осуществляемые ими операции.
5. Надо четко выделять отступами структуры циклов, условные структуры if, case, составные операторы, заключенные в операторные скобки begin ... end. Но этого мало, так как часто в коде имеется множество вложенных составных операторов, и в окне Редактора Кода не помещается одновременно начало и конец каждого оператора. В этих случаях совершенно необходимо обозначать каким-то одинаковым комментарием соответствующие друг другу

строки begin и end, чтобы было ясно, где можно что-то добавлять в тот или иной оператор.

6. В коде не должно содержаться никаких числовых констант, которые может потребоваться изменять в будущем. Все подобные константы должны быть оформлены именованными константами, снабженными комментарием. Тогда приложение будет масштабируемым, и его легко будет сопровождать.

Изложенные выше требования — минимальные. Многое в этом отношении будет добавлено в последующих главах.

Еще один момент, на который следует обращать внимание при разработке приложения — достоверность и точность вычислений. Среди пользователей, далеких от программирования, широко распространено безусловное доверие к машинным расчетам. Вероятно, вы не раз слыхали фразу: «Это рассчитала машина», за которой лежит подтекст: «Это абсолютная истина». Грешат таким подходом и начинающие программисты. В действительность, к сожалению, все обстоит не так. При разработке приложения надо серьезно задумываться о выборе типов различных параметров и переменных с учетом их ожидаемых значений. В разд. 2.4.8 и 2.9 вы могли видеть примеры, в которых компьютер выдает совершенно неверные значения. В подобных случаях обязательно надо включать в программы соответствующие проверки и предупреждать пользователя о недопустимых исходных данных. В примерах данной главы это не делалось, чтобы подобные детали не заслоняли собой основную идею алгоритмов. Но в серьезных приложениях такие проверки совершенно необходимы. Впрочем, даже если все сделано аккуратно, учтите, что за счет ошибок округления в результате проведения множества арифметических операций, точность получаемого результата редко превышает 5–6 десятичных разрядов.

Важнейшее условие разработки программы — продумывание для нее *системы тестов*. Тесты должны предусматривать самые различные сочетания исходных данных, самые фантастические ошибки пользователя. Во всех ситуациях программа должна работать нормально, или подсказывать пользователю, что он не так сделал. Необходимо продумывать так называемую «защиту от дурака», т.е. защиту от любых неправомерных действий пользователя. Лозунг: «Пользователь всегда прав» должен быть девизом любого программиста. Если пользователь часто ошибается, работая с вашей программой, значит, вы плохо спроектировали пользовательский интерфейс, и надо совершенствовать свою программу.

И последнее (а иногда и первое) — всегда думайте об эффективности алгоритмов и программ. Особенно это касается циклов. Из тела цикла, которое может проходиться множество раз, надо выносить все вычисления, не изменяющиеся от одного прохода цикла до другого. Все подобные расчеты надо выполнить один раз до начала циклической части кода и запомнить их результаты в переменных. Впрочем, повторных расчетов надо избегать в любой части кода, а не только в циклах.

Серьезно задуматься об эффективности надо, если вы решили применить рекурсию. В этом случае обязательно надо оценить сложность алгоритма и убедиться, что она линейная, а не экспоненциальная.

В заключение несколько советов по разработке и отладке алгоритмов.

Совет

Серьезные алгоритмические задачи целесообразно решать технологией нисходящей разработки с пошаговой детализацией.

Только очень простенькие алгоритмы можно создавать сразу на компьютере. Более серьезные задачи надо начинать на бумаге. Обычно целесообразно использовать технологию так называемой *нисходящей разработки алгоритма с пошаговой детализацией*. Сначала записываются основные этапы и разветвления алгоритма, без какой-либо детализации. При этом могут использоваться различные формы записи: в виде блок-схемы, или в виде последовательности шагов, написанной на псевдокоде. Но заведомо не стоит сразу записывать коды реального языка программирования, так как при этом возникнет множество второстепенных деталей, за которыми трудно будет уследить основную логику работы.

Запись последовательности шагов на псевдокоде может иметь примерно такой вид:

Шаг 1. Чтение введенных пользователем данных ... (указывается, что именно надо ввести).

Шаг 2. Проверка допустимости данных. Если неверно ..., то переход на шаг ... .

Шаг 3. Вычисление ... . Если ..., то переход на шаг ... .

Шаг 4. Цикл, пока выполняется условие ... .

Шаг 5. Если ..., то задать ..., иначе переход на шаг ... .

Шаг 6. ...

Подобная запись позволяет четко проследить логику работы и выявить, в частности, наиболее часто встречающиеся ошибки, связанные с тем, что при проходах по каким-то ветвям алгоритма некоторые данные могут отсутствовать (какие-то переменные не инициализируются).

После того как выверена основная схема алгоритма, можно начинать наполнять ее деталями — именами переменных, реализациями циклов (тоже на псевдокоде) и т.д. Эта постепенная детализация алгоритма и есть нисходящая разработка. На каждом этапе полезно проигрывать работу алгоритма так, как это будет впоследствии делать компьютер. Выпишите в виде заголовков таблицы имена основных переменных программы, поставьте сначала вместо их значений, например, символы "?", а потом проходите по шагам вашего алгоритма, изменяя соответствующим образом значения переменных. Подобная имитация работы программы позволит вам, кроме проверки правильности работы, получить тестовые результаты, которые впоследствии вы сможете сравнить с результатами работы приложения на компьютере. И позволит устранить возможные ошибки, связанные с отсутствием инициализации переменных в каких-то ветвях программы. Такие ошибки наиболее сложно отлавливать в окончательном коде, так как они вызывают при выполнении приложения *перемежающиеся ошибки* времени выполнения. Результат работы приложения при этом зависит от случайного состояния памяти. Сейчас приложение может сработать правильно, а через пару часов или на следующий день начнет выдавать неверные результаты. Или на вашем компьютере может работать правильно, а когда вы передали приложение пользователю на другой компьютер, оно откажется работать.

Только после завершения подобной бумажной подготовительной работы можно садиться за компьютер и реализовать алгоритм программно. И не смотрите с зави-

стью на тех «крутых» программистов, которые пренебрегают бумажной работой и сразу пишут программу. Может быть, они и сами себе кажутся крутыми. Но в действительности это не более чем начинающие программисты, никогда не создававшие сложных программ и не набившие на этом должное количество шишек. Со временем они поймут, что так серьезные программы не создаются. Но чтобы минимизировать в будущем затраты собственного времени и сил, и избавить себя от многих неприятностей, приучайтесь сразу работать грамотно. Не надо только доводить это до абсурда и использовать нисходящее проектирование для задач, вся реализация которых укладывается в несколько операторов.

И еще один очень важный совет:

Совет

Овладейте всем инструментарием отладки, имеющимся в ИСР Delphi, и всегда проводите тщательную отладку всех ветвей вашего алгоритма, даже если вы выполнили только что созданное приложение, и оно показало правильный результат.

Правильный результат выполнения приложения не является гарантией отсутствия ошибок. Не так уж редко получается, что в коде имеется несколько ошибок, которые, накладываясь друг на друга, дают при определенном сочетании исходных данных правильный результат. Кроме того, любой серьезный алгоритм имеет разветвления, и в каждом конкретном тесте срабатывают только некоторые ветви программы. Так что надо тщательно, по шагам пройти все ветви, задавая такие сочетания исходных данных, чтобы они срабатывали. Только убедившись в правильной работе каждого оператора, вы можете быть уверены в безошибочности программы. И то лишь с некоторой вероятностью, так как часто в тестах трудно предусмотреть все мыслимые сочетания исходных данных. А если вы не пользовались в полной мере данным ранее советом о нисходящей разработке и ручном прогоне алгоритма на бумаге, то в приложении могут остаться рассмотренные выше перемежающиеся ошибки, со всеми вытекающими отсюда печальными последствиями.

Иногда просмотр промежуточных результатов в окне наблюдения Delphi не дает полного представления о работе сложной программы. К тому же, нередко результаты отладки надо запомнить, чтобы потом, не спеша проглядеть и найти ошибки. В этих случаях полезно включать в приложение отладочную печать, которая помогает следить за ходом процесса. В Delphi для этого, на мой взгляд, удобен компонент **Memo**. В него можно заносить отладочные данные и тут же, во время выполнения приложения просматривать их. А затем можно скопировать их в буфер обмена, и запомнить с помощью любого текстового редактора в файле для дальнейшего анализа. Чтобы компонент **Memo** не занимал место на форме, его можно выводить в отдельное окно. Приведу без особых комментариев, как это можно сделать. Многое в приведенном коде, наверное, будет пока не понятно, так как соответствующие возможности приложений Delphi мы еще не рассматривали. Но чтобы не возвращаться в дальнейшем к проблемам отладки, приведу этот код. Вы можете его использовать в работе, а детали его реализации поймете позже.

Введите в программу глобальные переменные:

```
var FDebug:TForm;   // форма отладочного окна
    Memo1:TMemo;    // компонент вывода в отладочном окне
    LDebug:Word=0; // уровень отладки
```

Задайте обработчик события **OnCreate** формы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 SetExceptionMask([exInvalidOp, exDenormalized, exZeroDivide,
                   exOverflow, exUnderflow, exPrecision]);
 // установка уровня отладки; можно закомментировать
 LDebug := 1;
 if LDebug > 0 then begin
  FDebug := TForm.Create(Self);
  FDebug.Caption := 'Отладка';
  Memo1 := TMemo.Create(FDebug);
  Memo1.Parent := FDebug;
  Memo1.ScrollBars := ssVertical;
  Memo1.Align := alClient;
  FDebug.Show;
 end;
end;
```

Задайте обработчик события **OnDestroy** формы:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
 if LDebug > 0 then FDebug.Release;
end;
```

Тогда в тех местах вашего приложения, где требуется отладка, вы можете заносить в окно **Memo1** информацию о работе программы. Покажем это на примере рекурсивной функции **fib2**, разработанной в разд. 2.9:

```
function fib2(n: integer): Int64;
begin
 if n < 2
  then Result := n
 else Result := fib2(n-1) + fib2(n-2);
  // отладочная печать при уровне = 1
  if LDebug=1 then
     Memo1.Lines.Add('n = ' + IntToStr(n) +
                     ', число Фибоначчи = ' + IntToStr(Result));
end;
```

В приведенном коде прокомментировано все, относящееся к отладке. В обработчике события формы **OnCreate** задается уровень отладки оператором

```
LDebug := 1;
```

Уровень может определять степень подробности вывода отладочной информации. Если отладка вообще не нужна, этот оператор можно закомментировать.

При уровне отладки **LDebug** > 0 создается окно отладки и в нем — компонент **Memo1**, в который и выводится отладочная информация. В данном примере вывод этой информации вставлен в цикл.

В событии формы **OnDestroy** предусмотрено уничтожение отладочного окна.

Попробуйте сделать в приложении указанные изменения и запустить его на выполнение. Вы увидите в процессе выполнения окно, в которое по мере работы приложения будут заноситься данные. При этом вы получите информацию, на основании которой сможете увидеть последовательность рекурсивного расчета. И наглядно увидеть, почему он в данном примере неэффективен. Только тестируйте

отладку при небольших значениях **n**, например, при **n** = 5. Вы уже знаете, что это приложение работает долго при больших **n**. А вывод отладочной печати еще больше замедлит его работу.

После того как вы отладили приложение, вы можете легко удалить отладочную печать, убрав всего один оператор, задающий уровень отладки. Недостатком такого подхода является то, что в тексте все-таки останутся лишние операторы отладочной печати, которые чуть-чуть увеличат размер исполняемого модуля и замедлят работу. Правда, это ничтожное ухудшение эффективности приложения вы вряд ли заметите. Если вы все-таки хотите удалить из окончательного варианта отладочные печати, то можно воспользоваться директивами условной компиляции. В этом случае операторы отладки включаются в условную директиву следующим образом:

```
{$IFDEF Debug}
  операторы отладки
{$ENDIF}
```

Тогда, если в начале программы вы поместите директиву

```
{$DEFINE Debug}
```

операторы отладки будут компилироваться и выполняться. Но когда вы уберете эту директиву, или закомментируете ее, все операторы отладки исчезнут из текста.

Опыт показывает, что отладочные операторы обычно в ничтожной степени снижают эффективность работы приложения. Поэтому я обычно специально оставляю отладочную печать в готовом приложении. Дело в том, что сложное приложение, увы, не может быть сделано без ошибок. Ошибки могут проявляться иногда через месяцы и годы работы у пользователя в каких-то экзотических ситуациях. Тогда-то и поможет отладка. Включать ее можно, например, созданием текстового файла с условным именем *Debug.txt*, в котором указывать число — уровень отладки. В этом случае в приведенном ранее коде надо заменить оператор

```
LDebug := 1;
```

на

```
if FileExists('Debug.txt') then begin
  AssignFile(f, 'Debug.txt');
  Reset(f);
  Readln(f, LDebug);
end;
```

и добавить объявление переменной

```
var f:TextFile;
```

Попробуйте это сделать. Пока в каталоге, из которого вы запускаете свое приложение, нет файла *Debug.txt*, отладки не будет. Но если вы откроете текстовый файл (для этого надо выполнить команду File | New и на странице New выбрать пиктограмму Text), запишете в него единицу и сохраните под именем *Debug.txt*, то отладка появится. И никакой перекомпиляции не требуется. Вы можете включить отладку и посмотреть, как работает приложение, даже на компьютере пользователя, где нет никаких исходных текстов и нет Delphi.

2.13 Проверьте себя

2.13.1 Вопросы для самопроверки

1. Для чего используется предложение **uses**. Что такое циклические ссылки и как их избежать?
2. Ваше приложение содержит два модуля: *Unit1 и Unit2*. При попытке выполнить его вы получили сообщение компилятора об ошибке: «[Fatal Error] Unit1.pas(3): Circular unit reference to 'Unit1'». Что это означает, и какие исправления надо внести в проект?
3. Что такое масштабируемость приложения и как ее обеспечить?
4. Как обеспечивается автоматическое приведение типов?
5. Как можно преобразовать действительное число в целое?
6. Чем различаются передачи параметров в функции по значению и по ссылке. В каких случаях какой вариант передачи параметров надо использовать?
7. Как реализуются функции с параметрами по умолчанию?
8. Как передать в функцию имя (указатель) другой функции?
9. Как осуществляется перегрузка функций?
10. Поясните разницу между локальными функциями, глобальными функциями, и функциями, объявленными в классе формы.
11. Какими общими свойствами обладают порядковые типы? Почему действительные типы не являются порядковыми?
12. В каких случаях полезен оператор with?
13. Можно ли в любых случаях заменить оператор case операторами if?
14. Зачем и как вводятся перечислимые типы?
15. Как объявляются одномерные и двумерные массивы?
16. В каких случаях какие операторы циклов предпочтительнее?
17. Зачем и как можно осуществлять прерывание циклов?
18. Каким требованиям должен удовлетворять рекурсивный алгоритм?
19. Что такое линейная и экспоненциальная сложность алгоритма?
20. Что такое перемежающиеся ошибки времени выполнения и как с ними бороться?

2.13.2 Задачи

1. Создайте процедуру, заполняющую прямоугольную матрицу заданного размера N x M случайными числами, лежащими в диапазоне $0 \div 1$.
2. Создайте приложение, вычисляющее сумму и разность двух прямоугольных матриц размера N x M.
3. Создайте приложение, позволяющее подобрать ключ, использованный при шифровке текста простым алгоритмом, рассмотренным в разд. 2.8.7.2.
4. Создайте приложение, шифрующее текст ключом, состоящим из пяти чисел, применяющихся циклически к символам текста (см. разд. 2.8.7.2).

5. В приложение калькулятора, созданное в разд. 2.4.7, 2.8.4.3, 2.8.5, введите вычисление функции с двумя параметрами: возведение произвольного числа в произвольную степень.
6. Введите в функции расчета факториала и чисел Фибоначчи проверки, обеспечивающие бессбойную работу в любых ситуациях. При недопустимых данных функции должны информировать пользователя об ошибке.
7. Приложение поиска корня уравнения методом дихотомии, созданное в разд. 2.8.7.4, не обеспечивает проверку исходных данных, и предполагает задание только абсолютной допустимой погрешности по аргументу. Введите в приложение проверку и корректировку исходных данных. При ошибочном задании чисел надо сообщить об этом пользователю. Если пользователь задал a > b, приложение должна переставить эти значения, чтобы поиск происходил нормально. Если левая часть уравнения имеет на концах начального интервала неопределенности одинаковые знаки, надо предупредить пользователя о возможном отсутствии корня на интервале или наличии четного числа корней на нем. Надо предусмотреть задание и абсолютной, и относительной погрешностей. Тогда критерием окончания должно являться удовлетворение хотя бы одной из них (не обеих).
8. Создайте приложение поиска корня уравнения методом дихотомии, в котором было бы предусмотрено решение нескольких различных уравнений, и пользователь мог бы выбирать необходимое ему уравнение, щелкая на той или иной кнопке.
9. Создайте приложение поиска корня уравнения методом дихотомии для решения следующей задачи. Имеется банк, в который вы можете положить вклад с процентом годовых **P**. Вы должны заранее оговорить срок вклада — число полных месяцев. Вы знаете, что инфляция прогнозируется в **I** процентов в месяц. Значение **P** / 12 несколько больше, чем **I** (иначе класть деньги на вклад бессмысленно). На какой срок, судя по прогнозам, надо класть вклад? Пользователь приложения должен иметь возможность задавать значения **P** и **I**. Подсказка: учтите, что инфляция вычисляется с помощью сложных процентов, т.е. описывается показательной функцией.
10. Создайте приложение поиска корня уравнения методом дихотомии для решения следующей задачи. Имеется 2 банка, в которые вы можете положить вклад. Один банк выплачивает **P1** процентов годовых с первоначальной суммы взноса. Второй банк выплачивает **P2** процентов годовых (**P2** несколько меньше **P1**), но проводит ежемесячную капитализацию процентов, т.е. приплюсовывает накопившиеся проценты к вкладу, и на следующий месяц считает проценты от этой увеличившейся суммы. При каких сроках вклада выгоднее класть деньги в первый банк, а при каких — во второй? Пользователь приложения должен иметь возможность задавать значения **P1** и **P2**.
11. Создайте приложение поиска корня уравнения, подобное созданному в разд. 2.8.7.4, но реализующее метод хорд. Суть этого метода, хорошо работающего для функций, близких к линейным, сводится к следующему. Сначала проводятся два эксперимента на концах начального интервала неопределенности [a, b]. Затем по полученным результатам f(a) и f(b) (если на интервале есть корень, эти значения имеют разные знаки) намечается путем линейной интерполяции точка очередного эксперимента: $x = a - (b - a) \cdot f(a) / [f(b) - f(a)]$.

После расчета функции в этой точке в зависимости от полученного знака f(x) оставляется или интервал [a, x], или интервал [x, b]. Иначе говоря, меняется текущее значение или верхней границы b, или нижней границы a. В новом интервале, т.е. на следующей итерации опять находится точка очередного эксперимента. Итерации прекращаются, когда на двух последовательных итерациях значение x изменилось меньше, чем заданная допустимая погрешность по аргументу (надо помнить значение x на предыдущей итерации), или значение f(x) стало по модулю меньше допустимой погрешности по функции. Программа не должна допускать лишних обращений к функции левой части (в реальных задачах каждое обращение может быть очень длительным, если связано с каким-то моделированием сложных систем). На каждой итерации назначается только она новая точка x, и, значит, должно быть только одно обращение к функции. Значения функции на краях текущего интервала неопределенности должны запоминаться.

Глава 3

Более сложные элементы языка Object Pascal

В этой главе:

- *вы научитесь работать с массивами и с динамическими массивами*
- *научитесь проводить статистическую обработку данных и статистическое моделирование*
- *освоите работу со строками и текстами*
- *узнаете о записях и их использовании*
- *узнаете о связных списках, очередях, стеках и освоите различные способы их формирования*
- *научитесь работать с файлами различных видов*
- *получите информацию о классах и научитесь создавать собственные классы*
- *освоите объектно-ориентированный подход к проектированию*
- *построите много интересных и полезных приложений*

3.1 Массивы

3.1.1 Статические массивы

В разд. 2.8.7.1 уже давались предварительные сведения о массиве — структуре данных, позволяющей хранить под одним именем совокупность данных некоторого типа. Статическими будем называть массивы, размер которых фиксирован. В разд. 2.8.7.1 приводился синтаксис объявления одномерного статического массива:

```
var <имя массива>: array <ограниченный тип> of <тип элементов>;
```

Например, предложение:

```
var A: array [1..10] of integer;
```

объявляет массив с именем A, содержащий 10 целых чисел. Рассматривался также доступ к элементам массива через индексы и приводился ряд примеров работы с массивами. Отметим только, что диапазон индексов совершенно не обязательно должен начинаться с 1. Например, можно объявить массив следующим образом:

```
var B: array [0..10] of integer;
```

Надо только не запутаться в количестве элементов такого массива. Поскольку индексы изменяются от 0 до 10, то в таком массиве 11 чисел.

В приведенных примерах объявлялись непосредственно переменные, являющиеся массивами. Можно объявить и тип массива, а затем объявлять переменные этого типа. Например, операторы

```
Type Arr = array [0..10] of integer;
Var A,B:Arr;
```

объявляют тип **Arr** как массив целых чисел размером 11, и объявляют две переменные **A** и **B** этого типа.

Объявление переменной типа массива можно совмещать с заданием элементам массива начальных значений. Эти значения перечисляются после знака равенства, разделяются запятыми и заключаются в круглые скобки. Например:

```
Var A1:array[0..10] of integer = (1,2,3,4,5,6,7,8,9,10,11);
```

Аналогом одномерных массивов в математике являются векторы. Поэтому операции векторной алгебры легко реализуются с помощью массивов. Векторы часто встречаются во многих приложениях: в задачах механики, электротехники, радиотехники, физики, в инструментарии принятия решений, в задачах многомерной визуализации и т.п. Векторы с точки зрения программирования — это обычные массивы действительных чисел. Поэтому N-мерный вектор можно рассматривать как массив с числом элементов **N**, имеющих индексы от 0 до **N**−1. Например, приведенный ниже код вычисляет модуль (абсолютную величину, длину) трехмерного вектора V. Как известно, модуль равен корню из суммы квадратов координат вектора.

```
var V: array[0..3] of real;
    M: real;
    i: word;
...
V[0] := 3.;
V[1] := 4.;
V[2] := 5.;
M := sqr(V[0]);
for i := 1 to 2 do M := M + sqr(V[i]);
M := sqrt(M);
```

Попробуйте сами написать реализацию других операций с трехмерными векторами: сложения, вычитания, скалярного произведения, умножения на скаляр.

Можно объявлять многомерные массивы, т.е. массивы, элементами которых являются массивы. Например, двумерный массив можно объявить таким образом:

```
Var A2: array[1..10] of array[1..3] of integer;
```

Этот оператор описывает двумерный массив, который можно представить себе как таблицу, состоящую из 10 строк и 3 столбцов. То же самое можно объявить более компактно:

```
Var A2: array[1..10,1..3] of integer;
```

Обычно используется именно такая форма объявления многомерных массивов. Как и в одномерных массивах, элементы могут иметь любой тип, и индексы тоже могут иметь любой ограниченный тип.

Доступ к значениям элементов многомерного массива обеспечивается через индексы, перечисляемые через запятую. Например, **A2[4,3]** — значение элемента, лежащего на пересечении четвертой строки и третьего столбца.

Так же, как и в случае одномерных массивов, можно определять не непосредственно переменные типа многомерных массивов, а сначала определять соответствующий тип, а затем — переменные или типизированные константы этого типа. Например:

```
type Ar3 = array[1..4,1..3,1..2] of integer;
var A1,A2: Ar3;
const A3: Ar3=(((0,1),(2,3),(4,5)),
               ((6,7),(8,9),(10,11)),
               ((12,13),(14,15),(16,17)),
               ((18,19),(20,21),(22,23)));
```

При задании начальных условий список значений по каждой размерности заключается в скобки. Приведенный выше пример типизированной константы создает трехмерный массив **A3**, четыре строки которого являются матрицами вида

0	1
2	3
4	5

6	7
8	9
10	11

12	13
14	15
16	17

18	19
20	21
22	23

Например, элемент **A3[1,2,1]** равен 2, элемент **A3[4,1,2]** равен 19 и т.д.

Двумерные массивы представляют собой прямоугольные матрицы. Так что с их помощью вы можете запрограммировать все операции линейной алгебры с матрицами: сложение, вычитание, умножение, умножение на скаляр. Попробуйте реализовать и отладить соответствующие коды.

Для массивов можно использовать уже знакомые вам (см. разд. 2.5) функции **High** и **Low**. Если передать в них тип или переменную массива, они возвращают соответственно максимальное или минимальное значение индекса. Это можно использовать для записи операторов в общем виде, не зависящем от размера массива. Например, следующий код заполняет числовой массив любого размера числами Фибоначчи (см. разд. 2.8.7.3 и 2.8.7.5), начиная с F_1.

```
A[Low(A)] := 1;
A[Low(A)+1] := 1;
for i:=Low(A)+2 to High(A) do A[i] := A[i-2]+A[i-1];
```

Для массивов определена также функция **Length**, которая возвращает число элементов массива. Очевидно, что всегда **Length = High − Low + 1**.

Для массивов одного типа определена операция присваивания. Например, если массивы объявлены как

```
var A,B:array[1..3] of integer;
```

то в результате выполнения оператора

```
A := B;
```

значения элементов массива **B** присвоятся соответствующим элементам массива **A**. Прежние значения элементов **A** будут затерты. Таким образом, произойдет копи-

рование **B** в **A**. Сами массивы при этом останутся самостоятельными и в дальнейшем их элементы могут изменяться независимо друг от друга.

Аналогично будет работать оператор присваивания, если переменные массивов объявлены следующим образом:

```
type Ar=array[1..3] of integer;
var A:Ar;
    B:Ar;
```

Но если объявить массивы следующим образом:

```
var A: array[1..3] of integer;
    B: array[1..3] of integer;
```

то при попытке присваивания **A:=B** компилятор выдаст синтаксическую ошибку с сообщением «Incompatible types» — несовместимые типы. Дело в том, что компилятор считает, что переменные имеют один тип только в случаях, если они явным образом определены через некоторый поименованный тип, как сделано во втором примере, или если они объявлены в одном списке, как это имеет место в первом примере. А в третьем примере ни то, ни другое условие не соблюдается.

3.1.2 Передача массивов как параметров в функции и процедуры

При создании функции или процедуры работы с массивами в ее объявление нельзя включать описание индексов. Например, объявление

```
procedure MyProc(A: array[1..10] of Integer);
```

будет расценено как синтаксическая ошибка и вызовет соответствующее сообщение компилятора. Правильным будет объявление

```
type ta = array[1..10] of Integer;
procedure MyProc(A: ta);
```

Функции и процедуры в Object Pascal могут воспринимать в качестве параметров не только массивы фиксированного размера, но и так называемые *открытые массивы*, размер которых неизвестен. В этом случае в объявлении функции или процедуры они описываются как массивы базовых типов без указания их размерности. Например:

```
procedure SumArray(A:array of integer;
                   var B: array of integer);
```

При таком определении передаваемый в процедуру первый массив будет копироваться и с этой копией — массивом **A**, будет работать процедура. Второй открытый массив определен как **var**. Этот массив передается по ссылке (см. разд. 2.7.2), т.е. он не копируется, и процедура будет работать непосредственно с исходным массивом.

Выбирая, как передавать открытый массив в функцию, надо учитывать, что если его размер очень велик, то при копировании будут большие затраты памяти и времени, а может произойти и аварийное завершение, если в стеке не хватит места.

Массив, переданный как открытый, воспринимается в теле процедуры или функции как массив с целыми индексами, начинающимися с 0. Размер массива может быть определен функциями **Length** — число элементов и **High** — наибольшее значение индекса.

В качестве примера рассмотрим процедуру, которая принимает два одинакового размера открытых массива целых чисел, суммирует их и заносит результат во второй из переданных массивов.

```
procedure SumArray(A:array of integer;
                   var B: array of integer);
var i:word;
begin
 for i:=0 to High(A) do B[i]:=A[i]+B[i];
end;
```

Вызов этой процедуры может иметь вид:

```
var A1,A2: array [1..3] of integer;
begin
 <операторы заполнения массивов>
 SumArray(A1,A2);
end;
```

Обратите внимание на то, что массивы **A** и **B**, передаваемые в качестве аргументов, имеют значения индексов от 1 до 3, а процедура оперирует с индексами в диапазоне 0 – 2. Однако никакой путаницы не возникнет. Просто, например, элемент **A[0]** в теле процедуры будет соответствовать элементу **A1[1]** в массиве **A1**.

При вызове функции или процедуры с параметром в виде открытого массива можно использовать в качестве аргумента *конструктор открытого массива*, который формирует массив непосредственно в операторе вызова. Список элементов такого конструктора массива заключается в квадратные скобки, а значения элементов разделяются запятыми. Например, функцию **Sum**, суммирующую элементы числового массива, можно вызвать следующим образом:

```
S := Sum([1.2,4.45,0.1]);
```

В качестве примера использования открытых массивов запишем функцию, вычисляющую скалярное произведение двух векторов A и B одинаковой, но произвольной размерности. Она может выглядеть так:

```
function MultV(A, B: array of real): real;
var i, N: word;
begin
 Result := A[0] * B[0];
 N := High(A);
 if N <> High(B) then
 begin
  ShowMessage('Нельзя умножать векторы разных размерностей: ' +
              IntToStr(N+1) + ' и ' + IntToStr(High(B)+1));
  exit;
 end;
 for i := 1 to N do Result := Result + A[i] * B[i];
end;
```

В заключение остановимся коротко еще на одной возможности передачи массивов в функции и процедуры: в виде *открытых массивов констант*. Они, в отличие от всех других массивов, позволяют передавать в процедуру или функцию массив различных по типу значений. При передаче такого массива он объявляется как **array of const**. Например, в разд. 3.2.1 будет рассмотрена функция **Format**, которые может создавать форматированную строку из разных типов аргументов: целых и действительных чисел, символов, строк и др. Объявление функции имеет вид:

```
function Format(const SFormat: string;
                const Args: array of const): string;
```

Ее вторым аргументом **Args** является открытый массив констант, содержащий различные типы аргументов. Это может быть комбинация чисел, строк, указателей. Передаваемые в подобные функции аргументы перечисляются списком, разделяемым запятыми и заключенным в квадратные скобки. Например:

```
S := Format(SF, [5, s1, 7.3]);
```

Подробнее об этой функции см. в разд. 3.2.1.

3.1.3 Динамические массивы

3.1.3.1 Одномерные динамические массивы

Динамические массивы введены в Object Pascal, начиная с Delphi 4. Они отличаются от обычных статических массивов тем, что в них не объявляется заранее длина — число элементов. Поэтому динамические массивы удобно использовать в приложениях, где объем обрабатываемых массивов заранее неизвестен и определяется в процессе выполнения в зависимости от действий пользователя или объема перерабатываемой информации.

Объявление динамического массива содержит только его имя и тип элементов — один из базовых типов. Синтаксис объявления:

```
<имя> array of <тип элементов>
```

Например

```
var A: array of integer;
```

объявляет переменную **A** как динамический массив целых чисел.

При объявлении динамического массива место под него не отводится. Прежде, чем использовать массив, надо задать его размер процедурой **SetLength**. В качестве аргументов в нее передаются имя массива и целое значение, характеризующее число элементов. Например

```
SetLength(A,10);
```

выделяет для массива **A** место в памяти под 10 элементов и задает нулевые значения всех элементов.

Индексы динамического массива — всегда целые числа, начинающиеся с 0. Таким образом, в приведенном примере массив содержит элементы от **A[0]** до **A[9]**.

Повторное применение **SetLength** к уже существующему в памяти массиву изменяет его размер. Если новое значение размера больше предыдущего, то все значения элементов сохраняются и просто в конце добавляются новые нулевые элементы. Если же новый размер меньше предыдущего, то массив усекается, и в нем остаются значения первых элементов. Например, для приведенной ниже программы размерность и значения элементов массива при выполнении различных операторов показаны в комментариях к тексту.

```
var A: array of integer;
    N,i: integer;
begin
 N:=5;
 SetLength(A,N);                    // массив A(0,0,0,0,0)
```

```
 for i:=0 to N do A[i]:=i+1; // массив A(1,2,3,4,5)
 N:=7;
 SetLength(A,N);             // массив A(1,2,3,4,5,0,0)
 N:=3;
 SetLength(A,N);             // массив A(1,2,3)
 N:=4;
 SetLength(A,N);             // массив A(1,2,3,0)
end;
```

Впрочем, усечение динамического массива лучше проводить функцией **Copy**, присваивая ее результат самому массиву. Например, оператор

```
A := Copy(A, 0, 3);
```

усекает динамический массив **A**, оставляя неизменными первые три его элемента.

Если динамический массив уже размещен в памяти, к переменной этого массива можно применять стандартные для массивов функции **Length** — длина, **High** — наибольшее значение индекса (очевидно, что всегда **High** = **Length** − 1) и **Low** — наименьшее значение индекса (всегда 0). Если массив имеет нулевую длину, то **High** возвращает −1, т.е. при этом получается, что **High** < **Low**.

Удалить из памяти динамический массив можно одним из следующих способов: присвоить ему значение **nil** (о смысле этого значения см. в разд. 2.10), использовать функцию **Finalize** или установить нулевую длину. Таким образом, эквивалентны следующие операторы:

```
A := nil;
```

или

```
Finalize(A);
```

или

```
SetLength(A,0);
```

Если динамические массивы определены как переменные одного типа, например

```
var A,B: array of integer;
```

и размер массива **A** не меньше размера массива **B** или **A** = **nil**, то возможно присваивание вида

```
B := A;
```

которое приводит к тому, что переменная **B** начинает указывать на тот же самый массив, что и **A**, т.е. получается как бы два псевдонима для одного массива. А содержимое массива **B** при этом теряется. В этом коренное различие присваивания статических и динамических массивов.

Если динамические массивы объявлены не как переменные одного типа, т.е.

```
var A: array of integer;
    B: array of integer;
```

то присваивание

```
B := A;
```

вообще не допускается.

В операциях сравнения динамических массивов сравниваются только сами указатели, а не значения элементов массивов. Таким образом, выражение **A = B** вернет **true** только в случае, если **A** и **B** указывают на один и тот же массив. А вот выражение **A[0] = B[0]** сравнивает значения первых элементов двух массивов.

Динамические массивы могут передаваться в качестве параметров в те функции и процедуры, в описаниях которых параметр объявлен как массив базового типа без указания индекса, т.е. открытый массив. Например, функция

```
function CheckStrings(A: array of string): Boolean;
```

может работать в равной степени и со статическими, и с динамическими массивами.

Сочетание динамических массивов с открытыми массивами позволяет создавать универсальны функции, принимающие и возвращающие массивы произвольного размера. Правда, в качестве возвращаемого значения нельзя задать динамический массив. Но можно объявить тип динамического массива и его указать в качестве типа возвращаемого значения.

Ниже приведена в качестве примера функция, возвращающая сумму двух векторов произвольной, но одинаковой размерности:

```
type TVector = array of real;

function SumV(const A, B: array of real): TVector;
// Сложение двух векторов
var i, N: word;
    Vector: TVector;
begin
 if(Length(A) = 0) or (Length(B) = 0)
 then begin
   ShowMessage('Не задан вектор');
   exit;
 end;
 N := High(A);
// Размерность векторов должна быть одинаковой
 if(N <> High(B))
 then begin
   ShowMessage('Нельзя складывать векторы разных размерностей:'
          + IntToStr(N+1) + ' и ' + IntToStr(Length(B)));
   exit;
 end;
 SetLength(Vector, N+1);
 for i:=0 to N do
    Vector[i] := A[i] + B[i];
 Result := Vector;
end;
```

В этом коде объявляется тип **TVector** динамического массива действительных чисел. Этот тип указывается как тип возвращаемого значения функции **SumV**. Параметры **A** и **B** этой функции тоже можно было бы объявить типа **TVector**. Но это не стоит делать. При таком объявлении аргументы в вызове функции тоже обязательно должны быть того же типа, так что обычные массивы передать в функцию было бы нельзя. А при объявлении **array of real** функция примет и обычные массивы, и динамические, и массивы типа **TVector**.

Перед параметрами функции задан спецификатор **const**. Это несколько оптимизирует код, создаваемый компилятором (см. разд. 2.7.2). Можно было бы задать

спецификатор **var**, что позволило бы избежать затрат времени и памяти на копирование массивов в функцию (см. тот же разд. 2.7.2). Впрочем, для небольших массивов заметить это было бы трудно.

В функции объявляется локальная переменная **Vector** типа **TVector**. Функцией **SetLength** этому массиву задается тот же размер, который имеют массивы входных параметров. Далее в цикле в этот массив заносится результат сложения векторов. Последний оператор функции присваивает возвращаемому значению **Result** указатель на этот массив.

Вызов подобной функции может быть оформлен различными способами. Возможен такой вызов:

```
var A, B: array[1..3] of real;
    C: TVector;
...
// Заполнение массивов
A[1] := ...;
...
C := SumV(A, B);
```

Здесь параметры передаются в функцию как обычные массивы. Возможен другой вариант:

```
var A, B, C: TVector;
...
// Задание размера динамических массивов
SetLength(A, 3);
SetLength(B, 3);
// Заполнение массивов
A[1] := ...;
...
C := SumV(A, B);
```

Еще один вариант:

```
var C: TVector;
...
C := SumV([1, 2, 3], [4, 5, 6]);
```

В данном случае массивы параметров задаются не в качестве переменных, а как конструкторы открытых массивов.

Я советую сделать по приведенной схеме функции, реализующие другие операции с векторами произвольной размерности: вычисление модуля, вычитание векторов, умножение вектора на скаляр, вычисление скалярного и векторного произведений. Подобные функции имеет смысл внести в вашу библиотеку функций, созданную в разд. 2.8.7.4. Они могут вам пригодиться при решении многих практических задач.

Полезно также создать ряд процедур, выполняющих такие стандартные операции над динамическими массивами, как вставка нового элемента в указанное место массива (тогда позиции всех последующих элементов должны увеличиться на 1, и размер массива тоже увеличивается на 1), удаление элемента из указанной позиции (массив уплотняется и его размер уменьшается на 1), перестановки двух элементов в указанных позициях, замена значения указанного элемента. В качестве примера рассмотрим процедуру вставки указанного элемента на указанное место массива действительных чисел. Она может иметь следующий вид:

```
type AReal = array of real;

function AInsrert(var A: AReal; R: real; Ind: integer):
                                                        boolean;
// Вставка элемента R в позицию Ind динамического массива A
// При ошибочном индексе возвращается false
var i, N: integer;
begin
 N := Length(A);
 Result := not ((Ind < 0) or (Ind > N));
 if not Result
  then begin
   ShowMessage('Ошибочный индекс ' + IntToStr(Ind) +
               ' в функции AInsrert');
   exit;
  end;
 // Увеличение размера массива
 SetLength(A, N + 1);
 // Сдвиг элементов
 for i := N-1 downto Ind do
  A[i+1] := A[i];
 // Вставка элемента
 A[Ind] := R;
end;
```

В этом коде все операции прокомментированы. Обратите внимание на то, что в операторе **for** используется цикл с уменьшающимся счетчиком, так как сдвиг элементов с индексом большим, чем **Ind**, надо начинать с последнего элемента.

Вызов подобной функции может осуществляться, например, так:

```
var Ar: AReal;
...
if AInsrert(Ar, StrToFloat(Edit1.Text), StrToInt(Edit2.Text))
then ...
```

В этом операторе предполагается, что вставляемый элемент задан в окне **Edit1**, а индекс вставки — в окне **Edit2**. Создайте приложение, тестирующее эту функцию. Информацию об исходном и результирующем массиве выводите в окно **Memo**. Имеет смысл реализовать также упомянутые выше функции удаления, перестановки и изменения элементов. Такие функции стоит занести в вашу библиотеку.

3.1.3.2 Многомерные динамические массивы

Многомерный динамический массив определяется как динамический массив динамических массивов динамических массивов и т.д. Например:

```
var A2: array of array of integer;
```

определяет двумерный динамический массив.

Один из способов отвести память под такой массив — использовать ту же процедуру **SetLength**, которая использовалась для одномерного массива, но передавать ей в качестве параметров не один, а несколько размеров. Например, оператор

```
SetLength(A2,3,4);
```

задает размер массива 3 на 4.

Доступ к элементам многомерных динамических массивов осуществляется так же, как и для статических массивов. Например, **A2[1,2]** — элемент, лежащий на пересечении второй строки и третьего столбца (индексы считаются от нуля, так что индекс 1 соответствует второй строке, а индекс 2 — третьему столбцу).

Можно создавать и более интересные объекты — непрямоугольные массивы, в которых, например, второй размер не постоянен и варьируется в зависимости от номера строки. В этом случае сначала процедурой **SetLength** задается первый размер. Например, оператор

```
SetLength(A2,3);
```

задает первый размер — 3. В памяти отводится место под 3 строки — 3 динамических массива **A2[0]**, **A2[1]** и **A2[2]**. Размеры каждого из этих массивов еще не определены. Далее можно, например, задать размер первого из них равным 4:

```
SetLength(A2[0],4);
```

а размер следующего, например, равным 5:

```
SetLength(A2[1],5);
```

В качестве примера приведем программу построения и заполнения нижней треугольной матрицы произвольного размера **N**.

```
var A2: array of array of integer;
    N,i1,i2,m: integer;
begin
 N:=3;
 m:=1;
 SetLength(A2,N);          // Задание числа строк = N
 for i1:=0 to N do         // Цикл по строкам
  begin
   // Число столбцов равно номеру строки
   SetLength(A2[i1],i1+1);
   for i2:=0 to i1 do
    begin
     A2[i1,i2]:=m;         // Заполнение строки
     Inc(m);               // Увеличение m на 1
    end;
  end;
end;
```

Программа формирует двумерный массив, в котором число строк равно значению **N**, а число столбцов в каждой строке равно номеру строки. В результате получается следующая матрица:

1		
2	3	
4	5	6

В качестве примера работы с двумерными динамическими массивами, которые с успехом можно использовать для представления прямоугольных матриц произвольного размера, реализуем функцию, складывающую две матрицы. Такая функция может иметь следующий вид:

```
type TMatrix = array of array of real;

function SumMatrix(const A, B: TMatrix): TMatrix;
// Сумма двух матриц
var i, j, M, N :word;
    Matr: TMatrix;
begin
 if (Length(A) = 0) or (Length(B) = 0)
 then begin
   ShowMessage('Не задана матрица');
   exit;
 end;
 M := High(A);
 N := High(A[0]);
// Размеры матриц должны быть одинаковыми
 if(N <> High(B[0])) or (M <> High(B))
 then begin
   ShowMessage('Заданы матрицы разных размеров: '+
               IntToStr(M+1) + ' x ' + IntToStr(N+1) +
               ' и ' + IntToStr(Length(B)) + ' x ' +
               IntToStr(Length(B[0])));
   exit;
 end;
 SetLength(Matr, M+1, N+1);
 for i:=0 to M do
   for j:=0 to N do
     Matr[i, j] := A[i, j] + B[i, j];
 Result := Matr;
end;
```

В этом коде объявляется тип **TMatrix** двумерного динамического массива действительных чисел. Этот тип указывается как тип параметров **A** и **B** этой функции, и как тип возвращаемого ею значения. Перед параметрами функции задан спецификатор **const**. Это несколько оптимизирует код, создаваемый компилятором (см. разд. 2.7.2). Можно было бы задать спецификатор **var**, что позволило бы избежать затрат времени и памяти на копирование массивов в функцию (см. тот же разд. 2.7.2).

В функции объявляется локальная переменная **Matr** типа **TMatrix**. Функцией **SetLength** этому двумерному массиву задаются те же размеры, которые имеют матрицы **A** и **B**. Далее в цикле в эту матрицу заносится результат сложения матриц **A** и **B**. Последний оператор функции присваивает возвращаемому значению **Result** указатель на эту матрицу.

Вызов подобной функции может быть оформлен следующим образом:

```
var A, B, C: TMatrix;
...
// Задание размеров динамических массивов матриц 3 x 4
SetLength(A, 3, 4);
SetLength(B, 3, 4);
// Заполнение матриц A и B
...
C := SumMatrix(A, B);
```

Я бы посоветовал вам сделать по приведенной схеме функции, реализующие другие операции с матрицами произвольного размера: вычитание и умножение матриц, умножение матрицы на скаляр, вычисление определителя матрицы. Подобные функ-

ции имеет смысл внести в вашу библиотеку функций, созданную в разд. 2.8.7.4. Они могут вам пригодиться при решении многих практических задач.

3.1.4 Сортировка массивов

В качестве примера работы с массивами рассмотрим сортировку — упорядочивание массива в последовательности возрастания или убывания его элементов. Сортировка нужна, прежде всего, для упорядоченного представления данных массива пользователю. Но она часто требуется не сама по себе, а как предварительная процедура, упорядочивающая массив для облегчения последующей обработки данных. Пусть, например, в массиве содержатся результаты некоторого эксперимента. Тогда они требуют предварительного упорядочивания для облегчения последующего построения гистограмм, графиков, поиска значения, близкого к указанному, и т.п.

Наиболее простой способ решения задачи сортировки — алгоритм, получивший название *пузырьковая сортировка* или *сортировка погружением*. Названия связаны с тем, что в процессе работы алгоритма наименьшее значение постепенно «всплывает», продвигаясь к вершине (началу) массива, подобно пузырьку воздуха в воде, тогда как наибольшее значение погружается на дно (конец) массива. Этот алгоритм требует нескольких проходов по массиву. При каждом проходе сравнивается пара следующих друг за другом элементов. Если пара расположена в возрастающем порядке или элементы одинаковы, то мы оставляем значения как есть. Если же пара расположена в убывающем порядке, значения меняются местам в массиве.

Пусть, например, нам надо сортировать массив **A** из 5 элементов, имеющих значения, показанные во второй строке табл. 3.1. Первый проход алгоритма состоит в попарном сравнении элементов, начиная с последней пары (A[4], A[5]), и кончая первой (A[1], A[2]). Если оказывается, что первый элемент пары больше второго, то эти элементы переставляются. В результате, как видно из табл. 3.1, наименьший элемент перемещается на первое место («всплывает» вверх, как пузырек воздуха в воде). В общем случае, если размер массива **N**, то первый проход требует провести **N** − 1 сравнение. Ну, а число требуемых перестановок зависит от конкретных значений элементов. Если, например, массив сразу был упорядочен, то не потребуется ни одной перестановки. А если сначала минимальный элемент был расположен на последнем месте, то каждая проверка будет завершаться перестановкой.

Поскольку после первого прохода наименьший элемент уже расположен на первом месте, во втором проходе надо сравнивать пары, начиная с последней (A[4], A[5]), и кончая второй (A[2], A[3]). Это требует **N** − 2 сравнения. В результате второй по величине элемент займет второе место в массиве. Далее следует третий проход и т.д., причем в каждом проходе число проверок сокращается на 1. Последний (N − 1)-ый проход состоит всего из одной проверки. Нетрудно посчитать, что весь процесс сортировки требует **N** • (**N** − 1) / 2 проверок. А число перестановок зависит от начального расположения элементов. Так в приведенном примере было сделано 7 перестановок.

Таблица 3.1. Иллюстрация работы алгоритма пузырьковой сортировки

		A[1]	A[2]	A[3]	A[4]	A[5]
	Исходное состояние	5	1	4	3	2
Проход 1	Сравнение A[4] и A[5]	5	1	4	2	3
	Сравнение A[3] и A[4]	5	1	2	4	3
	Сравнение A[2] и A[3]	5	1	2	4	3
	Сравнение A[1] и A[2]	1	5	2	4	3
Проход 2	Сравнение A[4] и A[5]	1	5	2	3	4
	Сравнение A[3] и A[4]	1	5	2	3	4
	Сравнение A[2] и A[3]	1	2	5	3	4
Проход 3	Сравнение A[4] и A[5]	1	2	5	3	4
	Сравнение A[3] и A[4]	1	2	3	5	4
Проход 4	Сравнение A[4] и A[5]	1	2	3	4	5

Реализация рассмотренного алгоритма может быть такой:

```
var A:array[1..5] of integer = (5,1,4,3,2);
    i, j, N, Rtmp: integer;
...
N := High(A);
for i:=1 to N do
 for j:=N-1 downto i do
  if A[j] > A[j+1]
    then begin
     // Перестановка элементов A[j] и A[j+1]
     Rtmp := A[j+1];
     A[j+1] := A[j];
     A[j] := Rtmp;
    end;
```

Приведенный код достаточно прост. В начале функцией **High** определяется максимальное значение индекса массива и заносится в переменную **N**. Затем выполняются вложенные циклы: цикл проходов со счетчиком **i** и цикл попарных сравнений элементов со счетчиком **j**.

Создайте приложение, осуществляющее сортировку массива. Предусмотрите вывод несортированного и сортированного массивов в окно **Memo**. Так как вывод надо осуществлять дважды: до сортировки и после, удобно написать процедуру вывода элементов массива в окно **Memo**. А поскольку эта процедура должна работать с компонентом формы, удобно включить ее объявление в класс формы (вспомните, как это делается — см. разд. 2.7.1).

Рассмотренный алгоритм сортировки можно усовершенствовать. Дело в том, что если на очередном проходе не было перестановок, т.е. все элементы оказались расположенными по порядку, то не имеет смысла делать следующие проходы, так как в них, естественно, тоже не будет перестановок. Значит, надо прекращать сортировку, если на очередном проходе перестановок не было. Тогда в случае, если все элементы сразу оказались расположенными по порядку, потребуется всего один проход, чтобы убедиться в этом.

Ниже приведен вариант организации цикла в подобном алгоритме:

```
var ...
    noswap: boolean;

N := High(A);
for i:=0 to N-1 do
 begin
  noswap := true;
  for j:=N-1 downto i do
   if A[j] > A[j+1]
    then begin
     Rtmp := A[j+1];
     A[j+1] := A[j];
     A[j] := Rtmp;
     noswap := false;
    end;
    if(noswap) then break;
 end;
```

Булева переменная **noswap** является флагом, показывающим, были ли перестановки при очередном проходе. В начале каждого прохода переменной **noswap** присваивается значение **true**. Если это значение к концу прохода сохранилось, значит, на протяжении прохода не было ни одной перестановки элементов, так как только при перестановке задается **noswap = false**. Поэтому, если в конце прохода **noswap = true**, осуществляется прерывание сортировки и выход из цикла оператором **break**.

Рассмотренные алгоритмы начинают сравнение с последней пары элементов и продвигаются к началу массива, перемещая к началу элемент с наименьшим значением. Можно, конечно, организовать алгоритм иначе, начиная со сравнения первой пары элементов и перемещая к концу массива элемент с наибольшим значением.

3.1.5 Случайные числа и статистическая обработка данных

3.1.5.1 Функции статистической обработки

Имеется ряд стандартных функций и процедур, позволяющих производить статистическую обработку данных, хранящихся в числовых массивах. Они находят широкое применение в случаях, если массив содержит данные какого-то эксперимента или результаты наблюдения какого-то процесса. Все рассмотренные ниже функции объявлены в модуле *Math*, который автоматически к приложению не подключается. Поэтому при использовании этих функций его надо вручную связать с приложением оператором **uses**.

Функции **MaxIntValue**, **MaxValue**, **MinIntValue**, **MinValue** возвращают максимальные (**MaxIntValue** и **MaxValue**) и минимальные (**MinIntValue** и **MinValue**) значения со знаком элементов, хранящихся в массивах целых (**MaxIntValue** и **MinIntValue**) и действительных (**MaxValue** и **MinValue**) чисел. Например, оператор

```
B := MaxValue(A);
```

присваивает действительной переменной **B** максимальное из значений элементов, хранящихся в массиве действительных чисел **A**.

Предупреждение

Все описанные в данном разделе функции статистической обработки, относящиеся к массивам действительных чисел, требуют массив типа Double. Несмотря на то, что тип Real эквивалентен Double, при объявлении типа элементов массива надо использовать именно Double. Использование Real вызовет сообщение об ошибке: «Incompatible types» — несовместимые типы.

Функции **SumInt** и **Sum** возвращают суммы значений элементов массивов соответственно целых и действительных чисел. Функция **SumOfSquares** возвращает сумму квадратов значений элементов массива действительных чисел.

Процедура **SumsAndSquares**, определенная как

```
procedure SumsAndSquares(const Data: array of Double;
                         var Sum, SumOfSquares: Extended);
```

рассчитывает для массива **Data** одновременно сумму **Sum** и сумму квадратов **SumOfSquares** значений элементов. Время вычислений по процедуре **SumsAndSquares** меньше, чем при последовательном вызове функций **Sum** и **SumOfSquares**. Поэтому, если требуется знать и сумму, и сумму квадратов, то лучше использовать именно эту процедуру.

Функция **Norm** возвращает эвклидову норму: корень из суммы квадратов значений элементов массива.

Теперь перейдем к процедурам и функциям, вычисляющим характеристики законов распределения случайных величин, хранимых в массивах. Функция **Mean** возвращает среднее арифметическое значение (математическое ожидание) элементов массива действительных чисел **Data**. Например, оператор

```
B := Mean(A);
```

присваивает действительной переменной **B** значение математического ожидания элементов, хранящихся в массиве действительных чисел **A**. Если массив **A** содержит **n** элементов, то среднее значение рассчитывается по формуле

$$\sum_{i=1}^{n} A[i] / n.$$

Функция **StdDev** возвращает несмещенную оценку среднего квадратического отклонения элементов массива действительных чисел **Data**. Например, оператор

```
B := StdDev(A);
```

присваивает действительной переменной **B** значение среднего квадратического отклонения элементов, хранящихся в массиве действительных чисел **A**. Если массив **A** содержит **n** элементов, то среднее квадратическое отклонение рассчитывается по формуле

$$\sqrt{\sum_{i=1}^{n} (A[i] - \overline{A})^2 / (n-1)},$$

где $\overline{A}$ — среднее значение (математическое ожидание) элементов массива. Это несмещенная оценка, статистически более точная, чем корень из суммы квадратов отклонений, деленной на **n**. Если массив содержит всего один элемент, функция **StdDev** возвращает значение **NAN**.

Процедура **MeanAndStdDev**, объявленная как

```
procedure MeanAndStdDev(const Data: array of Double;
                        var Mean, StdDev: Extended);
```

рассчитывает для массива **Data** одновременно математическое ожидание **Mean** и среднее квадратическое отклонение **StdDev**. Время вычислений по процедуре **MeanAndStdDev** вдвое меньше, чем при последовательном вызове функций **Mean** и **StdDev**. Поэтому, если требуется знать и математическое ожидание, и среднее квадратическое отклонение, то лучше использовать именно эту процедуру.

Мы рассмотрели только некоторые из функций и процедур обработки статистических данных. Полный их список см. во встроенной справке Delphi, или в справке [3].

3.1.5.2 Генерация случайных чисел

Теперь остановимся на заполнении массивов случайными числами. Это нередко приходится делать при тестировании различных программ, при генерации случайных процессов, подаваемых на вход различных устройств или приложений, при расчетах разброса параметров устройств и моделей и т.п. Следует учесть, что компьютер никогда не генерирует действительно случайные числа. Генерируются только так называемые псевдослучайные числа. Они вырабатываются из некоторого начального числа применением к нему различных преобразований. Псевдослучайные числа отличаются от случайных следующими двумя свойствами:

- При одинаковом начальном числе каждый раз генерируется одна и та же случайная последовательность. Поэтому, если не принять специальных мер, то ваше приложение при каждом своем запуске будет генерировать одни и те же случайные числа. Это удобно при отладке, но не годится во время реальной работы. Чтобы избежать такого повторения случайных последовательностей, надо применять так называемую рандомизацию, сводящуюся, фактически, к случайному заданию начального числа при каждом выполнении приложения.
- Генераторы квазислучайных чисел имеют вполне конечный, хотя обычно очень большой, отрезок апериодичности. Отрезок апериодичности — это то количество случайных чисел, последовательность которых не повторяется. А если использовать в программе большее количество чисел, то они начнут повторяться с некоторым периодом. Считать подобные числа случайными невозможно. Поэтому количество используемых случайных чисел не должно превышать отрезка апериодичности. Если вам все-таки требуется больше чисел, то вы должны при приближении к границе отрезка апериодичности (а лучше задолго до нее) обновить последовательность чисел с помощью функций рандомизации.

Функция **Random**, объявленная в модуле *System* как

```
function Random (Range: Integer);
```

позволяет генерировать последовательности случайных целых и действительных чисел. Параметр **Range** не является обязательным. Если этот параметр не задан, то функция возвращает случайные действительные числа X, равномерно распределенные в интервале 0 <= X < 1. Например, оператор

```
for i:=Low(A) to High(A) do
  A[i] := Random;
```

заполняет массив **A** равномерно распределенными действительными числами в диапазоне от 0 до 1. Если вам требуются действительные случайные числа X, равномерно распределенные в каком-то другом диапазоне A1 <= X < A2, то такие действительные числа легко получить, сдвинув начальное значение чисел и умножив генерируемые числа на длину интервала. Например, операторы

```
A1, A2: double;
...
A1 := 50;
A2 := 150;
for i:=Low(A) to High(A) do
  A[i] := A1 + (A2 - A1) * Random;
```

обеспечивают заполнение массива действительными случайными числами X, лежащими в диапазоне 50 <= X < 150.

Если в функции **Random** задан параметр **Range**, то функция возвращает случайные целые числа в диапазоне 0 <= X < **Range**. Например, оператор

```
for i:=Low(M) to High(M) do
  M[i] := Random(101);
```

заполняет массив **M** целыми числами, равномерно распределенными в диапазоне от 0 до 100. А оператор

```
for i:=Low(M) to High(M) do
  M[i] := 100 + Random(101);
```

заполняет массив **M** целыми числами, равномерно распределенными в диапазоне от 100 до 200.

Функция **RandG**, объявленная в моуле *Math* как

```
function RandG(Mean, StdDev: Extended): Extended;
```

генерирует квазислучайные действительные числа, распределенные по нормальному закону (закону Гаусса) с математическим ожиданием **Mean** и средним квадратическим отклонением **StdDev**. Например, следующие операторы заполняют массив **M** действительными числами, распределенными по нормальному закону с математическим ожиданием 100 и средним квадратическим отклонением 10, а затем функцией **MeanAndStdDev** проверяются значения этих характеристик:

```
uses math;

const N = 100;
var   M: array[1..N] of Double;
      i: integer;
      Mean, StdDev: Extended;
...
for i:=Low(M) to High(M) do
  M[i] := RandG(100, 10);
MeanAndStdDev(M, Mean, StdDev);
ShowMessage(FloatToStr(Mean) + ' ' + FloatToStr(StdDev));
```

Изменяя в этом тесте размер массива **N**, вы можете наглядно посмотреть, что достоверные статистические данные получаются только при достаточно больших объемах выборки. Например, при **N** = 1000 рассчитанные по массиву математическое ожидание и среднее квадратическое отклонение достаточно близки к заданным. А при **N** = 100 уже имеется заметная погрешность в среднем квадратическом отклонении.

Поскольку генерируемые рассматриваемыми функциями числа являются псевдослучайными, то при каждом новом запуске вашего приложения будет вырабатываться одна и та же последовательность чисел. Это очень удобно для отладки приложения, но не подходит для его окончательного варианта. Если требуется, чтобы при каждом запуске приложения генерировались новые случайные числа, надо рандомизировать генератор чисел, т.е. задавать ему каждый раз новое случайное исходное число. Рандомизацию осуществляет функция **Randomize**. Достаточно вставить где-то в текст программы (например, в событие **OnCreate** формы) оператор

```
Randomize;
```

чтобы при каждом запуске приложения генерировалась новая последовательность чисел. Вставьте этот оператор в приведенный выше пример, и вы легко почувствуете действие функции **Randomize**.

Генерация случайных чисел широко используется при отладке различных алгоритмов. Например, при тестировании алгоритма сортировки в разд. 3.1.4 целесообразно было бы заполнять массив случайными числами. Это не было сделано просто потому, что случайные числа к тому моменту еще не рассматривались. Если вы проектируете приложение, обрабатывающее какие-то случайные данные, поступающие из внешнего источника, сложно отлаживать ваши алгоритмы сразу на реальном источнике данных. Гораздо проще отлаживать их на числах, генерируемых компьютером, тем более что без рандомизации они будут повторяться при каждом выполнении приложения, что облегчит отладку.

Но случайные числа широко используются также в различных расчетах. Примеры этого вы найдете в следующих разделах.

3.1.5.3 Пример — карточная игра

Простейший пример использования массивов случайных чисел — моделирование тасования колоды карт при разработке программы какой-то карточной игры. Давайте попробуем реализовать простейшую игру в очко. Конечно, это не слишком интеллектуальная и мало почтенная игра. Но для нее легко составить алгоритм. И мы будем иметь возможность освоиться с массивами. А приобретя на этом примере опыт разработки подобных программ, вы самостоятельно сможете попробовать свои силы в чем-либо более интеллектуальном.

Напомню правила игры. Используется колода из 36 карт, начиная с шестерок. Каждая карта, независимо от масти, приносит свои очки: шестерки — 6, ..., десятки — 10, валеты — 2, дамы — 3, короли — 4, тузы — 11. Один участник игры, в нашем примере — компьютер, тасует колоду и сдает одну карту партнеру (сдает втемную, не видя ее). Партнер может попросить еще одну карту, еще и т.д. Его задача — набрать 21 очко. Если набрал ровно 21 (очко), он выиграл. Если набрал больше (перебор), он проиграл. Если набрал меньше и не хочет брать еще одну карту, он, не открывая своих карт, говорит: «Хватит». Тогда сдающий (компьютер) аналогичным образом открывает поочередно карты себе. Если наберет очков меньше, чем у партнера (он не видит, сколько у того очков) — он проиграл. Если сумма очков окажется больше 21 (перебор) — он тоже проиграл. А если наберет очков не более 21, но более чем у партнера — он выиграл.

Вот такая незамысловатая игра. Форма этого приложения может иметь вид, который вы видите на рис. 3.1. Кнопка Тасовать осуществляет случайное тасование колоды и выдает первую карту пользователю — заносит ее название в окно **Memo**. Для

облегчения действий пользователя в это окно выводится и текущая сумма очков, набранная игроком. Кнопка Еще дает пользователю следующую карту. Кнопка Хватит инициирует игру компьютера. Мы, естественно, будем реализовывать честную игру, в которой компьютер как бы не знает, какие карты он сдал пользователю, и какая карта лежит следующей в колоде. Компьютер открывает некоторое количество очередных карт, после чего подсчитывается набранное им число очков, сравнивается с числом очков пользователя, и определяется, кто победил. Информация о картах, открытых компьютером заносится в окно **Memo**. После сообщения о результатах игры колода опять перемешивается, и пользователю сдается новая карта.

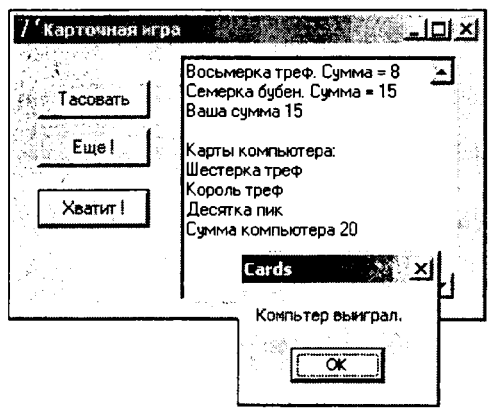


Рис. 3.1
Приложение во время выполнения

В начале проектирования каждого приложения надо четко представлять, какие операции (действия) надо в нем реализовать. В данном примере, очевидно, надо реализовать две вспомогательные функции (такие функции называются *утилитами*):

- Отображение в окне **Memo** информации о заданной карте.
- Определение цены заданной карты.

Эти две утилиты целесообразно оформить отдельными функциями, так как обращение к ним потребуется из разных мест программы: при сдаче карты пользователю и при сдаче карты компьютеру.

Помимо этих утилит надо будет, естественно, реализовать, обработчики щелков на кнопках и рандомизацию генератора случайных чисел, используемого при тасовании колоды.

После того как вы выяснили состав действий, которые надо реализовать в программе, надо продумать форму хранения данных. Давайте отведем для хранения колоды массив целых чисел **A**, содержащий 36 элементов:

```
var A: array[0..35] of integer; // колода
```

В нормальном, не перемешанном состоянии в массиве хранятся целые числа, соответствующие индексами от 0 до 35. Будем полагать, что первые 9 индексов относятся к картам масти пик, следующие 9 — к масти треф, далее — лежат карты червей и бубен. В каждой масти карты лежат в последовательности: 6, 7, ..., 10, валет, дама, король, туз. В дальнейшем колода (элементы массива **A**) может быть перемешана, но соответствие между картами и значениями элементов при этом останутся

прежними. Например, значение 0 может оказаться присвоенным элементу с индексом 32. Тогда мы будем знать, что 33-я карта в колоде (индексы отсчитываются от 0) — это шестерка пик.

Разобравшись с хранение информации, мы можем написать требуемые нам утилиты. Функция **cost**, которая определяет число очков заданной карты, может иметь вид:

```
function cost(N: integer): integer;
begin
 case N mod 9 of
  0, 1, 2, 3, 4: Result := N mod 9 + 6;
  5, 6, 7:       Result := N mod 9 - 3;
  8:             Result := 11;
 end;
end;
```

Функция принимает целое число **N**, обозначающее карту, и возвращает число соответствующих ей очков. Структура **case** производит выбор на основании выражения **N mod 9**. Это выражение дает порядковый номер карты в масти. Например, при **N** равном 0, 9, 18, 27 оно возвращает 0, соответствующий шестерке. А метки в структуре **case** обеспечивают вычисление числа очков карты в соответствии с рассмотренными ранее правилами игры.

Отметим одно обстоятельство. Во время компиляции проекта компилятор выдаст замечание: «Return value of function 'cost' might be undefined» — «Возвращаемое значение функции 'cost' может быть не определено». Компилятор просто «не замечает» что выражение **N mod 9** может возвращать только числа от 0 до 8, и «предполагает», что могут быть другие значения, при которых **Result** окажется неопределенным. От этого замечания легко было бы избавиться, заменив, например, оператор метки 8 оператором:

```
else Result := 11;
```

В работе алгоритма ничего не изменилось бы, а замечание исчезло бы. Но это тот редкий случай, когда вряд ли стоит прислушиваться к замечанию компилятора. Код алгоритма станет менее понятным. А наглядность кода, как уже не раз говорилось, очень важна.

Функция **card**, возвращающая описание карты в виде строки, может иметь вид:

```
function card(N: integer): string;
const face: array[0..8] of string =
                        ('Шестерка', 'Семерка', 'Восьмерка',
                         'Девятка', 'Десятка', 'Валет',
                         'Дама', 'Король', 'Туз');
const suit: array [0..3] of string =
                        ('пик', 'треф', 'червей', 'бубен');
begin
 Result := face[N mod 9] + ' ' +suit[N div 9];
end;
```

В этой функции вводится две константы в виде массивов строк. Массив **face** содержит слова, обозначающие карту, а массив **suit** содержит обозначения мастей в родительном падеже. Параметр **N**, как и в предыдущей утилите, — это число, обозначающее карту. Функция **card** возвращает строку, составленную из соответствующих элементов массивов **face** и **suit** с разделительным пробелом между ними. Индекс карты в масти, соответствующий индексу массива **face**, рассчитывается,

как и в предыдущей функции, выражением **N mod 9**. А масть определяется целочисленным делением **N div 9**, возвращающим 0 при **N** < 9, 1 при 8 < **N** < 18 и т.д.

Теперь давайте подумаем над алгоритмом тасования колоды.

Посмотрите сначала следующий код:

```
for i := 0 to 35 do A[i] := 0;
for i := 0 to 35 do A[Random(36)] := i;
```

Сначала массив **A** очищается. Затем в цикле перебираются все карты (переменная **i**), для каждой из них определяется функцией **Random** случайное место в колоде (в массиве **A**), и карта помещается в это место.

Подумайте, в чем ошибка подобного алгоритма? Я думаю, что вы ответили на этот вопрос правильно: какие-то карты могут лечь друг на друга, и значит, некоторые позиции в массиве останутся пустыми. Точнее, в них будет лежать 0, т.е. в колоде окажется несколько шестерок пик.

Очевидно, надо определять, не занята ли очередная случайная позиция. Если занята, то следует для данной карты генерировать новые случайные числа до тех пор, пока не найдется пустая позиция. А как определить, свободна ли позиция в массиве? Стандартный подход к подобным задачам: заполнить сначала массив какими-то условными числами, которые не могут встретиться в данной задаче. Тогда перед размещением очередной карты можно будет проверить значение, лежащее в намеченной позиции. Если оно равно условному числу, значит позиция пустая.

Так что окончательный алгоритм тасования может выглядеть так:

```
for i := 0 to 35 do A[i] := -1;
for i := 0 to 35 do
begin
 j := Random(36);
 while A[j] <> -1 do j := Random(36);
 A[j]:= i;
end;
```

Массив заполняется отрицательными числами −1, затем в цикле перебираются все карты (переменная **i**) и для каждой из них определяется случайное место в колоде — переменная **j**. Если это место не свободно (значение элемента массива не равно −1) то цикл **while** генерирует случайные значения, пока не найдется пустая позиция. В эту позицию помещается карта **i**.

А теперь рассмотрим другой возможный алгоритм. Сначала карты в колоде размещаются в стандартной последовательности, т.е. по порядку. Затем берется карта в первой позиции и переставляется с другой картой, расположенной в случайно выбранной позицией. Затем карта из второй позиции аналогичным образом переставляется с другой случайной картой. Цикл перестановок продолжается, пока не будет переставлена карта в последней позиции. Следующий код реализует этот алгоритм:

```
for i := 0 to 35 do A[i] := i;
for i := 0 to 35 do
begin
 j := Random(36);
 Tmp := A[i];
 A[i] := A[j];
 A[j] := Tmp;
end;
```

Оба рассмотренных алгоритма честно тасуют колоду. Подумайте, какому из них следует отдать предпочтение с точки зрения эффективности работы. В первом алгоритме для того, чтобы найти свободную позицию, надо многократно вызывать функцию генерации случайного числа. Особенно много раз приходится вызывать ее на последних проходах цикла. Например, когда в массиве осталась всего одна свободная позиция, придется вызывать функцию **Random** до тех пор, пока она случайно не выдаст номер этой единственной позиции. А ведь случайность есть случайность: может оказаться, хотя и маловероятно, что потребуются сотни вызовов. Подобные алгоритмы, в которых затраты времени неопределенны, использовать нежелательно. Во втором алгоритме все более определенно. Цикл будет пройден ровно 36 раз, и затраты при каждом проходе невелики: один вызов функции **Random** и 4 операции присваивания. Так что никаких сомнений нет: второй алгоритм лучше.

Я остановился на этом столь подробно, чтобы на примере показать цепь рассуждений, приводящую к созданию надежного и эффективного алгоритма. А теперь приведу сразу весь текст модуля нашего приложения.

```
var A: array[0..35] of integer; // колода
    Count: word = 0;            // число взятых карт
    Sum: word = 0;              // сумма очков

function cost(N: integer): integer;
begin
 case N mod 9 of
  0, 1, 2, 3, 4: Result := N mod 9 + 6;
  5, 6, 7:       Result := N mod 9 - 3;
  8:             Result := 11;
 end;
end;

function card(N: integer): string;
const face: array[0..8] of string =
                        ('Шестерка', 'Семерка', 'Восьмерка',
                         'Девятка', 'Десятка', 'Валет',
                         'Дама', 'Король', 'Туз');
const suit: array [0..3] of string =
                        ('пик', 'треф', 'червей', 'бубен');
begin
 Result := face[N mod 9] + ' ' +suit[N div 9];
end;

procedure TForm1.Button1Click(Sender: TObject);
var i, j, Tmp: word;
begin
 for i := 0 to 35 do A[i] := i;
 for i := 0 to 35 do
 begin
  j := Random(36);
  Tmp := A[i];
  A[i] := A[j];
  A[j] := Tmp;
 end;
 Count := 0;
 Sum := cost(A[0]);
```

```
 Memo1.Clear;
 Memo1.Lines.Add(card(A[0]) + '. Сумма = ' + IntToStr(Sum));
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 Inc(Count);
 Sum := Sum + cost(A[Count]);
 Memo1.Lines.Add(card(A[Count]) + '. Сумма = ' + IntToStr(Sum));
 if Sum > 21
  then begin
   ShowMessage('Перебор !!! Вы проиграли.');
   Button1Click(Sender);
  end
  else if Sum = 21
  then begin
   ShowMessage('Очко !!! Вы выиграли!');
   Button1Click(Sender);
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
var MySum: word;
begin
 Memo1.Lines.Add('Ваша сумма ' + IntToStr(Sum));
 Memo1.Lines.Add('');
 Memo1.Lines.Add('Карты компьютера:');
 MySum := 0;
 repeat
  Inc(Count);
  MySum := MySum + cost(A[Count]);
  Memo1.Lines.Add(card(A[Count]));
 until 22 - MySum < 7;
 Memo1.Lines.Add('Сумма компьютера ' + IntToStr(MySum));
 if MySum > 21
  then ShowMessage('Перебор !!! Компьютер проиграл !')
  else if MySum > Sum then ShowMessage('Компьютер выиграл.')
  else if MySum < Sum then ShowMessage(
                    'Компьютер проиграл. Поздравляю !!!')
  else ShowMessage('Ничья !!!');
  Button1Click(Sender);
end;

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

Рассмотрим приведенный код. Глобальными переменными, помимо массива **A**, объявляются **Count**, в которой фиксируется число взятых из колоды карт, и **Sum**, в которой будет накапливаться сумма очков пользователя.

Функции **cost** и **card** уже были рассмотрены ранее. Процедура **Button1Click** является обработчиком щелчка на кнопке Тасовать. Она реализует рассмотренный ранее алгоритм тасования колоды. Затем сбрасывается на 0 счетчик числа карт

Count, в переменную **Sum** заносится число очков первой карты, рассчитанное функцией **cost**, окно **Memo1** очищается и в него заносится с помощью функции **card** информация о первой сданной карте.

Процедура **Button2Click** является обработчиком щелчка на кнопке Еще. В ней увеличивается процедурой **Inc** счетчик **Count** и информация об очередной сданной карте заносится в переменную **Sum** и в окно **Memo1**. Затем проверяется накопленная сумма очков. Если она превышает 21, пользователь проиграл. Ему показывается соответствующее сообщение, после чего вызывается описанная ранее процедура **Button1Click**, т.е. начинается новая игра. Если сумма набранных очков равна 21, то пользователь извещается о том, что он выиграл, после чего также начинается новая игра.

Процедура **Button3Click** является обработчиком щелчка на кнопке Хватит. Локальная переменная **MySum** служит для накопления суммы очков компьютера. В начале процедуры в окно **Memo1** заносится итог игры пользователя и информация о начале игры компьютера. Счетчик **MySum** обнуляется. Затем в цикле **repeat** набираются карты компьютера. Критерий окончания цикла, записанный в условии **until**, определяет стратегию игры компьютера. Стратегия, реализованная в приведенном коде, основывается на следующих соображениях. В полной колоде сумма очков всех карт равно $4 \cdot (6 + 7 + \ldots + 10 + 2 + 3 + 4 + 11) = 240$. Если разделить эту сумму на число карт (36), то получится математическое ожидание числа очков одной карты. Оно равно 6,(6), т.е. примерно 7. Если сумма очков, уже набранная компьютером, плюс 7 ожидаемых очков превышают 21, то брать очередную карту не надо. Конечно, нетрудно уточнить эту оценку учетом уже взятых компьютером карт. Для этого надо из суммы очков всех карт 240 вычесть сумму уже набранных компьютером очков и разделить результат на число карт 36 минус число карт, взятых компьютером. Попробуйте сами реализовать такую стратегию. Может быть также сверх осторожная стратегия, исключающая перебор. Тогда надо прекращать набор карт при набранных 11 очках. При такой стратегии перебора гарантированно не будет, но и на выигрыш компьютеру рассчитывать вряд ли придется. Могут быть и другие стратегии, учитывающие число карт, набранных партнером. Число, но не сумму очков, так как честный компьютер этой суммы «не знает». Ну и, конечно, может быть масса нечестных стратегий, начиная с «подглядывания» суммы партнера, «подглядывания» очередной карты, и кончая откровенной подтасовкой колоды. Подобные «стратегии» я рассматривать не буду, так как очень не люблю жульничества в программировании, да и в жизни тоже. Поэтому у меня очень скептическое отношение к хакерам, которые многим начинающим программистам кажутся мастерами. За исключением очень немногих ассов, хакеры вовсе не являются сильными программистами. Для взламывания программ и баз данных особого умения не требуется. Есть множество инструментария для этой цели, известны стандартные накатанные методики и приемы. Так что сильный программист, я убежден, в хакеры не пойдет — ему это просто будет скучно. Хакерство — это ремесло, а программирование — творчество и искусство.

Вернемся после этого лирического отступления, к процедуре **Button3Click**. Она завершается достаточно очевидными подсчетами очков, выявлением победителя и переходом к новой игре с помощью вызова процедуры **Button1Click**.

Последняя процедура приведенного кода **FormCreate** является обработчиком события **OnCreate** формы. Это событие, которое возникает один раз в момент создания формы, мы уже использовали в нескольких примерах. В данном случае обработчик этого события рандомизирует генераторы случайных чисел вызовом

функции **Randomize**. Затем вызывается процедура **Button1Click**, начинающая новую игру.

Реализуйте это приложение, опробуйте разные честные стратегии игры. А набравшись опыта, попробуйте реализовать какую-нибудь игру посложнее. Ее создание и выработка соответствующей стратегии будут прекрасным полигоном для продвижения вперед в искусстве программирования.

3.1.5.4 Примеры метода статистических испытаний Монте-Карло

Метод статистических испытаний (метод Монте-Карло) широко используется при решении различных вычислительных задач, при моделировании случайных процессов (в частности, экономических), а также в процессе проектирования различных технических систем и устройств. Рассмотрим пару достаточно простых примеров применения метода Монте-Карло.

Этот метод может использоваться для вычисления определенного интеграла ограниченной функции. Пусть имеется неотрицательная функция f(x), ограниченная на интервале [a, b], и нам надо вычислить интеграл $\int_a^b f(x)dx$. Верхнюю и нижнюю границы функции (не обязательно точные границы) обозначим как F_{max} и F_{min} соответственно. Применение метода Монте-Карло для решения этой задачи поясняет рис. 3.2. Значения a, b, F_{max} и F_{min} выделяют прямоугольник, внутри которого лежит функция. Площадь этого прямоугольника равна $(F_{max} - F_{min}) \cdot (b - a)$. Генерируется достаточно большое число N случайных чисел, равномерно распределенных в этом прямоугольнике. При этом фиксируется число экспериментов N_1, в которых точка оказалась ниже линии f(x). Очевидно, что при большом N отношение площади прямоугольника под кривой f(x) ко всей площади прямоугольника равно N_1 / N. Поэтому за оценку площади прямоугольника под кривой можно принять значение $(F_{max} - F_{min}) \cdot (b - a) \cdot N_1 / N$. Интеграл, который надо вычислить, равен этой площади плюс площадь под прямоугольником, равная $F_{min} \cdot (b - a)$.

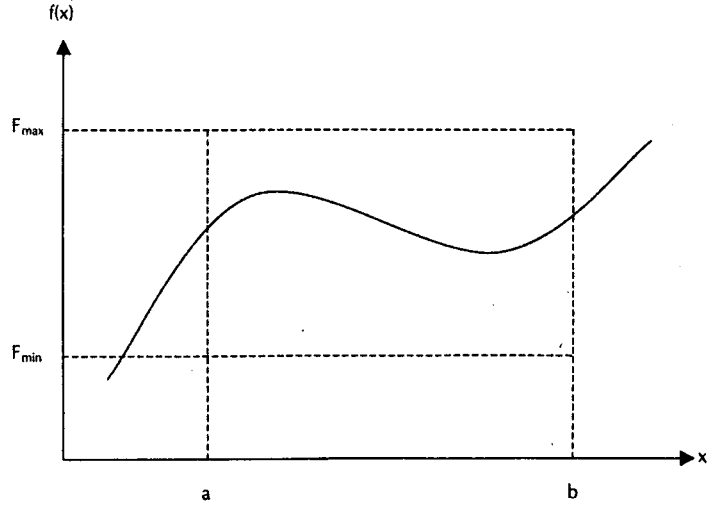


Рис. 3.2
Иллюстрация вычисления определенного интеграла методом Монте-Карло

Точность расчета интеграла зависит от числа экспериментов N. Для достижения приемлемой точности число экспериментов должно быть не менее 100, а желательно еще больше.

Реализация функции расчета определенного интеграла методом Монте-Карло при постоянном объеме выборки N может быть оформлена, например, следующим образом:

```
type TFunc = function(X: real): real;

function Integr(a, b, Fmin, Fmax: real; Func: TFunc): real;
const N = 10000;
var   N1, i: integer;
begin
 N1 := 0;
 for i:= 1 to N do
 begin
  if Func(a + (b - a) * Random) >= Fmin + (Fmax - Fmin) * Random
   then Inc(N1);
 end;
 Result := ((Fmax - Fmin) * N1 / N + Fmin) * (b - a);
end;
```

Объявление типа **TFunc** функции позволяет передавать имя функции в качестве параметра. С этим вы уже знакомы по реализации метода дихотомии в разд. 2.8.7.4. Параметры **a**, **b**, **Fmin**, **Fmax** определяют интервалы по аргументу и функции. Переменная **N1** используется для накопления числа экспериментов, в которых случайная точка расположена ниже функции. Сравнение функции со случайной точкой осуществляется условием оператора **if**. В этом условии вычисляется значение функции **Func** при случайном значении аргумента в интервале **[a, b]**. Генерация таких случайных действительных чисел рассмотрена в разд. 3.1.5.2. Далее это значение функции сравнивается с другим случайным числом в интервале **[Fmin, Fmax]**. Если значение функции больше этого числа, то счетчик **N1** увеличивается на 1. После окончания цикла значение интеграла определяется по формуле, рассмотренной ранее.

В качестве функции типа **TFunc** может использоваться любая неотрицательная ограниченная на заданном интервале аргумента функция. Например, функция exp(x) / x, интеграл от которой, как известно, выражается только бесконечным рядом. Расчет интеграла от этой функции на любом интервале, не включающем 0 (при x = 0 функция неограниченна), может быть оформлен так:

```
function F(X: real): real;
begin
 Result := exp(X) / X;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 Label1.Caption := FloatToStr(Integr(StrToFloat(Edit1.Text),
             StrToFloat(Edit2.Text), StrToFloat(Edit3.Text),
             StrToFloat(Edit4.Text), F));
end;
```

Тут предполагается, что интервал по аргументу и границы функции заданы в соответствующих окнах редактирования **Edit**. В функциях, подобных приведен-

ной, а также в любых монотонных функциях известно, что максимальные значения достигаются на краях заданного интервала по аргументу. Так что можно избавить пользователя от необходимости задавать границы функции, проведя предварительное тестирование функции на краях интервала:

```
Label1.Caption := FloatToStr(Integr(StrToFloat(Edit1.Text),
     StrToFloat(Edit2.Text), 0.,
     Max(F(StrToFloat(Edit1.Text)), F(StrToFloat(Edit2.Text))),
     F));
```

В этом операторе аргумент **Fmin** задается равным 0, а в качестве **Fmax** указывается максимальное из граничных значений функции. Только для использования подобного оператора надо подключить к приложения оператором **uses** модуль *Math*, в котором объявлена функция **Max**.

Вы создали полезную функцию, и, может быть, имеет смысл включить ее в вашу библиотеку, созданную в разд. 2.8.7.4. Впрочем, как известно, для вычисления определенных интегралов функций одного аргумента есть значительно более эффективные методы: прямоугольников, трапеций, парабол (Симпсона). Так что если в дальнейшем вам потребуется вычислять подобные интегралы, то имеет смысл реализовать и их. Но вычисление определенных интегралов методом Монте-Карло нетрудно распространить на многомерный случай. И тут, пожалуй, он становится самым эффективным. Если надо рассчитать интеграл от функции $f(x_1, x_2, ..., x_m)$ по многомерному гиперпараллелепипеду $a_i < x_i < b_i$ при $i = 1, ..., m$, то процедура вычисления аналогична рассмотренной для одномерного случая. Надо только предусмотреть массивы **A**, **B** и **X** для хранения векторов аргумента и интервалов.

```
type TFuncM =,function(X: array of real): real;

function IntegrM(A, B: array of real; Fmin, Fmax: real;
                 Func: TFuncM): real;
// Расчет определенного интеграла в гиперпараллелепипеде
// с пределами a и b от функции Func.
// Функция неотрицательна и ограничена, границы Fmin и Fmax.
const N = 10000;
var   N1, M, i, j: integer;
      X: array of real;
begin
 M := High(A);
 SetLength(X, M+1);
 N1 := 0;
 for i:= 1 to N do
 begin
  // Задание случайного вектора X
  for j := 0 to M do
   X[j] := A[j] + (B[j] - A[j]) * Random;
  // Проверка соотношения Func(X) и случайной точки
  if Func(X) >= Fmin + (Fmax - Fmin) * Random
   then Inc(N1);
 end;
 Result := ((Fmax - Fmin) * N1 / N + Fmin);
 for j := 0 to M do
  Result := Result * (B[j] - A[j]);
end;
```

В этом коде объявлен тип функции **TFuncM**, принимающей открытый массив действительных чисел (см. разд. 3.1.2) и возвращающей действительное число. Это функция, от которой берется интеграл. А функция вычисления интеграла **IntegrM** отличается от рассмотренной ранее функции **Integr** следующим. Границы по координатам она принимает в виде открытых массивов **A** и **B** действительных чисел. В функции вводится локальная переменная **X**, в которой формируется случайный вектор аргументов. Переменная **X** является динамическим массивом (см. разд. 3.1.3.1). Размер этого массива определяется размером массивов границ **A** и **B**. Вместо задания одного случайного значения аргумента в функции **Integr**, в данном случае задается в цикле случайный вектор. Результат расчета умножается на произведение длин интервалов по отдельным координатам. А сам алгоритм расчета тот же, что в функции **Integr**. Так что вы, вероятно, без особого труда в нем разберетесь.

Теперь рассмотрим применение метода Монте-Карло в моделировании и проектировании. На любую систему — биологическую, экономическую, техническую всегда влияют случайные воздействия. Поэтому состояние любой системы всегда в определенной мере случайно. Игнорирование этой вероятностной природы систем может привести к грубым ошибкам при их моделировании и прогнозе их развития. А при проектировании технической системы игнорирование разброса ее выходных параметров может привести к созданию неработоспособного или ненадежного устройства.

Применение метода Монте-Карло в моделировании сводится к тому, что состояние системы многократно рассчитывается при различных значениях случайных воздействий. Законы распределения этих воздействий считаются известными. В результате такого моделирования можно определить вероятности того или иного состояния системы.

Рассмотрим несколько условный простенький пример, так как области интересов читателей данной книги могут быть очень различны, и трудно подыскать пример реальной системы, который был бы понятен и интересен всем. Пусть имеется некоторый сигнал, который надо усилить и передать через какую-то линию связи. Линия связи вносит затухание, так что после прохождения через нее сигнал ослабляется. В итоге коэффициент передачи сигнала T через усилитель и линию связи равен $K \bullet F$, где K — коэффициент усилителя, F — коэффициент передачи линии, причем $F < 1$. Характеристики линии связи (значение F) заданы, и изменять их невозможно. А усилитель проектируется. Причем его коэффициент усиления K должен быть таким, чтобы общий коэффициент передачи системы был не меньше заданного значения $T_{мин}$, т.е. надо обеспечить выполнение неравенства $T = K \bullet F \geq T_{мин}$. Так как увеличение коэффициента усиления K связано с удорожанием и повышением нестабильности системы, излишний запас по K нежелателен.

В детерминированной постановке эта задача тривиальна. Из приведенных соотношений следует, что коэффициент усиления проектируемого усилителя K надо выбрать равным $T_{мин} / F$. Но это было бы неверное решение. Положим, что проектируемая система будет многократно тиражироваться, так что надо обеспечить ее надежную работу во всех ситуациях. Коэффициент передачи F в каждом экземпляре линии, естественно, будет разным из-за множества случайных факторов, влияющих на него. Предположим, что его случайные значения подчиняются нормальному закону распределения с математическим ожиданием, равным той средней величине F, для которой проводится расчет, и с некоторым достаточно большим средним квадратическим отклонением. Коэффициент усиления тоже будет в раз-

ных экземплярах усилителя различным, так как все элементы, из которых построен усилитель, имеют разброс параметров. Правда, усилители могут проходить двухстороннюю отбраковку по коэффициенту усиления на выпускающем их заводе. В этом случае закон распределения их коэффициента усиления близок к равномерному в пределах некоторого поля допусков.

В силу описанного разброса параметров линии связи и усилителя окажется, что если выбрать $K = T_{мин} / F$, подразумевая под K и F математические ожидания (средние значения), то в 50% случаев система будет неработоспособна, т.е. ее коэффициент передачи окажется меньше $T_{мин}$. Вряд ли потребители будут благодарны проектировщику за такой усилитель. Значит, прежде всего, встает задача моделирования: определить для выбранного проектировщиком коэффициента K вероятность работоспособности системы. Но неплохо бы было решить и вторую задачу — задачу проектирования: по заданной, достаточно большой вероятности работоспособности найти требуемое значение K.

Попробуем решить обе эти задачи. Форма соответствующего приложения, может выглядеть так, как показано на рис. 3.3. Сразу должен сказать, что ее внешний вид крайне неудовлетворительный. Но мы еще не умеем проектировать красивый и удобный графический интерфейс пользователя. Эти вопросы будут рассмотрены в гл. 5. Так что пока сделайте так, как умеете, но постарайтесь, насколько возможно, спроектировать форму аккуратно и удобно для пользователя.

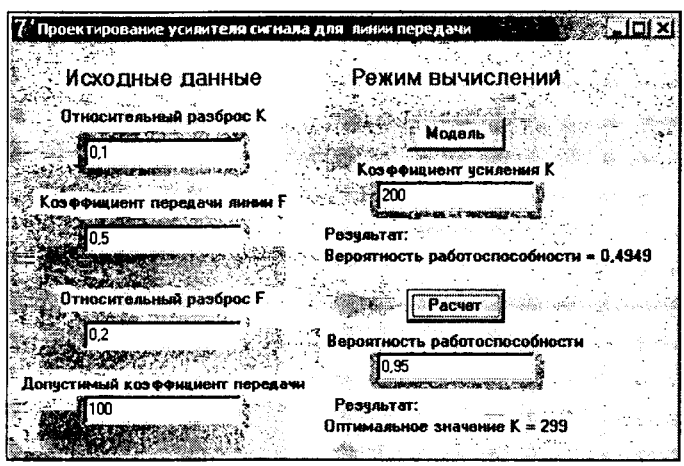


Рис. 3.3
Приложение для моделирования и проектирования усилителя

Исходными данными для расчета, задаваемыми в соответствующих окнах редактирования, являются относительный разброс коэффициента усиления усилителя, среднее значение коэффициента передачи линии, относительная величина его среднего квадратического отклонения, минимально допустимый коэффициент передачи системы. Кроме того, при решении задачи моделирования, осуществляемом щелчком на кнопке Модель, задается коэффициент усиления. А при решении задачи проектирования, осуществляемом щелчком на кнопке Расчет, задается вероятность работоспособности системы.

Ниже приведен код этого приложения.

```
uses math, MyLib1;
var K, DK, F, DF, Tmin, Pdop: double;

function Model(K, F: double): double;
// Модель системы: расчет T = K * F
begin
 Result := K * F;
end;

function P: real;
// Расчет вероятности удовлетворения системой
// технического задания: T = K * F > Tmin
const N = 10000;
var   N1, i: integer;
begin
 N1 := 0;
 for i:= 1 to N do
 begin
  if Model(K * (1 + DK * (2. * Random - 1)), RandG(F, F * DF)) > Tmin
   then Inc(N1);
 end;
 Result := N1 / N;
end;

procedure TForm1.Button1Click(Sender: TObject);
// Моделирование: расчет вероятности работоспособности системы
// при заданном пользователем коэффициенте усиления K
begin
 K := StrToFloat(Edit1.Text);
 DK := StrToFloat(Edit2.Text);
 F := StrToFloat(Edit3.Text);
 DF := StrToFloat(Edit4.Text);
 Tmin := StrToFloat(Edit5.Text);
 Pdop := StrToFloat(Edit6.Text);
 Label1.Caption := 'Вероятность работоспособности = ' +
                   FloatToStr(P);
end;

function Fun(X: real): real;
// Функция левой части уравнения в методе дихотомии
begin
 K := X;
 Result := P - Pdop;
end;

procedure TForm1.Button2Click(Sender: TObject);
// Решение уравнения P(X) - Pdop = 0 методом дихотомии
begin
 DK := StrToFloat(Edit2.Text);
 F := StrToFloat(Edit3.Text);
 DF := StrToFloat(Edit4.Text);
 Tmin := StrToFloat(Edit5.Text);
 Pdop := StrToFloat(Edit6.Text);
 Label2.Caption := 'Оптимальное значение K = ' +
                   FloatToStr(Round(Dihot(200, 1000, 1,Fun)));
end;
```

Рассмотрим этот код. Предложение **uses** подключает модуль *Math*, в котором описана используемая в приложении функция **RandG**, и разработанную в разд. 2.8.7.4. вашу библиотеку *MyLib1*, так как приложение использует описанную в ней функцию метода дихотомии.

Глобальные переменные **K**, **DK**, **F**, **DF**, **Tmin** и **Pdop** хранят соответственно заданные пользователем среднее значение коэффициента усиления усилителя (в задаче моделирования) и его относительный разброс, среднее значение коэффициента передачи линии и относительную величину его среднего квадратического отклонения, минимально допустимый коэффициент передачи системы и вероятность работоспособности системы (в задаче проектирования).

Функция **Model** описывает модель системы. В нашем примере она принимает в качестве параметров значения коэффициента усиления **K** и коэффициента передачи линии **F**, и возвращает выходной параметр системы — полный коэффициент передачи. Обратите внимание на то, что параметр **K** в функции и глобальная переменная **K** имеют разный смысл и значения. Параметр **K** — это случайное значение коэффициента усиления в каком-то экземпляре системы. А глобальная переменная **K** — это среднее значение коэффициента усиления спроектированного усилителя.

Функция **P** рассчитывает вероятности удовлетворения системой технического задания: T = K • F > Tmin. Функция подобна той, которая использовалась в приведенных ранее примерах применения метода Монте-Карло. Она как бы моделирует производство 10000 штук систем, содержащих усилитель и линию связи, и определяет, какова при этом доля работоспособных систем. Общий коэффициент передачи для каждого экземпляра системы определяется вызовом функции **Model**, в которую передаются случайные значения коэффициента усиления усилителя и коэффициента передачи линии связи. Закон распределения коэффициента усиления принят равномерный в интервале K • (1 ± DK), где K — среднее значение коэффициента усиления, DK — заданный пользователем относительный разброс. Закон распределения коэффициента передачи линии связи принят нормальным со средним значением F и средним квадратическим отклонением равным F • DF, где DF — заданная пользователем относительная величина, характеризующая разброс.

Процедура **Button1Click** является обработчиком щелчка на кнопке Модель. В этой процедуре сначала читаются и заносятся в глобальные переменные данные, введенные пользователем в окна редактирования. А затем в метку **Label1** заносится значение, которое возвращает функция **P**. Как видно на рис. 3.3, коэффициент 200, следующий из детерминированной постановки задачи, обеспечит работоспособность только примерно 50% систем.

Функция **Fun** относится к решению задачи проектирования: найти среднее значение коэффициента усиления, при котором вероятность работоспособности системы P(K) равна заданной величине Pdop. Для решения этой задачи надо найти корень уравнения P(K) = Pdop. Это уравнение можно решить относительно K методом дихотомии, который был реализован в разд. 2.8.7.4. Для этого уравнение записывается в виде P(K) – Pdop = 0. Левую часть этого уравнения рассчитывает функция **Fun**, объявление которой соответствует объявленному в разд. 2.8.7.4 типу **TFunc**. Параметром **X**, передаваемым в функцию, является текущее приближение значения коэффициента усиления. Как видим, в данном примере выражение в левой части уравнения не описывается аналитически. Оно может быть вычислено только методом Монте-Карло, реализованным рассмотренной ранее функцией **P**. Поэтому в функции **Fun** сначала задается значение глобальной переменной **K**, равное переданному

в **Fun** параметру **X**. А затем вызывается функция **P** и формируется значение левой части уравнения.

Процедура **Button2Click** является обработчиком щелчка на кнопке Расчет. В этой процедуре сначала читаются и заносятся в глобальные переменные данные, введенные пользователем в окна редактирования. А затем в метку **Label2** заносится значение, которое возвращает функция **Dihot**, содержащаяся в вашей библиотеке *MyLib1*. Как видно на рис. 3.3, оптимальный коэффициент усиления равен примерно 300, т.е. в полтора раза больше, чем следовало из детерминированной постановки задачи. В качестве аргументов в функцию **Dihot** передаются ожидаемые пределы коэффициента усиления (от 200 до 1000) и погрешность вычисления — 1. Обратите внимание на использованную в последнем операторе процедуры функцию **Round**, округляющую результат выполнения процедуры **Round**. Если бы результат не округлялся, пользователю был бы показан ответ 299,21875, т.е. 8 значащих цифр результата. Но это было бы неграмотно, так как вы задаете в методе дихотомии абсолютную погрешность 1. Следовательно, результат можно представлять только с точность до целого значения.

Хороший стиль программирования

Результаты расчета следует предоставлять пользователю с той точностью, которая обеспечивается в алгоритме расчета. Учтите, что все операции с действительными числами выполняются с округлением. Так что если вычисление связано с большим количеством арифметических операций, то точность вычислений может не превышать 5–6 десятичных знаков. Поэтому выдавать результаты с большим числом знаков — это значит вводить пользователя в заблуждение. К тому же, представление результатов с большим числом десятичных знаков обычно неудобно пользователю. Возможности форматирования вывода чисел рассмотрены в разд. 3.2.1.

Реализуйте это приложение, опробуйте его в работе. А затем реализуйте подобное приложение для какой-то системы из той области знаний, которая вам ближе. Изменения, которые вам придется сделать в рассмотренном коде, в основном сведутся к следующему.

В общем случае у вас будет не 2, как в рассмотренной задаче, а N случайных параметров. Их средние значения целесообразно хранить в N-мерном массиве. В другом массиве того же размера могут храниться характеристики относительного разброса случайных параметров. Если для каких-то параметров используется нормальный закон распределения, а для других — равномерный, то можно создать еще один массив того же размера с элементами булева типа, причем **true**, например, будет означать нормальный закон распределения соответствующего параметра, а **false** — равномерный закон.

В функции **P** должен формироваться массив случайных значений параметров примерно так же, как это делалось в функции **IntegrM** при вычислении многомерного интеграла, но по формулам, реализованным в рассмотренной функции **P**. При вызове функции **Model** в нее должен передаваться этот массив случайных значений, а не два параметра, как в рассмотренном примере. Исходя из переданных значений, функция **Model** должна рассчитывать выходную характеристику системы. Это все изменения, которые надо сделать для режима моделирования. А режим расчета оптимального значения одного из параметров практически ничем не отли-

чается от рассмотренного. Только в функции **Fun** параметр **X** должен заноситься не в переменную **K**, а в соответствующую позицию массива средних значений параметров.

3.2 Строки и тексты

3.2.1 Форматирование строк

Строки представляют собой последовательность символов. Начнем мы рассмотрение строк с описания строковых констант, хотя вы и знакомы с ними, поскольку практически во всех уже созданных приложениях использовали их. Как вы знаете, значения строковых констант заключаются в одинарные кавычки и могут содержать последовательность любых символов. Если в текст строки надо включить символ одинарной кавычки, то он повторяется дважды. Например

```
const MyFile = 'Имя моего файла ''Mf.txt'''
```

Объявленная таким образом константа соответствует тексту «Имя моего файла 'Mf.txt'». Впрочем, чтобы не путаться, я бы советовал в подобных случаях использовать внутри текста не одинарные, а двойные кавычки.

Если строковая константа состоит из последовательности нескольких строк (например, строка настолько длинная, что при кодировании ее желательно разбить на несколько строчек), то отдельные строки могут объединяться операцией "+"; Например, оператор:

```
const C= 'Это начало строки, ' +
         'а это ее продолжение';
```

объявляет константу, содержащую текст «Это начало строки, а это ее продолжение».

В значение строковой константы можно включать как печатаемые, так и непечатаемые символы, указывая после символа "#" десятичное или шестнадцатеричное число от 0 до 255, соответствующее коду ASCII нужного символа. Коды некоторых символов приведены в табл. 2.4 в разд. 2.6. Например, обозначение #13 соответствует символу перевода строки, а обозначение #9 — символу табуляции.

В записи константы можно чередовать записи в кавычках и записи с символом #. Например, константа

```
'строка 1'#13'строка 2'
```

соответствует двум строкам:

```
строка 1
строка 2
```

Обратите внимание на то, что благодаря символам #13 понятие строки в языке программирования несколько отличается от этого понятия в обычном разговорном языке: одна строка может отображаться несколькими строчками текста.

Ниже приведен более сложный пример, отображающий функцией **ShowMessage** диалоговое окно с табулированным текстом:

```
ShowMessage('Символ'#9#9'Индекс'#13'Нулевой'#9'0'#13+
            'BackSpace'#9'8'#13'Tab'#9#9'9'#13'Enter'#9#9'13');
```

Этот оператор показывает пользователю диалоговое окно, представленное на рис. 3.4. Учтите только, что символ табуляции сдвигает очередной символ к очередной позиции табуляции, заданной по умолчанию. Поэтому для хорошего выравнивания таблицы, столбцы которой содержат строки разной длины, иногда после строк меньшей длины надо указывать несколько подряд символов табуляции, как сделано в приведенном примере. Поэкспериментируйте с этим примером, и вы поймете принцип задания символов табуляции.

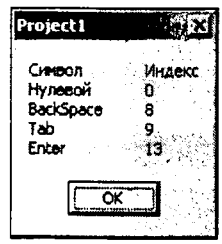


Рис. 3.4
Пример вывода табулированного текста

Кстати, такие строки, состоящие из нескольких строчек, можно отображать и в метках **Label**. Только для этого надо установить в **false** свойство **AutoSize**, устанавливающее автоматическую адаптацию размеров метки к размерам отображаемой надписи. После этого надо увеличить размеры метки так, чтобы в ней могло поместиться несколько строк. Тогда оператор

```
Label4.Caption := 'Символ'#9#9'Индекс'#13'Нулевой'#9'0'#13+
                  'BackSpace'#9'8'#13'Tab'#9#9'9'#13'Enter'#9#9'13';
```

обеспечит отображение в метке такой же таблицы, какую вы видите на рис. 3.4. Впрочем, если в дополнение к **AutoSize = false** вы установите в **true** свойство **WordWrap**, задающее перенос по словам, то длинный текст, превышающий ширину метки, будет автоматически размещаться в нескольких строках. При этом значение свойства **Alignment** будет определять выравнивание текста внутри метки: **taLeftJustify** — влево, **taRightJustify** — вправо, **taCenter** — по центру. Опробуйте эти варианты на какой-нибудь длинной строке текста.

Если в строку, отображаемую в диалоговом окне, в метке или в окне редактирования, необходимо вставить числа, это может осуществляться функциями **IntToStr** и **FloatToStr**. Например, если **X** — действительная переменная, представляющая результат каких-то вычислений, представить ее пользователю можно оператором:

```
Label1.Caption := 'Результат расчета: ' + FloatToStr(X);
```

Но у этого оператора есть один недостаток: результат будет представлен с точностью в 15 значащих цифр, если, конечно, они есть в числе. Можете проверить это, задав, например, оператор:

```
X := Pi;
```

Правда, число π, возвращаемое функцией **Pi**, имеет 20 достоверных значащих цифр. Так что если вы хотите предоставить пользователю возможность познакомиться с этим числом, то результат, даваемый функцией **FloatToStr**, пожалуй, недостаточно точен. А если **X** — это действительно результат расчета, то 15 значащих цифр — это слишком много. В разд. 3.1.5.4 уже говорилось о том, что нехорошо

представлять пользователю результат с точностью, превышающей реальную точность расчета. Хотелось бы иметь инструмент, позволяющий управлять форматом преставления чисел.

Таким инструментом является функция **Format**, которая объявлена в модуле *Sysutils* следующим образом:

```
function Format(const SFormat: string;
                const Args: array of const): string
```

Функция **Format** возвращает отформатированную строку, представляющую собой результат применения строки описания формата **SFormat** к открытому массиву аргументов (см. разд. 3.1.2) **Args**. Параметр **Args** задается как заключенный в квадратные скобки список аргументов, разделяемых запятыми.

Строка описания формата **SFormat** определяет способ форматирования аргументов **Args**. Она содержит в общем случае два типа объектов: обычные символы, которые непосредственно копируются в форматированную строку, и спецификаторы формата, которые определяют формат записи в результирующую строку списка аргументов.

Каждый спецификатор формата имеет вид

```
%[<индекс>:][-][<ширина>][.<точность>]<тип>
```

Спецификатор формата начинается с символа "%". Затем без пробелов следует ряд необязательных полей:

[<индекс>:]	определяет индекс (номер) аргумента в заданном списке, к которому относится данный спецификатор формата
[–]	индикатор выравнивания влево
[<ширина>]	устанавливает ширину поля
[.<точность>]	спецификатор точности

Затем так же без пробела записывается единственное обязательное поле <тип>, определяющее, как и в каком формате будет интерпретироваться аргумент.

Ниже приводится таблица некоторых *спецификаторов типа*.

d	Десятичное целое. Значение преобразуется в строку десятичных цифр. Если спецификатор формата содержит поле точности, то результирующая строка должна содержать количество цифр, не менее указанного значения. Если форматируемое значение содержит меньше цифр, оно дополняется слева нулями.
e	Научный формат действительного значения. Значение преобразуется в формат вида "–d,ddd...E+ddd", где "d" означает цифру. Иначе говоря, число представляется в виде $-d,ddd...\cdot 10^{+ddd}$. Отрицательные числа начинаются со знака "–". Перед десятичной запятой всегда имеется одна цифра. Общее число цифр (включая цифру перед запятой) равно числу, указанному спецификатором точности. По умолчанию (в отсутствие спецификатора точности) точность равна 15. После символа "E" обязательно следует знак "+" или "–" и не менее трех цифр.
f	Формат с фиксированной запятой действительного значения. Значение преобразуется в формат вида "–ddd.ddd...". Отрицательные числа начинаются со знака "–". Число цифр после десятичной запятой равно числу, указанному спецификатором точности. По умолчанию (в отсутствие спецификатора точности) точность равна 2.

n	Формат, подобный формату с фиксированной запятой для действительных чисел, но отличающийся наличием в результирующей строке разделителей тысяч. Иными словами, число представляется в форме "–d,ddd,ddd.ddd...".
g	Обобщенный формат действительных чисел. Значение преобразуется в формат научный или с фиксированной запятой, в зависимости от того, какой из них дает более короткую запись. В научном формате общее число цифр (включая цифру перед запятой) равно числу, указанному спецификатором точности. По умолчанию (в отсутствие спецификатора точности) точность равна 15. Заключающие нули после десятичной запятой в результирующей строке отбрасываются, а сама десятичная запятая появляется, только если это необходимо. Результирующая строка использует формат с фиксированной запятой, если число цифр слева от десятичной запятой не превышает заданной точности и если значение числа не меньше 0.00001. В остальных случаях используется научный формат. Обобщенный формат действительных чисел наиболее удобен в большинстве случаев.
s	Формат строки для аргументов вида символ или строка. Строка или символ просто вставляются в результирующую строку. Если задан спецификатор точности, то он определяет максимальное число вставляемых символов. Если вставляемая строка длиннее, она усекается.

Все спецификаторы типа могут записываться как строчными, так и прописными буквами.

Спецификатор индекса задает порядковый номер аргумента в списке, к которому применяется преобразование. Индекс первого аргумента считается равным 0. Применение индексов позволяет пропускать какие-то аргументы в списке или форматировать один и тот же аргумент несколько раз.

Спецификатор ширины задает минимальное число символов результата данного преобразования. Если результирующая строка короче заданной ширины, то лишние позиции заполняются пробелами. По умолчанию используется выравнивание вправо и пробелы ставятся перед преобразованным значением. Но если указан индикатор левого выравнивания (символ "–" перед спецификатором ширины), то пробелы вносятся после преобразованного значения.

Действие *спецификатора точности* на различные форматы уже рассмотрено выше.

Во всех форматах действительных чисел символы, используемые для отделения целой части числа от дробной и для разделения тысяч, берутся из глобальных переменных **DecimalSeparator** и **ThousandSeparator**.

Посмотрим применение функции **Format** на ряде примеров. Если переменная X равна числу π, то оператор

```
Label1.Caption := Format('Результат расчета: %g', [X]);
```

даст результат: «Результат расчета: 3,14159265358979». Это такой же результат, который давал приведенный ранее оператор, использующий функцию **FloatToStr**. Но если вы выполните оператор

```
Label3.Caption := Format('Результат расчета: %.5g', [X]);
```

то результат будет следующий: «Результат расчета: 3,1416». Вы ограничили точность результата пятью значащими цифрами, и именно такой результат получили. В этом и состоит преимущество функции **Format**: вы можете управлять точностью и выполнять рекомендацию, данную в разд. 3.1.5.4 — представлять пользователю результат с точностью, соответствующей реальной точности расчета. В приведен-

ном примере вы можете заметить, что отображаемое число не просто усечено до заданного числа цифр, но округлено до единицы последнего отображаемого разряда.

Формат

```
Format('%e %0:f %0:n %0:g', [10000.])
```

дает результат

```
1,00000000000000E+004 10000,00 10 000,00 10000
```

а формат

```
Format('%e %0:f %0:g', [0.0001])
```

дает результат

```
1,00000000000000E-004 0,00 0,0001
```

В этих двух примерах спецификатор индекса использован, чтобы отобразить одно и то же число в разных форматах. Примеры показывают, что применение формата **e** с точностью по умолчанию дает очень неудобную форму представления. Второй из примеров показывает, что применение формата **f** с точностью по умолчанию (2) к числам, меньшим 1, может отсечь значащие цифры. Наиболее удобен во всех случаях формат **g**.

3.2.2 Обработка строк

В Delphi имеется несколько типов строк, но мы остановимся, в основном, только на одном из них: **AnsiString**. Вместо этого идентификатора можно использовать другой, эквивалентный ему — **String**. При принятых по умолчанию настройках компилятора Delphi эти два типа идентичны. Так что, пользуясь во всех предыдущих примерах типом **String**, мы неявным образом использовали тип **AnsiString**.

Тип **AnsiString (String)** обеспечивает очень удобные операции со строками. Переменные этого типа могут хранить огромные строки — до 2 Гбайт. Вы уже знаете, что это могут быть не строки в обыденном понимании, а длинные тексты, содержащие, если необходимо, внутри себя символы перехода к новой строке.

Переменная типа **AnsiString** является указателем на динамически выделяемую область памяти, в которой хранится строка. Сама строка представляет собой массив символов с индексами, начинающимися с 0. После значащих символов строки в ней хранится нулевой символ #0, указывающий на ее окончание.

Переменные типа **AnsiString** объявляются обычным образом:

```
var S1: AnsiString;
    S2: string;
```

Объявление глобальных переменных или типизированных констант можно сочетать с их инициализацией, причем нулевой символ в конце строки добавляется автоматически, так что думать о нем не приходится:

```
var S1: AnsiString = 'текст';
    S2: string = 'начальный текст';
```

Иногда в объявлении после ключевого слова **String** используют указание числа символов в квадратных скобках. Например:

```
var S: string[4];
```

Это уже другой тип строк — **ShortString** (короткие строки). Длина этих строк не может превышать 255 символов. Хранятся они не так, как строки типа **AnsiString**. В них нет завершающего нулевого символа, символы индексов начинаются с 1, а под нулевым индексом хранится число символов. Так что, изменяя значение **S[0]**, можно изменять размер памяти, отведенный под строку. Короткие строки — это устаревший тип, сохраняемый в Object Pascal только для обратной совместимости с прежними версиями Pascal. Так что далее этот тип мы не будем рассматривать.

Вернемся к строкам типа **AnsiString**. Доступ к отдельным символам строки осуществляется как к символьному массиву по индексам. Индексы отсчитываются от 1 (обратите на это внимание, так как в некоторых других типах строк индексы отсчитываются от 0). Например, для объявленной выше строки **S1** выражение **S1[1]** возвращает значение первого символа строки 'т', **S1[2]** — второго и т.д. Если индекс превышает число символов в строке, возвращается нулевой символ '#0'.

Функция **Length** возвращает число символов строки, не считая нулевого символа. Так что для приведенной выше переменной **S1** выражение **Length(S1)** вернет 5.

Процедура **SetLength**, объявленная как

```
procedure SetLength(var S; NewLength: Integer);
```

задает новый размер **NewLength** строки **S**. Если **NewLength** меньше числа символов в строке, то текст строки усекается до указанного числа символов. Если **NewLength** больше числа символов в строке, то прежний текст сохраняется, а после него записываются нулевые символы. Впрочем, такое увеличение длины строки для типа **AnsiString** не имеет смысла. При каждом присваивании переменной нового значения длина строки автоматически делается равной длине присвоенного текста. Например, оператор

```
S1 := 'Новый текст';
```

изменит длину строки **S1** до 11 символов, не считая нулевого.

Сцепление текстов двух строк типа **AnsiString** осуществляется операцией "+". Например, операторы

```
var S1, S2, S: AnsiString;
...
S1 := 'Текст 1';
S2 := 'Текст 2';
S := S1 + ' и ' + S2;
```

занесут в **S** строку: «Текст 1 и Текст 2».

Для обработки строк типа **AnsiString** имеется ряд библиотечных функций. Полный их список и описания вы можете найти во встроенной справке Delphi или в справке [3]. Рассмотрим основные функций обработки строк.

При анализе текстовых строк часто надо найти в одной из строк фрагмент текста, заданный в другой строке. Этот фрагмент, например, может быть некоторым ключевым словом, символом и т.п. Эту задачу позволяет решить функция

```
function Pos(Substr: string; S: string): Integer;
```

которая ищет в строке **S** первое вхождение текста строки **Substr** и, если поиск прошел удачно, возвращает указатель на первый символ этого вхождения. Если же текст не был найден, возвращается нуль.

Сделайте, например, приложение, содержащее кнопку **Button1**, многострочное окно редактирования **Memo1**, окно **Edit1** и метку**Label1**. Окно **Memo1** содержит некоторый текст. При нажатии **Button1** требуется найти в этом тексте сочетание символов, введенное пользователем в окно **Edit1**, и отобразить в метке **Label1** результаты поиска. Это можно сделать следующим обработчиком щелчка на кнопке:

```
procedure TForm1.Button1Click(Sender: TObject);
var i:integer;
begin
 i := Pos(Edit1.Text, Memo1.Text);
 if i > 0
  then Label1.Caption := 'Текст найден в позиции ' + IntToStr(i)
  else Label1.Caption := 'Текст не найден';
end;
```

В этом коде использованы функция **Pos** и известная вам функция **IntToStr**. В результате поиска в метке отображается текст вида «Текст найден в позиции 5» или «Текст не найден».

В приведенном коде поиск происходил с учетом регистра. Если поиск надо проводить без учета регистра, можно воспользоваться функциями **AnsiLowerCase** или **AnsiUpperCase**, приводящими строки соответственно к нижнему или верхнему регистру. Для этого в приведенном выше коде достаточно заменить оператор вычисления индекса вхождения подстроки оператором:

```
i := Pos(AnsiLowerCase(Edit1.Text), AnsiLowerCase(Memo1.Text));
```

или

```
i := Pos(AnsiUpperCase(Edit1.Text), AnsiUpperCase(Memo1.Text));
```

В обоих случаях текст и подстрока будут приведены к одному регистру и, следовательно, поиск будет осуществляться без учета регистра.

Теперь давайте решим более сложную задачу. Пусть нам надо найти в тексте **Memo1.Text** первое вхождение текста **Edit1.Text** и, если поиск прошел удачно, то заменить найденный фрагмент на текст, содержащийся в компоненте **Edit2.Text**. Иначе говоря, требуется произвести контекстную замену в **Memo1.Text** текста **Edit1.Text** на текст **Edit2.Text**.

Для решения подобных задач можно воспользоваться процедурами **Delete** и **Insert**, позволяющими соответственно удалять и вставлять подстроку. Процедуры определены следующим образом:

```
procedure Delete(var S: string; Index, Count:Integer)
procedure Insert(Source: string; var S: string; Index: Integer);
```

Процедура **Delete** удаляет из строки **S** подстроку, начинающуюся с индекса **Index** и содержащую **Count** символов. А процедура **Insert** вставляет в строку **S** подстроку **Source**, начиная с индекса **Index**. Символы, расположенные в строке выше, сдвигаются к концу строки.

Эти процедуры позволяют осуществить контекстную замену следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var i:integer;
    S: string;
begin
 S:= Memo1.Text;
```

```
 i := Pos(Edit1.Text, S);
 if i > 0
 then begin
   Delete(S, i, Length(Edit1.Text));
   Insert(Edit2.Text, S, i);
 end;
 Memo1.Text := S;
end;
```

Если вхождение текста **Edit1.Text** в **Memo1.Text** найдено, то процедура **Delete** удаляет его, используя функцию **Length** для определения числа удаляемых символов, а процедура **Insert** вставляет на место удаленного текста строку **Edit2.Text**. Некоторое усложнение приведенного кода получилось за счет введения локальной переменной **S**. В нее читается текст **Memo1.Text**, затем в строке **S** осуществляется контекстная замена и в заключение эта строка возвращается в **Memo1.Text**. Введение переменной **S** потребовалось, так как компилятор не разрешает непосредственно передавать в процедуры по ссылке (как **var**) значение свойства **Memo1.Text**.

Другой возможный вариант решения той же задачи контекстной замены связан с использованием функции **Copy**. Эта функция, объявленная как

```
function Copy(S; Index, Count: Integer): string;
```

возвращает подстроку строки **S**, начинающуюся с индекса **Index** и содержащую **Count** символов. При использовании этой функции решение задачи имеет вид:

```
procedure TForm1.Button1Click(Sender: TObject);
var i:integer;
begin
 i := Pos(Edit1.Text, Memo1.Text);
 if i > 0
 then
   Memo1.Text := Copy(Memo1.Text, 1, i-1) + Edit2.Text +
                 Copy(Memo1.Text, i + Length(Edit1.Text), 1000);
end;
```

Первый вызов **Copy** заносит в **Memo1.Text** первую часть текста с первой позиции по **i** – 1, где **i** — индекс начала вхождения текста **Edit1.Text** в **Memo1.Text**. Затем операцией склеивания "+" к этому тексту добавляется текст **Edit2.Text**. И в заключение добавляется текст, расположенный в исходном тексте после вхождения в него **Edit1.Text**. Начало этой части определяется индексом **i + Length(Edit1.Text)**. А число индексов задано равным 1000. Дело в том, что если в функции **Copy** задать число копируемых символов больше, чем содержится в строке, то ничего страшного не произойдет: скопируется столько символов, сколько их остается до конца строки. Так что в приведенном коде полагается, что длина текста в **Memo1.Text** не превышает 1000 символов.

Попробуем еще усложнить наш пример. Потребуем, чтобы в тексте **Memo1.Text** была произведена замена подстрок **Edit1.Text** на **Edit2.Text** во всех вхождениях, а не только в первом, как это делалось до сих пор. Ниже приведен код, решающий эту задачу.

```
procedure TForm1.Button1Click(Sender: TObject);
var i: integer;
begin
 i := Pos(Edit1.Text, Memo1.Text);
```

```
 while i > 0 do
 begin
  Memo1.Text := Copy(Memo1.Text, 1, i-1) + Edit2.Text +
                Copy(Memo1.Text, i + Length(Edit1.Text), 1000);
  i := Pos(Edit1.Text, Memo1.Text);
 end;
end;
```

В приведенном коде использованы те же операторы контекстной замены, что и в предыдущей задаче. Только они заключены в цикл **while** ... **do**, который обеспечивает просмотр всех вхождений искомой строки в исходный текст.

Достаточно часто встречающаяся задача, особенно, при сортировке массивов строк — лексикографическое сравнение двух строк. Под лексикографическим сравнением понимается следующее. Сравнение строк осуществляется по символам, начиная с первого. Если очередные символы не равны друг другу, то строка, в которой символ больше, считается больше другой строки, и функция возвращает соответствующее значение. Заглавные буквы (символы, набранные в верхнем регистре) считаются больше прописных (набранных в нижнем регистре). Сравнение символов кириллицы производится в соответствии с русским алфавитом. Считается, что латинские символы меньше символов кириллицы, символы цифр меньше символов букв, символы пунктуации (включая пробел) меньше символов цифр. Если в процессе сравнения оказывается, что в одной строке символы закончились, а в другой еще имеются, строка с меньшим числом символов считается меньшей.

Если строки содержат символы кириллицы, то сравнение строк можно проводить функциями **AnsiCompareStr** и **AnsiCompareText**:

```
function AnsiCompareStr(const S1, S2: string): Integer;
function AnsiCompareText(const S1, S2: string): Integer;
```

Функция **AnsiCompareStr** сравнивает строки **S1** и **S2** с учетом регистра, а **AnsiCompareText** — без его учета. Обе функции возвращают значение < 0, если **S1** < **S2**, 0, если **S1** = **S2**, и > 0, если **S1** > **S2**. Реально при **S1** < **S2** результат равен −1, а при **S1** < **S2** равен +1, хотя в документации это не оговорено.

Например, следующие операторы обеспечивают различные действия (обозначены многоточиями) в зависимости от сравнения текстов в окнах **Edit1** и **Edit2**.

```
var S1, S2: AnsiString;
...
S1 := Edit1.Text;
S2 := Edit2.Text;
if AnsiCompareStr(S1, S2) < 0
 then ...
 else if AnsiCompareStr(S1, S2) = 0
then ...
else ...;
end;
```

Приведенный выше текст можно было бы упростить, удалив из него переменные **S1** и **S2** и заменив операторы **if** следующими:

```
if AnsiCompareStr(Edit1.Text, Edit2.Text) ...
```

В приведенной ниже таблице даны различные варианты строк **S1** и **S2** и результаты их сравнения.

S1	S2	AnsiCompareStr возвращает	AnsiCompareText возвращает
'строка 1'	'строка 2'	-1	-1
'строка '	'строка'	+1	+1
'строка 1'	'Строка 1'	-1	0
'строка'	'строки'	-1	-1

Мы рассмотрели обработку строк типа **AnsiString**. В Object Pascal строки с нулевым символом в конце могут задаваться также как массивы символов с индексами, начинающимися с 0. В массиве хранятся символы строки и в конце нулевой символ (#0), указывающий на окончание строки.

Таким образом, тип строки с нулевым символом в конце может объявляться как статический массив символов, например:

```
type TS = array[0..15] of Char;
```

При этом размер массива должен быть таким, чтобы вместить значащие символы и нулевой символ в конце.

Обратите внимание на то, что минимальное значение индекса должно равняться нулю. Если, например, вы напишете

```
type TS = array[1..15] of Char;
```

компилятор выдаст сообщение об ошибке.

С массивами символов работать намного сложнее, чем со строками типа **AnsiString**. Во-первых, надо постоянно отслеживать размер отведенной по массив памяти и сопоставлять его с размером строки, которая в него записывается. Если размер строки окажется больше отведенного в массиве места, лишние символы сотрут какие-то другие данные, используемые в программе, и последствия при этом непредсказуемы. Точнее, ошибки гарантированы, но отыскать их будет очень трудно. Во-вторых, операции склеивания строк, да и многие другие использовать сложнее, чем это было рассмотрено для строк типа **AnsiString**.

Для работы с массивами символов с нулевым символом в конце предусмотрен тип указателей на них **PChar**. С этим указателем работать несколько более удобно, чем непосредственно с массивами символов. Объявление переменной типа **PChar** производится обычным образом:

```
var PCh: PChar;
```

Объявление может сопровождаться инициализацией:

```
var PCh: PChar = 'Привет!';
```

При этом не приходится думать о размере массива. Он устанавливается автоматически. Но как только вы захотите изменить размер, записав в переменную какую-то более длинную строку, вам опять придется заботиться о выделении дополнительного места в памяти.

Всеми этими сложностями мы заниматься не будем, поскольку тип строк **AnsiString** во всех случаях предпочтительнее массивов символов и типа указателей на строки **PChar**. Тем не менее, во многих задачах приходится работать и с типом **AnsiString**, и с типом **PChar**. Дело в том, что многие свойства компонентов Delphi имеют тип **string**, эквивалентный **AnsiString**, а многие функции, особенно те, которые инкапсулируют функции API Windows, требуют задания параметров

типа **PChar**. Так что в реальных приложениях нередко приходится переходить от одного типа строк к другому. Вот эту проблему нам следует рассмотреть.

Строки разных типов могут присваиваться друг другу и смешиваться в одном выражении. Компилятор при этом осуществляет автоматическое приведение типов. Например, строкам **AnsiString** можно непосредственно присваивать значения строк **PChar**:

```
var S1: AnsiString;
    P1: PChar;
...
S1 := P1;
```

Но непосредственное присваивание строкам **PChar** значения строк **AnsiString** не допускается компилятором. На оператор

```
P1 := S1;
```

компилятор выдаст сообщение «Incompatible types: 'String' and 'PChar'» — несовместимые типы 'String' и 'PChar'. Чтобы избежать такой ошибки, надо применять явное приведение типа:

```
P1 := PChar(S1);
```

Аналогичное явное приведение типа требуется при переносе в строку **PChar** текста какого-то компонента Delphi. Например,

```
P1 := PChar(Edit1.Text);
```

Но при переносе в текст компонента строки типа **PChar** приведение типов происходит автоматически:

```
Edit1.Text := P1;
```

Особенно часто приходится применять явное приведение типов при передаче параметров в функции, требующие тип **PChar**. В качестве примера можно привести функцию **Application.MessageBox**, чрезвычайно часто используемую для диалоговых окон, вызываемых приложением. Эта функция объявлена как

```
function MessageBox(Text, Caption: PChar; Flags: Longint): Integer;
```

Подробное рассмотрение этой функции, смысл флагов **Flags** и возвращаемого функцией значения мы отложим до разд. 4.11 в гл. 4. А сейчас нас будут интересовать первые параметры функции: **Text** — текст сообщения, и **Caption** — заголовок окна. Тип обоих этих параметров **PChar**. Если вы хотите передавать в качестве текста сообщения строковую константу, то никаких сложностей не возникнет:

```
Application.MessageBox('Ошибочный текст', 'Ошибка',
                                          MB_ICONEXCLAMATION);
```

Но если текст сообщения сформирован в некоторой строке **S** типа **string**, то оператор

```
Application.MessageBox(S, 'Ошибка', MB_ICONEXCLAMATION);
```

вызовет уже приведенное ранее сообщение компилятора о несовместимости типов **string** и **PChar**. Приведенный выше оператор надо переписать с использованием приведения типов:

```
Application.MessageBox(PChar(S), 'Ошибка',MB_ICONEXCLAMATION);
```

Аналогичное приведение типов надо использовать, если вы формируете строку сообщения непосредственно в вызове функции, используя операцию "+":

```
Application.MessageBox(PChar('Ошибочный текст "' + Edit1.Text +
                             '"'), 'Ошибка',MB_ICONEXCLAMATION);
```

Все сказанное выше относилось к 32-разрядным версиям Delphi. В Delphi 1 имеются существенные отличия: отсутствие типов длинных строк и отсутствие приведения типов строк, использующих и не использующих нулевой символ в конце. Поэтому для преобразования типов строк надо использовать специальные функции. Функция

```
function StrPas(Str: PChar): String;
```

копирует строку **Str** типа **PChar** в строку в стиле Pascal. Функция

```
function StrPCopy(Dest: PChar; Source: String): PChar;
```

копирует строку **Source** в стиле Pascal в строку **Dest** типа **PChar** и возвращает указатель на эту строку. Размер **Dest** должен по крайней мере на 1 превышать число символов в **Source**. Функция

```
function StrPLCopy(Dest: PChar; const Source: string;
                   MaxLen: Word): PChar;
```

осуществляет аналогичное копирование, но копирует не более **MaxLen** символов.

3.2.3 Форматирование текстов окон редактирования

В таких компонентах, как **Edit** и **Memo** формат (шрифт, его атрибуты, выравнивание) одинаков для всего текста и определяется свойством **Font**. Эти параметры можно устанавливать во время проектирования или программно. Например, операторы

```
with Memo1.Font do
begin
 Name := 'Arial';
 Size := 12;
 Style := Style + [fsBold, fsItalic];
 Color := clRed;
end;
```

устанавливают в окне **Memo1** красный полужирный курсив шрифта Arial размера 12. А следующий оператор возвращает шрифту нормальное начертание:

```
Memo1.Font.Style := Memo1.Font.Style - [fsBold, fsItalic];
```

Можно ввести в приложение какие-то кнопки, разрешающие пользователю подобным образом выбирать отдельные атрибуты форматирования. А можно предоставить ему возможность вызывать стандартный диалог Windows (см. рис. 3.5) для задания сразу всех атрибутов шрифта. Для этого нам понадобится компонент **FontDialog**. Этот компонент, как и другие компоненты стандартных диалогов, находится в библиотеке компонентов на странице *Dialogs*. Перейдите в палитре компонентов на эту страницу. Она расположена в палитре далеко справа, и сразу не видна. Чтобы добраться до нее, воспользуйтесь кнопкой со стрелочкой вправо, расположенной в правом конце палитры (см. рис. 1.2 в разд. 1.2.1). Компоненту **FontDialog** соответствует пиктограмма с символами "F". Перенесите этот компо-

нент на форму приложения. Компоненты диалогов относятся к так называемым невизуальным компонентам, которых пользователь не видит. Так что можете поместить **FontDialog** в любое место формы. Перенесите также на форму окно **Memo** и кнопку с надписью Шрифт. Мы хотим, чтобы при щелчке на этой кнопке пользователь мог задать атрибуты шрифта в окне **Memo1**.

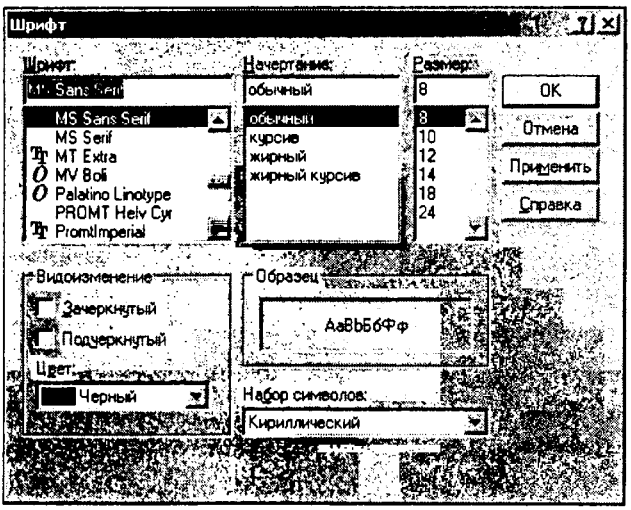


Рис. 3.5
Диалоговое окно выбора атрибутов шрифта

Основной метод, которым производится обращение к любому диалогу, — **Execute**. Эта функция открывает диалоговое окно и, если пользователь произвел в нем какой-то выбор, то функция возвращает **true**. При этом в свойствах компонента—диалога запоминается выбор пользователя, который можно прочитать и использовать в дальнейших операциях. Если же пользователь в диалоге нажал кнопку Отмена или клавишу Esc, то функция **Execute** возвращает **false**. Поэтому стандартное обращение к любому компоненту диалога имеет вид:

```
if <имя компонента-диалога> Execute
 then <операторы, использующие выбор пользователя>;
```

Основное свойство компонента **FontDialog** — **Font**, в котором вы можете задать при желании начальные установки атрибутов шрифта и в котором вы можете прочитать значения атрибутов, выбранные пользователем в процессе диалога. Так что в обработчик щелчка на кнопке Шрифт нашего приложения можете поместить операторы:

```
FontDialog1.Font.Assign(Memo1.Font);
if FontDialog1.Execute
 then Memo1.Font.Assign(FontDialog1.Font);
```

Первый из этих операторов не является обязательным. Он просто задает с помощью метода **Assign** (см. разд. 2.8.1) текущие атрибуты шрифта окна **Memo1** в качестве начальных значений при открытии окна диалога (рис. 3.5). Второй оператор вызывает диалог выбора атрибутов шрифта и, если пользователь произвел выбор, то значения всех выбранных атрибутов, содержащиеся в свойстве

FontDialog1.Font, присваиваются с помощью метода **Assign** атрибутам окна редактирования, содержащимся в свойстве **Memo1.Font**. Шрифт в окне редактирования немедленно изменится.

Свойство **Options** компонента **FontDialog** содержит множество опций, смысл которых вы можете посмотреть в справках. Отметим только опцию **fdApplyButton**. Если установить ее в **true**, то в окне диалога (см. рис. 3.5) появляется кнопка Применить. Когда пользователь щелкает на ней, в компоненте возникает событие **OnApply**. В его обработчике вы можете написать код, который применит выбранные пользователем атрибуты, не закрывая диалогового окна.

Установите в компоненте **FontDialog1** опцию **fdApplyButton**, и напишите обработчик события **OnApply**:

```
Memo1.Font.Assign(FontDialog1.Font);
```

Теперь пользователь сможет наблюдать изменения в окне **Memo1**, нажимая в диалоговом окне кнопку Применить и не прерывая диалога. Это очень удобно, так как позволяет пользователю правильно подобрать атрибуты шрифта.

3.2.4 Окно редактирования RichEdit

В разд. 3.2.3 рассмотрены весьма скромные возможности форматирования текста в окне **Memo**. Несравненно большими возможностями обладает компонент **RichEdit**, расположенный в библиотеке на странице Win32. Это *многострочное* окно редактирования, во многом похожее на **Memo**. Перенеся его на форму, вы увидите у него свойство **Lines**, которое, как и в окне **Memo**, содержит строки занесенного в окно текста. Добавлять строки в окно можно программно тем же методом **Add**, который вы использовали в **Memo**. Точно так же свойство только времени выполнения **Text** содержит весь текст окна в виде одной строки. Так что все приемы работы с текстами окна **Memo**, рассмотренные ранее, можно использовать и при работе с окном **RichEdit**. В любом приложении, которое вы до сих пор создали, можно заменить **Memo** на **RichEdit**, соответственно заменить в кодах имя компонента **Memo1** на **RichEdit1**, и все будет работать. Свойство **ScrollBars**, как и в **Memo**, определяет наличие полос прокрутки. Только в компоненте **RichEdit** это свойство более «интеллектуально»: полосы появляются только в тот момент, когда размер текста перестает целиком помещаться в окне.

Особенностью компонента **RichEdit** является то, что он может работать с текстом в обогащенном формате RTF. Этот формат позволяет раздельно форматировать отдельные фрагменты текста. Для любого фрагмента можно задать свои атрибуты шрифта: выделить какие-то слова или фразы полужирным шрифтом или курсивом, задать абзацу формат списка, выровнять отдельные абзацы по центру и т.п.

Прежде, чем рассматривать способы форматирования фрагментов текста, следует сказать о свойствах, описывающих выделенный текст в любых окнах редактирования: **Edit**, **Memo**, **RichEdit**. Это свойства только времени выполнения **SelLength**, **SelStart**, **SelText**, определяющие соответственно длину выделенного текста, позицию перед первым символом выделенного текста и сам выделенный текст. Например, если в окне имеется текст «выделение текста» и в нем пользователь выделил слово «текста», то **SelLength** = 6, **SelStart** = 10 и **SelText** = "текста". Если выделенного текста нет, то свойство **SelStart** просто определяет текущее положение курсора.

В окне **RichEdit** атрибуты выделенного фрагмента текста определяются свойством **SelAttributes**. Это свойство совместимо по типу со свойством **Font** и имеет те

же подсвойства. Так что если изменить значения атрибутов шрифта в свойстве **SelAttributes**, эти атрибуты немедленно будут применены к выделенному тексту. Если в тексте нет выделения, то новые атрибуты будут применяться к вновь вводимому тексту, начиная с текущей позиции курсора **SelStart**.

Давайте создадим на основе компонента **RichEdit** прототип текстового редактора. Пока мы еще не готовы реализовать полноценный редактор. Это будет сделано значительно позднее. Но его прототип — начальную версию можно сделать, и опробовать на ней возможности форматирования. Перенесите на форму окно **RichEdit**, диалог **FontDialog** и кнопку с надписью Шрифт. Введите в обработчик щелчка на этой кнопке операторы:

```
FontDialog1.Font.Assign(RichEdit1.SelAttributes);
if(FontDialog1.Execute)
then RichEdit1.SelAttributes.Assign(FontDialog1.Font);
```

А в обработчик события **OnApply** (см. разд. 3.2.3) компонента **FontDialog1** введите оператор:

```
RichEdit1.SelAttributes.Assign(FontDialog1.Font);
```

Запустите приложение, и вы увидите, что пользователь может теперь изменять атрибуты текста, выполняя отдельные фрагменты различными шрифтами, размерами, цветами, стилями. Щелкая в окне диалога на кнопке Применить, пользователь может сразу видеть результаты и осознанно подобрать желательные ему атрибуты шрифта. Устанавливаемые атрибуты влияют на выделенный текст или, если ничего не выделено, то на атрибуты нового текста, вводимого начиная с текущей позиции курсора.

Теперь рассмотрим возможности задания атрибутов форматирования отдельных абзацев. Эти атрибуты определяются свойством **Paragraph** окна **RichEdit**. Отметим следующие подсвойства свойства **Paragraph**:

Alignment	Определяет выравнивание текста. Может принимать значения **taLeftJustify** (влево), **taCenter** (по центру), или **taRightJustify** (вправо).
Numbering	Управляет вставкой маркеров, как в списках. Может принимать значения **nsNone** — отсутствие маркеров, **nsBullet** — маркеры ставятся.

Значения подсвойств свойства **Paragraph** можно задавать только в процессе выполнения приложения, например, в событии создания формы или при нажатии какой-нибудь кнопки.

Вы можете ввести в свое приложение три кнопки, обеспечивающие пользователю выравнивание текста текущего абзаца влево, по центру и вправо. Операторы обработчиков щелчков на этих кнопках могут содержать операторы:

```
RichEdit1.Paragraph.Alignment:=taLeftJustify;  // Влево
RichEdit1.Paragraph.Alignment:=taCenter;       // По центру
RichEdit1.Paragraph.Alignment:=taRightJustify; // Вправо
```

Введите также кнопку, каждый щелчок на которой вводит или удаляет из абзаца формат списка. Обработчик щелчка на этой кнопке может иметь вид:

```
if RichEdit1.Paragraph.Numbering = nsBullet
 then RichEdit1.Paragraph.Numbering := nsNone
 else RichEdit1.Paragraph.Numbering:=nsBullet;
```

Можете усовершенствовать этот оператор, добавив в него изменения надписи кнопки, чтобы пояснить пользователю ее назначение. Например: «Вкл. список» и «Выкл. список».

Вы создали приложение на основе компонента **RichEdit**, позволяющее пользователю осуществлять форматирование шрифта и абзацев. Это приложение можно считать первым эскизом текстового редактора. Конечно, в редактор надо бы ввести возможность запоминания набранного пользователем форматированного текста в файле и чтение текстов из файлов. Это будет сделано в разд. 3.4.2. Желательно также ввести возможности контекстного поиска и замены. Это мы сделаем в разд. 3.2.5. Но главный недостаток нашего текстового редактора — убогий и несовременный интерфейс пользователя, отсутствие меню, инструментальных панелей, полосы состояния. Оформлением интерфейса нашего редактора мы займемся позднее в гл. 4 и 5.

3.2.5 Диалоги поиска и замены текстов в окнах редактирования

В разд. 3.2.2 мы рассмотрели алгоритмы поиска и замены фрагмента текста в строке. Хотелось бы применить эти алгоритмы к текстовым редакторам на основе компонентов **Memo** и **RichEdit**, связав со стандартными диалогами Windows поиска и замены. Это позволяют сделать компоненты **FindDialog** и **ReplaceDialog**, вызывающие диалоги поиска и замены фрагментов текста (рис. 3.6 и 3.7). Как и другие компоненты диалогов они расположены в библиотеке на странице *Dialogs*. Перенесите эти компоненты на форму вашего текстового редактора, разработанного в разд. 3.2.4 и использующего окно **RichEdit**. Посмотрите свойства компонентов **FindDialog** и **ReplaceDialog**. Они одинаковы, кроме одного, задающего заменяющий текст в компоненте **ReplaceDialog**. Такое сходство не удивительно, поскольку **ReplaceDialog** — производный класс от **FindDialog**.

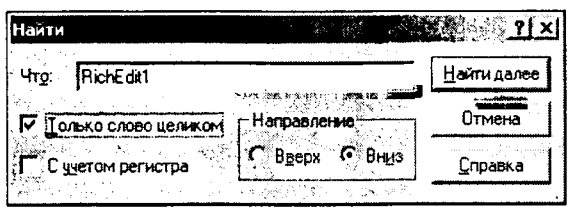


Рис. 3.6
Диалоговое окно поиска фрагмента текста

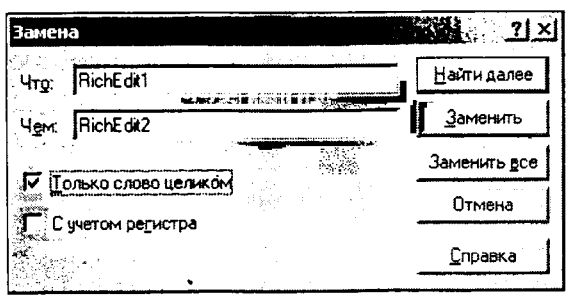


Рис. 3.7
Диалоговое окно замены фрагмента текста

Свойство **FindText** определяет текст, заданный для поиска или замены. Он может быть установлен программно перед вызовом диалога как начальное значение, предлагаемое пользователю. Свойство **ReplaceText**, имеющееся только в компоненте **ReplaceDialog**, определяет текст, который должен заменять **FindText**.

Параметр **Options** — может содержать следующие свойства:

frDisableMatchCase	Делает недоступным индикатор С учетом регистра в диалоговом окне.
frDisableUpDown	Делает недоступными в диалоговом окне кнопки Вверх и Вниз группы Направление, определяющие направление поиска.
frDisableWholeWord	Делает недоступным индикатор Только слово целиком в диалоговом окне.
frDown	Выбирает кнопку Вниз группы Направление при открытии диалогового окна. Если эта опция не установлена, то выбирается кнопка Вверх.
frFindNext	Эта опция включается автоматически, когда пользователь в диалоговом окне щелкает на кнопке Найти далее, и выключается при закрытии диалога.
frHideMatchCase	Удаляет индикатор С учетом регистра из диалогового окна.
frHideWholeWord	Удаляет индикатор Только слово целиком из диалогового окна.
frHideUpDown	Удаляет кнопки Вверх и Вниз из диалогового окна.
frMatchCase	Этот флаг включается и выключается, если пользователь включает и выключает опцию С учетом регистра в диалоговом окне. Можно установить эту опцию по умолчанию во время проектирования, чтобы при открытии диалога она была включена.
frReplace	Применяется только для **ReplaceDialog**. Этот флаг устанавливается системой, чтобы показать, что текущее (и только текущее) найденное значение **FindText** должно быть заменено значением **ReplaceText**.
frReplaceAll	Применяется только для **ReplaceDialog**. Этот флаг устанавливается системой, чтобы показать, что все найденные значения **FindText** должны быть заменены значениями **ReplaceText**.
frShowHelp	Задает отображение кнопки Справка в диалоговом окне.
frWholeWord	Этот флаг включается и выключается, если пользователь включает и выключает опцию Только слово целиком в диалоговом окне. Можно установить эту опцию по умолчанию во время проектирования, чтобы при открытии диалога она была включена.

Вызов диалогов осуществляется уже известным вам методом **Execute**. Например:

```
FindDialog1.Execute;
```

Перед вызовом вы можете задать значение искомого текста как выделенный фрагмент в окне редактирования. Например:

```
FindDialog1.FindText := RichEdit1.SelText;
```

Сами по себе компоненты **FindDialog** и **ReplaceDialog** не осуществляют ни поиска, ни замены. Они только обеспечивают интерфейс с пользователем. А поиск и замену надо осуществлять программно. Для этого можно пользоваться событием **OnFind**, происходящим, когда пользователь нажал в диалоге кнопку Найти далее, и событием **OnReplace**, возникающим, если пользователь нажал кнопку Заменить

или Заменить все. В обработчиках этих событий в свойствах **FindText** и **ReplaceText** вы найдете соответственно искомый и замещающий тексты. В событии **OnReplace** узнать, какую именно кнопку нажал пользователь, можно по значениям флагов **frReplace** и **frReplaceAll**. Завершить работу с диалогом можно методом **CloseDialog**, имеющимся у диалогов **FindDialog** и **ReplaceDialog**.

Попробуйте самостоятельно написать обработчики событий, обеспечивающие поиск и замену в окне **Memo**. Алгоритмы такого поиска и замены были рассмотрены в разд. 3.2.2. Они основаны на использовании функций **Pos**, **AnsiLowerCase** (чтобы обеспечить независимость от регистра), **Delete** и **Insert**. Для упрощения задачи исключите опцию поиска только целых слов и опцию поиска вверх от положения курсора.

Рассмотренное в разд. 3.2.2 можно применить к поиску в любом окне редактирования: **Memo** или **RichEdit**. Однако в компоненте **RichEdit** имеется метод **FindText**, существенно облегчающий организацию поиска. Этот метод объявлен следующим образом:

```
function FindText(const SearchStr: string;
                  StartPos, Length: Integer;
                  Options: TSearchTypes): Integer;
```

Его первым параметром **SearchStr** является текст искомого фрагмента. Поиск ведется в свойстве **Text** компонента **RichEdit**, начиная с позиции, указанной параметром **StartPos**, на протяжении **Length** символов. Таким образом, поиск можно вести не во всем тексте, а на некотором его участке. Так что, как будет показано ниже, легко организовать поиск всех вхождений указанного фрагмента в текст. Параметр **Options** является множеством опций, которое может содержать опции **stWholeWord** — поиск только целого слова, и **stMatchCase** — поиск с учетом регистра.

Ниже приведен код, обеспечивающий поиск и замену в окне **RichEdit1**.

```
var Dialog: TFindDialog;
    SPos:integer;
...
procedure TForm1.Button1Execute(Sender: TObject);
begin
// запоминание позиции курсора
 SPos:=RichEdit1.SelStart;
 if (Sender as TButton) = Button1
  then Dialog := FindDialog1
  else Dialog := ReplaceDialog1;

 with Dialog do begin
  // начальное значение текста поиска - текст,
  // выделенный в RichEdit1
  FindText:=RichEdit1.SelText;
  // удаление из диалога кнопок "Вверх", "Вниз"
  Options:=Options + [frHideUpDown];
  // выполнение
  Execute;
 end;
end;

procedure TForm1.FindDialog1Find(Sender: TObject);
var Opt: TSearchTypes;
```

```
begin
 with Dialog do begin
 if frMatchCase in Options
  // поиск с учетом регистра
  then Opt := Opt + [stMatchCase];
  if frWholeWord in Options
  // поиск только слова целиком
   then Opt := Opt + [stWholeWord];
  SPos:=RichEdit1.FindText(FindText, SPos,
                           Length(RichEdit1.Lines.Text), Opt);
  if SPos >= 0
   then begin
    // выделение найденного текста
    RichEdit1.SelStart := SPos;
    RichEdit1.SelLength := Length(FindText);
    Inc(SPos);
    if (Dialog = ReplaceDialog1) and (frReplaceAll in Options)
     then ReplaceDialog1Replace(Sender);
   end
   else begin
    ShowMessage(
          'Поиск завершен. Текст "'+FindText+'" не найден.');
    CloseDialog;
   end;   else
    if MessageDlg(
          'Текст "'+FindText+'" не найден. Продолжать диалог?',
          mtConfirmation, mbYesNoCancel, 0) <> mrYes
    then CloseDialog;
 end;
 RichEdit1.SetFocus;
end;

procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
begin
 if RichEdit1.SelText<>''
  then RichEdit1.SelText := ReplaceDialog1.ReplaceText;
 if frReplaceAll in ReplaceDialog1.Options
  then FindDialog1Find(AReplace);
end;

procedure TForm1.RichEdit1KeyUp(Sender: TObject; var Key: Word;
                                Shift: TShiftState);
begin
 SPos := RichEdit1.SelStart;
end;

procedure TForm1.RichEdit1MouseUp(Sender: TObject;
     Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 SPos := RichEdit1.SelStart;
end;
```

Код написан в общем виде, чтобы не повторять одинаковые операторы для режимов только поиска и поиска с заменой. Поэтому в модуле объявлена глобальная переменная **Dialog** типа **TFindDialog**. Этот тип является типом компонента **FindDialog**. А для компонента **ReplaceDialog** это родительский класс. Так что

в переменную **Dialog** можно в зависимости от ситуации записать указатель на компонент **FindDialog1**, или на компонент **ReplaceDialog1**.

В приложении имеются две кнопки, одна из которых обеспечивает поиск, а другая — поиск и замену. Процедура **Button1Execute** является общим обработчиком щелчка для обеих кнопок. Первый оператор этой процедуры запоминает в глобальной переменной **SPos** текущую позицию курсора **SelStart**. Затем анализируется источник события — переменная **Sender**. Если источник события — кнопка **Button1**, то в переменную **Dialog** заносится указатель на диалог **FindDialog1**. В противном случае в эту переменную заносится указатель на диалог **ReplaceDialog1**. Таким образом, весь последующий код применим в равной степени и для операции поиска, и для операции замены.

Далее в свойство **FindText** диалога заносится текст, выделенный в окне **RichEdit1** — свойство **SelText**. Включается опция **frHideUpDown**, удаляющая из окна диалога кнопки Вверх и Вниз, задающие направление поиска. Дело в том, что приведенный код обеспечивает поиск только вниз от текущей позиции курсора. Поиск вверх требует дополнительного усложнения кода и в данном примере не реализован.

После задания значений всех этих свойств вызывается метод **Execute**, инициирующий диалог.

В обоих компонентах-диалогах **FindDialog1** и **ReplaceDialog1** в качестве обработчика события **OnFind**, возникающего после каждого щелчка пользователя на кнопке Найти далее в окне диалога, задана процедура **FindDialog1Find**. В начале этой процедуры формируется множество опций поиска **Opt** в соответствии с заданными пользователем условиями, возвратившимися в свойстве **Options**. Затем описанным выше методом **FindText** осуществляется поиск заданного пользователем фрагмента, содержащегося в свойстве **FindText**. Поиск начинается с позиции **SPos**. Результат поиска заносится в ту же переменную **SPos**. Если значение **SPos** получилось отрицательным, значит очередное вхождение фрагмента не найдено. В этом случае пользователю выдается сообщение об этом и работа с диалогом завершается методом **CloseDialog**. Если же значение **SPos** неотрицательное, значит очередное вхождение фрагмента найдено. Тогда это вхождение выделяется в окне **RichEdit1**. Значение **SPos** увеличивается на 1, чтобы следующий поиск проводился, начиная с позиции, превышающей найденную. Далее определяется, является ли диалог, с которым проводится работа, диалогом замены **ReplaceDialog1** и не задана ли для него опция **frReplaceAll**, включающаяся при щелчке пользователя на кнопке Заменить все в окне диалога. Если задана, то вызывается процедура **ReplaceDialog1Replace**, которая будет описана далее.

В конце процедуры **FindDialog1Find** фокус передается окну **RichEdit1**, чтобы пользователь мог увидеть выделенный фрагмент и что-то с ним сделать.

Процедура **ReplaceDialog1Replace** является обработчиком события **OnReplace** компонента **ReplaceDialog1**. В нем выделенный в окне **RichEdit1** текст заменяется текстом, указанным пользователем и содержащимся в свойстве **ReplaceText**. А затем, если опция **frReplaceAll** включена, производится опять вызов описанной выше процедуры **FindDialog1Find**. Таким образом, процедуры **ReplaceDialog1Replace** и **FindDialog1Find** вызывают друг друга до тех пор, пока не закончится поиск всех вхождений заданного фрагмента текста.

Процедуры **RichEdit1KeyUp** и **RichEdit1MouseUp** являются обработчиками событий **OnKeyUp** и **OnMouseUp**, возникающих, когда пользователь отпустил клавишу или кнопку мыши в окне **RichEdit1**. Это может быть связано с тем, что

пользователь переместил курсор в окне и хочет, чтобы поиск проводился, начиная с этой новой позиции курсора. Поэтому в обработчиках этих событий значение глобальной переменной **SPos** делается равным текущей позиции курсора.

Реализуйте это приложение и проверьте его в работе.

3.3 Записи и списки

3.3.1 Объявление и использование записей

Запись (record), называемая в некоторых языках *структурой*, представляет собой объединенный общим именем набор данных различных типов. Отдельные данные записи называются *полями*. Все это напоминает запись в базе данных, только хранящуюся в оперативной памяти компьютера.

Тип записи объявляется следующим образом:

```
type
 <имя типа> = record
  <список имен полей> : <тип>;
  ...
  <список имен полей> : <тип>;
 end;

Например:

type
 TPers= record
  Fam,Name,Par : string;
  Year         : integer;
  Staf         : boolean;
  Dep          : string;
 end;
```

Эти операторы объявляют тип записи **TPers**, содержащей сведения о сотруднике некоторой организации: его фамилию, имя, отчество (поля **Fam**, **Name**, **Par**), год рождения (поле **Year**), штатный или нештатный сотрудник (поле **Staf**), отдел, в котором работает (поле **Dep**). Можете при желании считать, что это не список сотрудников, а список студентов или учащихся — что вам ближе. Тогда поле **Dep** — это студенческая группа (класс), а поле **Staf** — это, например, является ли студент (учащийся) отличником.

Область видимости идентификаторов полей только внутри записи. Так что обращаться к полям можно только через объект записи.

Когда определен тип записи, можно определить переменные этого типа. Например:

```
var Pers, Pers1 : TPers;
```

Можно объявлять запись и непосредственно в объявлении переменной, не прибегая к объявлению типа:

```
var <имя переменной> = record
     <список имен полей> : <тип>;
     ...
     <список имен полей> : <тип>;
    end;
```

Доступ к отдельным полям записи осуществляется указанием на соответствующую переменную типа записи и после символа точки "." — имя поля. Например:

```
Pers.Fam := 'Иванов';
Pers.Nam := 'Иван';
Pers.Par := 'Иванович';
Pers.Year:= 1960;
Pers.Staf:= true;
Pers.Dep := 'Цех 1';
```

При групповых операциях с полями удобно использовать оператор **with** (см. разд. 2.8.3), позволяющий не указывать перед каждым полем имя переменной. С помощью **with** приведенный выше пример можно записать компактнее:

```
with Pers do
 begin
  Fam := 'Иванов';
  Nam := 'Иван';
  Par := 'Иванович';
  Year:= 1960;
  Staf:= true;
  Dep:= 'Цех 1';
 end;
```

3.3.2 Связные списки, очереди, стеки, самоадресуемые записи

Теперь рассмотрим самоадресуемые записи, которые можно объединять в различные списки (последовательности). Нередко в памяти надо динамически размещать (см. разд. 2.11) последовательность записей, как бы формируя некий фрагмент базы данных, предназначенный для оперативного анализа и обработки. Поскольку динамическое размещение проводится в непредсказуемых местах памяти, то такие записи надо снабдить полями, содержащими указатели на следующую аналогичную запись. Такие записи со ссылками на аналогичные записи и называются *самоадресуемыми*. Ниже приведена схема соединения таких записей в связный список. Полю указателя в последней записи обычно присваивается значение **nil**, что является признаком последней записи при организации поиска в списке. Подобный список называется *линейным односвязным списком*, поскольку каждый элемент связан только с одним следующим элементом. Если указатель последнего элемента не равен **nil**, а указывает на первый элемент списка, то такой список называется *циклическим односвязным списком*.

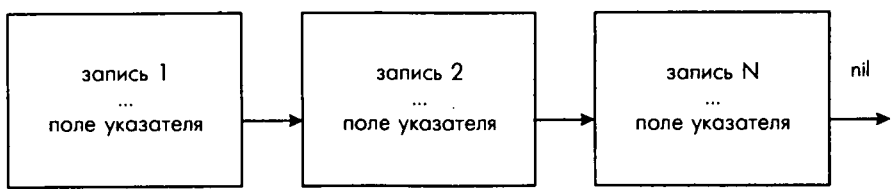


Рис. 3.8
Структура линейного односвязного списка

Сформировав список в памяти, далее легко его просматривать, проходя в цикле по указателям. Легко также делать перестановки записей, их удаление и т.п. Для всех этих операций не надо ничего перемещать в памяти. Достаточно только изменять соответствующие ссылки. Сравните это с вариантом динамического массива записей. В разд. 3.1.3.1 мы обсуждали функции вставки и удаления элементов в динамических массивах. При этом приходится перемещать множество элементов, чтобы освободить место для вставляемого, или чтобы занять место удаляемого элемента. Если массив большой, все это может занять немало времени. А в связном списке для этих операций надо просто изменить ссылку в предшествующем элементе.

Недостатком односвязных списков является то, что их можно просматривать только в одном направлении. Это снижает производительность некоторых операций. Поэтому иногда в каждый элемент вводят два указателя, один из которых указывает на следующий, а другой — на предыдущий элемент. Такие списки называются *двусвязными*.

Часто используется вариант линейного списка, называемый *очередью*. Когда надо вставить новый элемент в очередь, он вставляется в конец списка. А когда надо забрать элемент из очереди, то выбирается первый элемент списка. Таким образом реализуется дисциплина обслуживания очереди, называемая FIFO (First In First Out — первым вошел, первым вышел). Это подобно обычной очереди за какими-то благами: кто первым занял очередь, тот первым и обслуживается. И без очереди никто влезть не может.

Очереди используются в приложениях достаточно часто. Например, они применяются, если в ваше приложение поступают какие-то сообщения от других приложений. Пока приложение не готово к их обработке, они накапливаются в очереди, добавляясь в ее конец. А при обработке они поочередно извлекаются из начала очереди, т.е. в порядке их поступления.

Работа с очередью организуется проще, чем со списком общего вида. Для операции вставки нового элемента (она обычно называется Push) достаточно знать только конец очереди — адрес последнего элемента. После него вставляется новый элемент, а в бывшем последнем указатель начинает указывать на вставленный элемент. Для операции извлечения элемента (она обычно называется Pop) достаточно знать только начало очереди. Извлекаемый элемент удаляется, а началом очереди становится тот элемент, на который указывал удаленный.

Еще одним частным случаем списка является *стек*, в котором и добавление и извлечение элементов производится в начале списка. В этом случае первым обрабатывается элемент, который был поставлен в список последним. Такая дисциплина обслуживания называется LIFO (Last In First Out — последним вошел, первым вышел). Это как бы очередь на увольнение из фирмы: фирма дорожит сотрудниками со стажем, и первым кандидатом на увольнение является тот, который поступил на фирму последним. Стеки используются при обработке вложенных запросов, вложенных вызовов каких-то процедур, при грамматическом разборе выражений. Во всех этих случаях обработку надо начинать с самого внутреннего из вложенных элементов, пришедшего последним, поскольку результат обработки должен передаваться во внешний элемент.

Работа со стеком организуется еще проще, чем работа с очередью. Достаточно знать только начало очереди. Вставка нового элемента сводится к тому, что его указатель начинает указывать на тот элемент, который до этого был первым, а началом списка становится новый элемент. Извлечение элемента из стека не отличается от описанного ранее для очереди.

Теперь посмотрим, как это все реализуется в языке Object Pascal. Объявление типа самоадресуемой записи, способной включаться в односвязные списки, выглядит так:

```
type
 <имя типа указателя на запись> = ^<имя типа записи>
 <имя типа записи> = record
  ...
  <имя поля указателя>: <имя типа указателя на запись>;
  ...
 end;
```

Например:

```
type
 TPPers = ^TLPers
 TLPers = record
  Fam,Name,Par : string;
  Year         : integer;
  Staf         : boolean;
  Dep          : string;
  pf           : TPPers;
 end;
```

Подобное описание является исключением из общего правила языка Pascal, по которому нельзя использовать в предложениях еще не объявленные типы и переменные. В данном случае первое предложение объявляет тип **TPPers** как указатель на запись типа **TLPers**, объявленного после этого. Сразу за этим предложением следует объявление самой записи. В этом объявлении имеется поле **pf**, представляющее собой указатель на следующую запись аналогичного вида.

Для реализации работы со списками нам потребуются указатели на начало списка **P0**, на его конец **PLast**, и указатель **P**, с помощью которого можно будет перемещаться по списку. Такой указатель, перемещающийся по списку, обычно называется *итератором*. Все эти переменные в нашем примере могут быть объявлены так:

```
var P0: TPPers = nil;
    PLast, P: TPers;
```

Переменная **P0** будет всегда указывать на первую запись в списке, или равняться **nil**, если список пуст. Переменная **PLast** в непустом списке будет указывать на его последнюю запись. Итератор **P** может указывать на какой-то элемент списка. Например, если выполнить для непустого списка оператор

```
P := P0;
```

то итератор будет указывать не первый элемент. Если мы захотим перейти к следующему элементу, надо будет выполнить оператор:

```
P := P^.pf;
```

Тогда **P** станет равным значению поля **pf** данного элемента. А в этом поле записан указатель на следующий элемент списка. Многократное выполнение такого оператора позволит пройти по всем элементам списка, чтобы, например, найти требуемый. Перемещение должно заканчиваться, когда в очередной записи значение поля **pf** равно **nil**, т.е. когда достигнут последний элемент списка.

Итератор в подобном списке называется *однонаправленным*, поскольку может перемещаться только в одном направлении — от предыдущего к последующему элементу. Чтобы сделать итератор *двунаправленным*, надо формировать двусвязный список. Например, в нашу запись можно добавить еще оно поле **pr** типа **TPPers**, в которое заносить указатель на предыдущий элемент списка. Тогда оператор

```
P := P^.pr;
```

будет смещать итератор на одну позицию к началу списка.

Впрочем, все это нужно для работы со списками общего назначения. Если же список должен отображать очередь, то итерации по элементам списка могут и не потребоваться, если только не возникает задача просмотра всей очереди. А в нормальном режиме работы очередь имеет дело только с первым и последним элементами, на которые указывают **P0** и **PLast**. В случае стека не нужна и переменная **PLast**, так как вся работа веется с началом списка, на который указывает **P0**.

Давайте создадим приложение, работающее со списками. Форма этого приложения может иметь вид, показанный на рис. 3.8.а. На рисунке специально не удалены тексты окон редактирования, чтобы были понятны имена, данные различным окнам. Вам, конечно, надо эти тексты удалить.

Приложение содержит один компонент, который мы пока не использовали. Это индикатор с флажком **CheckBox**, который расположен в библиотеке на странице *Standard*. Индикаторы используются в приложениях в основном для того, чтобы пользователь мог включать и выключать какие-то опции, или для индикации состояния. В нашем случае индикатор призван показать, является ли сотрудник штатным, или нештатным. При каждом щелчке пользователя на индикаторе его состояние изменяется, проходя в общем случае последовательно через три значения: выделение (появление черной галочки), промежуточное (серое окно индикатора и серая галочка) и не выделенное (пустое окно индикатора). Этим трем состояниям соответствуют три значения свойства компонента **State**: **cbChecked**, **cbGrayed**, **cbUnchecked**. Впрочем, эти три состояния допускаются только при значении другого свойства **AllowGrayed** равном **true**. Если же **AllowGrayed** = **false** (значение по умолчанию), то допускается только два состояния: выделенное и не выделенное. И **State**, и **AllowGrayed** можно устанавливать во время проектирования или программно во время выполнения.

Проверять состояние индикатора можно не только по значению **State**, но и по значению свойства **Checked**. Если **Checked** равно **true**, то индикатор выбран, т.е. **State** = **cbChecked**. Если **Checked** равно **false**, то **State** равно **cbUnchecked** или **cbGrayed**. Установка **Checked** в **true** во время проектирования или выполнения автоматически переключает **State** в **cbChecked**.

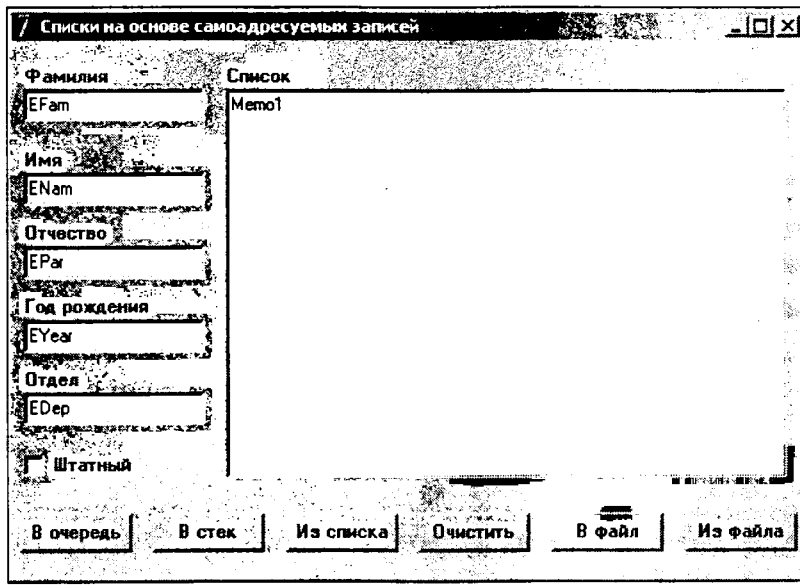
Надпись в индикаторе задается свойством **Caption**, а ее размещение по отношению к индикатору — свойством **Alignment**: **taLeftJustify** — слева, **taRightJustify** — справа (как на рис. 3.8).

Приложение содержит следующие кнопки:

Надпись	Имя в коде	Назначение
В очередь	BQPush	Добавление записи в очередь, т.е. в конец списка.
В стек	BSPush	Добавление записи в стек, т.е. в начало списка.
Из списка	BQPop	Извлечение записи из начала списка и ее удаление.
Очистить	BClear	Очистка списка — удаление из него всех записей.

Надпись	Имя в коде	Назначение
В файл	BSave	Запоминание списка в файле. В данном разделе не рассматривается (см. разд. 3.4.5).
Из файла	BOpen	Чтение списка из файла. В данном разделе не рассматривается (см. разд. 3.4.5).

а)



б)

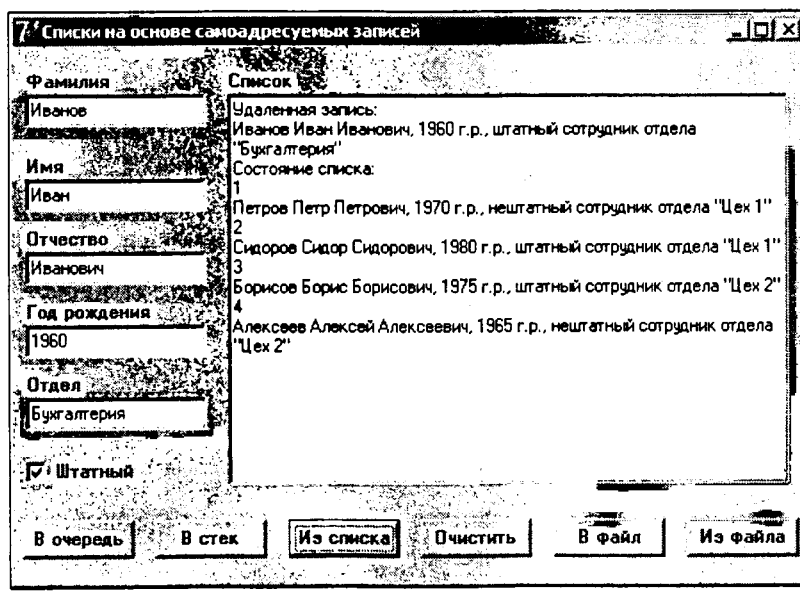


Рис. 3.8

Форма приложения работы со списками (а) и приложение во время выполнения (б)

Во втором столбце таблицы указаны имена кнопок, которые используются в приведенном далее коде. Две последние кнопки, связанные с запоминанием списка в файле и чтением из файла, будут рассмотрены в разд. 3.4.5. Так что пока можете ввести эти кнопки в приложение, но обработчики щелчков на них мы в этом разделе рассматривать не будем.

При добавлении записи в список значения полей читаются из окон редактирования. В окне **Memo1** после каждой операции со списком отображается его текущее состояние. При удалении записи ее поля отображаются в окнах редактирования, а в окне **Memo1** дается информация об удаленной записи (именно этот момент показан на рис. 3.8.6).

Ниже приведен код этого приложения с некоторыми очевидными купюрами:

```
unit Unit1;

interface

...

type
  TPPers = ^TLPers;
  TLPers = record
   Fam, Nam, Par: string[20];
   Year         : integer;
   Staf         : boolean;
   Dep          : string[15];
   pf           : TPPers;
  end;

  TForm1 = class(TForm)
  ...
    procedure BQPushClick(Sender: TObject);
    procedure BQPopClick(Sender: TObject);
    procedure BSPushClick(Sender: TObject);
    procedure BClearClick(Sender: TObject);
    procedure BSaveClick(Sender: TObject);
    procedure BOpenClick(Sender: TObject);
  private
    { Private declarations }
    function RecordPrint(P: TPPers): string;
    procedure ListPrint;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

var P0: TPPers  = nil;
    PLast, P: TPPers;
```

```
function TForm1.RecordPrint(P: TPPers): string;
begin
 with P^ do
 begin
  Result := Fam + ' ' + Nam + ' ' + Par +
         ', ' + IntToStr(Year) + ' г.р., ';
  if Staf then Result := Result + 'штатный'
     else Result := Result + 'нештатный';
  Result := Result + ' сотрудник отдела "' + Dep + '"';
 end;
end;

procedure TForm1.ListPrint;
var i: integer;
begin
 P := P0;
 i := 1;
 Memo1.Lines.Add('Состояние списка:');
 while P <> nil do
  with P^ do
  begin
   Memo1.Lines.Add(IntToStr(i));
   Inc(i);
   Memo1.Lines.Add(RecordPrint(P));
   P := pf;
  end;
end;

procedure TForm1.BQPushClick(Sender: TObject);
begin
 New(P);      // выделение памяти под новую запись
with P^ do
 begin
  Fam := EFam.Text;
  Nam := ENam.Text;
  Par := EPar.Text;
  Year:= StrToInt(EYear.Text);
  Staf:= CheckBox1.Checked;
  Dep := EDep.Text;
  pf  := nil;
 end;
 if P0=nil then P0:=P // P0 - указатель на первую запись
  else PLast^.pf:=P;  // указатель на очередную запись
 PLast:=P;
 Memo1.Clear;
 ListPrint;
end;

procedure TForm1.BSPushClick(Sender: TObject);
begin
 New(P);      // выделение памяти под новую запись
 with P^ do
 begin
  Fam := EFam.Text;
  Nam := ENam.Text;
  Par := EPar.Text;
```

```
  Year:= StrToInt(EYear.Text);
  Staf:= CheckBox1.Checked;
  Dep:= EDep.Text;
  if P0=nil
   then pf := nil
   else pf := P0;
 end;
 P0:=P;    // P0 - указатель на первую запись
 Memo1.Clear;
 ListPrint;
end;

procedure TForm1.BQPopClick(Sender: TObject);
begin
 Memo1.Clear;
 if P0 = nil
  then begin
   Memo1.Lines.Add('Список пуст');
   exit;
  end;
 Memo1.Lines.Add('Удаленная запись:');
 Memo1.Lines.Add(RecordPrint(P0));
 with P0^ do
 begin
  EFam.Text := Fam;
  ENam.Text := Nam;
  EPar.Text := Par;
  EYear.Text := IntToStr(Year);
  EDep.Text := Dep;
  CheckBox1.Checked := Staf;
 end;
 P := P0;
 P0 := P^.pf;
 Dispose(P);
 ListPrint;
end;

procedure TForm1.BClearClick(Sender: TObject);
begin
 while P0 <> nil do
 begin
  P := P0;
  P0 := P^.pf;
  Dispose(P);
 end;
 Memo1.Clear;
 Memo1.Lines.Add('Список пуст');
end;
```

Начнем анализ этого кода. В класс формы мы хотим включить вспомогательную функцию **RecordPrint**, которая будет возвращать в виде строки информацию о записи, на которую указывает ее аргумент **P** типа **TPPers**. Конечно, можно было бы не включать эту функцию, а также вспомогательную процедуру **ListPrint**, отображающую список в окне **Memo1**, в класс формы, тем более что классы мы пока рассматривали очень бегло. Но в разд. 2.7.1 мы рассматривали преимущества

включения в класс формы объявлений функций, работающих с компонентами формы. А функция **RecordPrint** и процедура **ListPrint** работают с окнами редактирования. Так что логично их объявления поместить в класс формы.

Но тут возникает одна проблема. Функция **RecordPrint** имеет аргумент типа **TPPers**. Чтобы компилятор понял этот тип, объявление типа должно быть сделано раньше, т.е. до объявления класса формы. Поэтому в приведенном коде перед объявлением класса вставлено объявление типов **TPPers** и **TLPers**.

Объявление типа записи **TLPers** отличается от приведенного в предыдущих примерах тем, что тип строковых полей в нем указан короткими строками, содержащими по 20 и 15 символов. Для примера, рассматриваемого в данном разделе, можно было бы оставить прежний тип **string**. Но для записи списка в файл это не годится по причинам, рассмотренным в разд. 3.4. Поэтому, чтобы не переделывать в дальнейшем этот пример, в нем использованы короткие строки.

Раздел **implementation** начинается с объявления рассмотренных ранее переменных **P0**, **PLast**, **P**. Затем следует реализация вспомогательной функции **RecordPrint**. Ее код, вероятно, вам очевиден, а результирующие строки вы можете видеть на рис. 3.8б в окне **Memo1**.

Далее следует вспомогательная процедура **ListPrint**. В ней реализован рассмотренный ранее цикл, в котором итератор **P** проходит по всему списку. В окно **Memo1** заносится порядковый номер записи и строка, формируемая для каждой записи функцией **RecordPrint**.

Процедура **BQPushClick** является обработчиком щелчка на кнопке В очередь. Ее назначение — сформировать новую запись в соответствии с данными, указанными пользователем, и включить эту запись в конец очереди. В начале процедуры выделяет в памяти место для новой записи оператором **New(P)**. Затем поля этой записи, на которую указывает указатель **P**, заполнятся текстами, введенными пользователем в окна редактирования. Значение поля **Staf** задается равным состоянию индикатора **CheckBox1**. Если пользователь включил индикатор, то в поле заносится **true**. В поле **pf** заносится **nil**. Это указывает на то, что данная запись будет последней в списке.

После этого надо включить новую запись в список. Если список пока пустой (**P0 = nil**), то указателю **P0** присваивается ссылка **P** на вновь размещенную запись. В противном случае ссылка на новую запись присваивается полю **pf** предыдущей записи в списке. Указателем на эту запись, которая была до этого последней в списке, содержится в переменной **PLast**. Таким образом новая структура включается в общий список. В заключение переменной **PLast** присваивается значение **P**, так что теперь концом списка становится новая запись. После включения записи в список вызывается рассмотренная ранее процедура **ListPrint**, отображающая в окне **Memo1** новый состав списка.

Процедура **BSPushClick** является обработчиком щелчка на кнопке В стек. Ее назначение — сформировать новую запись в соответствии с данными, указанными пользователем, и включить эту запись в начало очереди. Первые операторы не отличаются от рассмотренных в процедуре **BQPushClick**. Только поле **pf** заполняется иначе. Если список пока пуст, то новая запись и первая, и последняя. Как свидетельство этого в поле **pf** заносится значение **nil**. А если список уже содержит записи, то в поле **pf** заносится **P0** — указатель на запись, которая была первой до этого. Таким образом новая запись включается перед прежней первой записью. И эта новая запись в любом случае становится первой в списке, поскольку переменной **P0** присваивается **P** — указатель на новую запись.

Процедура **BQPopClick** является обработчиком щелчка на кнопке Из списка. Ее назначение — удалить из списка первый элемент. Эта операция извлечения очередного элемента из списка одинакова для очереди и для стека. Первые операторы процедуры отображают в окнах редактирования и в индикаторе **CheckBox1** содержимое первой записи. Затем переменной **P** присваивается значение **P0**, т.е. в **P** запоминается указатель на первую запись, которую надо удалить. Затем переменной **P0** присваивается значение поля **pf** этой записи, так что **P0** начинает указывать на запись, которая ранее была второй (если в списке была только одна запись, то **P0** становится равным **nil**). После этого бывшая первая запись удаляется процедурой **Dispose**. Эти три оператора должны выполняться именно в этой последовательности. Если сначала удалить запись, то уже нельзя будет прочитать значение ее поля **pf**, которое надо присвоить переменной **P0**. А если не запомнить сначала в переменной **P** указатель на первую запись, то после изменения значения **P0** до этой записи уже невозможно будет добраться, так как указатель на нее будет утерян.

Последняя процедура **BClearClick** является обработчиком щелчка на кнопке Очистить. Она в цикле проходит по всем записям списка и удаляет их. Операторы тела этого цикла повторяют рассмотренные выше операторы удаления записи в процедуре **BQPopClick**.

На процедуру **BClearClick** надо сослаться также как на обработчик события **OnDestroy** формы. Это событие наступает в момент завершения приложения, когда его форма удаляется из памяти. В обработчике этого события надо уничтожать все объекты, динамически размещенные в памяти. Иначе могут быть неоправданные потери памяти после завершения приложения.

Хороший стиль программирования

Следует предусматривать при завершении приложения удаление всех объектов, динамически размещенных в памяти процедурами **New**, **GetMem**, функцией **GetMem**, явно вызываемыми конструкторами объектов **Create**. Если вы не освобождаете динамически распределяемую память от уже ненужных объектов, возможны бессмысленные затраты этой памяти, так называемая *утечка памяти*.

Мы рассмотрели основные операции со стеками и очередями. Попробуйте дополнить это приложение операциями работы со списками общего назначения: вводом новой записи на указанное пользователем место в списке, удалением из списка записи, указанной пользователем как ее место или как фамилия сотрудника, перестановкой двух указанных записей.

Реализуя эти операции, вы увидите, что однонаправленные итераторы, присущие связным спискам на основе самоадресуемых записей, не очень удобны для работы со списками общего назначения. Чтобы добраться до какого-то конкретного элемента, приходится многократно перемещать итератор от начала списка, так как перемещать его можно только на одну позицию. Возможна принципиально иная организация списков, обеспечивающая *итераторы произвольного доступа*. В этом случае список представляется в виде динамического массива указателей на записи. Записи при этом используются не самоадресуемые, а обычные. Впрочем, вместо записей могут фигурировать любые другие объекты: числа, массивы, компоненты. Если надо вставить в указанную позицию новый элемент, это легко сделать, перемещая только элементы массива (подобная задача решалась в разд. 3.1.3.1), т.е.

указатели, и вставляя в нужную позицию указатель на новую запись. Так же просто, без перемещения в памяти самих записей можно реализовать удаление любого элемента списка, сортировку списка и т.п. При этом итерации по списку можно делать так же легко, как в обычных массивах, в любую сторону и на любое число позиций, можно сразу получить доступ к любому элементу и т.п. Это и есть реализация итератора произвольного доступа.

Подобная реализация, пожалуй, избыточна для очередей и стеков, но для списков произвольного назначения много удобнее, чем использование самоадресуемых записей. Такой список нетрудно было бы создать для рассмотренного выше примера. Но этого даже не надо делать, так как создатели Delphi позаботились о разработке классов подобных списков. Они будут рассмотрены в следующих разделах.

3.3.3 Списки указателей TList

Объекты класса **TList** предназначены для хранения и управления списком указателей на любые объекты. Свойства и методы **TList** позволяют добавлять и удалять элементы списка, изменять их расположение в списке, сортировать элементы и проводить другие манипуляции с данными.

Для того чтобы создать в приложении объект класса **TList**, надо вызвать его конструктор **Create**. Например, так:

```
var MyList: TList;
...
MyList := TList.Create;
```

После того как объект стал не нужен, его надо удалить, чтобы освободить память. Объекты классов удаляются из памяти с помощью их метода **Free**:

```
MyList.Free;
```

Если объект нужен все время, пока выполняется приложение, удаление его из памяти надо осуществить в обработчике события **OnDestroy** формы, наступающего в момент завершения приложения.

Доступ к указателям, имеющимся в списке, можно осуществлять через свойство **Items**, являющееся индексированным массивом:

```
property Items[Index: Integer]: Pointer;
```

Индексы начинаются с 0. Обратите внимание на то, что свойство возвращает нетипизированный указатель типа **Pointer** (см. разд. 2.10). Так что перед его применением надо явным образом приводить его тип к типу указателя на соответствующий объект. Это станет ясно на примерах, которые будут рассмотрены позднее.

Свойство **Items** — это так называемое свойство по умолчанию. Поэтому с равным успехом можно применять индексы к самому объекту списка. Например, выражения **MyList.Items[1]** и **MyList[1]** эквивалентны и возвращают второй указатель списка.

Если в приведенных выражениях и во всех последующих методах задан недопустимый индекс, не соответствующий элементу, имеющемуся в списке, генерируется исключение **EListError**.

Свойство **Count** возвращает число указателей в списке. При добавлении или удалении элементов списка значение **Count** увеличивается или уменьшается автоматически. Если намеренно увеличить значение **Count**, то в список добавится соот-

ветствующее число нулевых указателей **nil**. Если намеренно уменьшить значение **Count**, то соответствующее число последних элементов списка будет удалено.

Имеется похожее на **Count** свойство **Capacity**, которое показывает, сколько указателей может храниться в массиве, т.е. емкость списка. Поэтому значение **Count** всегда не больше значения **Capacity**.

При добавлении в массив нового элемента **Count**, как уже говорилось, автоматически увеличивается на 1. Если при этом значение **Count** превышает значение **Capacity**, то **Capacity** тоже автоматически увеличивается, причем с запасом (обычно запас равен 3). Так что свойство **Capacity** вообще можно нигде в приложении не задавать. Смысл задания **Capacity** заключается только в том, чтобы предотвратить слишком частое перераспределение памяти при добавлении новых элементов списка. Так что если вы можете предположить, сколько примерно новых элементов будет далее включаться в список, имеет смысл задать соответствующее значение **Capacity**. Это ускорит выполнение, так как каждое автоматическое увеличение размера списка, связанное с перераспределением памяти, требует затрат времени. Например, оператор

```
MyList.Capacity := MyList.Count + 10;
```

увеличит емкость списка, разрешая включать в него до десяти новых указателей без перераспределения памяти.

При удалении элементов списка **Count** автоматически уменьшается, а значение **Capacity** остается неизменным. Таким образом, при сокращении длины списка получаются излишние затраты памяти. Если в ближайшее время не планируется новое увеличение числа элементов, то в этих случаях целесообразно уменьшить **Capacity** до значения **Count**:

```
MyList.Capacity := MyList.Count;
```

Это сократит затраты памяти.

Увеличение емкости списка можно производить также методом **Expand**, объявленным следующим образом:

```
function Expand: TList;
```

Если список не заполнен, т.е. количество элементов **Count** меньше емкости списка **Capacity**, то вызов метода **Expand** не расширяет список. Если же **Count** = **Capacity**, то функция **Expand** увеличивает емкость **Capacity** согласно следующему алгоритму. Если значение **Capacity** меньше 4, то **Capacity** увеличивается на 4. Если значение **Capacity** больше 4, но меньше 9, то **Capacity** увеличивается на 8. Если значение **Capacity** больше 8, то **Capacity** увеличивается на 16.

Поскольку функция **Expand** возвращает расширенный список, то ее можно также использовать для создания дубликата имеющегося списка. Например, следующий оператор создает объект **List1**, являющийся копией списка **MyList**:

```
List1 := MyList.Expand;
```

Если исходный список был заполнен, то оба списка **MyList** и **List1** оказываются расширенными.

Ниже приведены основные методы класса **TList**, позволяющие манипулировать указателями, хранящимися в списке:

Метод	Объявление / Описание
Add	**function Add(Item: Pointer): Integer;** Добавление нового указателя **Item** в список.
Clear	**procedure Clear;** Очистка списка.
Delete	**procedure Delete(Index: Integer);** Удаление из списка элемента по его индексу **Index**.
Exchange	**procedure Exchange(Index1, Index2: Integer);** Взаимная перестановка двух элементов списка с индексами **Index1** и **Index2**.
First	**function First: Pointer;** Возвращает первый указатель списка.
IndexOf	**function IndexOf(Item: Pointer): Integer;** Возвращает индекс первого вхождения в список заданного указателя **Item**.
Insert	**procedure Insert(Index: Integer; Item: Pointer);** Вставляет элемент **Item** в список в заданную позицию **Index**.
Last	**function Last: Pointer;** Возвращает последний указатель списка.
Move	**procedure Move(CurIndex, NewIndex: Integer);** Изменяет текущую позицию **CurIndex** элемента в списке на **NewIndex**.
Remove	**function Remove(Item: Pointer): Integer;** Удаляет из списка первое вхождение элемента **Item**. Возвращает индекс, который был у удаленного элемента.
Sort	**type TListSortCompare = function (Item1, Item2: Pointer): Integer;** **procedure Sort(Compare: TListSortCompare);** Сортирует список с использованием указанной функции сравнения **Compare**. Функция **Compare** должна возвращать отрицательное число, если элемент **Item1** должен располагаться раньше, чем **Item2**, положительное число в противном случае, и должна возвращать 0, если приоритет обоих элементов одинаков.

Рассмотрим применение этих методов на примерах. Следующие предложения объявляют тип записи **TPers**, аналогичный рассмотренному в разд. 3.3.1, и тип указателя **Trec** на эту запись:

```
type
 TPers= record
  Fam,Name,Par : string;
  Year         : integer;
  Staf         : boolean;
  Dep          : string;
 end;
 Trec = ^TPers;
```

Тип **Trec** в данном случае надо вводить обязательно, так как он потребуется при работе с указателями списка.

Следующие предложения объявляют переменную списка **MyList** и переменную указателя на запись **PRec**:

```
var MyList: TList;
    PRec: Trec = nil;
```

Учтите, что эти объявления не создают список и запись. Список **MyList** еще предстоит создать вызовом конструктора класса **TList**. А указателю **PRec** надо будет присвоить адрес созданного объекта записи.

Создание списка **MyList** осуществляется оператором:

```
MyList := TList.Create;
```

Создание записи, заполнение ее полей и включение указателя на нее в список может осуществляться операторами:

```
// Выделяется место под запись, адрес передается в PRec
New(PRec);
// Заполняются поля записи
with PRec^ do
begin
 Name := 'Петров';
 Year := 1960;
 ...
end;
// Указатель на запись заносится в список
MyList.Add(PRec);
```

Метод **Add** (см. приведенную выше таблицу) добавляет новый элемент в список. Если список не сортированный (сортировка списков будет рассмотрена позднее), то элемент добавляется в конец списка. Если же список сортированный, то новый элемент добавляется в позицию, которая определяется сортировкой. Функция **Add** возвращает индекс добавленного элемента и увеличивает значение свойства **Count** на 1.

Аналогичным образом можно сформировать несколько записей, занося указатели на них в конец списка. Как вы знаете из материала разд. 3.3.2, такое добавление новых элементов характерно для очередей. Но в списки класса **TList** новый элемент можно заносить в любую указанную позицию методом **Insert**. Например, оператор

```
MyList.Insert(0, PRec);
```

занесет указатель **PRec** в начало списка, что характерно для стеков. А оператор

```
MyList.Insert(3, PRec);
```

занесет указатель в четвертую позицию. При выполнении метода **Insert** элементы списка с индексами больше указанного автоматически сдвигаются, освобождая место для нового элемента.

Доступ к элементам списка можно получить через индексы описанного ранее свойства **Items**, или через индексы самого объекта списка. Например, следующие два эквивалентных оператора отображают в метке **Label1** поле **Name** второй записи списка:

```
Label1.Caption := TRec(MyList[1])^.Name;
Label1.Caption := TRec(MyList.Items[1])^.Name;
```

Обратите внимание на то, как обеспечивается доступ к записи. Выражения **MyList[1]** и **MyList.Items[1]** возвращают второй указатель, хранящийся в спи-

ске. Но это нетипизированный указатель типа **pointer** (см. разд. 2.10). Программа не знает, адрес объекта какого типа в нем записан. Так что прежде, чем использовать этот указатель, надо осуществить явное приведение его типа к типу указателя на нашу запись. Выражение **TRec(MyList[1])** возвращает указатель типа **TRec** на запись типа **TPers**. Через этот указатель можно получить доступ к записи с помощью операции разыменования ^. Так что **TRec(MyList[1])^** — это объект записи. Далее с помощью обычной операции точки можно получить доступ к любому полю записи.

Аналогичным образом можно изменить значение любого поля любой записи. Например, следующий оператор изменяет год рождения во второй записи списка:

```
TRec(MyList[1])^.Year := 1970;
```

Следующий пример демонстрирует поиск в списке самого молодого сотрудника:

```
PRec := MyList.Items[0];
for i:=1 to MyList.Count - 1 do
  if (TRec(MyList[i])^.Year > PRec^.Year)
    then PRec := MyList[i];
ShowMessage('Самый молодой - ' + PRec^.Name);
```

Первый из приведенных операторов можно заменить эквивалентным ему:

```
PRec := MyList.First;
```

Цикл, аналогичный приведенному, можно использовать для отображения записей, включенных в список, например, в окне редактирования **Memo** (см. пример отображения списка в разд. 3.3.2, только в данном случае код будет проще). Можно также отображать список с фильтрацией: например, только те записи, в которых год рождения сотрудника лежит в заданных пределах.

Метод **Exchange** меняет местами два элемента списка. Например, следующий элемент меняет местами первый и второй элементы:

```
MyList.Exchange(0, 1);
```

Метод **Move** перемещает элемент из позиции, указанной первым аргументом, в позицию, указанную вторым аргументом. Например, следующий оператор перемещает шестой элемент в начало списка:

```
MyList.Move(5, 0);
```

Позиции всех остальных элементов изменяются так, чтобы список остался непрерывным.

Удалить элемент списка можно методом **Delete**. Например, следующий оператор удалит из списка второй элемент:

```
MyList.Delete(1);
```

Все элементы списка с индексами, большими указанного, сдвинутся, так что позиция с указанным индексом займется следующим элементом списка. При этом уменьшится на 1 значение свойства **Count**, но свойство **Capacity** останется неизменным. Так что если в дальнейшем добавление новых элементов не ожидается, рационально устранить избыточные затраты памяти, выполнив оператор:

```
MyList.Capacity := MyList.Count;
```

Обратите внимание на очень важное обстоятельство. Метод **Delete** удаляет из списка указатель на объект, но не сам объект. Так что в нашем примере запись, ад-

рес которой был записан в удаленном указателе, останется в памяти. Если в приложении нет другого указателя, соответствующего этой записи, то получить в дальнейшем доступ к этой записи будет невозможно. Получится в чистом виде так называемая *утечка памяти* — наличие в области динамически распределяемой памяти блоков, к которым нет доступа и которые не нужны для работы приложения. Так что если надо удалить не только указатель из списка, но и запись, на которую он указывает, код должен быть таким:

```
Dispose(MyList[1]);
MyList.Delete(1);
```

Удалить сразу все элементы списка можно методом **Clear**. Он удалит все указатели из списка и сделает равными нулю значения **Count** и **Capacity**. Но при применении метода **Clear** надо учитывать сделанное выше замечание: если надо удалить не только указатели из списка, но и сами записи, то код очистки списка должен быть следующим:

```
for i := 0 to (MyList.Count - 1) do
 Dispose(MyList[i]);
MyList.Clear;
```

Аналогичный код должен выполняться и перед удалением из памяти самого списка:

```
for i := 0 to (MyList.Count - 1) do
 Dispose(MyList[i]);
MyList.Free;
```

Хороший стиль программирования

Нельзя допускать в программе утечку памяти — наличие в области динамически распределяемой памяти блоков, к которым нет доступа и которые не нужны для дальнейшей работы приложения. Подобные блоки должны своевременно удаляться процедурами **Dispose** или **FreeMem**. А объекты классов, созданные в процессе выполнения приложения явным вызовом конструкторов, должны своевременно уничтожаться методом **Free**.

Нам осталось рассмотреть только метод **Sort**, обеспечивающий сортировку списка по заданному критерию. Для применения этого метода следует написать функцию типа **TListSortCompare**. Метод сортировки будет автоматически вызывать эту функцию для различных пар элементов списка **Item1** и **Item2**. Если элемент **Item1** должен располагаться в списке до **Item2**, функция должна вернуть отрицательное число. Если элемент **Item1** должен располагаться после **Item2**, функция должна вернуть положительное число. Если приоритет обоих элементов одинаков, функция должна вернуть 0.

Пусть, например, мы хотим расположить элементы в алфавитной последовательности сотрудников. Тогда функция может иметь следующий вид:

```
function Comp(Item1, Item2: pointer): integer;
var S1, S2: string;
begin
 with TRec(Item1)^ do
  S1 := Fam + ' ' + Name + ' ' + Par;
 with TRec(Item2)^ do
```

```
  S2 := Fam + ' ' + Name + ' ' + Par;
 Result := AnsiCompareText(S1, S2);
end;
```

Имя функции (в данном примере **Comp**) может быть произвольным. Два первых оператора формируют строки **S1** и **S2**, состоящие из фамилии, имени и отчества одного и другого сотрудника. А затем функция **Comp** возвращает результат сравнения этих строк без учета регистра функцией **AnsiCompareText**, описанной в разд. 3.2.2.

Если вы ввели такую функцию в приложение, то сортировка списка осуществляется оператором:

```
MyList.Sort(Comp);
```

Попробуйте сами составить аналогичную функцию, которая сортировала бы список по подразделениям, а внутри каждого подразделения — по алфавиту.

Мы рассмотрели возможности списков, созданных с помощью класса **TList**. Сравнив возможности таких списков и рассмотренных в разд. 3.3.2 связных списков, использующих самоадресуемые записи, вы можете увидеть, что **TList** имеет несомненные преимущества. Напомню также, что с помощью **TList** можно создавать не только списки записей, но и, например, списки чисел с возможностью их упорядочивания, вставки и удаления любых элементов и т.п.

Развернутый пример использования списков **TList** мы рассматривать не будем. Предоставляю вам возможность самим повторить на основе таких списков пример, рассмотренный в разд. 3.3.2. Создайте также пример с возможностью сортировки по указанному пользователем критерию: алфавиту, году рождения, отделу с алфавитной последовательностью фамилии сотрудников внутри каждого отдела. Подобные примеры позволят вам закрепить сведения о классе **TList**, изложенные выше.

3.3.4. Списки строк TStrings и TStringList

В Delphi имеется два класса, представляющие объекты списков строк. Один из них — **TStrings**, с которым вы уже не раз имели дело. Именно такого типа свойство **Lines** в окнах редактирования **Memo** и **RichEdit**, а также во многих других компонентах, с которыми мы познакомимся позднее. И когда вы добавляли строки в окна редактирования методом **Add**, вы использовали метод класса **TStrings**.

Класс **TStrings** используется в Delphi только для свойств различных компонентов. Объект этого класса вы не можете создавать в своем приложении. Но имеется другой класс списков строк — **TStringList**. Он наследует классу **TStrings**, наследует все свойства и методы своего родительского класса, и добавляет некоторые новые возможности: сортировку строк и контроль наличия в списке дубликатов — одинаковых строк.

Основное свойство обоих классов, дающее доступ к отдельным строкам списка — **Strings[Index: Integer]: string**. Это индексированное множество строк. Индексы отсчитываются от 0. Так что **Strings[0]** — это первая строка списка. Свойство **Text** содержит все строки списка в виде одной строки с разделителями. Свойство **Count** содержит число строк списка. В отличие от рассмотренного ранее класса **TList**, в списках строк это свойство только для чтения. Оно автоматически изменяется при добавлении и удалении строк, но программно изменять его нельзя. Свойство **Capacity** указывает число строк, которое может содержать список. Это свойство аналогично такому же свойству в **TList**.

Каждой строке списка может быть сопоставлен некоторый объект, наследующий базовому классу всех объектов в Delphi **TObject**. Далее у нас будет пример, в котором мы будем использовать подобные объекты. Доступ к объектам дает свойство **Objects[Index: Integer]: TObject**. Это индексированное множество, причем индексы совпадают с индексами строк в свойстве **Strings**. Так что, например, объект **Objects[0]** соответствует строке **Strings[0]**.

Класс **TStringList** имеет несколько свойств, отсутствующих в **TStrings**. Булево свойство **Sorted** указывает, должны ли строки в списке автоматически сортироваться по алфавиту. По умолчанию оно равно **false**, так что по умолчанию список не сортированный. Свойство **Duplicates** определяет, могут ли добавляться в сортированный список дубликаты строк. Возможные значения этого свойства: **dupIgnore** — игнорирование добавления дубликата (это значение по умолчанию), **dupAccept** — разрешение добавления дубликата, **dupError** — генерация исключения **EListError** при попытке добавления дубликата строки. Значения **dupAccept** и **dupError** никак не реагируют на уже имеющиеся в списке дубликаты. На несортированный список свойство **Duplicates** не оказывает никакого влияния. Булево свойство **CaseSensitive** определяет, учитывается ли регистр при сортировке и выявлении дублей. По умолчанию **false** — не учитывается.

Остановимся на этих свойствах и обсудим методы рассматриваемых классов. Основные методы приведены в следующей таблице:

Метод	Объявление / Описание
Add	**function Add(const S: string): Integer;** Добавляет строку **S** в конец несортированного списка или в необходимую позицию сортированного списка. Возвращает индекс добавленной строки.
AddObject	**function AddObject(const S: string; AObject: TObject): Integer;** Добавляет строку **S** и связанный с ней объект **AObject** в конец несортированного списка или в необходимую позицию сортированного списка. Возвращает индекс добавленной строки.
AddStrings	**procedure AddStrings(Strings: TStrings);** Добавляет в данный список строки и связанные с ними объекты из другого списка **Strings** класса **TStrings**.
Clear	**procedure Clear;** Очищает список.
CustomSort	**type TStringListSortCompare = function(List: TStringList; Index1, Index2: Integer): Integer;** **procedure CustomSort(Compare: TStringListSortCompare);** Обеспечивает нестандартную сортировку списка **List** с помощью функции сравнения **Compare**. Функция **Compare** должна возвращать отрицательное число, если строка с индексом **Index1** должна располагаться раньше строки с индексом **Index2**, положительное число в противном случае, и должна возвращать 0, если приоритет обеих строк одинаков.
Delete	**procedure Delete(Index: Integer);** Удаляет из списка строку с индексом **Index**.
Exchange	**procedure Exchange(Index1, Index2: Integer);** Переставляет местами строки списка с индексами **Index1** и **Index2**.

Метод	Объявление / Описание
Find	**function Find(const S: string; var Index: Integer): Boolean;** Определяет, имеется ли заданная строка **S** в сортированном списке, и, если имеется, то возвращает в параметр **Index** индекс этой строки. Для не сортированных списков следует использовать метод **IndexOf**.
IndexOf	**function IndexOf(const S: string): Integer;** Возвращает индекс указанной строки **S**. Если такой строки нет в списке, возвращается –1.
Insert	**procedure Insert(Index: Integer; const S: string);** Вставляет указанную строку **S** в заданную позицию **Index**. Если **Index** = 0, строка вставляется в первую позицию.
Move	**procedure Move(CurIndex, NewIndex: Integer);** Перемещает строку из позиции **CurIndex** в позицию **NewIndex**.
Sort	**procedure Sort;** Сортирует строки списка, свойство **Sorted** которого установлено в **false**, в возрастающей алфавитной последовательности. Если **Sorted** = **true**, то список сортируется автоматически.

Многие из этих методов аналогичны рассмотренным в разд. 3.3.3 для класса **TList** и вряд ли нуждаются в пояснении. Так что можно приступать сразу к созданию приложений на основе **TStringList**. Начните с предельно простого приложения, содержащего окно **Edit**, окно **Memo** и одну кнопку. Пусть по щелчку на кнопке строка, записанная в окне **Edit**, добавляется в список, а затем этот дополненный список отображается в окне **Memo**.

Введите в модуль глобальную переменную **List** класса **TStringList**:

```
var List: TStringList;
```

В обработчик события **OnCreate** формы введите оператор:

```
List := TStringList.Create;
```

Этот оператор создает список **List**, вызывая конструктор **Create** класса **TStringList**.

В обработчик события **OnDestroy** формы введите оператор:

```
List.Free;
```

Этот оператор удаляет из памяти список **List**. Способы создания и удаления объектов уже обсуждались в раз. 3.3.3.

В обработчик щелчка на кнопке введите код:

```
List.Add(Edit1.Text);
Memo1.Lines.Assign(List);
```

Первый из этих операторов добавляет строку в список. Второй оператор присваивает содержимое списка свойству **Lines** окна **Memo1**. Это свойство имеет тип **TStrings**, так что может воспринимать список строк.

Приложение завершено. Можете выполнить его и убедиться, что список заполняется строками. Чтобы посмотреть сортировку, можете добавить в обработчик события **OnCreate** формы оператор:

```
List.Sorted := true;
```

Тогда вы сможете убедиться, что строки вставляются в список по алфавиту, а дубли строк вставить невозможно. Можете также поварьировать свойствами **Duplicates** и **CaseSensitive**, чтобы увидеть, как они влияют на сортировку и попытки включения дублей строк.

Можете опробовать другие методы списка. Например, можете добавить в приложение кнопку, по щелчку на которой из списка должна удаляться строка, текст которой записан в окне **Edit1**. Обработчик щелчка на этой кнопке может иметь вид:

```
if List.IndexOf(Edit1.Text) > 0
 then List.Delete(List.IndexOf(Edit1.Text));
Memo1.Lines.Assign(List);
```

Структура **if** проверяет, имеется ли в списке требуемая строка. Если имеется (метод **IndexOf** возвращает неотрицательный индекс строки), то строка удаляется методом **Delete**.

А теперь давайте создадим значительно более сложное приложение, вид которого показан на рис. 3.9. Верхняя часть формы отображает список подразделений некоторой организации (или список учебных групп, классов — что вам больше нравится). Кнопка Добавить позволяет добавить в список строку, текст которой записан в верхнем окне редактировании **Edit**. Кнопка Удалить позволяет удалить соответствующую строку из списка.

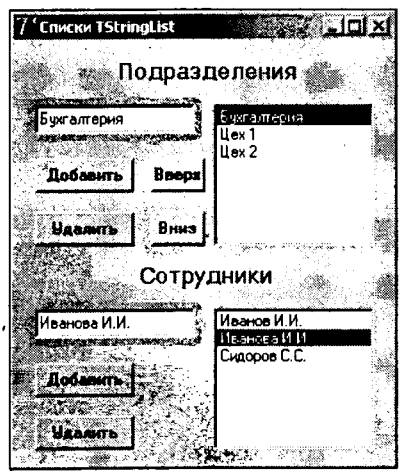


Рис. 3.9
Приложение, основанное на списках TStringList

Список подразделений не упорядочен по алфавиту, так как их последовательность в списке может определяться пользователем из каких-то своих соображений. Например, он может группировать подразделения по степени их важности, по группам сходных функций (управление, производство) и т.п. Поэтому в приложение введены кнопки Вверх и Вниз, позволяющие перемещать в списке выделенную пользователем строку.

Нижняя часть формы отображает список сотрудников (студентов, учащихся) подразделения, соответствующего строке, которую пользователь выбрал в верхнем списке. Так что, перемещаясь по верхнему списку, пользователь будет видеть в нижнем списке сотрудников соответствующего подразделения. Кнопка Добавить в нижней части формы позволяет добавить в соответствующее подразделение сотрудника, фамилия которого записана в нижнем окне редактировании **Edit**. Кнопка Удалить в нижней части формы позволяет удалить соответствующего сотрудника из списка подразделения. Строки в списке располагаются в алфавитном порядке.

Все основные операции этого приложения, как вы увидите, будут выполнены с помощью методов и свойств класса **TStringList**. Но для отображения списков нам потребуется компонент **ListBox**, который пока не рассматривался. Этот компонент вы найдете на странице *Standard* палитры компонентов. Его основное свойство **Items** имеет тип **TStrings**, так что может отображать наши списки строк. Свойство только времени выполнения **ItemIndex** указывает индекс строки, которую выделил пользователь, работая со списком. Если **ItemIndex** = −1, значит пользователь не выделил ни одну строку. Свойство только времени выполнения **Count** указывает число строк в списке. Остальные свойства компонента **ListBox** в данном приложении нам не потребуются. Так что рассмотрим их много позднее в другой главе.

Теперь давайте подумаем, как можно реализовать задуманное приложение. Нам надо иметь вложенные списки. Основной список содержит названия подразделений. Каждой его строке, т.е. каждому подразделению соответствует свой список сотрудников. Тут нам поможет описанное ранее свойство **Objects**. Для каждой строки основного свойства мы можем создавать свой список сотрудников и включать его как объект, связанный с данной строкой.

Ниже приведен код этого приложения.

```
var List, LTmp: TStringList;

procedure TForm1.FormCreate(Sender: TObject);
begin
 // Создание основного списка
 List := TStringList.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
var i: integer;
begin
 // Удаление списков, связаных со строками списка List
 for i:=0 to List.Count - 1 do
  List.Objects[i].Free;
 // Удаление списка List
 List.Free;
end;

procedure TForm1.BAdd1Click(Sender: TObject);
// Добавление строки в основной список
begin
 // Если строка уже есть в списке - уход
 if(List.IndexOf(Edit1.Text) >= 0) then exit;
 // Создание вспомогательного списка
 LTmp := TStringList.Create;
 LTmp.Sorted := true;
```

```
 // Добавление строки и вспомогательного списка в List
 List.AddObject(Edit1.Text, LTmp);
 // Передача основного и вспомогательного списков в ListBox
 ListBox1.Items.Assign(List);
 ListBox1.ItemIndex := List.IndexOf(Edit1.Text);
 ListBox2.Items.Assign(LTmp);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
// Обработка щелчка в ListBox1
var Ind: integer;
begin
 Ind := ListBox1.ItemIndex;
 // Если нет выделенной строки - уход
 if Ind < 0 then exit;
 // Занесение выделенной строки в Edit1
 Edit1.Text := ListBox1.Items[Ind];
 // Занесение в ListBox2 вспомогательного списка
 ListBox2.Items.Assign(List.Objects[Ind] as TStrings);
 // Если вспомогательный список не пустой - настройка ListBox2
 if (ListBox2.Count > 0
  then begin
   ListBox2.ItemIndex := 0;
   Edit2.Text := ListBox2.Items[0];
  end;
end;

procedure TForm1.BDel1Click(Sender: TObject);
// Удаление строки из основного списка
begin
 // Если строки из Edit1 нет в списке - уход
 if List.IndexOf(Edit1.Text) < 0 then exit;
 // Удаление вспомогательного списка и строки основного списка
 List.Objects[List.IndexOf(Edit1.Text)].Free;
 List.Delete(List.IndexOf(Edit1.Text));
 // Настройка ListBox1 и ListBox2
 ListBox1.Items.Assign(List);
 if List.Count > 0
  then ListBox1.ItemIndex := 0;
 ListBox1Click(Sender);
end;

procedure TForm1.ListBox2Click(Sender: TObject);
// Обработка щелчка в ListBox2
begin
 Edit2.Text := ListBox2.Items[ListBox2.ItemIndex];
end;

procedure TForm1.BAdd2Click(Sender: TObject);
// Добавление строки во вспомогательный список
begin
 LTmp := List.Objects[ListBox1.ItemIndex] as TStringList;
 LTmp.Add(Edit2.Text);
 ListBox2.Items.Assign(LTmp);
end;
```

```
procedure TForm1.BDel2Click(Sender: TObject);
// Удаление строки из вспомогательного списка
begin
 LTmp := List.Objects[ListBox1.ItemIndex] as TStringList;
 if LTmp.IndexOf(Edit2.Text) < 0 then exit;
 LTmp.Delete(LTmp.IndexOf(Edit2.Text));
 ListBox2.Items.Assign(LTmp);
 if LTmp.Count > 0
  then ListBox2.ItemIndex := 0;
end;

procedure TForm1.BUpClick(Sender: TObject);
// Обработка щелчка на кнопке Вверх
var Ind: integer;
begin
 Ind := ListBox1.ItemIndex;
 if Ind > 0
 then begin
  List.Move(Ind, Ind - 1);
  ListBox1.Items.Assign(List);
  ListBox1.ItemIndex := Ind - 1;
  ListBox1Click(Sender);
 end;
end;

procedure TForm1.BDownClick(Sender: TObject);
// Обработка щелчка на кнопке Вниз
var Ind: integer;
begin
 Ind := ListBox1.ItemIndex;
 if Ind < ListBox1.Count - 1
 then begin
  List.Move(Ind, Ind + 1);
  ListBox1.Items.Assign(List);
  ListBox1.ItemIndex := Ind + 1;
  ListBox1Click(Sender);
 end;
end;
```

Код снабжен развернутыми комментариями, так что можно ограничиться сравнительно краткими пояснениями. В приложении объявлены две глобальные переменные типа **TStringList**. Переменная **List** будет содержать основной список подразделений. А переменная **LTmp** будет использоваться для создания вспомогательных списков сотрудников.

В обработчике **FormCreate** события **OnCreate** формы создается основной список **List**. В обработчике **FormDestroy** события **OnDestroy** формы этот список удаляется. Но предварительно в цикле удаляются все вспомогательные списки **Objects[i]**, связанные со строками основного.

Процедура **BAdd1Click** является обработчиком щелчка на верхней кнопке Добавить. Первый оператор процедуры проверяет вызовом метода **IndexOf**, нет ли в списке строки, текст которой задан в окне **Edit1**. Дело в том, что мы не хотим, как было сказано ранее, сортировать основной список. Свойство **Duplicates** для несортированных списков не работает. Но нам все-таки надо предотвратить ошиб-

бочное добавление в список строк, которые там уже есть. Это и осуществляется первым оператором процедуры.

Второй оператор создает в переменной **LTmp** новый список класса **TStringList**. Следующий оператор устанавливает в списке **LTmp** свойство **Sorted**, чтобы этот список был сортирован по алфавиту. Далее методом **AddObject** строка текста из **Edit1** и созданный список добавляются в список **List**. В **ListBox1** заносятся строки обновленного списка **List**. Свойство **ItemIndex** задается равным индексу занесенной строки. В **ListBox2** передается список **LTmp**, пока пустой.

Процедура **ListBox1Click** является обработчиком щелчка пользователя в окне компонента **ListBox1**. Этот щелчок приводит к выделению в списке новой строки. Значит, в окно **Edit1** надо занести текст выделенной строки, а в компонент **ListBox2** занести строки соответствующего списка сотрудников. Первый оператор заносит в локальную переменную **Ind** индекс строки, выделенной в **ListBox1**. Если ничего не выделено, этот индекс равен −1, и выполнение процедуры заканчивается. При наличии выделения в окно **Edit1** и в **ListBox2** заносятся соответствующие данные. Обратите внимание, что при занесении списка строк в **ListBox2** использовано выражение **List.Objects[Ind] as TStringList**. Дело в том, что объекты в свойстве **Objects** имеют тип **TObject**. И для того, чтобы компилятор понял, что этот объект можно присвоить свойству типа **TStrings**, надо указать, что объект следует рассматривать как **(as)** объект типа **TStrings**. Далее, если список, занесенный в **ListBox2**, не пустой, то производится задание свойства **ItemIndex** компонента **ListBox2**, и заносится текст в окно **Edit2**.

Процедура **BDel1Click** является обработчиком щелчка на верхней кнопке Удалить. Первый оператор процедуры проверяет вызовом метода **IndexOf**, есть ли в списке строка, текст которой задан в окне **Edit1**. Если есть, то соответствующий ей объект удаляется методом **Free**, после чего удаляется методом **Delete** и сама строка. Измененный список передается в компонент **ListBox1**. Если список не пустой, то индекс **ItemIndex** компонента **ListBox1** устанавливается на первую строку. В заключение вызывается описанная выше процедура **ListBox1Click**, чтобы завершить настройку компонентов, отображающих списки.

Процедура **ListBox2Click** является обработчиком щелчка пользователя в окне компонента **ListBox2**. В ней просто заносится в окно **Edit2** текст строки, выделенной пользователем в **ListBox2**.

Процедуры **BAdd2Click** и **BDel2Click** являются обработчиками щелчков на нижних кнопках Добавить и Удалить. Вряд ли они нуждаются в особых комментариях. Отметим только, что так же, как в процедуре **ListBox1Click**, при работе со свойством **Objects** надо приводить объект операцией **as** к типу объекта **LTmp** — в данном случае к **TStringList**.

Процедуры **BUpClick** и **BDownClick** являются обработчиками щелчков на кнопках Вверх и Вниз. Начинаются эти процедуры с проверки индекса строки, выделенной в компоненте **ListBox1**, так как если индекс равен 0, то строку невозможно сдвинуть вверх, а если индекс соответствует последней строке, то строку невозможно сдвинуть вниз. Затем методом **Move** строка в списке **List** сдвигается в соответствующем направлении, и отображение в **ListBox1** приводится в соответствие с новым состоянием списка.

Опробуйте ваше приложение в работе. Оно очень неплохое и может быть приспособлено для решения самых различных задач. Но в нем пока имеется очень большой недостаток — сформированные списки исчезают, как только выполнение

приложения завершается. В это приложение, как и в некоторые другие, созданные вами ранее, совершенно необходимо включить запоминание результатов работы в файлах. В разд. 3.4 мы увидим, как это можно сделать.

3.4 Файлы

3.4.1 Диалоги открытия и сохранения файлов

Работа с файлами — неотъемлемая часть практически любого приложения. Какие бы операции пользователь ни выполнял с помощью приложения: редактирование текстов, изображений, осуществление каких-то расчетов, всегда требуется сохранить результаты работы в файле, а при следующем сеансе работы с приложением надо иметь возможность открыть файл и просмотреть результаты или продолжить работу. При открытии файла обычно надо предоставить пользователю возможность выбрать требуемый файл, а при сохранении — указать имя файла и каталог. Все это желательно осуществлять с помощью стандартных диалогов Windows. Поэтому мы начнем рассмотрение работы с файлами именно с этих диалогов.

В палитре компонентов Delphi на странице *Dialogs* имеются компоненты **OpenDialog** — диалог «Открыть файл» и **SaveDialog** — диалог «Сохранить файл как». Примеры вызываемых ими диалоговых окон приведены на рис. 3.10 и 3.11. Все свойства компонентов **OpenDialog** и **SaveDialog** одинаковы, только их смысл несколько различен для открытия и сохранения файлов. Основное свойство, в котором возвращается в виде строки выбранный пользователем файл, — **FileName**. Значение этого свойства можно задать и перед обращением к диалогу. Тогда оно появится в диалоге как значение по умолчанию в окне Имя файла (см. рис. 3.10, 3.11).

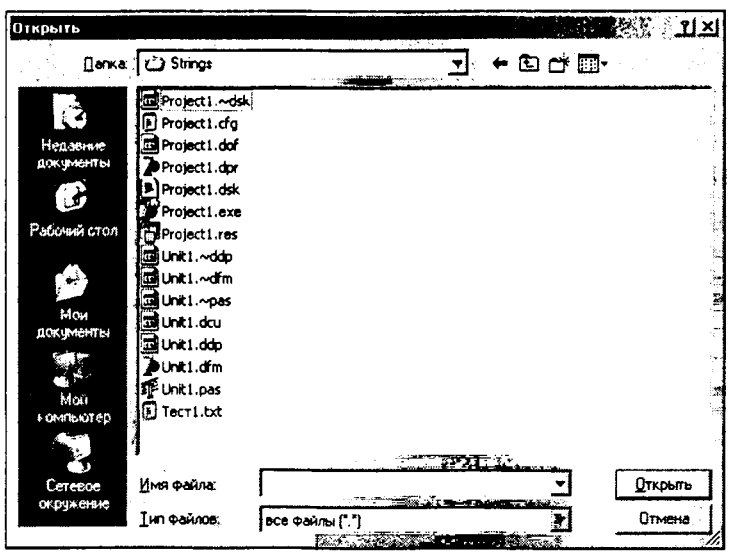


Рис. 3.10
Диалоговое окно открытия файла

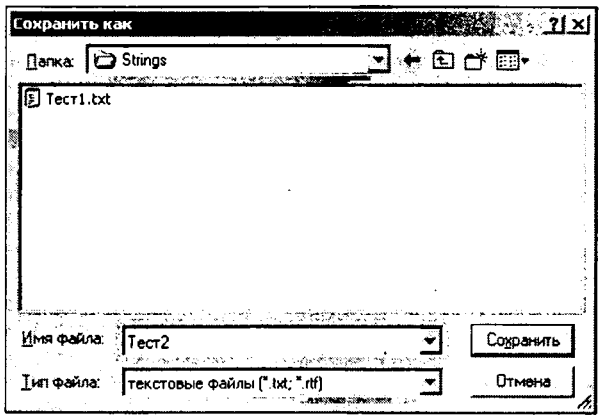


Рис. 3.11
Диалоговое окно сохранения файла

Типы искомых файлов, появляющиеся в диалоге в выпадающем списке Тип файла (рис. 3.10, 3.11), задаются свойством **Filter**. В процессе проектирования это свойство проще всего задать с помощью редактора фильтров, который вызывается нажатием кнопки с многоточием около имени этого свойства в Инспекторе Объектов. При этом открывается окно редактора, вид которого представлен на рис. 3.12. В его левой панели Filter Name вы записываете тот текст, который увидит пользователь в выпадающем списке Тип файла диалога. А в правой панели Filter записываются разделенные точками с запятой шаблоны фильтра. В примере рис. 3.12 задано два фильтра: текстовых файлов с расширениями *.txt* и *.rtf* и любых файлов с шаблоном "*.*".

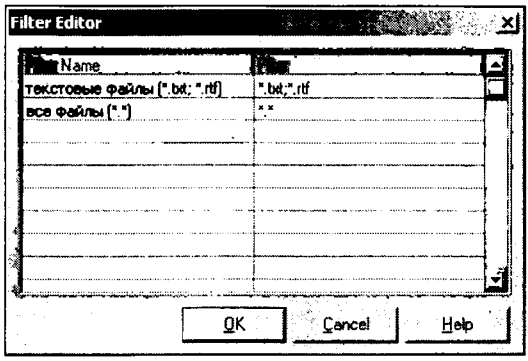


Рис. 3.12
Окно редактора фильтров

После выхода из окна редактирования фильтров заданные вами шаблоны появятся в свойстве **Filter** в виде строки вида:

```
текстовые файлы (*.txt; *.rtf)|*.txt;*.rtf|все файлы (*.*)|*.*
```

В этой строке тексты и шаблоны разделяются вертикальными линиями. В аналогичном виде, если требуется, можно задавать свойство **Filter** программно во время выполнения приложения.

Свойство **FilterIndex** определяет номер фильтра, который будет по умолчанию показан пользователю в момент открытия диалога. Например, значение **FilterIndex** = 1 задает по умолчанию первый фильтр.

Свойство **InitialDir** определяет начальный каталог, который будет открыт в момент начала работы пользователя с диалогом. Если значение этого свойства не задано, то открывается текущий каталог или тот, который был открыт при последнем обращении пользователя к соответствующему диалогу в процессе выполнения данного приложения.

Свойство **DefaultExt** определяет значение расширения файла по умолчанию. Если значение этого свойства не задано, пользователь должен указать в диалоге полное имя файла с расширением. Если же задать значение **DefaultExt**, то пользователь может писать в диалоге имя без расширения. В этом случае будет принято заданное расширение.

Свойство **Title** определяет заголовок диалогового окна. Если это свойство не задано, будут показаны стандартные заголовки «Открыть» и «Сохранить как», которые вы можете видеть на рис. 3.10 и 3.11.

Свойства **Options** и **OptionsEx** определяют условия выбора файла и вид диалога. Чаще всего все эти опции можно оставить без изменения. Отметим только опцию **ofExNoPlacesBar** в свойстве **OptionsEx**. Она запрещает появление в диалоге полосы, обеспечивающей доступ к папкам Недавние документы, Рабочий стол и т.п. В приведенных выше примерах на рис. 3.10 эта опция выключена, а на рис. 3.11 — включена. Как видите, эта опция кардинально меняет вид диалогового окна. По умолчанию она выключена, но на мой взгляд ее обычно стоит включать. Впрочем, эта опция и само свойство **OptionsEx** существуют, только начиная с Delphi 6. Учтите также, что вид диалоговых окон зависит от версии Windows, установленной на компьютере. Приведенные рисунки относятся к Windows XP.

3.4.2 Методы LoadFromFile и SaveToFile

У классов **TStrings**, **TStringList** и многих других имеются методы **LoadFromFile** и **SaveToFile**:

```
procedure LoadFromFile(const FileName: string);
procedure SaveToFile(const FileName: string);
```

Первая из этих процедур обеспечивает загрузку в объект содержимого файла **FileName**, а вторая сохраняет в файле **FileName** содержимое объекта. Аргумент **FileName** может содержать полное имя файла с путем. Если же указано только имя, то подразумевается текущий каталог.

В частности, свойства **Lines** окон **Memo** и **RichEdit** имеют тип **TStrings**, и, значит, имеют методы **LoadFromFile** и **SaveToFile**, обеспечивающие соответственно загрузку текста из файла и сохранение текста в файле.

Рассмотрим примеры использования методов **LoadFromFile** и **SaveToFile**, а также описанных в разд. 3.4.1 диалогов **OpenDialog** и **SaveDialog**. Откройте тот проект текстового редактора на основе компонента **RichEdit**, который вы создавали в разд. 3.2. Добавьте в него кнопки Открыть, Сохранить и Сохранить как. При щелчке на кнопке Открыть пользователь сможет загрузить в окно **RichEdit1** текстовый файл. При щелчке на кнопке Сохранить он сможет сохранить открытый ранее файл после каких-то изменений в тексте. А при щелчке на кнопке Сохранить как он сможет сохранить текст в файле с указанным им именем.

Поскольку после чтения файла вам надо запомнить его имя, чтобы знать, под каким именем потом его сохранять, вы можете определить для этого имени глобальную переменную, назвав ее, например, **FName**:

```
var FName:string = 'Неизвестный';
```

Переменная инициализируется текстом «Неизвестный», чтобы впоследствии при сохранении можно было понять, был ли загружен в окно **RichEdit1** какой-то файл, или сохраняется просто набранный в окне текст.

Обработка команды Открыть может сводиться к следующему (имя соответствующей кнопки принято равным **BOpen**):

```
procedure TForm1.BOpenClick(Sender: TObject);
begin
 if OpenDialog1.Execute
  then begin
   // запоминание имени файла
   FName := OpenDialog1.FileName;
   RichEdit1.Lines.LoadFromFile(FName);
end;
end;
```

Этот оператор вызывает диалог, проверяет, выбрал ли пользователь файл (если выбрал, то функция **Execute** возвращает **true**), после чего имя выбранного файла (**OpenDialog1.FileName**) сохраняется в переменной **FName** и файл загружается в текст **RichEdit1** методом **LoadFromFile**.

Обработка команды Сохранить как выполняется операторами (имя кнопки — **BSaveAs**):

```
procedure TForm1.BSaveAsClick(Sender: TObject);
begin
 // задание имени файла по умолчанию
 SaveDialog1.FileName := FName;
 if SaveDialog1.Execute
  then begin
   // сохранение имени файла в FName
   FName := SaveDialog1.FileName;
   RichEdit1.Lines.SaveToFile(FName);
  end;
end;
```

Обработка команды Сохранить выполняется операторами (имя кнопки — **BSave**):

```
procedure TForm1.BSaveClick(Sender: TObject);
begin
 if (FName = 'Неизвестный')
  // вызов процедуры Сохранить как
  then BSaveAsClick(Sender)
  else RichEdit1.Lines.SaveToFile(FName);
end;
```

Если в переменной **FName** записано имя «Неизвестный», то вызывается приведенная выше процедура **BSaveAsClick**, так как надо дать пользователю возможность задать имя файла. Если же в переменной **FName** записано имя файла, то текст окна **RichEdit1** сохраняется без вызова диалога.

Мы рассмотрели организацию работы с файлами применительно к окну редактирования **RichEdit**. Для окна **Memo** все может быть организовано точно так же, только имя окна, конечно, в приведенных кодах надо изменить.

Теперь рассмотрим использование методов **LoadFromFile** и **SaveToFile**: класса **TStringList**. В разд. 3.3.4 вы создали очень неплохое приложение (рис. 3.9), работающее со списками строк. Но у него был крупный недостаток — невозможность сохранять списки строк в файлах. Для того чтобы исправить этот недостаток, можно изменить в приложении обработчики событий **OnCreate** и **OnDestroy** формы следующим образом:

```
procedure TForm1.FormDestroy(Sender: TObject);
var i: integer;
begin
// Запоминание и удаление списков, связаных со строками List
 for i:=0 to List.Count - 1 do
 begin
  (List.Objects[i] as TStringList).SaveToFile(List.Strings[i] + '.txt');
  List.Objects[i].Free;
 end;
// Запоминание и удаление списка List
 List.SaveToFile('MyList.txt');
 List.Free;
end;

procedure TForm1.FormCreate(Sender: TObject);
var i: integer;
begin
 // Создание основного списка
 List := TStringList.Create;
 // Если имеется файл MyList.txt, то список читается из него
 if FileExists('MyList.txt')
  then begin
   // Чтение основного списка
   List.LoadFromFile('MyList.txt');
   for i:=0 to List.Count - 1 do
   begin
    // Создание и чтение вспомогательных списков
    LTmp := TStringList.Create;
    LTmp.Sorted := true;
    LTmp.LoadFromFile(List.Strings[i] + '.txt');
    List.Objects[i] := LTmp;
   end;
   // Передача списов в ListBox
   ListBox1.Items.Assign(List);
   ListBox1.ItemIndex := 0;
   ListBox1Click(Sender);
  end;
end;
```

Начнем с процедуры **FormDestroy**, являющейся обработчиком события формы **OnDestroy**. В этом обработчике, срабатывающем при завершении работы приложения, надо сохранить в файлах строки основного списка **List**, и вспомогательные списки, относящиеся к каждой строке. Кроме того, как и в прежнем приложении, надо освободить память от объектов основного и вспомогательных списков.

В начале процедуры **FormDestroy** в цикле просматриваются все объекты, связанные со строками основного списка. Вспомогательные списки, рассматриваемые как объекты класса **TStringList** (см. разд. 3.3.4), сохраняются их методом **SaveToFile** в файлах. Имя каждого файла формируется как имя соответствующей строки основного списка, плюс расширение ".txt". Например, "Бухгалтерия.txt". В том же цикле после запоминания объекты вспомогательных списков удаляются из памяти методом **Free**. После окончания цикла основной список запоминается в файле с именем "MyList.txt" и удаляется из памяти. Все файлы создаются в текущем каталоге, так как мы не добавляем путь в их имена.

Теперь рассмотрим процедуру **FormCreate**, являющейся обработчиком события формы **OnCreate**. В этом обработчике надо создать объект основного списка **List** и прочитать строки основного и вспомогательного списков из файлов. Первый оператор процедуры создает объект основного списка **List**. Далее проверяется, имеется ли на диске в текущем каталоге файл *MyList.txt*. Если имеется, то содержимое этого файла загружается в список **List** методом **LoadFromFile**. Затем следует цикл по строкам основного списка, для каждой строки создается объект вспомогательного списка, и в него загружаются строки из соответствующего файла.

Введите рассмотренные дополнения в ваше приложение, и вы получите хороший инструмент для решения самых различных задач. Это может быть список адресов и телефонов ваших друзей, сгруппированный по каким-то категориям. Или список ваших любимых популярных групп, и для каждой — список их песен с пометками, ест ли они у вас. Или список комплектующих какой-то системы и по каждому изделию списки поставщиков (или магазинов) с адресами, ценами и т.д. Словом, вы сами найдете применение для подобного приложения. Только, наверное, стоит добавить в него кнопки Изменить, по щелкам на которых можно изменять текст выделенной строки.

3.4.3 Организация файлового ввода/вывода

В разд. 3.4.2 была рассмотрена работа с текстовыми файлами, которые загружаются в окна редактирования и другие компоненты, чтобы пользователь мог их посмотреть и отредактировать. Но очень часто в приложениях используются файлы, которые не предназначены для непосредственного просмотра пользователем. Это могут быть результаты вычислений, списки строк или иных объектов, которые или используются программой в дальнейших вычислениях, или перед показом пользователю должны пройти какую-то обработку. Рассмотрим работу с подобными файлами.

Для работы с файлом должна быть определена файловая переменная, соответствующая типу файла. Затем эта файловая переменная связывается с конкретным файлом, и через нее осуществляются все операции с файлом.

Ниже приведены примеры объявления файловых переменных:

```
var F1: TextFile;
    F2: Text;
    F3: file of real;
    F4: file;
```

Первые два объявления вводят переменные **F1** и **F2**, которые могут связываться с *текстовыми файлами* (начиная с Delphi 6, тип **Text** исчез из языка, и можно использовать только тип **TextFile**). Переменная **F3** может связываться с файлом, содержащим действительные числа. Подобные файлы, содержащие значения одного

(любого) типа называются *типизированными файлами*. Переменная **F4**, в которой тип указан как просто **file**, может связываться с файлами, содержащими данные различного типа. Такие файлы называются *нетипизированными файлами*.

В последующих разделах мы рассмотрим особенности работы со всеми этими видами файлов. А данный раздел посвящен общим приемам работы с файлами любого вида.

Файловая переменная связывается с конкретным файлом процедурой **AssignFile** (в языке Pascal соответствующая процедура называлась **Assign**, но в Object Pascal ее имя было изменено, чтобы не возникало путаницы с одноименным методом — см. разд. 2.8.1). Эта процедура имеет синтаксис:

```
procedure AssignFile(var F: File, S: string);
```

где **F** — файловая переменная любого типа, **S** — строка, содержащая имя файла. Например, операторы

```
AssignFile(F1, 'Test.txt');
AssignFile(F2, OpenDialog1.FileName);
```

связывают файловые переменные **F1** и **F2** соответственно с файлом "Test.txt" и с файлом, имя которого записано в свойстве **FileName** компонента-диалога **OpenDialog1**.

Открытие существующего файла осуществляется процедурой **Reset**, формат которой (кроме нетипизированных файлов — см. разд. 3.4.6) следующий:

```
procedure Reset(var F: File);
```

Файловая переменная **F** перед обращением к этой процедуре должна быть связана с файлом.

Создание и открытие нового файла осуществляется процедурой **Rewrite**, формат которой (кроме нетипизированных файлов — см. разд. 3.4.6) следующий:

```
procedure Rewrite(var F: File);
```

Если файл уже существует, то его содержимое уничтожается.

После того как файл открыт, можно выполнять различных операций чтения и записи. Они будут рассмотрены в последующих разделах применительно к различным видам файлов. При чтении обычно надо проверять, есть ли в файле еще записи, не окончился ли он. Эта проверка осуществляется функцией **Eof(F: File)**, возвращающей **true** при достижении конца файла.

После окончания всех операций чтения/записи файл должен быть закрыт процедурой **CloseFile**:

```
procedure CloseFile(var F: File);
```

Предупреждение

Не забывайте закрыть файл процедурой CloseFile. Особенно это относится к файлам, в которые вы производите запись. Если вы завершите приложение, не закрыв такой файл, то он останется пустым, и все ваши записи пропадут.

При работе с файлами, при чтении и записи в них каких-то данных возможны различные ошибки. Если не принять соответствующих мер, то эти ошибки приведут к прерыванию работы программы. Предотвратить это можно двумя путями.

Первый предполагает обработку исключений **EInOutError**, поле **errorcode** которых содержит информацию о конкретном виде ошибки. Второй предполагает применение директивы компилятора (см. разд. 2.2) **{$I–}**, отключающей генерацию исключений ошибок ввода/вывода, и дальнейшее обращение к функции **IOResult**, возвращающей код ошибки и сбрасывающей его в 0 (так что повторное обращение к **IOResult** бессмысленно). После обработки операции ввода/вывода контроль ошибок можно снова включить опцией **{$I+}**. В этом случае общая организация работы с файлами строится примерно по такой схеме:

```
AssignFile(F, OpenDialog1.FileName);
{$I-}
Reset(F);
{$I+}
I:=IOResult;
if I <> 0 then <операторы обработки кода ошибки>
 else ...
...
{$I-}
<операторы чтения и записи>
{$I+}
I:=IOResult;
if I <> 0 then <операторы обработки кода ошибки>
...

CloseFile(F);
```

Проверку значения **IOResult** и обработку кода ошибки удобно оформить в виде функции, например, следующего вида:

```
function IO(s: string): boolean;
var i: integer;
begin
 i:=IOResult;
 Result := (i = 0);
 if i<>0 then
 begin
  case i of
      2: s:='Файл "' + s + '" не найден';
      3: s:='Ошибочное имя файла "' + s + '"';
      4: s:='Слишком много открытых файлов';
      5: s:= 'Файл "' + s + '" не доступен';
   100: s:= 'Достигнут конец файла "' + s + '"';
   101: s:= 'Диск переполнен при работе с файлом "' + s + '"';
   103: s:= 'Файл "' + s + '" не открыт';
   104: s:= 'Файл "' + s + '" не открыт для чтения';
   105: s:= 'Файл "' + s + '" не открыт для записи';
   106: s:= 'Ошибка ввода при работе с файлом "' + s + '"';
  end;
  ShowMessage(s);
 end;
end;
```

Параметр функции **s** — имя файла, с которым ведется работа. Возвращаемое функцией значение равно **true**, если ошибки нет. При наличии ошибки структура **case** формирует строку сообщения, после чего функция **ShowMessage** показывает

пользователю окно со сформированным пояснением ошибки. Из приведенного текста вы можете видеть, что означают различные коды ошибок.

Подобную функцию имеет смысл включить в вашу библиотеку функций и процедур. При наличии такой функции приведенная ранее схема организации работы с файлами может быть изменена следующим образом:

```
AssignFile(F, OpenDialog1.FileName);
{$I-}
Reset(F);
{$I+}
if IO(OpenDialog1.FileName)
 then
...
{$I-}
<операторы чтения и записи>
{$I+}
I:=IOResult;
if IO(OpenDialog1.FileName)
 then
...
CloseFile(F);
```

С каждой файловой переменной **F** связана запись типа **TTextRec**, содержащая всю информацию, необходимую для работы файловой системы Object Pascal. При обычной работе с файлами бывает полезно только одно поле этой записи — **Name**, в котором записано имя файла, связанного с данной файловой переменной. Так что выражение

```
TTextRec(F).Name
```

даст вам имя файла, связанного в данный момент с файловой переменной **F**.

3.4.4 Текстовые файлы

Текстовые файлы состоят из последовательностей символов, разбитых на строки. Файл для записи открывается с помощью описанной в разд. 3.4.3 процедуры **Rewrite**. Если файл уже существует, то его содержимое уничтожается. Кроме процедуры **Rewrite**, для текстовых файлов имеется процедура **Append**. Она открывает существующий текстовый файл для добавления записей в его конец.

Запись данных в текстовый файл осуществляется процедурой

```
procedure Write(var F: TextFile; <список выражений>);
```

Выражения могут быть типов **Char**, **Integer**, **Real**, **string**, **boolean**. При этом может использоваться форматирование (см. функцию **Format** в разд. 3.2.1). Например:

```
Write(F, 'Вам ', I, ' лет');
Write(F, ', а ему больше.');
```

Эти два оператора запишут в файл строку текста вида «Вам ... лет, а ему больше», причем строка еще не будет завершена. Чтобы начать следующую строку, надо будет указать символ перевода строки (см. разд. 3.2.1). Например:

```
Write(F, #13'Привет!');
```

Процедура **Writeln** подобна **Write**, но отличается тем, что после записи автоматически пишет в файл символ перехода на новую строку. Так что каждый вызов **Writeln** формирует одну строку.

Файл для чтения открывается с помощью описанной в разд. 3.4.3 процедуры **Reset**. Чтение данных из текстового файла осуществляется последовательно от его начала процедурой

```
procedure Read(var F: TextFile; <список переменных>);
```

где в списке перечисляются переменные, в которые читаются данные из файла. Например, если определить переменные **S1** и **S2** как

```
var S1,S2:string[4];
```

то чтение первой строки, записанной в файл приведенными выше операторами, если открыть этот файл и использовать оператор

```
Read(F, S1, I, S2);
```

даст значение **S1**, равное «Вам », значение **I**, равное записанному числу лет, и значение **S2**, равное « лет». В данном случае первое слово отделено от записанного в тексте числа, благодаря тому, что переменная **S1** объявлена как короткая строка, содержащая 4 символа. Если бы объявление было такое:

```
var S1,S2:string;
```

то в переменную **S1** прочиталась бы вся строка текста, а в переменные **I** и **S2** ничего не занеслось бы.

Число **I** в приведенном примере прочитается верно, потому что оно отделено от следующего элемента строки разделителем — пробелом.

Предупреждение

Если вы храните в текстовом файле числа, то не забывайте при записи отделять их друг от друга пробелами. Иначе эти числа невозможно будет прочитать.

Процедура **Readln** аналогична **Read**, и отличается только тем, что после чтения переводит текущую позицию в файле на новую строку. Если в процедуре **Readln** не задан список переменных, то она просто пропускает текущую строку и переходит к следующей. Например, если вам надо прочитать в переменную **S** 11-ю строку файла, а предыдущие 10 вас не интересуют, то это можно сделать так:

```
for i:=1 to 10 do Readln(F);
Readln(F, S);
```

В заключение рассмотрим пример процедуры, которая читает текстовый файл (например, файл *.pas* какого-то проекта Delphi) и все строки, в которых в любом регистре встречается слово «procedure», заносит в окно **RichEdit1**. В итоге в окне создается список всех процедур проекта.

```
procedure TForm1.Button1Click(Sender: TObject);
var F: TextFile;
    S: string;
const S1 = 'PROCEDURE';
begin
 if not OpenDialog1.Execute then exit;
 AssignFile(F, OpenDialog1.FileName);
 {$I-}
 Reset(F);
 {$I+}
 if not IO(OpenDialog1.FileName) then exit;
```

```
 RichEdit1.Clear;
 while not Eof(F) do
 begin
  Readln(F, S);
  if Pos(S1, AnsiUpperCase(S)) > 0
   then RichEdit1.Lines.Add(S);
 end;
 CloseFile(F);
end;
```

В данной процедуре вызывается диалог **OpenDialog1**, позволяющий пользователю выбрать файл. Далее процедурой **Reset** файл открывается для чтения. Проверка ошибки открытия файла осуществляется реализованной вами в разд. 3.4.3 функцией **IO**. В цикле до тех пор, пока не достигнут конец файла (проверяется функцией **Eof**), строки файла читаются в переменную **S**. Для каждой строки функцией **Pos** (см. разд. 3.2.2) проверяется, не содержится ли в строке подстрока **S1**. При этом используется функция **AnsiUpperCase** (см. разд. 3.2.2), приводящая строку к верхнему регистру и тем самым исключающая влияние регистра, в котором набран текст. Если в строке содержится подстрока «procedure», то эта строка заносится в окно **RichEdit1**.

3.4.5 Типизированные файлы

Типизированные файлы являются двоичными файлами, содержащими последовательность однотипных данных. Объявление файловых переменных таких файлов имеет вид:

```
var <имя файловой переменной>: file of <тип данных>;
```

Например:

```
var F: file of real;
```

Тип данных может быть не только простым типом, но и, например, типом записей или объявленным вами типом статического массива.

Процедуры чтения и записи **Read** и **Write** не отличаются от рассмотренных ранее для текстовых файлов. Только типы выражений при записи и переменных при чтении должны соответствовать объявленному типу данных файла. Процедур, аналогичных **Readln** и **Writeln** для типизированных файлов нет.

Рассмотрим пример:

```
procedure TForm1.Button1Click(Sender: TObject);
type TA = array[1..10] of real;
var A, B, A1, B1: TA;
    F: file of TA;
    i: integer;
begin
 // Имитация заполнения массивов A и B
 for i:= 1 to 10 do
 begin
  A[i] := i;
  B[i] := 10 * i;
 end;

 AssignFile(F, 'MyArrays.dat');
```

```
// Запись массивов A и B в файл "MyArrays.dat"
Rewrite(F);
Write(F, A, B);
CloseFile(F);

// Чтение из файла "MyArrays.dat" в массивы A1 и B1
Reset(F);
Read(F, A1, B1);
CloseFile(F);
end;
```

В этой процедуре объявлены тип массива действительных чисел **TA**, четыре массива этого типа и файловая переменная файла с элементами этого типа. Массивы **A** и **B** могут хранить какие-то результаты расчетов. В этом примере они просто заполняются соответствующими числами. Затем эти массивы одним вызовом функции **Write** записываются в файл с именем "MyArrays.dat". Последний фрагмент кода иллюстрирует чтение этих данных в массивы **A1** и **B1**. Чтение осуществляется одним оператором **Read**.

Мы видим, что в типизированных файлах можно хранить, в частности, числа. Но ведь их можно хранить и в рассмотренных в разд. 3.4.4 текстовых файлах. Что лучше? С точки зрения эффективности типизированные файлы имеют несомненное преимущество, так как при записи и чтении чисел в текстовые файлы тратится время на взаимное преобразование чисел и строк. Но с точки зрения отладки текстовые файлы удобнее, так как их можно просматривать любым текстовым редактором. Поэтому могу посоветовать следующее:

Совет

Если вы можете в равной степени использовать для хранения данных типизированные и текстовые файлы, то пока идет отладка проекта лучше работать с текстовыми файлами, в которых вы легко можете просматривать содержимое. Только не забывайте при записи чисел в текстовые файлы заносить пробелы межу ними. А когда отладка закончена, надо перевести текстовые файлы в типизированные, чтобы обеспечить хорошую эффективность приложения.

В этом совете я недаром оговорил: «Если вы можете ...». Дело в том, что текстовые файлы обладают неприятной особенностью: читать информацию в них можно только от начала файла к его концу. Так что если, например, вам надо в данный момент прочитать число, записанное в 100-й строке, вам придется предварительно 99 раз вызвать функцию **Readln** (см. разд. 3.4.4). А для типизированных файлов имеется процедура **Seek**, позволяющая перемещаться по файлу не только последовательно, как в текстовых файлах, но сразу переходить к требуемому элементу. Ее синтаксис:

```
procedure Seek(var F: File, N: Longint);
```

где **N** — номер элемента. Эта процедура перемещает текущую позицию в файле на нужный элемент. Номера позиций отсчитываются от 0. Так что для того, чтобы прочитать в файле действительных чисел 100-е число, достаточно выполнить операторы:

```
Seek(F, 99);
Read(F, R);
```

Каждый оператор **Read** или **Write** читает или записывает данные в текущую позицию в файле, и затем увеличивает эту позицию на 1.

Открытие типизированного файла процедурой **Reset** отличается от применения той же процедуры к текстовому файлу тем, что файл открывается не обязательно только для чтения. Режим открытия файла определяется глобальной переменной **FileMode**. Ей могут присваиваться значения ряда именованных констант, объявленных в модуле *SysUtils*. По умолчанию значение **FileMode** равно **fmOpenReadWrite** и файл открывается для чтения и записи. Однако если вы откроете в этом режиме файл с атрибутом «только для чтения», то возникнет ошибка. Такие файлы должны открываться в режиме **fmOpenRead** — для чтения. Для записи файл должен открываться в режиме **fmOpenWrite**.

Так что по умолчанию файл открывается процедурой **Reset** для чтения и записи. Поэтому вы можете, например, решить такую задачу: прочитать 100-е число из файла действительных чисел, умножить его на 10 и записать это увеличенное число в ту же позицию файла. Это можно сделать следующим кодом:

```
Reset(F);
Seek(F, 99);
Read(F, R);
Seek(F, 99);
R := R * 10;
Write(F, R);
```

Имеется функция **FilePos**:

```
function FilePos(var F): Longint;
```

которая возвращает текущую позицию в файле. Эту функцию можно использовать, чтобы запомнить текущую позицию и потом после каких-то перемещений по файлу вернуться к той же позиции. Например, приведенный выше фрагмент кода может выглядеть так:

```
Reset(F);
Seek(F, 99);
P := FilePos(F);
Read(F, R);
Seek(F, P);
R := R * 10;
Write(F, R);
```

Здесь текущая позиция запоминается в переменной **P**, которая должна быть объявлена как **Longint**. После вызова процедуры **Read** позиция смещается. А затем она возвращается к прежней процедурой **Seek**, в которую передается запомненная позиция. Конечно, в данном примере можно обойтись и без запоминания позиции, благо она и так известна — 99. Но очень часто после многочисленных операций с файлом надо вернуться к той позиции, с которой начинались эти операции. Тогда без функции **FilePos** обойтись трудно.

Теперь рассмотрим пример, в котором используется файл, содержащий не числа, а более сложные объекты — записи. В разд. 3.3.2 было рассмотрено приложение (рис. 3.8), работающее со связными списками. Оно позволяет формировать список сотрудников некоторой организации. В приложении были предусмотрены кнопки В файл и Из файла, позволяющие запоминать список в файле и читать из него. Теперь мы можем реализовать обработчики щелчков на этих кнопках.

Обработчик щелчка на кнопке В файл может быть реализован так:

```
procedure TForm1.BSaveClick(Sender: TObject);
var F: file of TLPers;
begin
 AssignFile(F, 'MyList.dat');
 Rewrite(F);
 P := P0;
 while P <> nil do
 begin
  Write(F, P^);
  P := P^.pf;
 end;
 CloseFile(F);
end;
```

Локальная переменная **F** объявлена как переменная типизированного файла с элементами типа **TLPers**. Напомню, что это тип введенной вами самоадресуемой записи, содержащей сведения о сотруднике некоторой организации. Первые выполняемые операторы процедуры открывают для записи файл с именем "MyList.dat". Затем организуется цикл по всем записям списка, и содержимое каждой записи (**P^**) заносится как единое целое в файл процедурой **Write**. В заключение файл закрывается.

Обработчик щелчка на кнопке Из файла может быть реализован так:

```
procedure TForm1.BOpenClick(Sender: TObject);
var F: file of TLPers;
begin
 AssignFile(F, 'MyList.dat');
 Reset(F);
 while not Eof(F) do
 begin
  New(P);                 // выделение памяти под новую запись
  Read(F, P^);
  P^.pf := nil;
  if P0=nil then P0:=P   // P0 - указатель на первую запись
   else PLast^.pf:=P;    // указатель на очередную запись
  PLast:=P;
 end;
 Memo1.Clear;
 ListPrint;
end;
```

Файл "MyList.dat", в котором был сохранен список, открывается для чтения. Организуется цикл **while**, повторяющийся до тех пор, пока файл не будет прочитан полностью. Создание каждой новой записи и включение ее в список осуществляется так же, как уже рассматривалось при анализе этого приложения. Отличие только в том, что поля каждой записи заполняются чтением процедурой **Read** данных, запомненных в файле.

Остановимся на одном тонком моменте, связанном с введением в это приложение работы с файлами. Тип записи **TLPers** в нем (см. листинг приложения в разд. 3.3.2) содержит строковые поля **Fam**, **Nam**, **Par**, **Dep**, для которых указан тип в виде коротких строк **string[20]** и **string[15]**. Попробуйте удалить из этого объявления указания числа символов, т.е. объявите тип этих полей как **string**. Вы получите на объявлениях локальных переменных **F** сообщение компилятора об

ошибке: «Type 'TLPers' needs finalization — not allowed in file type». Это сообщение связано с тем, что поля типа **string** могут иметь разный размер в разных элементах списка в зависимости от длины хранимых строк. Это недопустимо в типизированных файлах, в которых каждая запись должна иметь одинаковый размер. Использование коротких строк снимает эту проблему.

3.4.6 Нетипизированные файлы

Нетипизированные файлы — это двоичные файлы, которые могут содержать самые различные данные в виде последовательности байтов. Программист при чтении этих данных сам должен разбираться, какие байты к чему относятся.

Тип файловой переменной нетипизированного файла объявляется следующим образом:

```
var <имя файловой переменной>: file;
```

Открытие нетипизированных фалов осуществляется процедурами **Reset** и **Rewrite**, но их синтаксис несколько отличен от аналогичных процедур для других видов файлов:

```
procedure Reset(var F: File; Recsize: Word);
procedure Rewrite(var F: File; Recsize: Word);
```

В этих процедурах **Recsize** — необязательный параметр, который указывает размер одной записи в файле (одного элемента файла) в байтах. Если параметр **Recsize** не задан, то размер одной записи предполагается равным 128 байтам.

Вместо процедур записи и чтения **Read** и **Write** в нетипизированных файлах имеются процедуры **BlockRead** и **BlockWrite**, объявленные следующим образом:

```
procedure BlockWrite(var f: File; var Buf; Count: Word;
                     [var Result: Word]);
procedure BlockRead(var F: File; var Buf; Count: Word;
                    [var Result: Word]);
```

В этих процедурах **Buf** — любая переменная, содержимое которой записывается в файл или в которую читаются записи из файла. **Count** — выражение типа **Word**, определяющее число записываемых или читаемых записей. В необязательный параметр **Result** заносится число реально записанных или прочитанных записей. Это число может быть меньше или равно числу, указанному как **Count**. Недостаточное значение **Result** при чтении может, например, свидетельствовать, что конец файла достигнут ранее, чем прочиталось заданное число записей. А при записи это может означать, например, что диск переполнился.

Рассмотрим на примере, как можно записывать в нетипизированный файл и считывать из него данные различных типов. Ниже приведен код, который записывает в файл *MyFile.dat* два целых числа, строку текста и действительное число, а потом читает из файла эти значения.

```
var i, j, i1, j1: integer;
    a, a1: real;
    s, s1: string;
    F: file;
    ch: char;
...
i := 1;
j := 25;
```

```
s := 'Иванов';
a := 25e6;
AssignFile(F, 'MyFile.dat');
// запись в файл
Rewrite(F, 1);
BlockWrite(F, i, SizeOf(i));
BlockWrite(F, j, SizeOf(j));
BlockWrite(F, s[1], Length(s)+1);
BlockWrite(F, a, SizeOf(a));
CloseFile(F);
// чтение из файла
Reset(F, 1);
BlockRead(F, i1, SizeOf(i));
BlockRead(F, j1, SizeOf(i));
s1 := '';
BlockRead(F, ch, 1);
while ch <> #0 do
begin
 s1 := s1 + ch;
 BlockRead(F, ch, 1);
end;
BlockRead(F, a1, SizeOf(a));
CloseFile(F);
```

В приведенном коде файл и для записи, и для чтения открывается с указанием размера записи 1. Так что под записью подразумевается один байт. Поэтому в функциях **BlockWrite** число записей определяется функцией **SizeOf** (см. разд. 2.11). Можно, конечно, было бы просто указать число байтов, исходя из табл. 2.1 в разд. 2.4.1. Но поскольку от версии к версии размер памяти, отводимой под разные типы данных, может изменяться, использование функции **SizeOf** делает код более универсальным.

Особо следует остановиться на записи и чтении строки. Во-первых, как вы видите, источником данных указан первый символ строки **s[1]**, а не сама переменная **s**. Если бы мы указали **s**, то в файл занесся бы не текст строки, а ее адрес. А во-вторых, при записи строки в файл надо продумать, как ее потом читать. В нашем искусственном примере проблем нет, так как мы знаем длину записанной строки. А в реальных задачах при чтении встает вопрос, сколько байтов надо читать в строку. Один из путей решения этой проблемы — записывать в файл перед строкой целое число, равное числу символов в ней, а затем записывать указанное число символов. Тогда при чтении надо будет сначала читать число символов, а затем функцией **BlockRead** читать в строку заданное число байтов. Другой вариант, использованный в приведенном коде — записывать после строки нулевой символ. В примере это делается заданием числа записываемых символов **Length(s)+1**. Вспомните (разд. 3.2.2), что строки типа **string** содержат завершающий нулевой символ. Именно его мы и записали, увеличив число символов на 1. Тогда при чтении строка читается в цикле **while** по символам, пока не будет найден нулевой символ.

Чтение и запись нетипизированных файлов осуществляется быстрее, чем типизированных, за счет того, что в течение одного обращения к файлу можно передавать сразу большую порцию данных. Ниже приведен пример использования этой возможности для организации копирования произвольных файлов.

```
procedure TForm1.Button1Click(Sender: TObject);
const N = 1000;
```

```
var Fin, Fout: file;
    Buf: array[1..N] of byte;
    Res: integer;
begin
 OpenDialog1.Title := 'Выберите копируемый файл';
 SaveDialog1.Title := 'Укажите каталог и имя копии';
 if OpenDialog1.Execute
 then if SaveDialog1.Execute
  then begin
   AssignFile(Fin, OpenDialog1.FileName);
   Reset(Fin, 1);
   AssignFile(Fout, SaveDialog1.FileName);
   Rewrite(Fout, 1);
   while not Eof(Fin) do
   begin
    BlockRead(Fin, Buf, N, Res);
    BlockWrite(Fout, Buf, Res);
   end;
   CloseFile(Fin);
   CloseFile(Fout);
  end;
end;
```

В приведенной процедуре вводится константа **N**, определяющая число байтов, передаваемых в каждой операции чтения и записи. Объявляется буфер **Buf** соответствующего размера, который воспринимает прочитанные данные. Выполнение процедуры начинается с того, что задаются заголовки стандартных диалогов открытия и сохранения файла, после чего вызываются эти диалоги. Если пользователь указал в них имена файлов, то начинается копирование. Файлы открываются, причем в данном случае размер записи указывается равным 1. Так что одна запись соответствует одному байту. Затем в цикле функцией **BlockRead** читается по **N** байтов в буфер **Buf**. Прочитанное число байтов, занесенное функцией **BlockRead** в переменную **Res**, записывается функцией **BlockWrite** в копию файла. Использование переменной **Res** обусловлено следующим. Пока не достигнут конец файла, функция **BlockRead** читает по **N** байтов, и **Res** равно **N**. Но в последней прочитанной порции данных значение **Res** окажется меньше **N**, если только размер файла не кратен **N**. Так что если бы в функции **BlockWrite** число записей было равно **N**, то в копию файла записались бы лишние данные.

Создайте соответствующее приложение и поэкспериментируйте с величиной константы **N** при копировании не очень маленьких файлов. Вы сможете заметить, что при малых значениях **N** время копирования увеличивается. Так что чем большим задать значение **N**, тем более эффективным будет приложение. Правда, задание очень больших значений **N** вызовет излишние затраты памяти при копировании небольших файлов.

Вы сделали неплохую программу копирования любых файлов. Попробуйте сами оформить ее в виде процедуры, принимающей два параметра: имя исходного файла и копии, и осуществляющей копирование. Такую функцию стоит включить в вашу библиотеку. А в следующем разделе мы обсудим, как эту функцию можно усовершенствовать.

3.4.7 Функции и процедуры обработки файлов

В данном разделе дается обзор ряда функций, облегчающих работу с файлами, каталогами и дисками. В табл. 3.2 приведен список некоторых из этих функций. Полный перечень функций с их описаниями вы можете посмотреть во встроенных в Delphi справках или в справках [3].

Таблица 3.2. Некоторые функции и процедуры манипулирования файлами и каталогами

ChDir(S: string) Изменяет текущий каталог на заданный параметром S. При ошибке или генерирует исключение **EInOutError** (при включенной опции **{$I+}**), или устанавливает код ошибки (при включенной опции **{$I–}**), который можно прочитать функцией **IOResult** (см. также **SetCurrentDir**).
CreateDir(const Dir: string): Boolean Создает каталог **Dir**. Возвращает **false**, если создать каталог не удается, например, если отсутствуют промежуточные каталоги, указанные в пути **Dir** (см. также **ForceDirectories** и **MkDir**).
DeleteFile(const FileName: string): Boolean Удаляет файл **FileName** с диска, возвращая **true** в случае успеха.
DirectoryExists(const Directory: string): Boolean Возвращает **true**, если каталог **Directory** существует.
DiskFree(Drive: Byte): Integer Возвращает свободную область в байтах диска **Drive**, который задается следующим образом: 0 — текущий диск, 1 — диск A, 2 — диск B, 3 — диск C и т.д. При ошибочном задании **Drive** возвращается –1.
DiskSize(Drive: Byte): Integer Возвращает размер диска **Drive** в байтах. **Drive** задается следующим образом: 0 — текущий диск, 1 — диск A, 2 — диск B, 3 — диск C и т.д. При ошибочном задании **Drive** возвращается –1.
FileExists(const FileName: string): Boolean Возвращает **true**, если существует файл **FileName**.
FileIsReadOnly(const FileName: string): Boolean Возвращает **true**, если файл существует, но доступ к нему только для чтения.
FileSize(var F): Integer Возвращает число записей в файле, связанном с файловой переменной **F**. Если размер записи объявлен равным 1, то возвращаемое число — размер файла в байтах.
ForceDirectories(Dir: string): Boolean Создает каталог **Dir**, включая промежуточные каталоги, указанные в полном пути к нему. Возвращает **false**, если создать каталог не удается (см. также **CreateDir** и **MkDir**).
GetCurrentDir: string Возвращает текущий каталог текущего диска с полным путем к нему.

GetDir(D: Byte; var S: string) Возвращает **S** — текущий каталог диска, указанного параметром **D**, который может равняться: 0 — текущий диск, 1 — диск A, 2 — диск B, 3 — диск C и т.д. Каталог заносится в **S** без заключительного символа слэша, например, "c:\mydir". Если указан ошибочный номер диска, то вернется корневой каталог этого отсутствующего диска.
MkDir(S: string) Создает подкаталог, заданный с полным путем параметром **S**. Если создать каталог не удается, например, если отсутствуют промежуточные каталоги, указанные в пути **S**, то или генерирует исключение **EInOutError** (при включенной опции **{$I+}**), или устанавливает код ошибки (при включенной опции **{$I–}**), который можно прочитать функцией **IOResult** (см. также **CreateDir** и **ForceDirectories**).
RemoveDir(const Dir: string): Boolean Удаляет каталог **Dir**. Каталог должен быть пустым. Если он не пустой, или к нему нет прав доступа на удаление, или он вообще отсутствует, возвращается **false** (см. также **RmDir**).
RenameFile(const OldName, NewName: string): Boolean Переименовывает файл **OldName**, давая ему новое имя **NewName**. Если в **NewName** указан каталог, отличный от указанного в **OldName**, то функция переместит файл в новый каталог. При успешном завершении возвращает **true**.
RmDir(S: string) Удаляет пустой подкаталог, заданный параметром S. При ошибке или генерирует исключение **EInOutError** (при включенной опции **{$I+}**), или устанавливает код ошибки (при включенной опции **{$I–}**), который можно прочитать функцией **IOResult** (см. также **RemoveDir**).
SetCurrentDir(const Dir: string): Boolean Изменяет текущий каталог на заданный параметром **Dir**. При ошибке возвращает **false** (см. также **ChDir**).

Вероятно, большинство процедур и функций, приведенных в табл. 3.2, не нуждается в дополнительных пояснениях. Остановимся только на различиях некоторых групп процедур, выполняющих сходные функции. Например, создать каталог на диске можно с помощью функций **CreateDir**, **ForceDirectories** и процедуры **MkDir**. Различие между ними в следующем. Функция **CreateDir** и процедура **MkDir** могут создавать только подкаталог существующего каталога. Например, если имеется каталог *C:\tt*, то они могут создать каталог, заданный как "C:\tt\tt1", но не могут сразу создать каталог "C:\tt\tt1\tt2", если промежуточный каталог *C:\tt\tt1* отсутствует. Реакция на подобную ошибку различна: функция **CreateDir** возвращает **false**, а процедура **MkDir**, в зависимости от заданной директивы компилятора, или генерирует исключение, или задает код ошибки, который может быть получен функцией **IOResult**. Так что вы можете сами решать, что вам удобнее. Мне думается, что в большинстве случаев удобнее функция **CreateDir**. Только, работая с ней, надо не забывать проверять возвращаемый результат. Иначе каталог не создастся, а вы этого не заметите, и будете пытаться с ним работать, что вызовет дальнейшие ошибки.

Функция **ForceDirectories** создает каталог, включая все промежуточные каталоги, указанные в полном пути к нему. Так что в этом плане она мощнее, чем **CreateDir** и **MkDir**. Она не сможет создать каталог и вернет **false**, только если в пути указан несуществующий диск, или если у приложения нет соответствующих прав доступа для создания требуемых каталогов.

Различия между функцией **SetCurrentDir** и процедурой **ChDir**, функцией **RemoveDir** и процедурой **RmDir** аналогичны рассмотренным выше различиям между **CreateDir** и **MkDir**.

Обратите внимание на функцию **RenameFile**. Она не только может переименовать файл, но и может переместить его в другой каталог.

Среди функций табл. 3.2 единственная функция, связанная с файловой переменной, это функция **FileSize**. Для того чтобы использовать эту функцию, файл должен быть предварительно открыт. Функция возвращает число записей в файле, связанном с переменной **F**. Если тип файловой переменной объявлен как **file of byte**, или если **F** связана с нетипизированным файлом и при его открытии размер записи объявлен равным 1, то возвращаемое число — это размер файла в байтах. Если при открытии нетипизированного файла задан другой размер записи, то функция **FileSize** возвращает число таких записей в файле.

Работая с функциями, перечисленными в табл. 3.2, часто приходится осуществлять преобразования расширений и имен файлов и каталогов. Конечно, все эти преобразования можно осуществлять с помощью функций, оперирующих со строками и рассмотренных в разд. 3.2.2. Но поскольку подобные операции требуются часто, имеется ряд функций, облегчающих работу с именами файлов. Некоторые из этих функций приведены в табл. 3.3.

Таблица 3.3. Некоторые функции и процедуры обработки имен файлов

ChangeFileExt(const FileName, Extension: string): string Изменяет расширение файла **FileName** на **Extension**.
ExtractFileDir(const FileName: string): string Возвращает путь к каталогу (включая указание диска), вырезанный из строки с полным именем файла **FileName**.
ExtractFileDrive(const FileName: string): string Возвращает символы диска, вырезанные из строки с полным именем файла **FileName**. Если строка записана в формате UNC, то возвращается и путь в сети (сервер).
ExtractFileExt(const FileName: string): string Возвращает символы расширения, вырезанные из строки с полным именем файла **FileName**.
ExtractFileName(const FileName: string): string Возвращает имя файла с расширением, вырезанное из строки с полным именем файла **FileName**.
ExtractFilePath(const FileName: string): string Возвращает путь к каталогу (включая указание диска и слэш перед именем файла), вырезанный из строки с полным именем файла **FileName**.

Все приведенные в таблице функции вырезают различные фрагменты из строки **FileName**, содержащей полное имя файла. Например, если эта строка равна "D:\Program Files\Borland\Delphi7\Bin\delphi32.exe", то различные функции вернут следующие результаты:

Вызываемая функция	Возвращаемое значение
ExtractFileDir(FileName)	D:\Program Files\Borland\Delphi7\Bin
ExtractFileDrive(FileName)	D:
ExtractFileExt(FileName)	.exe
ExtractFileName(FileName)	delphi32.exe
ExtractFilePath(FileName)	D:\Program Files\Borland\Delphi7\Bin\
ChangeFileExt(FileName, '.bak')	D:\Program Files\Borland\Delphi7\Bin\delphi32.bak

Теперь, ознакомившись с различными функциями, давайте обсудим в качестве примера, как можно с их помощью усовершенствовать код копирования файлов, приведенный в разд. 3.4.6. Прежде всего, в этом коде не предусмотрены проверки различных ошибочных ситуаций. Пользователь может случайно задать для копии тот же файл, который является исходным. В этом случае программа испортит исходный файл и не создаст копии. Так что эту ошибку надо отлавливать, предупреждать о ней пользователя, и не проводить копирования. Пользователь может также указать в качестве исходного файл, которого нет. Это вызовет ошибку открытия файла с сообщением пользователю: «File not found». Пользователя, который не силен в английском, это может повергнуть в шок. Да и грамотному пользователю подобное сообщение мало что скажет, так как неизвестно, какой файл не найден. Так что эту ошибку тоже надо отлавливать и сообщать о ней пользователю на нормальном русском языке.

Проверить, существует ли исходный файл, можно приведенной в табл. 3.2 функцией **FileExists**. Если файла нет, функция возвращает **false**. Так что в нашем случае проверка может проводиться оператором:

```
if(not FileExists(OpenDialog1.FileName))
then ...
```

Еще одна ситуация, требующая разрешения, возникает, если пользователь указал в качестве копии имя существующего файла. В этом случае этот файл удалится и заменится созданной копией исходного файла. Но ведь пользователь мог ошибиться, и нехорошо наказывать его удалением, может быть, очень нужного ему файла. Обычно в подобных случаях желательно сохранять файл, изменив его расширение на *.bak*. Понять, существует ли указанный файл, можно функцией **FileExists**. А переименовать файл можно приведенной в табл. 3.2 функцией **RenameFile**. При этом новое имя файла, отличающееся от заданного только расширением, можно сформировать из заданного имени, применив к нему функцию **ChangeFileExt**, описанную в табл. 3.3.

Функция **RenameFile** вернет **true**, если переименование прошло успешно. А ошибка может возникнуть, если файл с новым расширением *.bak* уже существует. Так что желательно сначала проверить функцией **FileExists**, есть ли соответствующий файл. Если есть, то его следует удалить процедурой **DeleteFile**, описанной в табл. 3.2. А если эта функция вернула **false**, значит у приложения нет соответствующих прав доступа и следует отказаться от создания файла *.bak*.

Еще одна возможная ошибка: пользователь может при задании копии указать каталог, который не существует. Это можно проверить, вырезав из имени файла функцией **ExtractFilePath** (табл. 3.3) полный путь и применив к этому пути функцию **DirectoryExists** (табл. 3.2). Если требуемого каталога нет, его можно создать

функцией **ForceDirectories**. А если эта функция вернула **false**, значит у приложения нет соответствующих прав доступа, или пользователь указал неверный диск (можно проверить функцией **DirectoryExists**, передав в нее имя диска, вырезанное функцией **ExtractFileDrive**). Так что при ошибке надо известить о ней пользователя и отказаться от попыток копирования.

Попробуйте самостоятельно применить все изложенное к совершенствованию вашей функции копирования файлов. И такую модернизированную функцию, безусловно, имеет смысл включить в вашу библиотеку. Тип возвращаемого ею значения имеет смысл задать булевым, и возвращать **false** при невозможности копирования. Функцию с таким возвращаемым значением вам удобнее будет в дальнейшем использовать.

Вам может показаться излишеством вводить в функцию все рассмотренные выше проверки. Но программист должен исходить из принципа, что пользователь всегда прав. Вспомните, наверное, у вас бывали ситуации, когда какая-то программа, даже созданная солидной фирмой, отказывалась работать или выдавала какие-то заумные сообщения на английском языке. Вспомните ваши эмоции в подобных случаях, и постарайтесь, чтобы пользователи не испытывали стрессов при работе с вашими приложениями.

Хороший стиль программирования

Добротная программа ни в каких ситуациях не должны отказываться работать и выдавать пользователю непонятные сообщения об ошибках. В программе надо пытаться предусмотреть все возможные ошибки пользователя или системы, и во всех подобных случаях выдавать пользователю сообщение, поясняющее ему причины неправильной работы и советующее, что ему следует делать, чтобы исправить ситуацию.

И в заключение еще об одной модификации, которую можно сделать в функции копирования. В разд. 3.4.6 говорилось о зависимости времени копирования от числа записей **N**, которыми обмениваются с файлом процедуры **BlockRead** и **BlockWrite**. Чем больше это число, тем быстрее выполняется копирование. Но число, большее количества байтов в файле, приводит к излишним затратам памяти. В табл. 3.2 вы можете видеть функцию **FileSize**, которая позволяет определить размер файла. Так что число записей **N** можно задать равным числу байтов в файле. Только при этом надо в функции копирования создать буфер соответствующего размера. Это можно сделать с помощью динамического массива. Ниже приведен соответствующий код. В нем отсутствует оформление в виде функции и все рассмотренные выше проверки. Все это вы можете сделать сами. Жирным шрифтом в коде выделены операторы, отличающиеся от аналогичного кода, приведенного в разд. 3.4.6.

```
procedure TForm1.Button1Click(Sender: TObject);
var N: integer;
    Fin, Fout: file;
    Buf: array of byte;
begin
 OpenDialog1.Title := 'Выберите копируемый файл';
 SaveDialog1.Title := 'Укажите каталог и имя копии';
 if OpenDialog1.Execute
 then if SaveDialog1.Execute
```

```
 then begin
  AssignFile(Fin, OpenDialog1.FileName);
  Reset(Fin, 1);
  N := FileSize(Fin);
  SetLength(Buf, N);
  AssignFile(Fout, SaveDialog1.FileName);
  Rewrite(Fout, 1);
  BlockRead(Fin, Buf[0], N);
  BlockWrite(Fout, Buf[0], N);
  CloseFile(Fin);
  CloseFile(Fout);
 end;
end;
```

Сравните этот код с приведенным в разд. 3.4.6. Вместо константы **N** здесь введена соответствующая переменная. После того как исходный файл открыт, ей присваивается значение, возвращаемое функцией **FileSize**. Далее размер массива **Buf**, который в данном коде объявлен динамическим, задается равным **N**. Вместо цикла, в котором ранее читались записи, достаточно прочитать весь файл одним вызовом процедуры **BlockRead**. В качестве буфера в вызове этой процедуры задается **Buf[0]** — адрес первого элемента массива. Затем аналогичным вызовом процедуры **BlockWrite** содержимое массива записывается в выходной файл.

3.4.8 Вызов исполняемых файлов

Вы можете выполнить из своего приложения любой исполняемый файл — какую-то внешнюю программу. Самый простой способ запустить внешнюю программу — использовать функцию **WinExec**. Эта функция может работать в любых версиях Windows и выполнять любые файлы: приложения Windows, MS-DOS, файлы PIF и т.п. Функция **WinExec** определяется следующим образом:

```
function WinExec(CmdLine: PChar; CmdShow: integer): integer;
```

Параметр **CmdLine** является указателем на строку с нулевым символом в конце, содержащую имя выполняемого файла и, если необходимо, параметры командной строки. Если имя файла указано без пути, то Windows будет искать в каталогах выполняемый файл в следующей последовательности:

1. Каталог, из которого загружено приложение
2. Текущий каталог
3. Системный каталог Windows, возвращаемый функцией **GetSystemDirectory**
4. Каталог Windows, возвращаемый функцией **GetWindowsDirectory**
5. Список каталогов из переменной окружения PATH

Параметр **CmdShow** определяет форму представления окна запускаемого приложения Windows. Возможные значения этого параметра вы можете посмотреть во встроенной справке Delphi или в справке [3]. Чаще всего используется значение **SW_RESTORE**, при котором окно запускаемого приложения активизируется и отображается на экране. Если это окно в данный момент свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции. Для приложений не Windows, для файлов PIF и т.д. состояние окна определяет само приложение.

При успешном выполнении запуска приложения функция **WinExec** возвращает значение, большее 31. При неудаче могут возвращаться следующие значения:

Значение	Номер	Описание
0	0	не хватает памяти или ресурсов системы
ERROR_BAD_FORMAT	11	ошибочный формат файла *.exe*
ERROR_FILE_NOT_FOUND	2	указанный файл не найден
ERROR_PATH_NOT_FOUND	3	указанный каталог не найден

Достоинством функции **WinExec** является ее совместимость с ранними версиями Windows.

Приведем примеры применения **WinExec**.

Оператор

```
WinExec('file.exe',SW_RESTORE);
```

запускает программу *file.exe*. Оператор

```
WinExec('nc',SW_RESTORE);
```

запускает *Norton Commander*. Оператор

```
WinExec('COMMAND.COM',SW_RESTORE);
```

приводит к вызову окна MS-DOS.

Теперь приведем пример процедуры, обеспечивающей выполнение любой выбранной пользователем программы. Откройте новый проект и разместите на форме компонент **OpenDialog** и кнопку, при щелчке на которой пользователь может выбрать в стандартном диалоговом окне Открыть файл программу и выполнить ее. Обработчик события **OnClick** этой кнопки может иметь вид:

```
procedure TForm1.Button1Click(Sender: TObject);
var i: integer;
begin
 if (OpenDialog1.Execute) then
  begin
   i := WinExec(PChar(OpenDialog1.FileName), SW_RESTORE);
   if i < 32
    then
      case i of
        0: ShowMessage('Не хватает памяти или ресурсов ');
        ERROR_BAD_FORMAT:
                    ShowMessage('Ошибочный формат файла ' +
                                OpenDialog1.FileName);
        ERROR_PATH_NOT_FOUND:
                    ShowMessage('Не найден каталог ' +
                     ExtractFilePath(OpenDialog1.FileName));
        ERROR_FILE_NOT_FOUND:
                    ShowMessage('Не найден файл ' +
                                OpenDialog1.FileName)
        else ShowMessage('Ошибка при выполнении файла ' +
                         OpenDialog1.FileName)
      end;
 end;
end;
```

Обратите внимание на необходимость в вызове **WinExec** явного приведения строки с именем файла к типу **PChar** (см. разд. 3.2.2). Структура **case** в приведенном операторе анализирует результат вызова функции **WinExec** и в случае ошибки выдает пользователю соответствующее сообщение.

Запустите ваше приложение на выполнение и попробуйте вызывать из него различные программы Windows и MS-DOS.

Еще большие возможности запуска внешних программ дает функция **ShellExecute**. Она может не только выполнять заданное приложение, но и открывать документ и печатать его. Под термином «открыть файл документа» понимается выполнение связанного с ним приложения и загрузка в него этого документа. Например, обычно с документами, имеющими расширение *.doc*, связан Word. В этом случае открыть файл, например, с именем *file.doc* означает запустить Word и передать ему в качестве параметра имя файла *file.doc*. Кроме описанных возможностей функция **ShellExecute** позволяет открыть указанную папку. Это означает, что будет запущена программа «Проводник» с открытой указанной папкой.

Для использования функции **ShellExecute** в оператор **uses** надо добавить модуль *ShellAPI*. Автоматически Delphi не включает этот модуль в программу.

Функция **ShellExecute** определена следующим образом:

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

Параметр **Wnd** является *дескриптором* родительского окна, в котором отображаются сообщения запускаемого приложения. Дескриптором называется специального вида указатель, который присваивается Windows каждому окну и каждому компоненту. У формы и любого оконного компонента Delphi дескриптор содержится в свойстве **Handle**, доступном только во время выполнения и только для чтения. Так что при вызове **ShellExecute** в качестве аргумента **Wnd** можно указать просто **Handle** — дескриптор окна формы.

Параметр **Operation** указывает на строку с нулевым символом в конце, которая определяет выполняемую операцию. Эта строка может содержать текст «open» (открыть), «print» (напечатать), «explore» (исследовать — открыть папку). Если параметр **Operation** равен **nil**, то по умолчанию выполняется операция «open».

Параметр **FileName** указывает на строку с нулевым символом в конце, которая определяет имя открываемого файла или имя открываемой папки.

Параметр **Parameters** указывает на строку с нулевым символом в конце, которая определяет передаваемые в приложение параметры, если **FileName** определяет выполняемый файл. Если **FileName** указывает на строку, определяющую открываемый документ или папку, то этот параметр задается равным **nil**.

Параметр **Directory** указывает на строку с нулевым символом в конце, которая определяет каталог по умолчанию.

Параметр **ShowCmd** определяет режим открытия указанного файла. Как и для функции **WinExec**, обычно используется значение **SW_RESTORE**, при котором окно запускаемого приложения активизируется и отображается на экране.

При успешном выполнении функция **ShellExecute** возвращает целое значение, большее 32. Значение меньшее или равное 32 указывает на ошибку. Основные из этих значений те же, что были указаны для функции **WinExec**. Но имеется еще одно значение — 31, свидетельствующее о том, что нет приложения, связанного с файлом указанного типа, или нет файла, связанного с указанной операцией.

Рассмотрим примеры использования функции **ShellExecute**. Следующий оператор открывает файл документа с именем *file.doc*, т.е. запускает Winword (обычно именно он связан с файлами *.doc)*, загрузив в него указанный файл:

```
ShellExecute(Handle,nil,'file.doc',nil,nil,SW_RESTORE);
```

Если вы хотите не открыть, а напечатать документ, записывается аналогичный оператор, но изменяется значение параметра **Operation**:

```
ShellExecute(Handle, 'print','file.doc',nil,nil,SW_RESTORE);
```

Выполнение этого оператора будет протекать следующим образом. Запустится Winword, связанный с файлами *.doc*, в него загрузится файл *file.doc*, затем из Winword запустится печать с атрибутами по умолчанию, после чего файл *file.doc* выгрузится из Winword.

Приведенный ниже оператор открывает приложение Windows «Калькулятор»:

```
ShellExecute(Handle,'open','Calc',nil,nil,SW_RESTORE);
```

Следующий пример открывает папку *c:\Program Files\Borland*:

```
ShellExecute(Handle,'explore', 'c:\Program Files\Borland',
                nil,nil,SW_RESTORE);
```

Можете сделать универсальное приложение, подобное описанному выше для функции **WinExec**, но использующее функцию **ShellExecute**. Для этого в приведенном ранее коде достаточно изменить один оператор:

```
i := ShellExecute(Handle, 'open', PChar(OpenDialog1.FileName),
                  nil, nil, SW_RESTORE);
```

Только не забудьте подключить к вашему модулю предложением **uses** модуль *ShellAPI*. С помощью этого приложения вы сможете открывать любые документы Word, Excel, любые файлы изображений, звуковые и мультимедийные файлы. Так что это будет действительно универсальное приложение для просмотра любых документов.

Научившись в разд. 3.3.4 работать со списками, вы можете также создать себе каталоги своих любимых музыкальных произведений, фотографий своих друзей и т.д. В качестве примера на рис. 3.13 показан пример такого приложения. Сделаю некоторые подсказки для создания подобного каталога фотографий. А остальное додумайте сами.

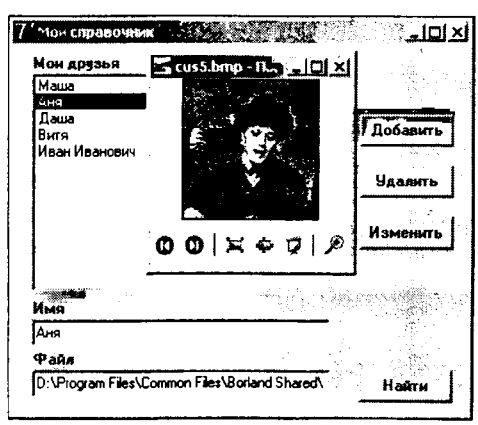


Рис. 3.13
Пример приложения для просмотра файлов

Начало приложения должно содержать операторы:

```
uses ShellAPI;
var FileList: TStringList; // Список имен файлов изображений
```

Обработчик события формы **OnCreate** должен создавать список **FileList** и читать в него и в компонент **ListBox1** запомненные имена файлов и имена ваших друзей. Например:

```
FileList := TStringList.Create;
if FileExists('Friends1.txt')
 then begin
  ListBox1.Items.LoadFromFile('Friends1.txt');
  FileList.LoadFromFile('Friends2.txt');
  ListBox1.ItemIndex := 0;
  ListBox1Click(Sender);
 end;
```

В этом коде «Friends1.txt» и «Friends2.txt» — имена файлов, в которых сохраняются списки.

Сохранение списков в файлах надо обеспечить в обработчике события **OnDestroy** формы:

```
ListBox1.Items.SaveToFile('Friends1.txt');
FileList.SaveToFile('Friends2.txt');
FileList.Free;
```

Обработчик двойного щелчка на строке компонента **ListBox1** (двойной щелчок вызывает событие **OnDblClick**), при котором показывается фотография из файла, записанного в списке **FileList** под тем же индексом, что и строка в **ListBox1**, может включать всего один оператор:

```
ShellExecute(Handle, 'open',
             PChar(FileList.Strings[ListBox1.ItemIndex]),
             nil, nil, SW_RESTORE);
```

Ну а обработчики остальных событий: в частности, щелчков на кнопках Добавить, Удалить, Изменить, управляющих списком, и на кнопке Найти, загружающей в окно **Edit** файл, указанный пользователем в диалоге открытия файла, попробуйте написать сами. Скажу только, что расширения графических файлов, в частности, могут быть *.bmp* (подробнее о графических файлах вы узнаете в разд. 5.5.1.2). С Delphi поставляется несколько подобных файлов, расположенных в каталоге *...Program Files\Common Files\Borland Shared\Data*.

3.4.9 Поиск файлов в каталогах

Нередко возникает задача поиска в каком-то каталоге файлов, удовлетворяющих некоторому шаблону или имеющих указанные атрибуты. Например, ваше приложение создает во время выполнения временные файлы с расширением *.tmp*, и в момент окончания работы надо их все отыскать и уничтожить, чтобы «убрать мусор», засоряющий диск ненужными файлами. Или надо найти все файлы документов с расширением *.doc*, чтобы показать их пользователю.

Подобные задачи решаются функциями **FindFirst**, **FindNext** и процедурой **FindClose**:

```
function FindFirst(const Path: string; Attr: Integer;
                   var F: TSearchRec): Integer;
function FindNext(var F: TSearchRec): Integer;
procedure FindClose(var F: TSearchRec);
```

Эти функции оперируют с записью типа **TSearchRec**, объявленного следующим образом;

```
type
TSearchRec = record
     Time: Integer;              // Время создания файла
     Size: Integer;              // Размер файла в байтах
     Attr: Integer;              // Атрибуты файла
     Name: TFileName;            // Имя файла
     ExcludeAttr: Integer;
     FindHandle: THandle;
     FindData: TWin32FindData;
end;
```

Начинается поиск вызовом функции **FindFirst**. Параметр **Path** определяет путь и шаблон искомых файлов. Например, если **Path** = "c:\test*.*", то будет проводиться поиск всех файлов в каталоге *c:\test*. А если **Path** = "c:\test*.tmp", то в каталоге *c:\test* будут искаться файлы с расширением *.tmp*. Параметр **Attr** определяет флаги атрибутов, которые должны иметь искомые файлы:

faReadOnly	файл только для чтения
faHidden	невидимый файл
faSysFile	системный файл
faVolumeID	идентификатор диска
faDirectory	каталог
faArchive	архивный файл
faAnyFile	любой файл

Флаги (см. разд. 2.4.10) могут объединяться операцией **or**. Например, если **Attr** = **faReadOnly or faHidden**, то будут искаться невидимые файлы только для чтения.

Функция **FindFirst** возвращает 0, если файл, удовлетворяющий условиям поиска, найден. В противном случае возвращается код ошибки.

Если файл найден, то сведения о нем заносятся в поля записи типа **TSearchRec**, определяемой параметром **F**. В поле **Name** этой записи можно найти имя файла вместе с его расширением. Например, "Test.txt". В поле **Time** заносится дата и время создания файла. Это время в формате DOS. Его можно перевести во время типа **TDateTime**, используемого в Delphi (см. разд. 2.9), функцией **FileDateToDateTime**, а если требуется перевести его в строку, то к полученному значению можно затем применить функцию **DateTimeToStr**. Таким образом, выражение вида

```
DateTimeToStr(FileDateToDateTime(F.Time))
```

вернет дату и время создания файла в виде строки.

Поле **Attr** записи **F** содержит атрибуты файла. Определить тип найденного файла можно комбинированием соответствующего флага с полем **Attr** по операции **and**. Если файл имеет данный атрибут, то результат этой операции будет больше 0.

Например, чтобы узнать, является ли найденный файл системным, надо записать выражение

```
(F.Attr and faSysFile > 0)
```

Это выражение вернет **true**, если файл системный.

Таким образом, вызов **FindFirst** позволяет найти первый файл, удовлетворяющий условиям поиска, или убедиться, что ни одного такого файла нет. Продолжение поиска осуществляется вызовом функции **FindNext** и передачей в нее в качестве параметра **F** той же записи, которая передавалась в **FindFirst**. Если **FindNext** вернет 0, значит нашелся еще один файл, удовлетворяющий условиям поиска. Информация об этом файле занесется в ту же запись **F**, после чего можно снова вызывать **FindNext** для поиска следующего файла. Если **FindNext** вернет ненулевое значение, значит больше нет файлов, удовлетворяющих условиям поиска. В этом случае надо вызвать процедуру **FindClose** с тем же параметром **F**. Эта процедура завершает поиск и освобождает ресурсы, выделенные для него.

Приведенный ниже пример обеспечивает занесение в список **ListBox1** имен всех файлов с расширением *.doc*, расположенных в каталоге «c:\test» и имеющих доступ только для чтения. В список **ListBox2** заносятся строки, отображающие дату и время создания этих файлов.

```
var F: TSearchRec;
    ires: integer;
...
ires := FindFirst('c:\test\*.doc', faReadOnly + faAnyFile, F);
while ires=0 do
 begin
  ListBox1.Items.Add(F.Name);
  ListBox2.Items.Add(DateTimeToStr(FileDateToDateTime(F.Time)));
  ires := FindNext(F)
 end;
FindClose(F);
```

Приведенный ниже код обеспечивает поиск и удаление из каталога, заданного переменной **sdirtmp**, всех файлов с расширением *.tmp*:

```
var sSR: TSearchRec;
    F: File;
    ires: integer;
    sdirtmp: string;
...
ires := FindFirst(sdirtmp+'\*.tmp', faAnyFile, sSR);
while ires=0 do
 begin
  if not DeleteFile(sdirtmp+sSR.name)
   then ShowMessage('Файл "' + sdirtmp + '\' + sSR.name +
                    '" не может быть удален');
  ires := FindNext(sSR)
 end;
FindClose(sSR);
```

Если вы укажете в приведенном коде шаблон "*.~*", то будут удалены в каталоге **sdirtmp** все файлы, расширение которых начинается с символа "~" — т.е. все резервные копии файлов проекта Delphi.

3.5 Классы

3.5.1 Объявление класса

Класс — это тип данных, определяемый пользователем. То, что в Delphi имеется множество предопределенных классов, не противоречит этому определению — ведь разработчики Delphi тоже пользователи Object Pascal.

Класс должен быть объявлен до того, как будет объявлена хотя бы одна переменная этого класса. Т.е. класс не может объявляться внутри объявления переменной.

В любом вашем приложении вы можете увидеть строки:

```
type
TForm1 = class(TForm)
    Button1: TButton;
    ...
    procedure Button1Click(Sender: TObject);
    ...
  end;
var
Form1: TForm1;
```

Это объявление класса **TForm1** вашей формы и объявление переменной **Form1** — объекта этого класса.

В общем случае синтаксис объявления класса следующий:

```
Type
  <имя класса> = Class (<имя класса - родителя>)
    public           // доступно всем
     <поля, методы, свойства, события>
    published        // видны в Инспекторе Объекта и изменяемы
      <поля, свойства>
    protected        // доступ только потомкам
     <поля, методы, свойства, события>
    private          // доступ только в модуле
     <поля, методы, свойства, события>
  end;
```

Имя класса может быть любым допустимым идентификатором. Но принято идентификаторы большинства классов начинать с символа "T". Имя класса — родителя может не указываться. Тогда предполагается, что данный класс является непосредственным наследником **TObject** — наиболее общего из предопределенных классов. Таким образом, эквивалентны следующие объявления:

```
type TMyClass = class
 ...
end;
```

и

```
type TMyClass = class(TObject)
  ...
end;
```

В приведенном ранее объявлении класса формы **TForm1** видно, что его родительским классом является класс **TForm**.

Класс наследует поля, методы, свойства, события от своих предков и может отменять какие-то из этих элементов класса или вводить новые. Доступ к объявляемым элементам класса определяется тем, в каком разделе они объявлены.

Раздел **public** (открытый) предназначен для объявлений, которые доступны для внешнего использования. Это открытый интерфейс класса. Раздел **published** (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и которые, следовательно, пользователь может устанавливать в процессе проектирования. Раздел **private** (закрытый), содержит объявления полей, процедур и функций, используемых только внутри данного класса. Раздел **protected** (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от конечного пользователя. Однако в отличие от закрытых, защищенные элементы остаются доступны для программистов, которые захотят производить от этого класса производные объекты, причем не требуется, чтобы производные объекты объявлялись в этом же модуле.

Объявления полей выглядят так же, как объявления переменных или объявления полей в записях:

```
<имя поля>: <тип>;
```

В приведенном ранее объявлении класса формы вы можете видеть строку

```
Button1: TButton;
```

Это объявление объекта (поля) **Button1** типа **TButton**.

Имеется одно очень существенное отличие объявления поля от обычного объявления переменной: в объявлении поля не разрешается его инициализация каким-то значением. Автоматически проводится стандартная инициализация: порядковым типам в качестве начального значения задается 0, указателям — **nil**, строки задаются пустыми. При необходимости задания других начальных значений используются конструкторы, описанные далее в разд. 3.5.3.

Объявления методов в простейшем случае также не отличаются от обычных объявлений процедур и функций.

3.5.2 Свойства

Поля данных, исходя из принципа инкапсуляции — одного из основополагающих в объектно-ориентированном программировании, всегда должны быть защищены от несанкционированного доступа. Доступ к ним, как правило, должен осуществляться только через свойства, включающие методы чтения и записи полей. Поэтому поля целесообразно объявлять в разделе **private** — закрытом разделе класса. В редких случаях их можно помещать в **protected** — защищенном разделе класса, чтобы возможные потомки данного класса имели к ним доступ. Традиционно идентификаторы полей совпадают с именами соответствующих свойств, но с добавлением в качестве префикса символа 'F'.

Свойство объявляется оператором вида:

```
property <имя свойства>:<тип> read <имя поля или метода чтения>
                              write <имя поля или метода записи>
                              <директивы запоминания>;
```

Если в разделах **read** или **write** этого объявления записано имя поля, значит предполагается прямое чтение или запись данных.

Если в разделе **read** записано имя метода чтения, то чтение будет осуществляться только функцией с этим именем. Функция чтения — это функция без параметра, возвращающее значение того типа, который объявлен для свойства. Имя функции чтения принято начинать с префикса **Get**, после которого следует имя свойства.

Если в разделе **write** записано имя метода записи, то запись будет осуществляться только процедурой с этим именем. Процедура записи — это процедура с одним параметром того типа, который объявлен для свойства. Имя процедуры записи принято начинать с префикса **Set**, после которого следует имя свойства.

Если раздел **write** отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь не может задавать его значение.

Директивы запоминания определяют, как надо сохранять значения свойств при сохранении пользователем файла формы *.dfm*. Чаще всего используется директива

```
default <значение по умолчанию>
```

Она не задает начальные условия. Это дело конструктора. Директива просто говорит, что если пользователь в процессе проектирования не изменил значение свойства по умолчанию, то сохранять значение свойства не надо.

Приведем пример. Мы уже не раз использовали объекты, содержащие сведения о сотруднике некоторой организации. Но раньше мы реализовывали эти объекты в виде записей (см. разд. 3.3). А теперь рассмотрим более общий подход — реализацию класса подобных объектов.

Начните новый проект. Объявление класса, который мы хотим создать, можно поместить непосредственно в файл модуля. Но если вы хотите создать класс, который будете использовать в различных проектах, лучше оформить его в виде отдельного модуля **unit**, сохранить в каталоге своей библиотеки или библиотеки Delphi, и подключать в дальнейшем к различным проектам с помощью предложения **uses**. Если вы забыли, как создается отдельный модуль, не связанный с формой, и как он сохраняется в библиотеке, посмотрите все это в разд. 2.8.7.4. Мы выберем именно этот вариант. Так что выполните команду File | New | Unit (в некоторых более старых версиях Delphi — File | New и на странице New выберите пиктограмму Unit). Сохраните сразу этот ваш модуль в библиотеке под именем, например, *MyClasses*. А в модуль формы введите оператор, ссылающийся на этот модуль:

```
uses MyClasses;
```

Теперь займемся созданием класса в модуле *MyClasses*. Текст этого модуля может иметь пока такой вид:

```
unit MyClasses;

interface
uses SysUtils, Dialogs, Classes;

type
  TPerson = class
  private
    FName: string;                  // Фамилия, имя и отчество
    FDep1, FDep2, FDep3: string;    // Место работы (учебы)
    FYear: word;                    // Год рождения
    FSex: char;                     // Пол: "м" или "ж"
    FAttr: boolean;                 // Булев атрибут
```

```
    FComment: string;                  // Комментарий
  protected
    procedure SetSex(Value: char);   // Процедура записи
  public
    property Name: string read FName write FName;
    property Dep1: string read FDep1 write FDep1;
    property Dep2: string read FDep2 write FDep2;
    property Dep3: string read FDep3 write FDep3;
    property Year: word read FYear write FYear;
    property Sex: char read FYear write SetSex default 'м';
    property Attr: boolean read FAttr write FAttr default true;
    property Comment: string read FComment write FComment;
  end;

implementation

procedure TPerson.SetSex(Value: char);
// Процедура записи пола
begin
if Value in ['м', 'ж']
  then FSex := Value
  else ShowMessage('Недопустимый символ "' + Value +
                   '" в указании пола');
end;
end.
```

Вглядимся в приведенный код. Интерфейсный раздел модуля **interface** начинается с предложения **uses**. Заранее включать это предложение в модуль не требуется. Но по мере написания кода вы будете встречаться с сообщениями компилятора о неизвестных ему идентификаторах функций, типов и т.п. Столкнувшись с таким сообщением, надо посмотреть во встроенной справке Delphi или в справке [3], в каком модуле объявлена соответствующая функция или класс. И включить этот модуль в приложение **uses**.

Теперь обратимся к объявлению класса. Объявленный класс **TPerson** наследует непосредственно классу **TObject**, поскольку родительский класс не указан. В закрытом разделе класса **private** объявлен ряд полей. Поле **FName** предполагается использовать для фамилии, имени и отчества человека. Поля **FDep1**, **FDep2**, **FDep3** будут использоваться под указание места работы или учебы. Поле **FYear** будет хранить год рождения, поле **FSex** — указание пола: символ "м" или "ж". Поле **FAttr** будет хранить какую-то характеристику: штатный — нештатный, отличник или нет и т.п. Поле **FComment** предназначено для каких-то текстовых комментариев. В частности, в нем можно хранить свойство **Text** многострочного окна редактирования **Memo** или **RichEdit**. Так что это может быть развернутая характеристика человека, правда, без форматирования.

В открытом разделе класса **public** объявлены свойства, соответствующие всем полям. Чтение всех свойств осуществляется непосредственно из полей. Запись во всех свойствах, кроме **Sex**, осуществляется тоже непосредственно в поля. А для поля **Sex** указана в объявлении свойства процедура записи **SetSex**, поскольку надо следить, чтобы по ошибке в это поле не записали символ, отличный от "м" и "ж". Соответственно в защищенном разделе класса **protected** содержится объявление этой процедуры. Как говорилось ранее, она должна принимать единственный параметр типа, совпадающего с типом свойства.

В раздел модуля **implementation** введена реализация процедуры записи **SetSex**. Ее заголовок повторяет объявление, но перед именем процедуры вводится ссылка на класс **TPerson**, к которому она относится. Не забывайте давать такие ссылки для методов класса. Иначе получите сообщение компилятора об ошибке: «Unsatisfied forward or external declaration: 'TPerson.SetSex'» — нереализованная ранее объявленная или внешняя функция 'TPerson.SetSex'.

Тело процедуры **SetSex** в особых комментариях не нуждается. В нем проверяется допустимость символа, переданного в процедуру через параметр **Value**. Если символ допустимый, то его значение заносится в поле **FSex**. Это поле, как и другие закрытые поля, открыто для методов данного класса. При ошибочном символе пользователю выдается сообщение об ошибке. Правда, лучше было бы в этом случае сгенерировать исключение, но пока мы не обсуждали, как это можно делать.

Вы создали класс в вашем модуле *MyClasses*. В дальнейшем мы его существенно модифицируем. Но пока давайте посмотрим, как можно использовать объекты нашего класса. Создайте в модуле формы *Unit1* вашего приложения тест класса **TPerson**. Введите в модуль операторы:

```
uses Class1;
var Pers: TPerson;
```

Они обеспечивают связь с модулем, описывающим класс, и объявляют переменную **Pers**, через которую вы будете связываться с объектом класса. Но так же, как при работе с другими объектами и записями, объявление этой переменной еще не создает сам объект. Это указатель на объект, и его значение равно **nil**.

Создание объекта вашего класса **TPerson** должно осуществляться вызовом его конструктора **Create**. Так что создайте обработчик события **OnCreate** вашей формы, и вставьте в него оператор:

```
Pers := TPerson.Create;
```

Вот теперь объект создан, и переменная **Pers** указывает на него. Чтобы не забыть очистить память от этого объекта при завершении работы приложения, сразу создайте обработчик события **OnDestroy** формы, и вставьте в него оператор:

```
Pers.Free;
```

Почему ваш объект воспринимает методы **Create** и **Free**? Ведь вы их не объявляли в классе **TPerson**! Это работает механизм наследования. В классе **TObject**, являющемся родительским для **TPerson**, методы **Create** и **Free** имеются. А поскольку вы их не перегружали, то ваш класс наследует их.

Теперь перенесите на форму четыре окна **Edit**, окно **Memo** и две кнопки. Пусть первая кнопка с надписью Запись заносит в объект **Pers** данные из окон редактирования: фамилию с именем и отчеством, пол, подразделение, в котором работает или обучается человек, год рождения, характеристику из окна **Memo**. Обработчик щелчка на ней может иметь вид:

```
with Pers do
begin
 Name := Edit1.Text;
 if Edit2.Text <> '' then Sex :=  Edit2.Text[1];
 Dep1 := Edit3.Text;
 Year := StrToInt(Edit4.Text);
 Comment := Memo1.Text;
end;
```

А вторая кнопка с надписью Чтение пусть осуществляет чтение информации из объекта в окна редактирования. Обработчик щелчка на ней может иметь вид:

```
with Pers do
begin
 Edit1.Text := Name;
 Edit2.Text :=  Sex;
 Edit3.Text := Dep1;
 Edit4.Text := IntToStr(Year);
 Memo1.Text := Comment;
end;
```

Выполните ваше приложение. Занесите в окна редактирования какую-то подходящую информацию и щелкните на кнопке Запись. А потом сотрите тексты всех окон редактирования и щелкните на кнопке Чтение. Информация в окнах должна восстановиться. Проверьте также реакцию на неверный символ, задающий пол.

3.5.3 Конструкторы и деструкторы

Конструкторы — это специальные методы, создающие и инициализирующие объект класса. Объект создается выделением для него памяти в динамически распределяемой области памяти heap (см. разд. 2.11). Объявление конструктора выглядит так же, как объявление процедуры, но предваряется ключевым словом **constructor**. В качестве имени конструктора обычно задают имя **Create**.

В реализации конструктора обычно первым идет вызов наследуемого конструктора с помощью ключевого слова **inherited**. В результате инициализируются все наследуемые поля. При этом порядковым типам в качестве начального значения задается 0, указателям — **nil**, строки задаются пустыми. После вызова наследуемого конструктора в процедуре инициализируются новые поля, введенные в данном классе.

Не обязательно при создании класса объявлять его конструктор. Если он не объявлен, то автоматически в момент создания объекта вызовется конструктор родительского класса. Мы это видели в примере класса **TPerson** и его тестового приложения, созданных в разд. 3.5.2. Там мы обошлись без конструктора, и все работало нормально. Но давайте несколько расширим возможности нашего класса, и посмотрим, сможем ли мы обойтись без конструктора. Добавим в класс открытые поля **AgeMin**, **AgeMax**, обозначающие минимальную и максимальную границы допустимого возраста. Подобные границы полезны, если приложение использует наш класс для регистрации претендентов на какую-то должность при приеме на работу, или для регистрации абитуриентов при приеме в учебное заведение. Пусть при задании года рождения класс автоматически проверяет, удовлетворяет ли регистрируемая личность поставленным возрастным ограничениям.

Для реализации этой идеи в объявлении класса надо произвести следующие изменения:

```
uses ..., DateUtils;
type
  TPerson = class
  private
  ...
  protected
   procedure SetYear(Value: word); // Процедура записи
  ...
```

```
  public
   AgeMin, AgeMax: word;
   ...
   property Year: word read FYear write SetYear;
  end;
```

В оператор **uses** вводится ссылка на модуль *DateUtils*. В этом модуле объявлены функции, которые мы будем использовать при записи года рождения. В защищенный раздел класса **protected** вводится объявление процедуры записи года рождения. В открытый раздел класса вводятся переменные **AgeMin** и **AgeMax**. И изменяется объявление свойства **Year**: теперь в него вводится ссылка на процедуру записи **SetYear**.

Реализация процедуры записи **SetYear** может быть следующей:

```
procedure TPerson.SetYear(Value: word);
// Процедура записи года рождения
var NowYear: word;
begin
 NowYear := YearOf(Date);
 if (NowYear - Value >= AgeMin) and (NowYear - Value <= AgeMax)
  then FYear := Value
  else ShowMessage('Недопустимый год рождения ' +
                   IntToStr(Value));
end;
```

В переменную **NowYear** заносится текущий год. Делается это так. Вызывается функция **Date**, которая возвращает текущую дату. Затем к этому результату применяется функция **YearOf**, которая выделяет из даты год. Функция **YearOf** объявлена в модуле *DateUtils*. Именно из-за нее этот модуль подключается предложением **uses**.

Остальное просто: проверяется, укладывается ли заданный год рождения **Value** в заданный диапазон, и если не укладывается, пользователю выдается соответствующее сообщение.

Возможно, вы уже заметили, почему все это пока не будет нормально работать. Если нет, введите описанные изменения в ваш класс и выполните тестовое приложение. Вы увидите, что можете занести в объект класса только того, кто родился в текущем году. А такой гражданин, пожалуй, слишком молод для приема на работу или учебу. Объясняется это просто: целые поля, такие как **AgeMin** и **AgeMax**, инициализируются нулевыми значениями. А нам нужны какие-то другие, более разумные возрастные рамки. Конечно, пользователь класса легко может этот устранить. Достаточно ввести, например, в обработчик события формы **OnCreate** после оператора создания объекта класса операторы вида:

```
Pers.AgeMax := 45;
Pers.AgeMin := 16;
```

Тем самым пользователь задает возрастные рамки для своего приложения. Ведь открытые поля **AgeMin** и **AgeMax** мы для того и вводили. Но хотелось бы, чтобы в случаях, если пользователю не требуется какой-то специфический возрастной диапазон, класс позволял бы работать с любыми реальными годами рождения. Но именно реальными, чтобы предотвратить случайную ошибку пользователя, когда он задаст, например, год рождения 3 или 3003.

Как указывалось в разд. 3.5.3, задать начальные значения полей в их объявлении невозможно. Вот для этого и служит конструктор класса. Добавьте в открытый раздел вашего класса объявление:

```
constructor Create;
```

А в раздел **implementation** добавьте реализацию конструктора:

```
constructor TPerson.Create;
const Unknown = 'неизвестный';
begin
 inherited;
 FSex := #0;
 AgeMax := 150;
 FName := Unknown;
 FDep1 := Unknown;
 FDep2 := Unknown;
 FDep3 := Unknown;
end;
```

Первый оператор вызывает с помощью ключевого слова **inherited** наследуемый конструктор родительского класса. А затем задаются начальные значения различных полей. Кроме задания максимального возраста 150 лет, тут исправляются еще некоторые недостатки нашего класса. Задается значение пола равное нулевому символу. По такому значению в дальнейшем можно проверять, был ли указан пол личности. И задаются строки «неизвестный» в качестве начального значения строковых полей. Так что теперь, если какие-то данные о личности неизвестны, вместо них будет выдаваться этот текст, а не пустая строка. Думаю, что пользователю это будет удобно.

Теперь рассмотрим *деструкторы*. Это специальные методы, уничтожающие объект и освобождающие занимаемую им память. Деструктор автоматически вызывается при выполнении метода **Free** объекта класса. Если в вашем классе деструктор не объявлен, вызывается деструктор родительского класса.

В большинстве случаев объявлять деструктор в классе не требуется. И мы до сих пор прекрасно без него обходились. Деструктор нужен в тех случаях, когда в конструкторе или в каком-то методе класса создается объект, динамически размещаемый в памяти. Тогда нужен деструктор, чтобы уничтожить этот объект и освободить занимаемую им память. Аналогично деструктор требуется, если объект создает и хранит информацию в каких-то временных файлах. Тогда в деструкторе надо уничтожить эти файлы, чтобы они не оставались на диске.

Давайте введем в наш класс **TPerson** еще одно добавление, которое расширит его возможности. Создадим возможность хранить форматированный текст в формате *.rtf*, который может поступать, как вы знаете, из окна **RichEdit** (см. разд. 3.2.4). Но форматированный текст можно хранить или в объекте класса **TRichEdit**, или в файле, в котором он записан методом **SaveToFile** свойства **Lines** окна **RichEdit**. Создать в нашем классе внутренний объект класса **TRichEdit** мы не можем. Так что остается вариант хранения во временном файле.

Добавьте в класс **TPerson** два поля:

```
FDoc: ^TRichEdit;
FileTMP: string;
```

Поле **FDoc** будет служить указателем на внешний компонент класса **TRichEdit**, из которого будет загружаться форматированный текст в файл, и в ко-

торый будет грузиться текст из файла. А в переменной **FileTmp** будем хранить имя временного файла. Чтобы компилятор принял объявление поля **FDoc**, он должен понять идентификатор **TRichEdit**. Этот класс объявлен в модуле *ComCtrls*. Так что добавьте ссылку на этот модуль в предложение **uses**.

Введите в открытый раздел класса объявления двух процедур:

```
procedure SetDoc(var Value: TRichEdit);
procedure GetDoc(var Value: TRichEdit);
```

Первая из них будет запоминать форматированный текст из окна **RichEdit**, переданного в нее как указатель на объект (вспомните смысл ключевого слова **var**). А вторая процедура будет заносить в окно, указанное аналогичным образом ее параметром, текст из файла.

Реализация этих функций может иметь вид:

```
procedure TPerson.SetDoc(var Value: TRichEdit);
begin
 if FileTMP = ''
  then FileTMP := FName + '.tmp';
   FDoc := @Value;
   FDoc^.Lines.SaveToFile(FileTMP);
end;

procedure TPerson.GetDoc(var Value: TRichEdit);
begin
 if FileTMP <> ''
  then FDoc^.Lines.LoadFromFile(FileTMP);
end;
```

В процедуре **SetDoc** сначала проверяется, задавалось ли уже имя временного файла. Если нет, то это имя формируется из строки **FName** (фамилия, имя, отчество) и расширения *.tmp*. Затем переменной **FDoc** задается адрес компонента **TRichEdit**. А дальнейшее понятно: форматированный текст записывается в файл с именем, хранящимся в переменной **FileTMP**.

Добавьте в свое тестовое приложение окно **RichEdit**. Можете добавить также компонент **FontDialog** и обеспечить возможность форматирования текста в окне **RichEdit** (см. разд. 3.2.4). В обработчик щелчка на кнопке Запись вставьте оператор:

```
SetDoc(RichEdit1);
```

А в обработчик щелчка на кнопке Чтение вставьте оператор:

```
GetDoc(RichEdit1);
```

Выполнив тестовое приложение, можете убедиться, что ваш класс стал намного мощнее и может теперь хранить форматированный текст. Но он пока не совсем правильно оформлен. Если в приложении выполнялся вызов процедуры **SetDoc**, то был создан временный файл. И когда приложение завершается, этот файл остается на диске. Вам нужно написать деструктор, который удалял бы этот файл.

Объявление деструктора выглядит так же, как объявление процедуры, но предваряется ключевым словом **destructor**. В качестве имени деструктора обычно задают имя **Destroy**. Реализация деструктора, как правило, завершается вызовом наследуемого деструктора с помощью ключевого слова **inherited**, чтобы освободить память, отведенную для наследуемых полей.

В нашем случае в открытый раздел класса следует ввести объявление деструктора:

```
destructor Destroy; override;
```

Смысл ключевого слова **override** вы узнаете позднее. Пока просто напишите его, не задумываясь о его назначении.

Реализация деструктора имеет вид:

```
destructor TPerson.Destroy;
begin
 if FileTMP <> ''
  then DeleteFile(FileTMP);
 inherited;
end;
```

Если файл создавался, то он уничтожается. Введение такого деструктора никак не отразится на внешнем поведении вашего тестового приложения. Но теперь вы не оставляете на диске временный и уже ненужный файл.

3.5.4 Методы, наследование классов, операции с классами

В предыдущих разделах мы уже создали ряд методов: чтения и записи свойств, конструктор, деструктор. Поскольку вы уже освоились с методами, давайте создадим еще два метода: **SaveToFile** и **LoadFromFile**, позволяющих сохранять сведения о личности в файле и читать их из файла. Без подобных методов наш класс явно неполноценный.

Введите в класс объявления открытых методов:

```
procedure SaveToFile(FileName: string = '');
procedure LoadFromFile(FileName: string);
```

Первый метод содержит значение параметра по умолчанию (см. разд. 2.7.3). Предполагается, что если при вызове метода параметр не задан, то имя файла будет совпадать с именем личности, записанным в поле **FName**.

Реализация объявленных методов может быть такой:

```
procedure TPerson.SaveToFile(FileName: string = '');
var F: TextFile;
begin
 if FileName = ''
  then FileName := FName + '.txt'
 else FileName := ChangeFileExt(FileName, '.txt');
 AssignFile(F, FileName);
 Rewrite(F);
 Writeln(F, FName);
 Writeln(F, FDep1);
 Writeln(F, FDep2);
 Writeln(F, FDep3);
 Writeln(F, IntToStr(FYear));
 Writeln(F, FSex);
 if FAttr then Writeln(F, 't')
  else Writeln(F, 'f');
 Writeln(F, FComment);
 CloseFile(F);
 if FDoc <> nil
```

```
  then FDoc^.Lines.SaveToFile(ChangeFileExt(FileName, '.rtf'));
end;

procedure TPerson.LoadFromFile(FileName: string);
var F: TextFile;
    s: string;
begin
 FileName := ChangeFileExt(FileName, '.txt');
 AssignFile(F, FileName);
 Reset(F);
 Readln(F, FName);
 Readln(F, FDep1);
 Readln(F, FDep2);
 Readln(F, FDep3);
 Readln(F, s);
 FYear := StrToInt(s);
 Readln(F, FSex);
 Readln(F, s);
 FAttr := (s = 't');
 Readln(F, FComment);
 CloseFile(F);
 if (FDoc <> nil) and FileExists(
                              ChangeFileExt(FileName, '.rtf'))
  then begin
   FDoc^.Lines.LoadFromFile(ChangeFileExt(FileName, '.rtf'));
   FDoc^.Lines.SaveToFile(FileTMP);
  end;
end;
```

Вы уже достаточно опытны, чтобы подобный код можно было не комментировать. Отмечу только, что в этом коде хранение данных полей предусмотрено в текстовом файле. Это не лучший вариант, но наиболее простой. Одним из недостатков такого хранения является то, что файлы доступны возможным злоумышленникам, которые могут прочитать их, получить доступ к конфиденциальным данным, а могут и изменить характеристику. Правда, все это они смогут сделать и с помощью вашего тестового приложения. Но приложение можно защитить паролем — позднее, в разд. 4.13.2 вы научитесь это делать. А текстовый файл ничем не защитишь. Так что может иметь смысл хотя бы зашифровать его в процедуре **SaveToFile** и дешифровать в процедуре **LoadFromFile**. Шифровка текстов рассмотрена в разд. 2.8.7.2.

Вызов метода **SaveToFile** в вашем тестовом приложении может быть оформлен оператором:

```
Pers.SaveToFile;
```

или

```
if SaveDialog1.Execute
 then Pers. SaveToFile(SaveDialog1.FileName);
```

В первом случае имя файла будет совпадать с именем личности. Во втором пользователь может указать в стандартном диалоге имя файла. В любом случае, вероятно, перед сохранением полезно вызвать обработчик щелчка на кнопке Запись, чтобы в объекте запомнились значения соответствующих окон редактирования.

Вызов метода **LoadFromFile** в вашем тестовом приложении может быть оформлен оператором:

```
if OpenDialog1.Execute
then Pers.LoadFromFile(OpenDialog1.FileName);
```

После него, вероятно, полезно вызвать обработчик щелчка на кнопке Чтение, чтобы прочитанные данные немедленно отобразились в окнах редактирования.

Все открытые и защищенные свойства и методы наследуются в классах, для которых данный класс является родительским. Давайте, например, построим два класса-наследника нашего класса **TPerson**. Они будут конкретизировать сведения об абстрактной личности, которую представляет класс **TPerson**. Один класс-наследник **TStudent** пусть представляет студента, а другой класс **TEmpl** представляет сотрудника некоторой организации. Объявления этих классов могут выглядеть так:

```
  TStudent = class(TPerson)
  public
   function PersonToStr: string;
  end;

  TEmpl = class(TPerson)
  public
   function PersonToStr: string;
  end;
```

Эти классы наследуют все свойства и методы своего базового класса **TPerson**. В обоих классах объявлены новые открытые методы **PersonToStr**. Предполагается, что их назначение — представить информацию о человеке в виде единой строки. Такая строка полезна во многих случаях. В частности, например, для подготовки шапки характеристики, как в приложении, которое вы разрабатывали в разд. 1.3.

Реализация этих методов может иметь вид:

```
function TStudent.PersonToStr: string;
const Unknown = 'неизвестный';
var S: string;
begin
 S := FName + ', ';
 if FYear = 0
  then S := S + Unknown
  else S := S + IntToStr(FYear);
 S := S + ' г.р., ';
 if FSex = 'ж'
  then S := S + 'студентка'
  else S := S + 'студент';
 S := S + ' группы ' + FDep1 + ' института ' + FDep2;
 Result := S;
end;

function TEmpl.PersonToStr: string;
const Unknown = 'неизвестный';
var S: string;
begin
 S := FName + ', ';
 if FYear = 0
  then S := S + Unknown
  else S := S + IntToStr(FYear);
```

```
 S := S + ' г.р., ';
 if FSex = 'ж'
  then S := S + 'сотрудница'
  else S := S + 'сотрудник';
 S := S + ' отдела "' + FDep1 + '" организации ' +
      FDep2;
 Result := S;
end;
```

Комментарии к этому коду, наверное, излишни. Имеет смысл сказать только о смысле свойств **Dep1** и **Dep2** в новых классах. Как видно из кода, свойство **Dep1** в классе **TStudent** указывает студенческую группу, а в классе **TEmpl** — отдел. Свойство **Dep2** указывает организацию (институт), в которой работает (учится) человек. Вы можете, конечно, воспринимать свойства методов иначе и соответственно переделать приведенные функции.

Можете добавить в ваше тестовое приложение еще одно окно **Edit** для задания свойства **Dep2**. Но главное сейчас — посмотреть соотношение родительского и производных классов.

Можете оставить прежнее объявление

```
var Pers: TPerson;
```

но при создании объекта указать какой-то производный класс, например:

```
Pers := TStudent.Create;
```

Все будет работать, т.е. производный класс присвоится переменной родительского класса. Но если вы попробуете записать оператор вида:

```
ShowMessage(Pers.PersonToStr);
```

то получите сообщение об ошибке: «Undeclared identifier: 'PersonToStr'», поскольку в классе **TPerson** не объявлен метод **PersonToStr**. Но все будет нормально, если вы измените этот оператор следующим образом:

```
ShowMessage((Pers as TStudent).PersonToStr);
```

В этом операторе вы используете операцию **as**, определенную для классов. Ее первым операндом является объект, вторым — класс. Если **A** — объект, а **C** — класс, то выражение **A as C** возвращает тот же самый объект, но рассматриваемый как объект класса **C**. Операция даст результат, если указанный класс **C** является классом объекта **A** или одним из наследников этого класса. В нашем случае это условие соблюдается, так что операция **as** возвращает объект **Pers**, рассматриваемый как объект **TStudent**. А в таком объекте метод **PersonToStr** имеется.

Если бы вы сразу объявили переменную **Pers** класса **TStudent**:

```
var Pers: TStudent;
```

то операция **as** не потребовалась бы. Выражение **Pers.PersonToStr** было бы воспринято нормально.

Для классов определена еще одна операция — **is**. Выражение **A is C** позволяет определить, относится ли объект **A** к классу **C** или к одному из его потомков. Если относится, то операция **is** возвращает **true**, в противном случае — **false**. Например, в нашем примере при любом объявлении класса создаваемого объекта **Pers** выражение **Pers is TPerson** вернет **true**. Но выражение **Pers is TStudent** вернет **true**, если объект создан как объект **TStudent**, и вернет **false**, если объект создан как объект **TPerson**.

Теперь посмотрим наследование методов. Давайте введем метод **PersonToStr** не только в производные, но и в базовый класс **TPerson**. Для этого класса его реализация может быть очень простой:

```
function TPerson.PersonToStr: string;
begin
 Result := FName;
end;
```

Теперь метод **PersonToStr** объявлен с одним и тем же именем и в родительском, и в производных классах. Иначе говоря, вы переопределили или перегрузили родительский метод в производных классах. Теперь вы сможете увидеть, что выражение **Pers.PersonToStr** будет всегда нормально срабатывать. Но вызываться будет метод того класса, который указан в объявлении переменной **Pers**. Если она объявлена как переменная класса **TPerson**, то всегда будет вызываться метод этого класса, даже если вы занесли в эту переменную указатель на объект класса **TStudent** или **TEmpl**. Если вы объявите переменную **Pers** класса **TStudent**, и занесете в нее указатель на объект класса **TStudent**, то вызываться будет метод этого класса. Если вы все-таки хотите в этом случае вызвать метод родительского класса, надо применить к переменной **Pers** явное приведение типа: **TPerson(Pers).PersonToStr**.

При реализации метода, переопределенного любым способом в классе-наследнике, можно вызывать метод класса-родителя. Для этого перед именем метода при его вызове записывается ключевое слово **inherited**. Например, оператор

```
inherited PersonToStr;
```

вызывает метод **PersonToStr** родительского класса.

Если записать слово **inherited** и после него не писать имя вызываемого метода, то будет вызываться наследуемый метод, совпадающий по имени с именем того метода, из которого он вызывается. Например, если в переопределяемом конструкторе встречается оператор

```
inherited;
```

то будет вызван конструктор родительского класса. Этим мы уже пользовались в разд. 3.5.3 при написании конструкторов и деструкторов класса **TPerson**.

3.5.5 Виртуальные методы, полиморфизм, абстрактные классы

Методы, с которыми вы имели дело в предыдущих разделах, называются *статическими*. По умолчанию все методы класса именно статические. Если в классе-наследнике переопределить такой метод (ввести новый метод с тем же именем), то для объектов этого класса новый метод отменит родительский. Если обращаться к объекту этого класса, то вызываться будет новый метод. Но если обратиться к объекту как к объекту родительского класса, то вызываться будет метод родителя.

Все это мы видели на примерах в предыдущем разделе. Но представим себе такую задачу. Вы объявили в приложении массив объектов типа **TPerson**:

```
var PersArray: array[1..10] of TPerson;
```

Далее заполняете этот массив вперемешку объектами классов **TStudent** и **TEmpl**, т.е. создаете, например, общий список учащихся и преподавателей. В разд. 3.5.4 вы видели, что это возможно, так как переменная базового класса мо-

жет принимать объекты производных классов. А затем хотите пройти в цикле элементы этого массива и отобразить в окне **Memo** информацию о них:

```
for i:=1 to 10 do
 Memo1.Lines.Add(PersArray[i].PersonToStr);
```

Или аналогичная задача с использованием списка **TList**: объявлена переменная

```
var List: TList;
```

в нее заносятся указатели на объекты классов **TStudent** и **TEmpl**, а затем вы хотите пройти в цикле элементы этого списка и отобразить их в окне **Memo**:

```
for i:=0 to List.Count - 1 do
  Memo1.Lines.Add(TPerson(List[i]).PersonToStr);
```

Реализуйте один из этих вариантов (не забудьте при завершении приложения удалять объекты из памяти), и посмотрите, что получится. А получится очевидный результат: во всех случаях сработает метод **PersonToStr** родительского класса **TPerson**, так как соответствующие объекты объявлены объектами этого класса. И программа, естественно, не знает, объекты какого класса хранятся в массиве или в списке.

Конечно, можно усложнить код, проверять каждый раз операцией **is** истинный класс объекта, и указывать операцией **as** этот класс (см. об этих операциях в разд. 3.5.4). Для массива это будет выглядеть так:

```
if PersArray[i] is TStudent
  then Memo1.Lines.Add((PersArray[i] as TStudent).PersonToStr)
  else if PersArray[i] is TEmpl
    then Memo1.Lines.Add((PersArray[i] as TEmpl).PersonToStr);
```

А для списка аналогичный код имеет вид:

```
if TPerson(List[i]) is TStudent
  then Memo1.Lines.Add(TStudent(List[i]).PersonToStr)
  else if TPerson(List[i]) is TEmpl
    then Memo1.Lines.Add(TEmpl(List[i]).PersonToStr);
```

Задача будет решена, но некрасиво. Во-первых, код усложняется. А во-вторых, если вы впоследствии решите создать какие-то новые классы, производные от **TPerson**, вам придется вводить соответствующие дополнительные проверки в эти коды.

Красивое решение подобных задач обеспечивается очень легко с помощью *виртуальных* методов. Виртуальные методы не связаны с другими методами с тем же именем в классах-наследниках. Если в классах-наследниках эти методы перегружены, то при обращении к такому методу во время выполнения будет вызываться тот из методов с одинаковыми именами, который соответствует истинному классу объекта. В приведенных примерах, если сделать методы **PersonToStr** виртуальными, то операторы

```
Memo1.Lines.Add(PersArray[i].PersonToStr);
```

и

```
Memo1.Lines.Add(TPerson(List[i]).PersonToStr);
```

будут автоматически вызывать методы **PersonToStr** того класса (**TStudent**, **TEmpl** или других производных от **TPerson**), к которому относится каждый объект. Такой подход, облегчающий работу с множеством родственных объектов, называется *полиморфизмом*.

Сделать метод родительского класса виртуальным очень просто. При объявлении в классе виртуальных методов после точки с запятой, завершающей объявление метода, добавляется ключевое слово **virtual**. Например, чтобы объявить в базовом классе **TPerson** метод **PersonToStr** виртуальным, надо в его объявление добавить слово **virtual**:

```
function PersonToStr: string; virtual;
```

Чтобы перегрузить в классе-наследнике виртуальный метод, надо после его объявления поставит ключевое слово **override**. В наших примерах объявления метода в классах и должны выглядеть так:

```
function PersonToStr: string; override;
```

И это все! Методы стали виртуальными и приведенные в начале данного раздела операторы, будут работать безо всяких проверок **if**.

Если в каком-то базовом классе метод был объявлен как виртуальный, то он остается виртуальным во всех классах-наследниках (в частности, и в наследниках классов наследников). Однако обычно для облегчения понимания кодов, перегруженные методы принято повторно объявлять виртуальными, чтобы была ясна их суть для тех, кто будет строить наследников данного класса. Например:

```
function PersonToStr: string; override; virtual;
```

В родительском классе виртуальный метод не обязательно должен быть реализован. Такой виртуальный метод, реализация которого не определена в том классе, в котором он объявлен, называется *абстрактным*. Предполагается, что этот метод будет перегружен в классах-наследниках. Только в тех классах, в которых он перегружен, его и можно вызывать.

Объявляется абстрактный метод с помощью ключевого слова **abstract** после слова **virtual**. Например, вы можете в классе **TPerson** объявить метод **PersonToStr** следующим образом:

```
function PersonToStr: string; virtual; abstract;
```

В этом случае вы должны удалить в этом классе реализацию метода.

Класс, содержащий абстрактный метод, сам становится абстрактным. Объекты такого класса создавать нельзя. Можно создавать объекты только тех классов-наследников, в которых абстрактный метод перегружен и реализован.

3.6 Некоторые итоги

В данной главе мы практически завершили рассмотрение языка Object Pascal. Вы уже можете считать себя грамотным программистом на этом языке (если, конечно, освоили изложенный материал и можете применять его на практике). Не страшно, если вы не помните наизусть какие-то синтаксические конструкции, имена свойств, методов, функций. Важнее, чтобы вы знали, какие задачи можно решать и как подойти к их решению. А имена функций, их параметры и т.п. вы всегда сможете посмотреть во встроенных справках Delphi, в справках [3], или с помощью оперативных подсказок в окне Редактора Кода.

Хотелось бы обратить ваше внимание на темы, являющиеся, на мой взгляд, наиболее важными в материале данной главы. Прежде всего, надо хорошо освоить работу с динамическими массивами. Это очень удобные объекты для решения множе-

ства задач. Но, все-таки, работа с массивами, строками, файлами — это чисто технические задачи, которыми надо просто достаточно свободно владеть. А основное в объектно-ориентированном проектировании и программировании — это объекты. Если говорить об элементах языка, то это записи и классы. И тут мало овладеть техникой. Надо постараться приучить себя формулировать в терминах объектов те задачи, которые вам приходится решать. Попробую пояснить, как осуществляется объектно-ориентированное проектирование. Не программирование, а именно проектирование — т.е. этап, который предшествует программированию.

Если перед вами стоит какая-то задача моделирования, надо представить себе моделируемый объект. Прежде всего, подумайте, не состоит ли он из ряда других объектов. Чаще всего приходится иметь дело с иерархией объектов. Какое-то устройство может состоять из ряда комплектующих изделий. Система может включать в себя ряд подсистем. В подобных случаях первый шаг — составление иерархии объектов. Самый верхний объект включает в себя множество объектов следующего уровня, те включают в себя более мелкие объекты и т.д.

Для каждого объекта надо составить список свойств, которыми он может обладать. Далее надо представить себе, какие операции могут потребоваться над этими свойствами. Только после того, как вся эта предварительная работа проделана и зафиксирована на бумаге (или в виде текстового и графического документа в памяти компьютера), можно начинать продумывать реализацию проекта.

Некоторые детали реализации зависят от того, будет ли у вас фиксированное число объектов нижних уровней, или это число заранее неизвестно. Например, рассмотренные в разд. 3.3.2, 3.3.3, 3.3.4 примеры списков записей о сотрудниках некоторой организации — типичные примеры, когда число объектов заранее неизвестно. То же относится к сведениям о продажах каких-то изделий или услуг, модели какого-то склада, производства и т.д. Если число объектов не фиксировано, значит для реализации потребуются или динамические массивы, или списки.

Реализация большинства объектов возможна или с помощью записей, или с помощью классов. В классах не представляет труда реализовать любые свойства (данные) и методы работы с ними. Реализация в виде записей позволяет хранить в них любые данные, и может быть дополнена набором функций, оперирующих с данными в записях. Пожалуй, наиболее принципиальный момент, различающий эти два подхода — доступ к данным. Работая с записями, программист и создаваемые им функции имеют прямой доступ к данным. Хорошо, если при этом в принципе невозможно занести в объект какие-то недопустимые значения, которые нарушат в нем целостность данных и функционирование. Если же такая опасность есть, предпочтение следует отдавать объектам, реализованным классами. В этом случае в полной мере можно реализовать принцип инкапсуляции — скрытия данных. Запись и чтение данных может осуществляться соответствующими функциями свойств. И если вы в дальнейшем надумаете изменить способы хранения данных и их обработки, то вы можете это сделать, не меняя открытые разделы класса. Тогда все потребители вашего класса даже не заметят изменения, никакого перепрограммирования приложений не потребуется. Просто эти приложения станут работать более эффективно, благодаря вашей модернизации класса.

Всю описанную работу надо проделать заранее, до начала программирования. После этого можно начинать реализацию. Она обычно происходит поэтапно. Сначала вместо большинства методов и функций ставятся так называемые «заглушки». Их объявления совпадают с объявлениями задуманных функций, а вместо тела записываются какие-то условные операторы, сообщающие вам о выполнении

задуманных действий и возвращающие правдоподобные результаты. Объект с такими заглушками позволяет оценить его работоспособность и взаимодействие с другими объектами. В результате могут быть уточнены структура и функции объектов. А затем заглушки поочередно заменяются реальными функциями, и в результате вы получаете задуманную прекрасно реализованную систему объектов.

Я не хочу приводить какой-то развернутый пример описанного объектно-ориентированного подхода к проектированию. Все зависит от той области, в которой вы специализируетесь. Но очень советую вам наметить задачу из знакомой вам области знаний, и попробовать применить к ней описанную методику. Не надо доводить дело до реализации, если, конечно, это не живая задача, которую жаждет решить какой-то заказчик. Но создать хотя бы раз, хотя бы на «бумаге» полноценный, выверенный проект, ориентированный на объекты, очень полезно каждому.

Все изложенное выше относится к программной реализации объектного подхода. А вторая задача любого проекта — создание удобного интерфейса пользователя, т.е. внешнего отображения ваших программистских усилий. Эта задача будет рассмотрена в следующих главах.

3.7 Проверьте себя

3.7.1 Вопросы для самопроверки

1. Как можно объявить массив, содержащий 5 действительных чисел, и задать в объявлении значения его элементов?
2. Как можно объявить двумерный массив размером 3 x 2, и задать в объявлении значения его элементов?
3. В каких случаях определена операция присваивания массивов друг другу?
4. В чем заключаются ошибки такого объявления функции:
   function F(A: array[1..5] of real): array[1..5] of real? Как их исправить?
5. Что такое открытые массивы? Как они передаются в функции и используются в них?
6. Что такое конструктор открытого массива?
7. Как можно объявить динамический массив действительных чисел, и задать значения 5-ти его элементов?
8. Как можно объявить и заполнить двумерный динамический массив действительных чисел размером 2 x 3?
9. Почему нельзя использовать типизированные файлы для хранения записей, содержащих поля длинных строк.
10. Файлы какого типа (текстовые, типизированные или нетипизированные) обеспечивают наибольшую скорость выполнения операций чтения и записи?
11. В чем различие объявлений методов и данных, помещенных в классе в разделы **public**, **published**, **protected** и **private**?
12. В каких случаях в класс надо включать конструктор?
13. Зачем и в каких случаях в класс включается деструктор?
14. Поясните различие операций **as** и **is**, применяемых к объектам классов.

15. Зачем вводятся в класс виртуальные методы?
16. Что такое абстрактный класс, и почему не льзя создавать объекты таких классов?

3.7.2 Задачи

1. Напишите функции вставки, удаления, перестановки и изменения элементов динамических массивов целых и действительных чисел по примеру функции, приведенной в разд. 3.1.3.1. Дополните их функциями, возвращающими в виде строки содержимое массивов. Включите все эти функции в вашу библиотеку.
2. Напишите для N-мерных векторов функции вычисления модуля, сложения, вычитания, скалярного и векторного произведения, умножения на скаляр, расчета угла межу векторами. Можете разделить создание таких функций между несколькими своими товарищами. Только интерфейс пользователя должён быть согласованным у всех разработчиков (полезно приучаться работать в коллективе программистов). Занесите разработанные функции в свою библиотеку и создайте для них тестовое приложение.
3. Напишите функции, реализующие основные операции с прямоугольными матрицами произвольного размера: сложение, вычитание, умножение, умножение на скаляр, вычисление определителя. Можете разделить создание таких функций между несколькими своими товарищами. Только интерфейс пользователя должен быть согласованным у всех разработчиков (полезно приучаться работать в коллективе программистов). Занесите разработанные функции в свою библиотеку и создайте для них тестовое приложение. С учетом функций, созданных в пункте 2, у вас получится неплохая библиотека функций для решения различных научных и технических задач.
4. Реализуйте алгоритм пузырьковой сортировки, начинающий просмотр со сравнения первой пары элементов и перемещающий к концу массива элемент с наибольшим значением.
5. Реализуйте алгоритм пузырьковой сортировки, упорядочивающий массив в последовательности убывания значений элементов.
6. Реализуйте метод Монте-Карло применительно к какой-то системе из той области знаний, которая вам ближе. Пояснения по разработке такого приложения даны в разд. 3.1.5.4.
7. Создайте вариант примера карточной игры (разд. 3.1.5.3), в котором начинал бы компьютер. Конечно, набранные им карты и очки пользователь должен увидеть только после того, как возьмет свои карты.
8. Напишите и протестируйте функцию, которая принимает строку с записанной фамилией, именем и отчеством, разделенными пробелами (может быть, несколькими), и возвращает строку с фамилией и инициалами, разделенными точками. Например: «Иванов А.Б.».
9. Создайте приложение, работающее со списком **TList** указателей на записи о сотрудниках некоторой организации или учащихся учебного заведения (см. разд. 3.3.3). Предусмотрите в нем возможность сортировки записей по алфавиту, по году рождения, по алфавиту подразделений, а внутри каждого подразеления — по алфавиту сотрудников.

10. Создайте на основе приложения, работающего со списками и рассмотренного в разд. 3.3.4 и 3.4.2, приложение, содержащее сведения по каким-то терминам языка Object Pascal. Основной список может содержать термины, вспомогательный — имена (осмысленные) текстовых файлов различных тем, связанных с этими терминами. При щелчке на строке вспомогательного списка соответствующий файл должен загружаться в окно **Memo** или **RichEdit**, которое должно быть на форме приложения. Более удобный, но немного более сложный вариант: вспомогательные списки содержат только названия тем, а каждой строке в них соответствует объект класса **TStringList**, содержащий имя файла.
11. Создайте приложение просмотра изображений, эскизно намеченное в разд. 3.4.8.
12. Введите в класс **TPerson**, разработанный в разд. 3.5, функцию **ShortName**, которая формировала бы из поля **FName** и возвращала строку с фамилией и инициалами. Например: «Иванов А.Б.». Предполагается, что в поле **FName** записаны фамилия, имя и отчество, разделенные пробелами (возможно, несколькими).
13. Создайте на основе класса **TList** класс списка, предназначенный для хранения объектов классов **TPerson** (см. разд. 3.5) и производных от него. Создаваемый класс должен иметь методы сохранения в файле и чтения из файла, выдачи содержимого списка в виде списка строк, методы сортировки данных по алфавиту, по году рождения, по подразделениям, а внутри подразделений — по алфавиту.
14. Проделайте описанные в разд. 3.6 подготовительные этапы объектно-ориентированной разработки какого-то проекта из области, близкой вам.

Часть 2

Приложения для Windows

Глава 4

Методика разработки приложений для Windows

В этой главе:

- *вы изучите методику построения хорошо структурированных приложений, доступных для модернизации и сопровождения*
- *научитесь создавать в приложении списки изображений*
- *узнаете о различиях стандартных и нестандартных действий*
- *освоите создание в приложении главного и контекстных меню*
- *освоите проектирование инструментальных панелей*
- *научитесь создавать файлы справок, и связывать их с приложениями Delphi*
- *научитесь управлять формами в приложениях с несколькими формами*
- *создадите тестовое приложение, удовлетворяющее всем требованиям изложенной методики*

4.1 Технология разработки приложений

Приложения Delphi для Windows можно, конечно, разрабатывать в любой последовательности — как бог на душу положит. Так что стоит ли мудрствовать: переноси компоненты на форму, пиши обработчики их событий и получай награду за прекрасно сделанное приложение.

Честно говоря, приведенные в предыдущих главах примеры так и строились. Там ставилась задача изучения программирования на языке Object Pascal. Так что цель создававшихся тестовых приложений была проста: обеспечить ввод исходных данных для расчета и отобразить результаты вычислений. Но настоящее приложение так проектировать нельзя. Бессистемно созданное приложение, даже очень хорошее, через некоторое время становится непонятным и самому разработчику, не говоря уж о проблемах сопровождения такого приложения кем-то из коллег. Подобное приложение, если возникает необходимость его доработки, проще создать заново, чем разбираться в его хитросплетениях.

Поэтому, чтобы избежать в дальнейшем лишней работы и нареканий в ваш адрес со стороны коллег, которым выпало несчастье модернизировать ваше приложение, лучше сразу приучить себя разрабатывать хорошо структурированные, полноценные приложения, которые в дальнейшем легко сопровождать и модернизировать.

Каждое серьезное приложение должно иметь:

- главное меню
- инструментальную панель быстрых кнопок, дублирующих основные разделы меню
- контекстные меню, всплывающие при щелчке пользователя правой кнопкой мыши на том или ином компоненте
- горячие клавиши для основных операций пользователя и клавиши быстрого доступа
- ярлычки подсказок, всплывающие при перемещении курсора мыши над быстрыми кнопками и иными компонентами
- полосу состояния, используемую часто для развернутых подсказок
- файл справки, темы которого отображаются при нажатии клавиши F1 и при выборе пользователем соответствующего раздела меню

Для того чтобы все многообразие перечисленных исполнительных элементов (различных меню, быстрых кнопок, управляющих кнопок, индикаторов) было хорошо структурировано, следует придерживаться определенной технологии разработки. Благо, в Delphi имеются для этого все возможности. Речь идет, прежде всего, о проектировании на основе списков действий, управляемых специальными компонентами.

В гл. 1 было рассказано об объектно-ориентированном программировании, а в гл. 3 были рассмотрены способы создания объектов в программах. В свое время объектная ориентация произвела революцию в программировании и сейчас уже трудно представить иное построение программы. Но увлечение объектами как-то отодвинуло временно на второй план функции, т.е. те действия, которые являлись основой предшествующего этапа в программировании. Сейчас интерес к действиям вновь возвращается. Ведь действия — это то, ради чего программа создается. И объекты интересуют только постольку, поскольку они могут облегчить действия пользователя.

Поэтому после того, как вами по методике объектно-ориентированного проектирования, рассмотренной в разд. 3.6, составлен план проекта, начинать его реализацию надо не с создания красивого и удобного пользовательского интерфейса, а с составления *списка действий*, которые пользователь сможет выполнять с помощью данного приложения. Конечно, по мере проектирования этот первоначальный список будет расширяться и уточняться. Но хотя бы начальный вариант такого списка должен являться отправной точкой проектирования. Без этого вряд ли можно создать что-нибудь стоящее.

Когда список действий составлен, можно начать размышлять над исполнительными элементами интерфейса, с помощью которых пользователь сможет инициировать действия. И здесь могут возникнуть определенные сложности. Обычно для одного и того же действия предусматривается несколько инициаторов. Например, как правило, некоторые разделы главного меню дублируются быстрыми кнопками инструментальных панелей, разделами контекстных меню, горячими клавишами, иногда обычными кнопками. Такое дублирование удобно пользователю. Но независимая разработка всех этих управляющих элементов часто приводит к неоправданному дублированию кодов, а главное — снижает возможности модернизации и сопровождения приложения. Действительно, если вы надумаете что-то изменить в одном из действий, вы должны будете вносить изменения сразу в нескольких

местах программы. И не дай Бог что-то пропустить: согласованная работа управляющих элементов нарушится.

Чтобы избежать подобных сложностей, в Delphi предусмотрены компоненты, осуществляющие *централизованное управление* действиями, их *диспетчеризацию*. Составленный вами список действий вы заносите в специальный компонент — диспетчер. Для каждого действия при этом вы можете задать множество свойств: надписи на соответствующих элементах управления, пиктограммы, тексты ярлычков подсказок, горячие клавиши и многое другое. Вы можете также написать обработчик, обеспечивающий выполнение задуманного действия. Впрочем, как вы увидите далее, для стандартных действий это даже обычно не нужно делать. Разработчики Delphi создали множество классов таких стандартных действий, в которых необходимые функции реализуются автоматически. Так что, не ознакомившись в деталях с этими возможностями, вы рискуете «изобретать велосипед». Вы потратите время и силы на их реализацию, а потом окажется, что то же самое можно было сделать несколькими движениями мыши, причем много лучше, чем вы это придумали сами.

Если вы сформировали подобный список действий, то последующее проектирование управляющих элементов существенно упрощается. В большинстве элементов имеется свойство **Action**. Достаточно сослаться в этом свойстве на одно из действий, как все свойства этого действия и обработчик, реализующий его, перенесутся в данный управляющий элемент. И вам не придется для каждого элемента задавать все это заново. Представляете, сколько времени вы сэкономите? А если вы в дальнейшем что-то измените в реализации действия или в его свойствах (например, смените горячие клавиши или пиктограмму), то вам даже не надо будет вспоминать, какие элементы в различных формах вашего приложения инициируют это действие. Все эти элементы автоматически воспримут внесенные изменения.

В Delphi, начиная с Delphi 4, предусмотрена подобная централизация управления приложением на нескольких уровнях. Если вы работаете с более младшими версиями Delphi, то, к сожалению, все, что будет далее сказано о диспетчеризации действий, можете пропустить. А в Delphi 4 для диспетчеризации действий появился компонент **ActionList**. Он упростил создание меню и инструментальных панелей, позволил делать программы более понятные и структурированные. Уже в Delphi 5 число стандартных действий, к которым можно обращаться из **ActionList**, было свыше двадцати. А в Delphi 7 их число достигло 66. Представляете, сколько стандартных операций реализовали за вас создатели Delphi?

Пока мы рассматривали только диспетчеризацию действий. Но централизация управления приложением имеет еще несколько уровней. Во-первых, надо сказать о списке изображений **ImageList**, который может централизованно снабжать изображениями и пиктограммами различные элементы приложения. Кроме того, в Delphi имеется такой объект, как **Application** — приложение. Этот объект управляет наиболее общими свойствами вашего проекта.

Все эти возможности и компоненты будут рассмотрены в данной главе. В общих чертах последовательность формирования списка действий и проектирования меню и инструментальных панелей сводится к следующим шагам:

1. Продумывается и составляется список действий, которые должны быть доступны будущему пользователю через разделы меню, инструментальные панели, кнопки и другие элементы управления.
2. Для тех нестандартных действий, которые должны быть доступны из быстрых кнопок инструментальной панели, готовится список пиктограмм на кнопках в компоненте **ImageList**.

3. На главную форму приложения переносится компонент диспетчеризации действий **ActionList**. Он связывается с **ImageList**. Формируется список стандартных и нестандартных действий.
4. Каждому действию задается набор характеристик: имя, надпись, пиктограмма, горячие клавиши, подсказки и др. Для нестандартных действий все эти характеристики вы записываете сами. Для стандартных действий они заносятся автоматически. Вам надо только перевести надписи и подсказки на русский язык и, может быть, исправить ссылки на не устраивающие вас стандартные изображения и комбинации горячих клавиш. А если у вас предусмотрена в приложении контекстная справка, то надо задать ссылки на соответствующие темы. Впрочем, в начале проектирования справки, конечно, еще нет. Так что соответствующие свойства вы можете задать позднее.
6. На форму переносится компонент **MainMenu** — главное меню, связывается с **ImageList**, в компоненте формируется меню, и в его разделах даются ссылки на действия, описанные в **ActionList**.
7. На форме создается инструментальная панель, или несколько панелей. Панели связываются с **ImageList**, а в их кнопках даются ссылки на действия, описанные в **ActionList**.
8. Записываются обработчики событий выполнения для всех нестандартных действий. Для этого в большинстве случаев предварительно надо перенести на форму компоненты диалогов, окна ввода и отображения информации, т.е. создать предварительный эскиз интерфейса. Иначе многие обработчики действий написать невозможно. Стандартные действия обрабатываются автоматически и для многих из них достаточно задать некоторые свойства обработки.

Вот в общих чертах последовательность операций при проектировании списка действий, меню и инструментальных панелей. В последующих разделах данной главы мы рассмотрим все эти операции подробнее. А после того, как все эти операции выполнены, можно приступать к доработке графического интерфейса пользователя, стремясь к тому, чтобы он был красивым и удобным. Но это уже тема следующей главы.

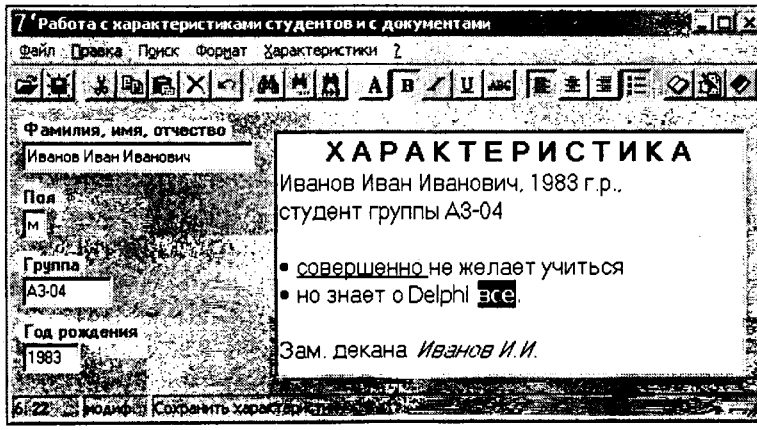


Рис. 4.1
Тестовое приложение

4.2 Список действий тестового примера

Рассматривать методику проектирования лучше, конечно, на каком-то примере. Мы будем проектировать приложение, вид которого показан на рис. 4.1. Приложение предназначено для просмотра информации и характеристик студентов (сотрудников, школьников) на основе классов, разработанных в разд. 3.5. Одновременно, чтобы можно был показать организацию некоторых операций с окном редактирования **RichEdit**, в приложении организован полноценный редактор на основе этого компонента. Так что в окно **RichEdit** можно загружать любые текстовые файлы и файлы *.rtf*, редактировать и форматировать тексты, сохранять их в файлах. Некоторые варианты форматирования показаны на рис. 3.5.

Это приложение содержит все атрибуты полноценного приложения Windows. Имеется полоса главного меню, контекстное меню, инструментальная панель, полоса состояния с подсказками внизу окна, справочная система. Не очень хорош интерфейс пользователя, но его совершенствованием мы займемся в следующей главе. А пока давайте составим список действий, которые надо реализовать в приложении.

Применительно к характеристикам реализуем три действия:

- загрузка из файла в окна редактирования
- занесение в окно **RichEdit** заготовки характеристики
- сохранение информации в файлах

Для произвольных текстовых документов в окне **RichEdit** реализуем:

- загрузку из файла
- сохранение в файле
- сохранение в файле с заданным именем
- печать

Реализуем еще два нестандартных действия:

- выбор атрибутов шрифта
- показ вспомогательного окна с информацией о программе

Помимо этого реализуем множество стандартных действий по форматированию шрифтов и абзацев, редактированию текста в **RichEdit**, поиску и контекстной замене фрагментов текста, показу справки. Впрочем, тем, кто работает с младшими версиями Delphi, придется ряд этих действий (а может быть и все) реализовывать тоже самим, или отказаться от них. Необходимые нам стандартные действия:

- вырезать выделенный фрагмент текста в буфер обмена
- копировать выделенный фрагмент текста в буфер обмена
- вставить текст из буфера обмена
- отменить последнюю операцию редактирования
- удалить выделенный текст
- найти заданный фрагмент текста
- продолжить поиск
- заменить заданный фрагмент заданным текстом
- установить полужирный шрифт
- установить курсив

- установить подчеркивание
- установить зачеркивание
- форматировать абзац списком
- выровнять влево
- выровнять по центру
- выровнять вправо
- показать справку
- завершить работу приложения

Как видим, список действий получился довольно внушительным, хотя приложение у нас достаточно простое.

Итак, первый шаг нами сделан — список действий составлен. Теперь перенесите на форму окна редактирования и расположите все примерно так, как показано на рис. 1.4. Далее напишите несколько операторов, которые нам в любом случае потребуются. Во-первых, сошлитесь на разработанный вами в разд. 3.5 модуль *MyClasses*, в котором реализован класс **TStudent**, с которым мы будем работать:

```
uses MyClasses;
```

Во-вторых, введите, как вы делали в разд. 3.5, глобальную переменную **Pers** класса **TStudent** и строковую переменную **FName**, в которой будет храниться имя открытого файла документа (см. разд. 3.4.2):

```
var Pers: TStudent;
    FName:string = 'Неизвестный';
```

И напишите обработчики событий **OnCreate** и **OnDestroy** формы, создающие объект **Pers**, связывающие его с окном **RichEdit1**, и уничтожающие объект:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Pers := TStudent.Create;
 Pers.SetDoc(RichEdit1);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 Pers.Free;
end;
```

Все эти операции, надеюсь, вам понятны, так как они рассматривались в гл. 3. А теперь мы можем приступить к грамотному проектирования нашего приложения.

4.3 Список изображений — компонент ImageList

Компонент **ImageList**, расположенный в палитре компонентов библиотеки Delphi на странице Win32, представляет собой набор изображений одинаковых размеров, на которые можно ссылаться по индексам, начинающимся с 0. Во многих компонентах — в меню, инструментальных панелях, списках и др. имеются свойства, представляющие собой ссылки на компонент **ImageList**. Этот компонент

позволяет организовать эффективное и экономное управления множеством пиктограмм и битовых матриц.

Изображения в компонент **ImageList** могут быть загружены в процессе проектирования с помощью редактора списков изображений. Окно редактора, представленное на рис. 4.2, вызывается двойным щелчком на компоненте **ImageList** или щелчком правой кнопки мыши и выбором команды контекстного меню ImageList Editor.

В окне редактора списков изображений вы можете добавить в списки изображения, пользуясь кнопкой Add, удалить изображение из списка кнопкой Delete, очистить весь список кнопкой Clear. При добавлении изображения в список открывается окно открытия файлов изображений, в котором вы можете выбрать интересующий вас файл. Только учтите, что размер всех изображений в списке должен быть одинаковым. Как правило, это размер, используемый для пиктограмм в меню, списках, кнопках.

С Delphi поставляется множество изображений для кнопок. Обычно они расположены в каталоге «...Program Files\Common Files\Borland Shared\Images». Выберите в этом каталоге какие-то изображения, которые нам понадобятся для некоторых нестандартных действий: открытия характеристики, занесения в окно **RichEdit** заготовки характеристики, сохранения характеристики в файле и задания атрибутов шрифта. На рис. 4.2 вы видите эти изображения с индексами от 0 до 3. Я выбрал для них файлы соответственно *BOOKOPEN.BMP*, *EDIT.BMP*, *BOOKSHUT.BMP* и *FONT.BMP*.

При добавлении в список изображений для кнопок надо иметь в виду, что они часто содержат не одно, а два и более изображений: для нажатого, отжатого, недоступного и др. состояний. В этих случаях при попытке добавить изображение в список **ImageList** задается вопрос: «Bitmap dimensions for ... are greater then imagelist dimensions. Separate into ... separate bitmaps?» (Размерность изображения ... больше размерности списка. Разделить на ... отдельных битовых матрицы?). Если вы ответите отрицательно, то все изображения уменьшатся в горизонтальном размере и лягут в список как одно изображение. Использовать его в дальнейшем будет невозможно. Поэтому на заданный вопрос надо отвечать положительно. Тогда загружаемая битовая матрица автоматически разделится на отдельные изображения, и потом вы можете удалить те из них, которые вам не нужны, кнопкой Delete.

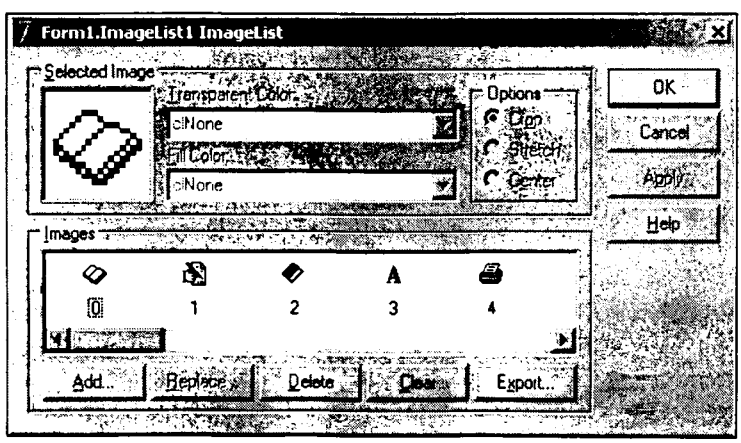


Рис. 4.2
Окно редактора списков изображений

Как видно из рис. 4.2, каждое загруженное в список изображение получает индекс. Именно на эти индексы впоследствии вы можете ссылаться в соответствующих свойствах разделов меню, списков, кнопок и т.д., когда вам надо загрузить в них то или иное изображение. Изменить последовательность изображений в списке вы можете, просто перетащив изображение мышью на новое место.

Мы занесли в **ImageList** немного изображений, так как остальные будут позднее добавлены в него автоматически стандартными действиями. Если в вашей версии Delphi не окажется каких-то стандартных действий, то вам придется позднее занести в список и другие изображения.

4.4 Диспетчер действий — компонент ActionList

Смысл диспетчеризации действий подробно рассматривался в разд. 4.1. В данном разделе мы обсудим, как это можно делать с помощью компонента **ActionList**, расположенного, начиная с Delphi 4, в библиотеке компонентов на странице Standard. В этот компонент заносится тот список действий, который, согласно рекомендациям разд. 4.1, вы составили в начале процесса проектирования. Перенесите этот компонент на форму, на которую в предыдущем разделе вы поместили **ActionList**. Сошлитесь в свойстве **Images** компонента **ActionList** на **ImageList**. Это полезно сделать в самом начале, так как описанные далее стандартные действия автоматически занесут в **ImageList** соответствующие им изображения.

Теперь сделайте на компоненте **ActionList** двойной щелчок. Вы попадаете в редактор действий (рис. 4.3), позволяющий вводить и упорядочивать действия. Только сначала это окно будет пустым.

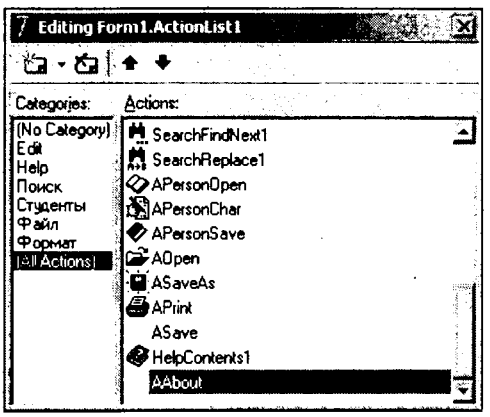


Рис. 4.3
Окно редактора действий

Щелчок правой кнопкой мыши или щелчок на маленькой кнопочке со стрелкой вниз правее первой быстрой кнопки окна редактора позволит вам выбрать одну из команд: New Action (новое действие) или New Standard Action (новое стандартное действие). Первая из них относится к вводу нового действия любого типа. По умолчанию эти действия будут получать имена **Action1**, **Action2** и т.д. Вторая ко-

манда открывает окно, в котором вы можете выбрать необходимое вам стандартное действие. Работу со стандартными действиями мы рассмотрим в разд. 4.5.

Каждое действие, которое вы внесли в список — это объект типа **TAction**. Выбрав в левом окне редактора ту или иную категорию или [AllActions] (все категории), а в правом — конкретное действие, вы можете увидеть в Инспекторе Объектов его свойства и события. Вы можете установить свойство **Name** (имя) — оно появится в правом окне редактора свойств и будет в дальнейшем фигурировать в заголовках обработчиков событий. При задании имени следует заботиться, чтобы оно было осмысленным, так как это облегчит вашу последующую работу. С другой стороны, желательно, чтобы имена не совпадали с какими-то именами функций, компонентов и т.п. Подобное совпадение в некоторых случаях может внести двусмысленность в код вашего приложения.

Итак, введите пока только четыре нестандартных действия и назовите их, например, так (эти имена используются в последующих кодах):

APersonOpen — загрузка характеристики из файла
APersonChar — занесение в окно **RichEdit** заготовки характеристики
APersonSave — сохранение информации в файлах
AFont — выбор атрибутов шрифта
AAbout — показ вспомогательного окна с информацией о программе

Для каждого из этих действий, кроме **AAbout**, задайте свойство **ImageIndex**, которое является индексом изображения, соответствующего данному действию в списке изображений **ImageList**. Этот индекс передастся в дальнейшем компонентам, связанным с данным действием — разделам меню, кнопкам. Эти же изображения появляются также в окне редактора действий (рис. 4.3).

Для каждого действия надо задать надпись **Caption**, которая далее будет появляться в инициаторах действия — кнопках, разделах меню и т.д. С этим свойством вы уже знакомы, но на некоторых его особенностях мы пока не останавливались. В надписях можно предусматривать использование клавиш ускоренного доступа, выделяя для этого один из символов надписи. Перед символом, который должен соответствовать клавише ускоренного доступа, ставится символ амперсанда «&». Этот символ не появляется в надписи, а следующий за ним символ оказывается подчеркнутым. Тогда пользователь может вместо щелчка на кнопке или на разделе меню нажать клавишу Alt совместно с клавишей выделенного символа. Правда, для раздела меню это будет действовать только в случае, если раздел в данный момент виден. Так что клавиши ускоренного доступа обязательно должны иметь все разделы меню, включая головные разделы. Весь этот аппарат придуман на случай, если у пользователя откажет мышь. И Microsoft требует непременного задания клавиш ускоренного доступа для всех действий.

Например, для действия **APersonOpen** надпись **Caption** можно задать равной «&Открыть характеристику ...». Тогда в соответствующем разделе меню, который будет связан с этим действием, надпись будет иметь вид «Открыть характеристику ...». И если пользователь нажмет клавиши Alt-О, то это будет эквивалентно выбору данного раздела меню.

Еще одно требование к надписям: если соответствующее действие вызывает диалоговое окно, то надпись должна заканчиваться многоточием. Поэтому в приведенном выше примере надписи для действия **APersonOpen** вы должны завершить надпись многоточием, так как действие открывает диалог выбора файла.

Для ряда действий можно задать горячие клавиши — свойство **ShortCut**. Их нажатие запускает действие, независимо от того, виден или нет соответствующий раздел меню или кнопка. Для нестандартных действий, связанных с характеристиками, можете придумать комбинации горячих клавиш сами. Впрочем, совершенно не обязательно задавать горячие клавиши для всех действий. А для традиционных действий горячие клавиши должны быть привычными пользователю по работе с другими программами Windows. Например, для выбора атрибутов шрифта (действие **AFont**) обычно используются горячие клавиши Ctrl-F.

Еще одно свойство действий — надписи на ярлычках подсказок и в строке состояний **Hint**. Это свойство есть также у кнопок, окон редактирования, да и практически у любых визуальных компонентов. Значение свойства **Hint** задается в виде строки текста, состоящей из двух частей, разделенных символом вертикальной черты '|'. Первая часть, обычно очень краткая, предназначена для отображения в ярлычке; вторая более развернутая подсказка предназначена для отображения в панели состояния или ином заданном месте экрана. Например, для действия **APersonChar** в свойстве **Hint** может быть задан текст: «Начать характеристику|Занести в окно начало характеристики». Как частный случай, в свойстве **Hint** может быть задана только первая часть подсказки без символа '|'.

Для того чтобы первая часть подсказки появлялась во всплывающем ярлычке, когда пользователь задержит курсор мыши над данным компонентом, например, над кнопкой, надо сделать следующее:

1. Указать тексты свойства **Hint** для всех действий и тех компонентов, для которых вы хотите обеспечить ярлычок подсказки.
2. Установить свойства **ShowHint** (показать подсказку) соответствующих компонентов (не действий, а компонентов, которыми мы займемся позже) в **true** или установить в **true** свойство **ParentShowHint** (отобразить свойство **ShowHint** родителя) и установить в **true** свойство **ShowHint** контейнера, содержащего данные компоненты.

При **ShowHint**, установленном в **true**, окно подсказки будет всплывать, даже если компонент в данный момент недоступен (свойство **Enabled = false**).

Если вы не задали значение свойства компонента **Hint**, но установили в **true** свойство **ShowHint** или установили в **true** свойство **ParentShowHint**, а в родительском компоненте **ShowHint = true**, то в окне подсказки будет отображаться текст **Hint** из родительского компонента.

Если в приложении предусмотрен файл справки, а у нас он будет предусмотрен, то для действий можно также указать свойства, связывающие его с конкретной темой справки **HelpContext**, **HelpKeyword** и **HelpType**. Но об этих свойствах будет рассказано в разд. 4.10.4.

Каждое действие имеет свойство **Category** — категория. Оно не имеет отношения к выполнению приложения. Задание категории просто позволяет в процессе проектирования сгруппировать действия по их назначению. Вы можете для каждого действия выбрать категорию из выпадающего списка, или написать имя новой категории и отнести к ней какие-то действия. Например, на рис. 4.3 видны введенные пользователем категории Поиск, Студенты, Файл, Формат. Обычно целесообразно называть категории по заголовкам будущих меню или инструментальных панелей. Можно выделить в правой панели окна на рис. 4.3 сразу группу действий, например, относящихся к характеристикам, и написать для них в окне Инспектора Объектов название категории, например, «Студенты». Тогда в левой панели окна

рис. 4.3 появится введенная вами категория, а если вы ее выделите, то на правой панели увидите относящиеся к ней действия.

На странице событий Инспектора Объектов для каждого действия определено событие **OnExecute**. Оно возникает в момент, когда пользователь инициализировал действие, например, щелкнув на компоненте (разделе меню, кнопке), связанном с данным действием. Обработчик этого события должен содержать процедуру, реализующую данное действие.

Для действия **AFont**, позволяющего пользователю задать атрибуты шрифта, надо, очевидно, перенести на форму диалог **FontDialog** (см. разд. 3.2.3). Тогда обработчик события **OnExecute** этого действия следует записать так:

```
procedure TForm1.AFontExecute(Sender: TObject);
begin
 FontDialog1.Font.Assign(RichEdit1.SelAttributes);
 if FontDialog1.Execute
  then RichEdit1.SelAttributes.Assign(FontDialog1.Font);
end;
```

Первый оператор заносит в диалог текущие атрибуты шрифта выделенного текста окна **RichEdit1**. А следующий оператор вызывает диалог и заносит в окно выбранный пользователем шрифт. Все это было рассмотрено в разд. 3.2.3 и 3.2.4. Обратите, кстати, внимание на имя обработчика — **AFontExecute**. Думаю, что оно понятнее, чем те бесконечные «Button...Click», которые были у вас ранее. А ведь в нашем приложении будет свыше двадцати кнопок, и еще больше разделов меню. И вы, почти наверняка, запутались бы в их обработчиках, если бы не более понятные имена, получающиеся за счет использования действий.

Обработчики событий **OnExecute** действий **APersonSave**, **APersonOpen** и **APersonChar** могут иметь следующий вид:

```
procedure TForm1.APersonSaveExecute(Sender: TObject);
begin
 with Pers do
 begin
  Name := Edit1.Text;
  if Edit2.Text <> '' then Sex := Edit2.Text[1];
  Dep1 := Edit3.Text;
  Year := StrToInt(Edit4.Text);
  Comment := RichEdit1.Text;
  SaveToFile;
 end;
end;

procedure TForm1.APersonOpenExecute(Sender: TObject);
begin
 if OpenDialog1.Execute
 then with Pers do
 begin
  LoadFromFile(OpenDialog1.FileName);
  Edit1.Text := Name;
  Edit2.Text :=  Sex;
  Edit3.Text := Dep1;
  Edit4.Text := IntToStr(Year);
  GetDoc(RichEdit1);
 end;
end;
```

```
procedure TForm1.APersonCharExecute(Sender: TObject);
var S: string;
begin
 with RichEdit1 do
 begin
  Clear;
  Paragraph.Alignment := taCenter;
  SelAttributes.Style := SelAttributes.Style + [fsBold];
  SelAttributes.Size := 14;
  Lines.Add('Х А Р А К Т Е Р И С Т И К А');
  Paragraph.Alignment := taLeftJustify;
  SelAttributes.Style := SelAttributes.Style - [fsBold];
  SelAttributes.Size := 12;
  S := Edit1.Text + ', ' + Edit4.Text + ' г.р., ';
  if Edit2.Text[1] = 'м'
   then S := S + 'студент'
   else S := S + 'студентка';
  S := S + ' группы ' + Edit3.Text;
  Lines.Add(S);
  SetFocus;
 end;
end;
```

Первые два из них комментировать не приходится, так как аналогичные обработчики вы писали в разд. 3.5.2 и 3.5.4. А последний обработчик отличается от прежних только тем, что задается форматирование строк окна **RichEdit1**. Первая строка выравнивается по центру и пишется полужирным шрифтом размера 14. А следующая выравнивается по левому краю и пишется обычным шрифтом (см. пример на рис. 4.1).

Мы написали обработчики введенных нами нестандартных действий, кроме действия **AAbout**. Но этим действием мы займемся много позже — в разд. 4.13.1.

4.5 Инструментальная панель ToolBar

Формирование списка действий в компоненте **ActionList** еще далеко не закончено. Но давайте прервемся, и сделаем инструментальную панель с кнопками, инициирующими хотя бы те действия, которые уже реализованы. Иначе нам придется слишком долго ждать, пока мы сможем выполнить наше приложение хотя бы в урезанном виде.

Инструментальную панель удобнее всего строить на основе компонента **ToolBar**, расположенного в библиотеке на странице Win32. Поместите этот компонент **ToolBar** на форму. Он займет ее верхнюю часть. Так что если у вас расположены там другие компоненты, сдвиньте их так, чтобы они были видны.

Первым делом свяжите панель свойством **Images** со списком изображений **ImageList1**. А теперь щелкните на **ToolBar** правой кнопкой мыши и выберите из всплывшего меню команду New Button. На форме появится кнопка — объект типа **TToolButton**. На кнопку занесется одно из изображений, хранящихся в **ImageList1**.

Чтобы связать кнопку с действием, надо выбрать его из выпадающего списка свойства кнопки **Action**. Как только вы выбрали действие, вы увидите, что в кнопку перенесся требуемый индекс изображения **ImageIndex**, и установятся значения

таких свойств, как **Hint** и **ShortCut**. На странице событий в окне Инспектора Объектов вы увидите, что в событии **OnClick** появится ссылка на введенный вами ранее обработчик действия. Так что никакой дополнительной настройки кнопки инструментальной панели не требуют: просто создаете новую кнопку и задаете ее свойство **Action**.

Обычно кнопки в инструментальной панели группируются по их назначению, и между группами вставляются разделители. Чтобы вставить разделитель, надо в меню, всплывающем при щелчке на панели **ToolBar** правой кнопкой мыши, выбрать раздел New Separator.

Свойство **Wrap** кнопки, установленное в **true**, приводит к тому, что после этой кнопки ряд кнопок на панели прерывается, и следующие кнопки размещаются в следующем ряду.

Из общих свойств компонента **ToolBar** следует отметить **ButtonHeight** и **ButtonWidth** — высота и ширина кнопок в пикселах. Установка свойств **AutoSize** и **Wrapable** в **true** обеспечивает автоматическую адаптацию размеров кнопок к содержащимся в них изображениям и автоматический перенос кнопок в следующий ряд панели, если они не помещаются в предыдущем. Такой перенос осуществляется и во время проектирования, и во время выполнения при изменении пользователем размеров окна, а следовательно, и размеров панели.

Свойства, определяющие вид панели **ToolBar**: **BorderWidth** — *ширина бордюра*, **EdgeInner** и **EdgeOuter** — стиль изображения внутренней и внешней части панели (утопленный или выступающий), **EdgeBorders** — определяет изображение отдельных сторон панели (левой, правой, верхней, нижней).

Не забудьте установить в **true** свойство панели **ShowHint**. Это, как пояснялось в разд. 4.4, обеспечит появление во всех кнопках всплывающих ярлычков с текстом, заданным свойством **Hint**. Связано это с тем, что во всех кнопках по умолчанию свойство **ParentShowHint** установлено в **true**.

Введите на панель кнопки, соответствующие всем созданным вами действиям, выполните приложение и убедитесь, что все работает правильно.

4.6 Работа со стандартными действиями

Теперь продолжим составление списка действий в компоненте **ActionList**. Откройте опять двойным щелчком окно редактора действий этого компонента, щелкните в нем правой кнопкой мыши и выберите раздел New Standard Action — новое стандартное действие. Перед вами откроется окно, показанное на рис. 4.4. Впрочем, такой вид имеет окно в Delphi 6 и 7. В Delphi 7 различных стандартных действий 66. В Delphi 5 их намного меньше — 26, и не все те, которые нам хотелось бы включить в приложение, имеются в этой версии. Но не огорчайтесь — в разд. 4.7 мы увидим, как можно обойтись и без стандартных действий. Хотя, конечно, с ними работать проще.

В окне рис. 4.4 вы можете выбрать из списка необходимое вам стандартное действие (или сразу несколько действий). Действия в списке сгруппированы по категориям. Стандартные действия охватывают операции редактирования текстов (категория Edit), форматирования текстов (категория Format), поиска в текстах (категория Search), работу со справками (категория Help), с файлами (категория File), с многостраничными панелями (категория Tab), списками (категория List), стан-

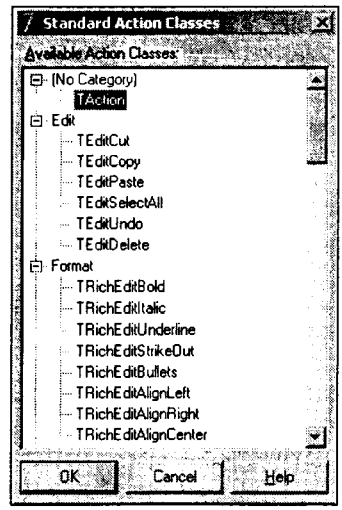


Рис. 4.4
Окно выбора стандартных действий в Delphi 7

дартными диалогами (категория Dialog), с Интернет (категория Internet), наборами данных (категория DataSet) и некоторые другие.

Нам желательно включить в список следующие стандартные действия:

Категория	Тип действия	Команда	Примечание
Edit Правка	**TEditCut**	Вырезать в буфер обмена	
	TEditCopy	Копировать в буфер обмена	
	TEditPaste	Вставить из буфера обмена	
	TEditUndo	Отменить операцию редактирования	
	TEditDelete	Удалить выделенный текст	
Format Формат	**TRichEditBold**	Полужирный шрифт	Начиная с Delphi 6
	TRichEditItalic	Курсив	Начиная с Delphi 6
	TRichEditUnderline	Подчеркнутый шрифт	Начиная с Delphi 6
	TRichEditStrikeOut	Зачеркнутый шрифт	Начиная с Delphi 6
	TRichEditBullets	Список	Начиная с Delphi 6
	TRichEditAlignLeft	Выровнять влево	Начиная с Delphi 6
	TRichEditAlignRight	Выровнять вправо	Начиная с Delphi 6
	TRichEditAlignCenter	Выровнять по центру	Начиная с Delphi 6
File Файл	**TFileExit**	Выход	
	TFileOpen	Открыть файл ...	Начиная с Delphi 6
	TFileSaveAs	Сохранить как ...	Начиная с Delphi 6

Категория	Тип действия	Команда	Примечание
Dialogs Диалоги	**TPrintDlg**	Печать ...	Начиная с Delphi 6, только для пиктограммы
Search Поиск	**TSearchFind**	Найти текст ...	Начиная с Delphi 6
	TSearchFindNext	Продолжить поиск ...	Начиная с Delphi 6
	TSearchReplace	Найти и заменить ...	Начиная с Delphi 6

Многие из указанных стандартных действий имеются, только начиная с Delphi 6. Так что тем, кто работает с более младшими версиями, придется ограничиться тем, что у них есть. А в разд. 4.7 вы увидите, как обойти эти трудности.

Выделив в окне редактора действий (рис. 4.4) любое из введенных вами стандартных действий, вы увидите, что для этих действий уже заполнены свойства **Caption**, **Hint**, **ShortCut**. Вам остается только перевести тексты на русский язык и изменить, если хотите, горячие клавиши. Заполнены также свойства **ImageIndex**. Если вы вернетесь в редактор компонента **ImageList1**, то увидите, что в списке автоматически появились стандартные изображения новых действий. Так что о пиктограммах вам тоже заботиться не придется, если только вы не надумаете их изменить. Но самое главное отличие стандартных объектов действий от нестандартных заключается в том, что для стандартных действий не надо писать обработчики событий **OnExecute**. Все операции, необходимые для выполнения стандартных действий уже заложены в их объекты. Они не только не требуют обработчиков событий **OnExecute**, но могут реализоваться через горячие клавиши даже без инициаторов действий — разделов меню, кнопок и т.п.

Введите в инструментальную панель кнопки, соответствующие действиям категорий Edit (Правка) и Format (Формат). Выполните ваше приложение. Пока в фокусе находится не окно **RichEdit**, кнопки форматирования будут недоступны, так как в окнах **Edit** форматирование невозможно. Если вы перейдете в окно **RichEdit**, то кнопки форматирования станут доступны и начнут работать. А все кнопки правки будут сначала недоступны. Это оправдано, поскольку никакой текст не выделен, так что нечего копировать или вырезать, и никакого редактирования не было, так что нечего отменять. Может оказаться доступной только кнопка Вставить (Paste, если вы не переводили подсказки на русский язык). Доступной она будет, если в буфере обмена в данный момент находится текст. Но нажмите, например, клавишу компьютера Print Screen, занеся в буфер обмена изображение экрана, и кнопка Вставить станет недоступной. Чувствуете, какое умное поведение. Если вы выделите текст в одном из окон редактирования, то, пока это окно в фокусе, будут доступны кнопки Копировать (Copy), Вырезать (Cut), Удалить (Delete). Если вы провели какую-то операцию редактирования, станет доступна кнопка Отменить (Undo). Но стоит вам переместить курсор в другое окно редактирования, где нет выделения и не было редактирования, как эти кнопки станут недоступны.

Таким образом, в стандартные действия заложено достаточно интеллектуальное поведение. Кнопки относятся всегда к тому элементу, который находится в фокусе (в данном примере — к находящемуся в фокусе окну редактирования), и доступность их в каждый момент автоматически определяется возможностью выполнения того или иного действия. Представляете, сколько кода вам надо было бы написать, чтобы реализовать подобное поведение, включая тестирование формата, занесенного в буфер обмена, определение окна, находящегося в фокусе, и определение

выделения в нем? Вы не написали ни одного оператора, а получили такой эффект благодаря возможностям, заложенным в стандартных действиях.

Но учтите, что почти весь «интеллект» стандартного действия может исчезнуть, если вы введете для такого действия свой обработчик события **OnExecute**. Тогда вам придется все реализовывать самому. Так что для многих стандартных действий не следует писать обработчиков событий **OnExecute**.

Теперь рассмотрим стандартные действия поиска. В них тоже практически ничего не надо настраивать, кроме перевода текстов **Caption** и **Hint** на русский язык и задания одной ссылки: в свойстве **SearchFind** действия **SearchFindNext1** надо сослаться на действие **SearchFind1**, начинающее поиск. И еще одну настройку надо произвести в окне **RichEdit1**: задать значение **false** в свойстве **HideSelection**. При заданном по умолчанию значении **true** свойства **HideSelection** выделение в окне **RichEdit1** будет невидимо, если это окно находится не в фокусе. Значит, когда пользователь будет работать с окнами поиска и замены, он не увидит в окне редактирования найденный текст. Задание **HideSelection = false** снимает эту проблему.

Введите на инструментальную панель кнопки действий поиска, продолжения поиска и замены. Выполните приложение. Вы сможете убедиться, что поиск и замена работают. Причем даже с большими возможностями, чем обеспечивали коды, рассмотренные в разд. 3.2.5. В частности, вы сможете проводить поиск текстов в прямом и обратном направлениях. И все это не потребовало от вас записи ни одного оператора Object Pascal.

Таким образом, стандартные действия, связанные с диалогами Windows, не требуют явного ввода в приложение компонентов-диалогов и явного их вызова. Впрочем, объект соответствующего диалога вводится в приложение автоматически. К этому объекту вы можете получить доступ через свойство **Dialog** соответствующего действия. Так что все настройки диалогов, которые были рассмотрены в разд. 3.4.1 и 3.2.5, вы можете провести и для стандартных действий.

Стандартные действия, связанные с вызовом диалогов Windows, имеют события **BeforeExecute** (наступает перед вызовом диалога), **OnAccept** (наступает, если пользователь в диалоге произвел выбор и нажал OK) и **OnCancel** (наступает, если пользователь в диалоге не произвел выбор и нажал кнопку Отказ или клавишу Esc).

Например, в обработчике события **BeforeExecute** действия **SearchFind1** вы можете записать оператор, обеспечивающий задание выделенного в окне текста как начального значения искомого текста:

```
SearchFind1.Dialog.FindText := RichEdit1.SelText;
```

В обработчике события **OnAccept** можно прочитать выбор пользователя. Например, для действия **FileOpen1**, настроив соответствующим образом объект в свойстве **Dialog**, в обработчике события **OnAccept** запишите код:

```
FName := FileOpen1.Dialog.FileName;
RichEdit1.Lines.LoadFromFile(FName);
```

А в обработчике события **OnAccept** действия **FileSaveAs1** запишите код:

```
FName := FileSaveAs1.Dialog.FileName;
RichEdit1.Lines.SaveToFile(FName);
```

В обработчике события **BeforeExecute** того же действия **FileSaveAs1** вы можете записать оператор, обеспечивающий задание начального значения файла:

```
FileSaveAs1.Dialog.FileName := FName;
```

Смысл всех этих операторов вам должен быть понятен, так как подобные операторы рассматривались в разд. 3.4.2. Только там выполнялись операции с компонентами диалогов, а не со стандартными действиями.

Введя в приложение все эти обработчики событий, добавьте соответствующие кнопки в инструментальную панель, выполните приложение, и убедитесь, что все работает правильно.

Из действий, связанных с файлами и намеченных нами в разд. 4.2, осталось реализовать сохранение в файле. Для этого введите в список действий **ActionList1** новое нестандартное действие. Обработчик его события **OnExecute** может иметь вид:

```
if (FName = 'Неизвестный')
 // вызов процедуры Сохранить как
 then FileSaveAs1Accept(Sender)
 else RichEdit1.Lines.SaveToFile(FName);
```

Если имя файла неизвестно, то вызывается описанный выше обработчик события **OnAccept** действия **FileSaveAs1**. В противном случае текст сохраняется в файле с именем, хранящемся в **FName**.

Теперь коротко о последнем из задуманных в приложении действий — печати документа из окна **RichEdit**. Окна класса **TRichEdit** имеют метод **Print**:

```
procedure Print(const Caption: string);
```

В этот метод передается единственный параметр типа строки, назначение которого заключается только в том, что при просмотре в Windows очереди печатаемых заданий принтера эта строка появляется как имя задания. Например, оператор

```
RichEdit1.Print('Printing of RichEdit1');
```

обеспечивает печать текста компонента **RichEdit1**, причем задание на печать получает имя «Printing of RichEdit1».

Печать воспроизводит все заданные особенности форматирования. Перенос строк и разбиение текста на страницы производится автоматически. Длина строк никак не связана с размерами компонента **RichEdit**, содержащего этот текст.

Ограничимся простейшим вариантом реализации действия печати. Введите в список действий **ActionList1** новое нестандартное действие, назовите его **APrint**. Сошлитесь в нем на пиктограмму принтера, автоматически внесенную в список изображений стандартным действием **TPrintDlg**. Само действие **TPrintDlg** можете удалить — мы его вводили только для того, чтобы получить пиктограмму. А обработчик события **OnExecute** действия **APrint** может состоять из одного оператора:

```
RichEdit1.Print(FName);
```

Внесите на инструментальную панель кнопку, соответствующую этому действию, выполните приложение, проверьте ег работу в самых различных режимах.

4.7 Дополнительные сведения о действиях и инструментальных панелях

Иногда надо, чтобы группа действий или кнопок работала согласованно как радиокнопки: если одна включается, то остальные выключаются. Примером служат рассмотренные в разд. 4.5 и 4.6 действия выравнивания абзаца влево, по центру и вправо. Очевидно, что в каждый момент один и только из этих видов выравнивания должен быть доступен. И включение одного из этих действий должно выключать два других.

Давайте на этом примере посмотрим, как можно реализовать подобные возможности. В разд. 4.6 мы просто использовали стандартные действия, в которых все это уже заложено. Но, во-первых, стандартные действия могли использовать только те, кто работает с версией старше, чем Delphi 5. А во-вторых, бывает немало аналогичных ситуаций, для которых стандартные действия не предусмотрены. И надо уметь решать подобные задачи без помощи стандартных действий.

Введите в компоненте **ActionList1** три нестандартные действия и назовите их **ALeft**, **ACenter**, **ARight**. Если ранее вводили соответствующие стандартные действия, укажите в новых действиях те же индексы изображений. Если соответствующих изображений у вас нет, введите в **ImageList1** какие-то подходящие изображения из числа поставляемых с Delphi.

Теперь обратите внимание на свойство **GroupIndex**, имеющееся у всех классов действий. По умолчанию это свойство равно 0. Но если вы зададите у группы действий какое-то одинаковое положительное значение **GroupIndex**, то эти действия начнут работать как согласованная группа. У каждого действия имеется свойство **Checked** — выбрано. Это свойство показывает, будет ли выбран соответствующий исполнительный элемент — кнопка или раздел меню (меню мы рассмотрим позднее). Кнопка, связанная с выбранным действием, представляется пользователю нажатой. Причем для группы действий, объединенной одинаковыми значениями свойства **GroupIndex**, значение **Checked** = **true** может быть задано только для одного действия. Как только в одном действии из группы вы зададите **Checked** = **true**, в остальных действиях установится **Checked** = **false**.

В нашем примере задайте для действий **ALeft**, **ACenter** и **ARight** значение **GroupIndex** = 1, а в действии **ALeft** установите **Checked** = **true**. В обработчики событий **OnExecute** этих действий внесите соответствующие операторы (см. разд. 3.2.4). Для **ALeft**:

```
RichEdit1.Paragraph.Alignment := taLeftJustify;
```

для **ACenter**:

```
RichEdit1.Paragraph.Alignment := taCenter;
```

для **ARight**:

```
RichEdit1.Paragraph.Alignment := taRightJustify;
```

Теперь сформируйте инструментальную панель быстрых кнопок, соответствующих введенным вами действиям. Вероятно, чтобы не портить ранее созданную панель, добавьте для наших экспериментов новый компонент **ToolBar**, и на нем формируйте соответствующие кнопки. Задайте им осмысленные имена **TBLeft**, **TBCenter**, **TBRight**. Установите для введенных вами кнопок свойство **Grouped** = **true**.

Это укажет на то, что кнопки образуют взаимодействующую группу. Кнопки, объединенные в группу, должны располагаться одна за другой. В свойстве **Style** кнопок выберите из выпадающего списка стиль **tbsCheck**. Этот стиль означает, что после щелчка пользователя на не нажатой кнопке она остается нажатой.

Можете выполнить приложение и убедиться, что введенные вами кнопки управляют выравниванием в окне **RichEdit1**. Но хотелось бы еще добавить в приложение обратную связь из окна **RichEdit1**: чтобы кнопки отображали выравнивание в том абзаце, в котором в данный момент находится курсор.

Положение курсора может измениться в результате таких действий пользователя, как нажатие или отпускание клавиши или кнопки мыши. Следовательно, именно в обработчики таких событий надо вставить операторы, изменяющие состояние быстрых кнопок инструментальной панели. При нажатии и отпускании клавиши возникают соответственно события **OnKeyDown** и **OnKeyUp**. При нажатии и отпускании кнопки мыши возникают соответственно события **OnMouseDown** и **OnMouseUp**. Так что вам надо поступить следующим образом. Выделите компонент **RichEdit1**, и в обработчик его события **OnKeyDown** запишите код:

```
case RichEdit1.Paragraph.Alignment of
 taLeftJustify: TBLeft.Down := true;
 taCenter: TBCenter.Down := true;
 taRightJustify: TBRight.Down := true;
end;
```

Сошлитесь на этот же обработчик в событии **OnKeyUp**. Далее введите аналогичный код в обработчик события **OnMouseDown**, и сошлитесь на тот же обработчик в событии **OnMouseUp**.

Выполните приложение. Теперь кнопки начнут отображать характер выравнивания в том абзаце, в котором находится курсор. Для полноты картины аналогичные коды можно ввести в обработчики действий, загружающих в окно файл или характеристику, чтобы кнопки сразу отображали выравнивание в загруженных текстах.

Осталось реализовать единственное, что отличает введенные нами действия от стандартных: желательно, чтобы эти действия были недоступны, пока пользователь не переключит фокус в окно **RichEdit1**, поскольку в любых других окнах выравнивание невозможно. При получении компонентом фокуса генерируется его событие **OnEnter**, а при потере фокуса возникает событие **OnExit**. Доступность действий, как и любых других компонентов, определяется их свойством **Enabled**. Так что вам надо для действий **ALeft**, **ACenter** и **ARight** задать значение **Enabled = false**, а обработчики событий **OnEnter** и **OnExit** окна **RichEdit1** оформить следующим образом:

```
procedure TForm1.RichEdit1Enter(Sender: TObject);
begin
 ALeft.Enabled := true;
 ACenter.Enabled := true;
 ARight.Enabled := true;
end;

procedure TForm1.RichEdit1Exit(Sender: TObject);
begin
 ALeft.Enabled := false;
 ACenter.Enabled := false;
 ARight.Enabled := false;
end;
```

Вот теперь введенные вами действия неотличимы от стандартных.

Рассмотрим возможность реализации некоторых других действий, которые ранее вы реализовывали как стандартные. Цель все та же — научиться создавать свои собственные действия, если соответствующих стандартных действий нет, или они вас не устраивают. Начнем с действия, форматирующего абзац как список. Введите в компоненте **ActionList1** новое нестандартное действие и назовите его **ABullets**. Это действие должно работать как переключатель: при каждом воздействии пользователя оно должно включать или выключать форматирование списка. Начиная с Delphi 6, в действиях имеется свойство **AutoCheck**, которое можно установить в **true**. Тогда при каждом вызове этого действия его свойство **Checked** будет попеременно переключаться в **true** и **false**. Так что реализация требуемого поведения введенного нами действия **ABullets** сводится к тому, что надо установить в **true** значение его свойства **AutoCheck** и написать следующий оператор в обработчик его события **OnExecute**:

```
if ABullets.Checked
  then RichEdit1.Paragraph.Numbering := nsBullet
  else RichEdit1.Paragraph.Numbering := nsNone;
```

В зависимости от состояния **ABullets** этот оператор включает или выключает форматирование списка (см. разд. 3.2.4).

Если вы работаете с версией младше Delphi 6, то свойство **AutoCheck** у действий отсутствует. Тогда вам надо перед приведенным выше оператором вставить:

```
ABullets.Checked := not ABullets.Checked;
```

Тем самым вы программно осуществляете переключение состояния действия при каждом его вызове.

Для осуществления обратной связи — влияния формата абзаца на поведение кнопки, связанной с **ABullets**, достаточно в рассмотренные ранее обработчики событий **OnKeyDown**, **OnKeyUp**, **OnMouseDown** и **OnMouseUp** окна **RichEdit1** вставить оператор:

```
ABullets.Checked := (RichEdit1.Paragraph.Numbering = nsBullet);
```

Изменение доступности действия ничем не отличается от рассмотренного для предыдущих действий. Введите кнопку, соответствующую действию **ABullets**, в инструментальную панель и протестируйте ее во время выполнения приложения.

Точно так же вы можете реализовать и иные действия, связанные с форматированием. Например, вы можете создать нестандартное действие **ABold**, управляющее полужирным шрифтом. Отличие от **ABullets** будет только в операторах обработчика события **OnExecute**:

```
if ABold.Checked
 then RichEdit1.SelAttributes.Style :=
                      RichEdit1.SelAttributes.Style + [fsBold]
 else RichEdit1.SelAttributes.Style :=
                      RichEdit1.SelAttributes.Style - [fsBold];
```

и в операторе, обеспечивающем связь из окна **RichEdit1**:

```
ABold.Checked := fsBold in RichEdit1.SelAttributes.Style;
```

На этом мы завершим рассмотрение действий и инструментальной панели. Вы легко сможете аналогичным образом реализовать остальные нестандартные действия форматирования. А осуществить действия поиска и замены при отсутствии

в вашей версии Delphi соответствующих стандартных действий вы можете с помощью приемов, рассмотренных в разд. 3.2.5.

4.8 Меню

В Delphi имеется два компонента, представляющие меню: **MainMenu** — главное меню, и **PopupMenu** — всплывающее меню. Оба компонента расположены на странице Standard. Эти компоненты имеют много общего. Начнем рассмотрение с компонента **MainMenu**. Перенесите его на форму тестового приложения, рассмотренного в предыдущих разделах. Это невизуальный компонент, т.е. место его размещения на форме в процессе проектирования не имеет никакого значения для пользователя — он все равно увидит не сам компонент, а только меню, сгенерированное им.

Обычно на форму помещается один компонент **MainMenu**. В этом случае его имя автоматически заносится в свойство формы **Menu**. Но можно поместить на форму и несколько компонентов **MainMenu** с разными наборами разделов, соответствующими различным режимам работы приложения. В этом случае во время проектирования свойству **Menu** формы присваивается ссылка на один из этих компонентов. А в процессе выполнения в нужные моменты это свойство можно изменять, меняя соответственно состав главного меню приложения.

Свяжите компонент **MainMenu1** его свойством **Images** со списком изображений **ImageList1**. Это необходимо, чтобы в меню отображались пиктограммы, связанные с действиями.

Основное свойство компонента **MainMenu** — **Items**. Его заполнение производится с помощью специального конструктора меню, вызываемого двойным щелчком на компоненте **MainMenu** или нажатием кнопки с многоточием рядом со свойством **Items** в окне Инспектора Объектов. В результате откроется окно, вид которого представлен на рис. 4.5. В этом окне вы можете спроектировать все меню. На рис. 4.6 показаны в работе меню, которые надо создать для нашего тестового приложения.

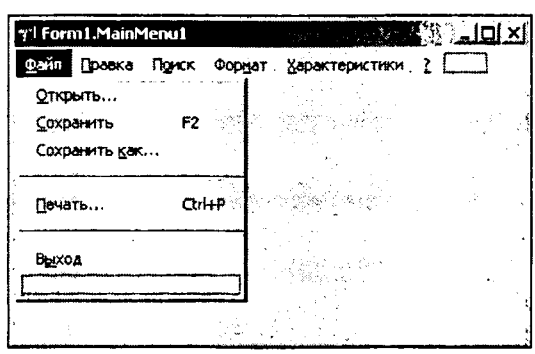


Рис. 4.5
Окно конструктора меню

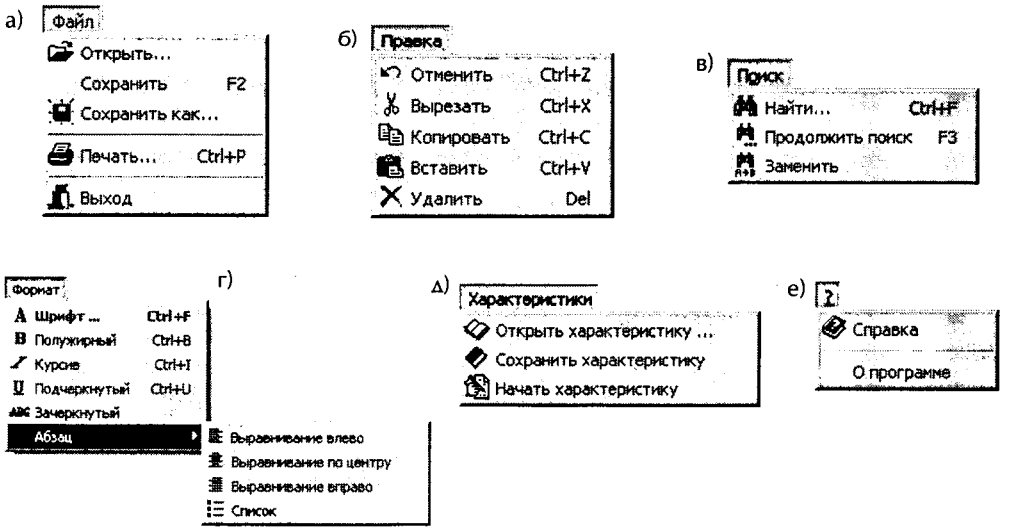


Рис. 4.6
Меню тестового приложения во время выполнения

В первый момент после открытия конструктора меню вы увидите пустое окно с рамкой из точек. Это место размещения головного раздела первого меню. В окне Инспектора Объектов вы можете увидеть свойства раздела. Это такие знакомые вам свойства, как **Caption**, **Hint**, **ShortCut**, **ImageIndex**, **Name**. Для головного раздела каждого меню надо задавать только надпись **Caption**, так как обычно с головным разделом не связано какое-то действие. При задании надписи любого раздела надо выделять амперсандом символ быстрого доступа (см. подробнее в разд. 4.4). Полезно также изменить на что-то осмысленное имя **Name** объекта, соответствующего разделу меню. Иначе вы скоро запутаетесь в ничего не говорящих именах типа **N21**. Куда понятнее имя типа **MFile**.

После того как вы ввели надпись головного раздела первого меню «&Файл», щелкните на рамке в окне конструктора меню. Снизу и справа появятся новые рамки из точек. В нижней вы можете вводить очередной раздел данного меню. В правой вы можете ввести головной раздел следующего меню.

Выделите нижнюю рамку. Если ориентироваться на меню, приведенное на рис. 4.5 и 4.6 а, то первый раздел первого меню соответствует команде открытия файла. Но у вас есть уже соответствующее действие **AOpen**. Так что единственное, что вам надо сделать — выбрать в окне Инспектора Объектов из выпадающего списка около свойства **Action** нужное действие. Вы увидите, что многие свойства объекта раздела сразу заполнятся значениями, заданными в объекте действия. Так что вам не потребуется задавать надпись, пиктограмму, горячие клавиши. Все это приходится задавать только в случаях, когда раздел не связан с каким-то действием. В этих случаях приходится также писать обработчик события **OnClick** раздела, чтобы определить операции, выполняемые при выборе данного раздела. В нашем приложении это не придется делать, так как в событие **OnClick** автоматически занесется ссылка на обработчик соответствующего действия.

Итак, сформируйте все меню, показанные на рис. 4.6. Единственный раздел, для которого мы пока не реализовывали действие — это раздел О программе (рис. 4.6 е). Но этим мы займемся позднее, в разд. 4.13.1.

Обычно разделы меню группируются по своему назначению и группы отделяются друг от друга разделителями, которые вы можете видеть на рис. 4.6. а (перед разделами Печать и Выход) и рис. 4.6 е (перед разделом О программе). Чтобы это реализовать, надо в качестве значения **Caption** очередного раздела ввести символ минус «-». Тогда вместо раздела в меню появится разделитель.

Иногда для какого-то раздела надо создать подменю (см. раздел Абзац на рис. 4.6 г). Для этого надо щелкнуть на разделе правой кнопкой мыши и выбрать из всплывшего контекстного меню команду Create Submenu.

Если при работе в конструкторе меню вы ошибочно ввели раздел не на том месте, где хотели, вы можете отбуксировать его мышью туда, куда вам надо. Другой путь ввода нового раздела в нужное место — использование контекстного меню, всплывающего при щелчке правой кнопкой мыши. Если вы предварительно выделите какой-то раздел меню и выберите из контекстного меню команду Insert, то рамка нового раздела вставится перед ранее выделенным.

Для одного раздела каждого меню или подменю вы можете установить в **true** его свойство **Default**. Тогда этот раздел выделится жирным шрифтом (см. раздел Шрифт на рис. 4.6 г) и станет разделом по умолчанию. Это значит, что он будет выполняться при двойном щелчке пользователя на родительском разделе.

Если вы реализовали в своем приложении главное меню, выполните приложение, чтобы убедиться, что все работает так, как было задумано.

Теперь рассмотрим контекстные всплывающие меню. Контекстное меню привязано к конкретному компоненту. Оно всплывает, если во время, когда данный компонент в фокусе, пользователь щелкнет правой кнопкой мыши. Обычно в контекстное меню включают те команды главного меню, которые в первую очередь могут потребоваться при работе с данным компонентом.

В нашем приложении, вероятно, полезно создать контекстное меню для окна редактирования **RichEdit**. В него можно поместить разделы, соответствующие меню Правка, или разделы меню Формат. Так что перенесите на форму со страницы Standard компонент **PopupMenu**, в котором формируется контекстное меню.

Поскольку в приложении может быть несколько контекстных меню, то и компонентов **PopupMenu** может быть несколько. Оконные компоненты: панели, окна редактирования, а также метки и др. имеют свойство **PopupMenu**, которое по умолчанию пусто, но куда можно поместить имя того компонента **PopupMenu**, с которым будет связан данный компонент. В нашем примере вам надо задать соответствующее значение свойства **PopupMenu** в компоненте **RichEdit1**.

Формирование контекстного всплывающего меню производится с помощью конструктора меню, вызываемого двойным щелчком на **PopupMenu**, точно так же, как это делалось для главного меню. Обратим только внимание на возможность упрощения этой работы. Поскольку разделы контекстного меню обычно повторяют некоторые разделы уже сформированного главного меню, то можно обойтись копированием соответствующих разделов. Для этого, войдя в конструктор меню из компонента **PopupMenu**, щелкните правой кнопкой мыши и из всплывшего меню выберите команду Select Menu (выбрать меню). Вам будет предложено диалоговое окно, в котором вы можете перейти в главное меню. В нем вы можете выделить нужный вам раздел или разделы (при нажатой клавише Shift выделяются разделы в заданном диапазоне, при нажатой клавише Ctrl можно выделить совокупность разделов, не являющихся соседними). Затем выполните копирование их в буфер обмена, нажав клавиши Ctrl-C. После этого опять щелкните правой кнопкой мыши,

выберите команду Select Menu и вернитесь в контекстное меню. Укажите курсором место, в которое хотите вставить скопированные разделы, и нажмите клавиши чтения из буфера обмена — Ctrl-V. Разделы меню вместе со всеми их свойствами будут скопированы в создаваемое вами контекстное меню.

В остальном работа с **PopupMenu** не отличается от работы с **MainMenu**.

4.9 Полоса состояния — компонент StatusBar

Как говорилось в разд. 4.1, обычно приложение должно содержать полосу состояния, в которой отображаются подсказки пользователю и иная информация, помогающая ему в работе. Полоса состояния создается компонентом **StatusBar**, имеющимся на странице Win32 библиотеки. Обычно эта полоса размещается внизу формы (см. рис. 4.1).

Полоса состояния может содержать одну или несколько панелей (на рис. 4.1 она содержит три панели). Свойство **SimplePanel** определяет, включает ли компонент одну панель, или несколько. Если **SimplePanel** = **true**, то вся полоса состояния представляет собой одну панель, текст которой задается свойством **SimpleText**. Давайте для начала создадим в нашем тестовом приложении, которое мы проектируем на протяжении этой главы, такую панель. Перенесите на форму компонент **StatusBar** и задайте в нем **SimplePanel** = **true**. Если вы установите также свойство **AutoHint** = **true**, это будет означать, что в полосе состояния автоматически будут отображаться подсказки — вторые части свойств **Hint** (см. разд. 4.4) тех компонентов, в которых свойство **Hint** задано и над которыми пользователь задержит курсор мыши. У нас такими компонентами являются кнопки инструментальной панели и разделы меню, если вы ввели значения свойств **Hint** в объекты действий, с которыми они связаны. Выполните приложение и убедитесь, что подсказки отображаются в полосе состояния.

В ряде случаев такой односекционной панели недостаточно для того, чтобы дать пользователю всю необходимую ему информацию. Например, при работе с текстовыми редакторами обычно желательно, чтобы в полосе состояния отображалась позиция курсора в тексте — номер символа и строки. Неплохо также отображать информацию о том, был ли изменен пользователем текст в окне. Это позволяет ему понять, надо ли сохранять текст в файле. Ну и подсказки тоже не помешают. Итого, хотелось бы разбить полосу на три панели, как показано на рис. 4.1.

Для полосы с несколькими панелями свойство **SimplePanel** надо задать равным **false**. Набор панелей задается свойством **Panels**. Свойства панелей можно задавать специальным редактором наборов. Вызвать редактор можно тремя способами: из Инспектора Объектов кнопкой с многоточием около свойства **Panels**, двойным щелчком на компоненте **StatusBar**, или из контекстного меню, выбрав команду Panels Editor. Во всех случаях появится окно, вид которого показан на рис. 4.7. Кнопка Add New (левая на рис. 4.7) добавляет новую панель. Вам надо добавить три панели. Выделив строку панели в окне редактора, вы увидите свойства этой панели в окне Инспектора Объектов.

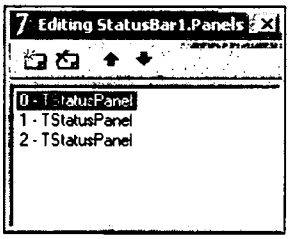


Рис. 4.7
Редактор набора — панелей полосы состояния

Основное свойство каждой панели — **Text**, в который заносится отображаемый в панели текст. Его можно занести в процессе проектирования, а затем можно изменять программно во время выполнения. В нашем случае занесение текста будет выполняться программно, так как информация об окне **RichEdit1** изменяется во время выполнения. Свойство **Width** определяет ширину панели.

Программный доступ к текстам отдельных панелей можно осуществлять двумя способами: через индексированное свойство **Panels** или через его индексированное подсвойство **Items**. Например, два следующих оператора дадут идентичный результат:

```
StatusBar1.Panels[0].Text := 'текст 1';
```

или

```
StatusBar1.Panels.Items[0].Text := 'текст 1';
```

Оба они напечатают текст «текст 1» в первой панели (индексы отсчитываются от 0).

Количество панелей полосы состояния можно определить из подсвойства **Count** свойства **Panels**. Например, следующий оператор очищает тексты всех панелей:

```
for i:= 0 to StatusBar1.Panels.Count -1 do
 StatusBar1.Panels[i].Text := '';
```

Для отображения в нашем примере положения курсора в окне **RichEdit1** можно воспользоваться свойством **CaretPos** окна. Это запись, доступная только во время выполнения, которая содержит координаты курсора: поле **X** определяет номер символа в строке, а поле **Y** — номер строки. Номера отсчитываются от 0. Так что для более понятного пользователю отсчета от 1 полезно, вероятно, прибавить к значениям этих полей 1. Поскольку положение курсора может изменяться пользователем в результате манипуляций мышью и клавишами, информацию о положении курсора следует обнвлять в обработчиках событий **OnKeyDown**, **OnKeyUp**, **OnMouseDown** и **OnMouseUp** компонента **RichEdit1**. Так что в эти обработчики, уже созданные вами в разд. 4.7, надо добавить оператор:

```
StatusBar1.Panels[0].Text:=IntToStr(RichEdit1.CaretPos.Y+1)
                      + ': '+IntToStr(RichEdit1.CaretPos.X+1);
```

Можете выполнить приложение и убедиться, что в первой панели полосы состояния отображаются координаты курсора в тексте окна **RichEdit1**. Теперь займемся второй панелью, в которой мы хотим отображать информацию о том, был ли изменен пользователем текст в окне **RichEdit1**.

Все окна редактирования имеют свойство только времени выполнения **Modified**, показывающее, был ли изменен текст в окне. В начале выполнения приложение это свойство равно **false**. Если пользователь что-то изменил в окне, то это свойство автоматически устанавливается в **true**. Так что для информации пользователя вы можете вставить в те же обработчики событий **OnKeyDown**, **OnKeyUp**, **OnMouseDown** и **OnMouseUp** компонента **RichEdit1** оператор:

```
if RichEdit1.Modified
 then StatusBar1.Panels[1].Text := 'модиф.'
 else StatusBar1.Panels[1].Text := '';
```

Но этого недостаточно. Ведь мы хотим информировать пользователя о наличии несохраненных изменений. Значит после того, как пользователь сохранил текст в файле, надо программно задать значение **Modified** равным **true**, так как несохраненных изменений в этом случае не осталось. То же самое надо сделать после загрузки в окно **RichEdit1** текста из файла. Так что в имеющиеся у вас обработчики действий **AOpen**, **ASave**, **ASaveAs**, **APersonOpen** и **APersonSave** следует добавить операторы:

```
RichEdit1.Modified := false;
```

Вот теперь две первые панели полосы состояния заполняются правильно. Отображение в третьей панели подсказок будет рассмотрено в разд. 4.11. А пока давайте объединим наши наработки по полосе состояния в единое целое. Хотелось бы, чтобы пока окно **RichEdit1** находится в фокусе, в полосе состояния отображались введенные нами панели. А когда пользователь переключает фокус на какой-то другой компонент, полоса состояния становилась несекционированной и отображала только подсказки. Для такой перестройки полосы состояния можно воспользоваться событиями **OnEnter** и **OnExit** окна **RichEdit1**. Обработчики этих событий вы уже писали в разд. 4.7. Теперь в обработчик события **OnEnter** надо добавить операторы:

```
StatusBar1.SimplePanel := false;
StatusBar1.AutoHint := false;
```

Они настраивают полосу состояния на многосекционный вариант. А в обработчик события **OnExit** надо добавить операторы:

```
StatusBar1.SimplePanel := true;
StatusBar1.AutoHint := true;
```

Эти операторы формируют простую односекционную полосу состояния.

4.10 Справочная система

Приложение должно предельно облегчать работу пользователя, снабжая его системой подсказок, помогающих сориентироваться в приложении. Эта система включает в себя:

- Ярлычки, которые всплывают, когда пользователь задержит курсор мыши над каким-то элементом окна приложения. В частности, такими ярлычками обязательно должны снабжаться быстрые кнопки инструментальных панелей, поскольку нанесенные на них пиктограммы часто не настолько выразительны, чтобы пользователь без дополнительной подсказки мог понять их назначение.

- Более развернутые подсказки в панели состояния или в другом отведенном под это месте экрана, которые появляются при перемещении курсора мыши в ту или иную область окна приложения.
- Встроенную систему контекстно-зависимой оперативной справки, вызываемую по клавише F1.
- Раздел меню Справка, позволяющий пользователю открыть стандартный файл справки Windows *.hlp*, содержащий в виде гипертекста развернутую информацию по интересующим пользователя вопросам.

О ярлычках и подсказках вы уже все знаете. Но эти средства дают довольно скудную информацию. Более подробные пояснения пользователю может дать контекстно-зависимая справка, встроенная в приложение. Она позволяет пользователю нажать в любой момент клавишу F1 и получить развернутую информацию о том компоненте, который в данный момент находится в фокусе. Для того чтобы это осуществить, надо разработать для своего приложения стандартными средствами Windows файл справки *.hlp*.

4.10.1 Файл тем справок

Файл тем справок создается с помощью текстового редактора, например, с помощью Microsoft Word. Каждая тема или кадр (в дальнейшем он будет отображаться в отдельном окне) занимает одну страницу. Друг от друга темы отделяются символом разрыва страницы. В Word это осуществляется или командой Вставка | Разрыв и выбором в диалоговом окне опции Начать Новую страницу, или нажатием клавиш Ctrl-Enter.

При написании темы можно использовать многие возможности Word: выбор атрибутов шрифта (полужирный, курсив), цвет шрифта, табуляцию, подчеркивание, включение в текст таблиц (правда, без сетки и без разноцветной заливки) и т.п.

Каждая тема содержит ряд рассмотренных ниже сносок, определяющих ее отображение в WinHelp — стандартной программе Windows.

Темы могут содержать так называемые *горячие области* (hotspot): выделенные слова, позволяющие пользователю переходить из данной темы в другие (позднее мы рассмотрим, как организуются эти выделения). При этом существует несколько возможностей переходов: прямой переход на заданную тему, переход с помощью макроса **KLink**, который может предложить пользователю выбор из тех тем, в **K**-сносках которых встречаются заданные ключевые слова, и с помощью макроса **ALink**, практически идентичного **KLink**, но сравнивающего ключевые слова с другим видом сносок — **A**-сносками.

В тему можно добавлять изображения в виде файлов типов *.bmp*, *.dib*, *.wmf*, *.shg*, *.mrb*. Для этого можно просто воспользоваться буфером обмена, занеся туда изображение из какой-нибудь графической программы и прочитав ее в Word командой Правка | Вставить.

Если в теме много строк, то для их просмотра пользователь будет пользоваться вертикальной прокруткой. Однако часто желательно указать какую-то область в начале текста темы, которая оставалась бы «замороженной» и не прокручивалась. Например, это может быть заголовок темы, шапка таблицы и т.п. Для того чтобы указать область, не подвластную прокрутке, надо выделить ее в Word, выполнить команду Формат | Абзац и в открывшемся диалоговом окне на странице Положение на странице установить опцию Не отрывать от следующего.

Текст тем справок, созданный в Word, надо сохранить в файле формата RTF — обогащенном текстовом формате. Для сохранения его в таком виде надо в Microsoft Word выполнить команду Файл | Сохранить как и в открывшемся диалоговом окне установить опцию Тип файла равной Текст в формате RTF.

Каждая тема снабжается сносками, определяющими ее наименование и ряд других свойств. Сноски в Word делаются командой Вставка | Сноска, или Вставка | Ссылка | Сноска. В отрывшемся диалоговом окне надо указать символ, используемый в сноске. Все сноски помещаются перед текстом темы, т.е. в первых позициях кадра. Чтобы наблюдать и редактировать тексты сносок Word должен быть переключен в режим Разметка страницы.

Теперь рассмотрим основные виды сносок. Сноска с символом # обозначает уникальный *идентификатор темы*, по которому на нее могут ссылаться другие темы. Этому идентификатору в дальнейшем может ставиться в соответствие номер, по которому на данную тему может ссылаться использующее справку приложение. Так что любой кадр должен снабжаться меткой #. Идентификатор, указываемый как текст этой ссылки, имеет чисто служебное назначение, пользователь его нигде не видит. Идентификатор может писаться латинскими или русскими буквами и состоять из одного или нескольких слов. Но иногда в WinHelp возможны сбои на переходах к темам с русскими идентификаторами. Так что все-таки лучше использовать латинские символы.

Сноска с символом **K** (заглавная латинская буква) должна включаться в кадр, если надо, чтобы тематика этого кадра отображалась при работе со справкой в окне справочной системы на странице «Предметный указатель» в списке, из которого пользователь может выбрать требуемую тему по ее первым буквам, или просто пролистав список. Те названия тем, которые пользователь видит в предметном указателе — это и есть тексты сносок **K** соответствующих кадров.

Текст сноски может состоять из одного или нескольких слов. Например, ^KМеню Правка. В этом случае при работе со справкой в списке указателей появится строчка «Меню Правка», по которой пользователь может выйти на данную тему. Можно также ввести несколько различных обозначений для одного и того же кадра, разделяя их точками с запятой. Например, ссылка ^KМеню Правка; Меню обеспечит две строки в указателе: «Меню правка» и «Меню», которые будут ссылаться на одну и ту же тему. Пользователь сможет выйти на нее по любой из этих строчек.

Элементы, указанные в сносках **K**, используются не только в списке указателя, но и при организации переходов между темами по ключевым словам с помощью макроса **KLink**, который будет рассмотрен позднее.

Сноска **A** (заглавная латинская буква) аналогична по синтаксису сноске **K**. Но ее тексты не включаются в указатель, как для сноски **K**, а используются только для переходов по ключевым словам с помощью рассмотренного далее макроса **ALink**. Таким образом, если тексты сносок **K** участвуют в двух различных процессах — поиске по указателю и поиске по ключевым словам, то сноски **A** позволяют развязать эти два процесса.

Сноска с символом **$** определяет заголовок данной темы. Этот заголовок используется в WinHelp в ряде режимов работы, в частности, в системе Поиск, в окне История, позволяющем пользователю вернуться к одной из уже просмотренных тем, и в ряде других режимов работы. Этот заголовок используется также во вспомогательном окне указателя тем в случаях, если какая-то строка основного указателя относится сразу к нескольким темам. Поясним это.

Выше были рассмотрены сноски **K** и **A**. Ничто не мешает в этих сносках для разных кадров указать одинаковый элемент. Это облегчает пользователю поиск нужной информации. Например, если вы описываете темы, связанные с главным меню приложения и его разделами Файл, Правка и др., вы можете во всех этих темах в сноски **K** и **A** внести элемент «Меню». Тогда строка «Меню» в предметном указателе будет относиться сразу к нескольким темам, и пользователь, выбравший эту строку, должен иметь возможность уточнить свой выбор, указав одну из конкретных тем. Для этого система справки автоматически предъявит пользователю окно Найденные разделы. Строки, обозначающие в этом окне конкретные темы задаются в кадрах с помощью сносок с символом **$**. Таким образом, сноски **$** имеет смысл включать в те кадры, которые в своих сносках **K** и **A** имеют элементы, используемые также в сносках **K** и **A** других кадров.

Рассмотрим теперь возможности создания в текстах «горячих областей», обеспечивающих переходы между темами. Для того чтобы выделить в тексте темы некоторое слово или сочетание слов, при щелчке на котором пользователь сразу перейдет на другую тему, надо сделать следующее. После выделяемых слов (без пробела) надо написать идентификатор темы, на которую надо перейти. Затем соответствующее слово или сочетание слов выделяется двойным подчеркиванием. Это подчеркивание не будет видно пользователю при работе со справкой — для него эта горячая область будет выделена цветом. Для того чтобы в Microsoft Word обеспечить двойное подчеркивание, надо выделить курсором требуемое слово или сочетание слов, выполнить команду Формат | Шрифт, и в открывшемся окне на странице Шрифт установить опцию Подчеркивание в положение Двойное.

После этого следующий за подчеркнутыми словами идентификатор темы перехода оформляется как скрытый (невидимый). Для этого выполняется та же команда Формат | Шрифт. В открывшемся окне на странице Шрифт в разделе Эффекты надо установить индикатор опции Скрытый и проследить, чтобы опция Подчеркивание имела значение (нет).

Для того чтобы вы сами видели этот скрытый текст и могли бы его редактировать, надо выполнить в Word команду Сервис | Параметры и на странице Вид установить индикатор Скрытый текст. Если вам захочется напечатать на принтере текст вашего файла, то дополнительно на странице Печать в разделе Печатать надо тоже установить индикатор Скрытый текст. Не забудьте все это проделать. Иначе вам будет очень трудно работать с вашим файлом.

Приведем пример вставки в текст подобного перехода. Пусть вы написали тему, в которой имеется фраза: «В приложении имеется полоса главного меню и инструментальная панель». Вы хотите выделить слова «полоса главного меню», чтобы при щелчке на них пользователь переходил к теме с описанием меню. Пусть идентификатор темы, где описано меню, — «MainMenu». Тогда эту фразу надо записать следующи образом: «В приложении имеется полоса главного менюMainMenu и инструментальная панель». В этом и последующих примерах скрытый текст показан бледным шрифтом. При работе со справкой скрытый текст не будет виден. Пользователь увидит только тот текст, который был до введения указателей переходов. В этом тексте будут выделены цветом и подчеркнуты слова «полоса главного меню». При щелчке на них пользователь перейдет к соответствующей теме.

Выше была описана организация непосредственного перехода на другую тему. Но иногда вам надо предоставить пользователю возможность выбрать одну из нескольких родственных тем. Например, вы хотите ввести в тему горячую область

«см. также», при щелчке на которой пользователь сможет выбрать одну из тем, описывающих главное меню и инструментальную панель.

Для решения подобной задачи предусмотрено два макроса — **KLink** и **ALink**, в которых может быть задан список идентификаторов тем. Различие этих макросов состоит в том, что первый из них ищет заданные ключевые слова в **K**-сносках, а второй — в **A**-сносках. В простейшем случае в макросы **ALink** и **KLink** передаются одно или несколько ключевых слов или словосочетаний, разделенных точками с запятой. Если хотя бы одно из этих словосочетаний содержит запятую, то весь список заключается в двойные кавычки.

Пример вызова макроса: **ALink(полоса главного меню; Panel)**. В этом примере подразумевается, что есть темы, в которых в **A**-сносках имеются тексты «полоса главного меню» и «Panel».

Макрос **KLink** ведет поиск в **K**-сносках сначала по первому слову. Если нашлось несколько тем, то пользователю показывается окно Найденные разделы, в котором пользователь может выбрать требуемый. Если нашлась только одна тема, осуществляется переход на нее. Если же не нашлось ни одной темы, начинается поиск по второму ключевому слову и т.д. Если ни одной темы ни по одному слову не нашлось, пользователю будет показано окно с соответствующим сообщением.

Алгоритм выполнения макроса **ALink** несколько иной. Поиск ведется сразу по всем ключевым словам в **A**-сносках. Если нашлась одна или несколько тем, пользователю показывается окно Найденные разделы. Если не нашлось ни одной темы, пользователю будет показано окно с соответствующим сообщением.

Вызов любого макроса начинается с символа восклицательного знака «!». Вызов помещается после текста горячей области, без пробела и форматируется как скрытый текст. Например: «см. также!Alink(полоса главного меню;Panel)».

Ограничимся этими сведениями о составлении файла тем справки. Если вас заинтересуют дополнительные возможности, а их очень много, посмотрите во встроенной справке программы Microsoft Help Workshop, которая будет описана в разд. 4.10.2. А пока попробуйте составить файл тем справки для того приложения, которое мы разрабатываем в данной главе. Ниже приводятся фрагменты возможного текста такого файла.

```
#$KA Обзор возможностей приложения
см. также!ALink(полоса главного меню; Panel; Контекст)
Приложение предназначено для просмотра информации и характеристик студентов (сотрудников, школьников). Одновременно, чтобы можно был показать организацию некоторых операций с окном редактирования RichEdit, в приложении организован полноценный редактор на основе этого компонента. Так что в окно RichEdit можно загружать любые текстовые файлы и файлы .rtf, редактировать и форматировать тексты, сохранять их в файлах.
Приложение содержит все атрибуты полноценного приложения Windows. Имеется полоса главного менюMainMenu, контекстное менюКонтекст в окне RichEdit, инструментальная панельPanel, полоса состояния с подсказками внизу окна, справочная система.

Сноски
# main
$ Обзор возможностей приложения
```

```
K обзор возможностей
A main

.....Разрыв страницы.....

# $ K A Полоса главного меню
см. также!ALink(main)
```

Полоса содержит меню **Файл** (работа с текстовыми файлами, печать документа), **Правка** (операции редактирования), **Поиск** (контекстный поиск и замена), **Формат** (атрибуты шрифта и абзаца), **Характеристики** (загрузка из файла, формирование заготовки, сохранение в файле), **?** (вызов справки, сведения о программе).

```
Сноски
# MainMenu
$ Полоса главного меню
K обзор возможностей
A полоса главного меню

.....Разрыв страницы.....
```

Это фрагменты только двух тем. Остальные тексты, более осмысленные и полезные, чем приведенные фрагменты, напишите сами. Первые две строки каждой темы (заголовок и «см. также») имеет смысл делать непрокручиваемыми, как было рассмотрено ранее.

Написав тексты, сохраните их в файле RTF с именем, например, *MyHelp.rtf*.

4.10.2 Компиляция и отладка проекта справки

Компиляция и отладка справки производится с помощью программы Microsoft Help Workshop, запускаемой из файла *Hcrtf*, расположенного в каталоге Delphi *...\Help\Tools*. Эта программа позволяет легко создать файл проекта справки, без которого ее нельзя компилировать, а если нужно, то создать еще некоторые вспомогательные файлы, например, файл содержания справки и ряд других. Далее программа позволяет скомпилировать файл справки и проверить его в работе. В принципе можно создавать файлы проекта, содержания и др. просто в любом текстовом редакторе. Но тогда надо детально изучить их синтаксис. А Help Workshop позволяет автоматизировать всю работу, не требуя знания синтаксиса, так как эта программа сама создает тексты файлов, отражающие действия разработчика.

Для компиляции справки и ее связывания с использующим ее приложением необходимо сформировать файл проекта справки. Это текстовый файл в формате ASCII. Для создания нового файла проекта, надо выполнить команду программы Microsoft Help Workshop File | New и в появившемся окне New выбрать опцию Help Project. Далее в окне Project File Name надо задать имя и каталог файла проекта справки. При этом учтите, что то же имя, какое вы зададите файлу проекта, будет присвоено компилятором и завершающему файлу справки *.hlp*. Так что для нашего примера назовите его, например, «MyHelp». Стандартное расширение файла проекта — *.hpj*.

В результате описанных действий перед вами появится окно заготовки файла проекта (рис. 4.8), в котором первоначально будет занесен только раздел [Options]. Теперь, щелкая на соответствующих кнопках в правой части этого окна, можно создавать и заполнять другие разделы файла проекта.

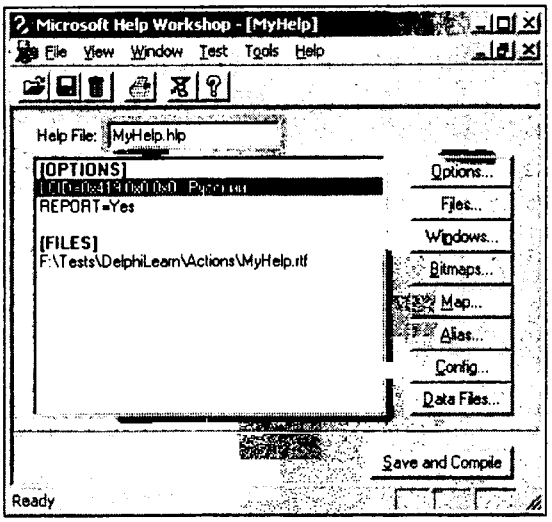


Рис. 4.8
Окно редактирования файла проекта справки

Кнопка Files в правой части окна рис. 4.8 (не путайте с аналогичным по названию разделом меню) позволяет указать имена файлов с текстами тем. Щелкните на этой кнопке. Перед вами появится окно Topic Files, в котором надо щелкнуть на копке Add и выбрать среди файлов подготовленный файл текстов тем справки, в нашем примере файл *MyHelp.rtf*. В результате в файле проекта появится раздел [Files] с внесенным в него именем файла. Если ваша справка компилируется из нескольких файлов тем, то аналогичным образом вы должны включить в раздел [Files] и остальные файлы.

Для файлов тем, расположенных в том же каталоге, где располагается файл проекта *hpj*, имена файлов могут указываться без путей. Если же они расположены в другом каталоге, то они указываются с путем.

Можно еще упомянуть о возможности создать таблицу соответствия номеров контекстной справки, задаваемых в приложении, и идентификаторов тем. Это имеет смысл делать в основном в случае, если вы работаете с версией Delphi, младше Delphi 6. Для создания такой таблицы надо щелкнуть на кнопке Map, в открывшемся диалоговом окне Map щелкнуть на кнопке Add, и в окне добавления нового входа Add Map Entry ввести идентификатор темы (в окно Topic ID) и произвольный номер (в окно Mapped numeric value), по которому будет осуществляться ссылка на эту тему из приложения. Можно также ввести комментарий в окно Comment. Этот комментарий просто будет занесен в файл проекта и никак не повлияет на справку. Но он очень пригодится вам, когда вы будете назначать значения **HelpContext** компонентам своего приложения. Затем надо щелкнуть на OK, вернуться в окно «Map» и опять щелкнуть на OK. В результате подобных действий в файле проекта появится раздел [MAP].

Когда файл проекта создан, надо щелкнуть на кнопке Save and Compile в нижнем правом углу окна рис. 4.8. Файл проекта будет сохранен и откомпилирован, в результате чего создастся файл справки *.hlp* с тем же именем, что и файл проекта. Кнопкой Save and Compile надо пользоваться каждый раз, когда изменяется файл проекта. Если же в дальнейшем вы меняете только тексты в файле тем, то компиляцию удобнее производить командой File | Compile или проще — нажав соответствующую быструю кнопку с пиктограммой мясорубки (вторая справа на рис. 4.8). Причем вы можете это делать, даже не открывая предварительно файла проекта. В результате выполнения этой команды откроется окно Compile a Help File (см. рис. 4.9), в котором вы можете выбрать нужный файл проекта и затем щелкнуть на кнопке Compile. Предварительно удобно установить флаг опции Automatically display Help file in WinHelp when done, которая обеспечивает сразу после компиляции просмотр скомпилированного файла справки. Если вы не установили опцию Minimize window while compiling, то во время компиляции вы будете видеть окно, с работающей мясорубкой, отражающей процесс компиляции. Установка указанной опции минимизирует это окно. Опция Turn off compression (reduces compile time) позволяет временно отключить сжатие файла. Это приведет к увеличению размера файла справки, но заметно уменьшит время компиляции больших справок. Опция Include .rtf filename and topic ID in Help file включит в файл справки имена файлов текстов тем и идентификаторы тем. Это позволит вам в процессе отладки, установив в меню File окна рис. 4.8 опцию Help Author, получать при просмотре справки оперативную информацию о каждом кадре справки.

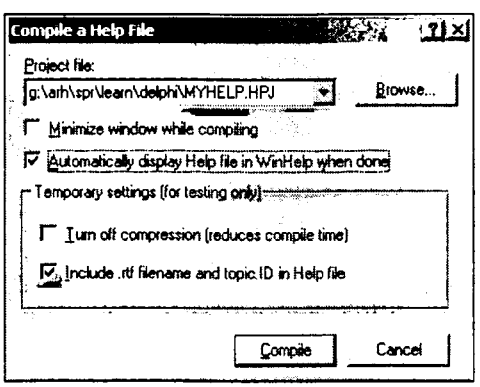


Рис. 4.9
Задание опций компиляции файла справки

После компиляции вы увидите в окне Help Workshop содержимое текстового файла Compilation, в котором содержатся сведения о результатах компиляции. Здесь же будут замечания и сообщения об ошибках, которые возникнут при неправильном исходном файле *.rtf*. Однако не спешите огорчаться, получив множество сообщений об ошибках. И не бросайтесь сразу искать их и исправлять в вашем файле тем. Посмотрите в конец выданной вам информации о результатах компиляции. Если вы увидите там фразу «Created ... .hlp, ... bytes», значит ваш файл справки скомпилирован, и вы можете его посмотреть. В процессе этого просмотра и отладки вы скорее найдете ошибки, чем просто отыскивая их в исходном тексте.

После компиляции при включенной указанной выше опции Automatically display Help file in WinHelp when done вы сразу же можете поработать с откомпилированным файлом справки. В дальнейшем просмотр и работа с этим файлом может осуществляться или с помощью команды File | Run WinHelp, или соответствующей быстрой кнопкой со знаком вопроса (крайняя правая в панели инструментов на рис. 4.8). В результате откроется диалоговое окно View Help File, из которого запуск просмотра осуществляется кнопкой View Help. В верхнем окне File устанавливается интересующий вас файл справки.

В процессе отладки удобно установить в меню File рис. 4.8 опцию Help Author. Тогда при просмотре справки вы можете в любом кадре нажать правую кнопку мыши, во всплывающем меню выбрать команду Сведения о разделе и увидеть окно, в котором указано название раздела (темы), файл справки, исходный файл тестов (файл раздела) и идентификатор этого раздела. Сведения о файле и идентификаторе темы содержатся в этом окне, если вы при компиляции установили опцию Include .rtf filename and topic ID in Help file. В том же меню, всплывающем при нажатии правой кнопки мыши в режиме Help Author, вы можете установить опцию Запросы при ссылках. Тогда при каждом переходе от темы к теме будет появляться окно, позволяющее вам отследить, в каких случаях и к каким темам осуществляется переход.

4.10.3 Файл содержания

Файл содержания (Contents file), имеющий расширение *.cnt*, является текстовым файлом, который можно проектировать с помощью Microsoft Help Workshop и присоединять его к файлу проекта справки. Этот файл обеспечивает при работе со справкой страницу Содержание в окне справочной системы, в которой в виде пиктограмм открытых и закрытых книг и пиктограмм тем отражается структура справки. Чтобы создать новый файл содержания, надо в Help Workshop выполнить

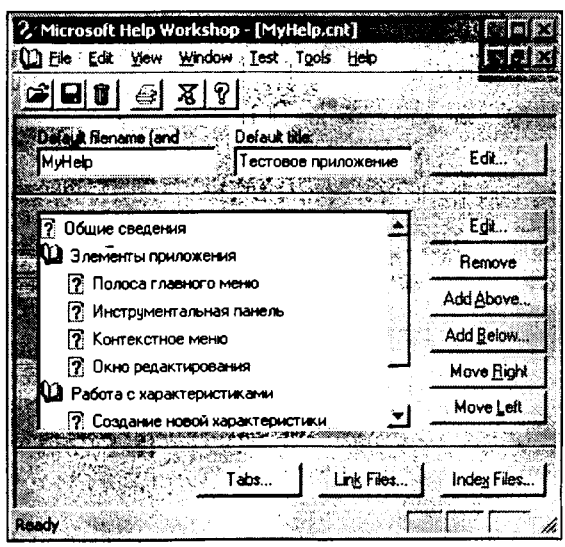


Рис. 4.10
Окно редактирования файла содержания

команду File | New и в окне New выбрать опцию Help Contens. Откроется окно с загруженным в него файлом содержания. Вид этого окна можно видеть на рис. 4.10; только в первый момент все его окна редактирования будут пустыми.

В верхних окнах надо задать: в левом — имя файла *.hlp*, в правом — заголовок, который будет появляться в окне Содержание при работе справочной системы. Заносить информацию в эти окна можно непосредственно, но проще щелкнуть на верхней кнопке Edit и работать в открывающемся при этом диалоговом окне.

Нижнее окно заполняется с помощью кнопок Add Above и Add Below. Первая из этих кнопок вставляет новую строку выше той, в которой находится в данный момент курсор, а вторая — ниже этой строки. При нажатии любой из этих кнопок открывается окно Edit Contents Tab Entry (см. рис. 4.11), в котором вы можете задать очередной заголовок (Heading), отображающийся в виде книги, или очередную тему (Topic). Для заголовка указывается только его текст (Title). Для темы записывается ее название (Title), которое появится на странице Содержание справки, и идентификатор темы (Topic ID), указанный в файле тем.

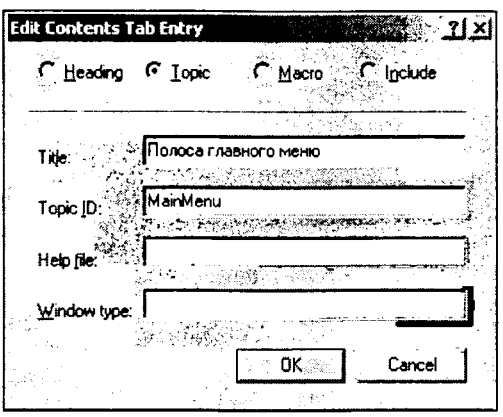


Рис. 4.11
Задание очередного элемента файла содержания

Вернемся к рис. 4.10 и рассмотрим некоторые другие кнопки окна редактирования файла содержания. Кнопки Move Right и Move Left позволяют сдвигать вправо и влево выделенные строки заголовков и тем, обеспечивая нужные отступы, характеризующие многоуровневую структуру. При этом строки тем, относящихся к одному заголовку, располагаются на один шаг правее своего заголовка и не могут сдвигаться друг относительно друга. Точнее, сдвинуть их можно, но это никак не отразится на их расположении при работе пользователя со справкой.

Кнопка Edit позволяет редактировать выделенную строку, а кнопка Remove удаляет строку.

Создайте файл содержания для нашего проекта. Затем сохраните его командой File | Save as. Теперь вам надо вернуться к файлу проекта и указать в нем, что проект имеет файл содержания. Откройте опять файл проекта, или просто вернитесь к нему, если вы его не закрывали. В окне рис. 4.8 щелкните на кнопке Options. В открывшемся диалоговом окне перейдите на страницу Files. На ней вы увидите окно Contents file. Укажите в нем с помощью кнопки Browse ваш файл содержания. Щелкните на OK. Вы вернетесь в окно рис. 4.8 и увидите в нем появившуюся стро-

ку, указывающую файл содержания. Теперь можно щелкнуть на кнопке Save and Compile, откомпилировать проект, и полюбоваться полноценной справкой, имеющей страницу содержания.

4.10.4 Связь приложения с файлом справки

Чтобы сослаться на созданный файл справки из приложения Delphi, надо выполнить команду Project | Options и в окне Project Options (опции проекта) на странице Application (приложение) установить значение опции Help file, равное имени подготовленного файла *.hlp*. Если предполагается, что файл справки будет расположен в том же каталоге, где находится само приложение, то имя файла надо задавать без указания пути. Иначе приложение, работающее на вашем компьютере, перестанет работать на компьютере пользователя, у которого каталоги не совпадают с вашими.

Введите ссылку на файл справки в тестовом приложении, которое разрабатывается в данной главе, и выполните приложение. Если вы использовали в приложении стандартное действие открытия справки **HelpContents**, то, выбрав раздел меню Справка вашего приложения, увидите окно справки.

Если вы не используете стандартное действие, вы можете в обработчик управляющего элемета, вызывающего справку, вставить оператор:

```
Application.HelpJump(<идентификатор темы>);
```

В качестве идентификатора задается значение, введенное в теме ссылкой #. Например:

```
Application.HelpJump('main');
```

Указанная тема будет открываться пользователю при щелчке на соответствующем управляющем элементе. А далее при желании он сможет перейти на любую другую тему.

Если вы назначите действию вызова справки традиционную горячую клавишу F1, то нажатием этой клавиши пользователь сможет в любой момент вызвать справку, даже не заходя в меню. Но нередко пользователю желательно получать контекстно зависимую информацию. Чтобы, нажав F1, когда какой-то компонент находится в фокусе, получить информацию именно об этом компоненте.

Большинство визуальных компонентов и все объекты действий имеют свойства, в которых можно указать, какая тема справки должна отображаться пользователю, когда он нажмет F1. В версиях, младше Delphi 6, такое свойство одно — **HelpContext**. Это номер темы, который задается в проекте справки специальной таблицей [MAP], содержащей эти условные номера и соответствующие им идентификаторы тем (см. в разд. 4.10.2). Начиная с Delphi 6, появилось еще два свойства: **HelpKeyword** и **HelpType**. Первое из них является идентификатором темы, содержащимся в **K**-сноске (это обозначения тем, которые видны в окне справки на странице Указатель). А второе определяет, каким свойством — **HelpContext** или **HelpKeyword** задается ссылка на тему. Если **HelpType** = **htContext** — используется свойство **HelpContext**; если **HelpType** = **htKeyword** — используется свойство **HelpKeyword**.

Опробуйте это, задав ссылки на темы вашей справки в различных действиях или компонентах. Только учтите, что в этом случае надо удалить горячую клавишу F1 из действия, связанного с показом справки. Одно из двух: или F1 обеспечивает контекстно зависимые справки, или она относится к показу всей справки.

4.11 Приложение — объект Application и компонент ApplicationEvents

В каждом приложении автоматически создается объект **Application** типа **TApplication** — приложение. Этот компонент отсутствует в палитре библиотеки только потому, что он всегда один в приложении. **Application** имеет ряд свойств, методов, событий, характеризующих приложение в целом.

Рассмотрим только некоторые свойства **Application**. Свойство **Title** определяет строку, которая появляется около пиктограммы свернутого приложения. Свойство **HelpFile** указывает файл справки, который используется в приложении в данный момент как файл по умолчанию. Свойство **ExeName** является строкой, содержащей имя выполняемого файла с полным путем к нему. Это свойство удобно использовать, чтобы определить каталог, из которого запущено приложение и который может содержать другие файлы (настройки, документы, базы данных и т.п.), связанные с приложением. Выражение **ExtractFilePath(Application.ExeName)** дает этот каталог. Обычно свойство **ExeName** тождественно функции **ParamStr(0)**, возвращающей нулевой параметр командной строки — имя файла с путем.

Ряд свойств объекта **Application** определяет ярлычки подсказок компонентов приложения. Свойство **Hint** содержит текст подсказки **Hint** того визуального компонента или раздела меню, над которым в данный момент перемещается курсор мыши. Смена этого свойства происходит в момент события **OnHint**, которое будет рассмотрено позднее. Во время этого события текст подсказки переносится из свойства **Hint** компонента, на который переместился курсор мыши, в свойство **Hint** объекта **Application**. Свойство **Application.Hint** можно использовать для отображения этой подсказки или для установки и отображения в полосе состояния текста, характеризующего текущий режим приложения. Далее вы увидите соответствующие примеры.

Теперь остановимся на некоторых методах объекта **Application**. Методы **Initialize** — инициализация проекта, и **Run** — запуск выполнения приложения, включаются в каждый проект автоматически — вы можете это увидеть в головном файле проекта, если выполните команду Project | View Source. Там же вы можете увидеть применение метода создания форм **CreateForm** для всех автоматически создаваемых форм проекта.

Метод **Terminate** завершает выполнение приложения. Если вам надо завершить приложение из главной формы, то вместо метода **Application.Terminate** вы можете использовать метод **Close** главной формы. Но если вам надо закрыть приложение из какой-то вторичной формы, например, из диалога, то надо применять метод **Application.Terminate**.

Метод **Minimize** сворачивает приложение, помещая его пиктограмму в полосу задач Windows.

В классе **TApplication** имеется еще немало методов, но часть из них используется в явном виде достаточно редко (вы можете посмотреть их во встроенной справке Delphi), часть будет рассмотрена в разд. 4.13, а часть будет рассмотрена ниже при обсуждении событий объекта **Application**. Хотелось бы только обратить внимание читателя на очень полезный метод **MessageBox**, позволяющий вызывать диалоговое окно с указанным текстом, указанным заголовком и русскими надписями на кнопках (в русифицированных версиях Windows). Это наиболее удачный полностью русифицируемый стандартный диалог. Он отображает диалоговое окно с заданными кнопками, сообщением и заголовком и позволяет проанализировать ответ пользователя.

Метод объявлен следующим образом:

```
function MessageBox(Text, Caption: PChar; Flags: Longint): Integer;
```

Параметр **Text** представляет собой текст сообщения, которое может превышать 255 символов. Для длинных сообщений осуществляется автоматический перенос текста. Параметр **Caption** представляет собой текст заголовка окна. Он тоже может превышать 255 символов, но не переносится. Так что длинный заголовок приводит к появлению длинного и не очень красивого диалогового окна.

Параметр **Flags** представляет собой множество флагов, определяющих вид и поведение диалогового окна. Этот параметр может комбинироваться операцией сложения по одному флагу из следующих групп.

Флаги кнопок, отображаемых в диалоговом окне

Флаг	Значение (в скобках даны надписи в русифицированных версиях Windows)
MB_ABORTRETRYIGNORE	Кнопки Abort (Стоп), Retry (Повтор) и Ignore (Пропустить).
MB_OK	Кнопка OK. Этот флаг принят по умолчанию.
MB_OKCANCEL	Кнопки OK и Cancel (Отмена).
MB_RETRYCANCEL	Кнопки Retry (Повтор) и Cancel (Отмена).
MB_YESNO	Кнопки Yes (Да) и No (Нет).
MB_YESNOCANCEL	Кнопки Yes (Да), No (Нет) и Cancel (Отмена).

Флаги пиктограмм в диалоговом окне

Флаг	Пиктограмма
MB_ICONEXCLAMATION, MB_ICONWARNING	Восклицательный знак (замечание, предупреждение).
MB_ICONINFORMATION, MB_ICONASTERISK	Буква «i» в круге (подтверждение).
MB_ICONQUESTION	Знак вопроса (ожидание ответа).
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	Знак креста на красном круге (запрет, ошибка).

Флаги, указывающие кнопку по умолчанию (которая в первый момент находится в фокусе)

Флаг	Кнопка
MB_DEFBUTTON1	Первая кнопка. Это принято по умолчанию.
MB_DEFBUTTON2	Вторая кнопка.
MB_DEFBUTTON3	Третья кнопка.
MB_DEFBUTTON4	Четвертая кнопка.

Функция возвращает нуль, если не хватает памяти для создания диалогового окна. Если же функция выполнена успешно, то возвращаемая величина свидетельствует о следующем:

Значение	Численное значение	Пояснение
IDABORT	3	Выбрана кнопка Abort (Стоп).
IDCANCEL	2	Выбрана кнопка Cancel (Отмена) или нажата клавиша Esc.
IDIGNORE	5	Выбрана кнопка Ignore (Пропустить).
IDNO	7	Выбрана кнопка No (Нет).
IDOK	1	Выбрана кнопка OK.
IDRETRY	4	Выбрана кнопка Retry (Повтор).
IDYES	6	Выбрана кнопка Yes (Да).

В дальнейшем мы не раз будем использовать метод **MessageBox**. Да и во всех примерах, разработанных вами в предыдущих главах, сообщения с помощью функции **ShowMessage** лучше заменить на вызов метода **MessageBox**. Например, на рис. 4.12 приведены диалоговые окна с сообщением о какой-то ошибке, вызываемые операторами

```
ShowMessage('Ошибочные данные');
Application.MessageBox('Ошибочные данные','Ошибка',
                       MB_ICONSTOP);
```

Думаю, что окно, вызываемое функцией **MessageBox**, предпочтительнее. Его заголовок содержит заданный текст, а не имя приложения. А пиктограмма не только украшает окно, но и концентрирует внимание пользователя на сообщении.

Но сновное преимущество метода **MessageBox**, конечно, не в оформлении окна, а в возможности анализа возвращенного значения, т.е. анализа действий

Рис. 4.12
Диалоговые окна, вызываемые функциями ShowMessage (а) и MessageBox (б)

пользователя. Ниже приведен несколько абстрактный пример, демонстрирующий это:

```
case Application.MessageBox(
                    '      'Вы согласны продолжать вычисления?',
                           'Ответьте на вопрос',
                           MB_YESNOCANCEL + MB_ICONQUESTION) of
  IDYES:                                                    '
     <действия, если пользователь щелкнул на кнопке "Да">
  IDCANCEL:
     <действия, если пользователь щелкнул на кнопке "Отмена">
  IDNO:
     <действия, если пользователь щелкнул на кнопке "Нет">
end;
```

Вызов **MessageBox** в этом примере формирует диалоговое окно, показанное на рис. 4.13. Структура **case** анализирует возвращенный результат и обеспечивает различные действия в зависимости от того, на какой кнопке щелкнул пользователь в окне рис. 4.13.

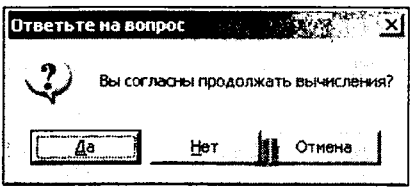


Рис. 4.13
Пример окна, формируемого функцией MessageBox

Вернемся к объекту **Application**. В его классе **TApplication** определено множество событий, которые очень полезны для организации приложения. До Delphi 5 для использования этих событий было необходимо вводить соответствующие обработчики и указывать на них объекту **Application** специальными операторами. В Delphi 5 введен компонент **ApplicationEvents**, существенно облегчивший эту задачу. Этот компонент перехватывает события объекта **Application** и, следовательно, обработчики этих событий теперь можно писать как обработчики событий невизуального компонента **ApplicationEvents**.

Поместите этот компонент, расположенный в библиотеке на странице Additional, на форму тестового приложения, разрабатываемого в данной главе. В окне Инспектора Объектов на странице событий вы можете увидеть множество событий, которые может обработать компонент **ApplicationEvents**. Рассмотрим только одно из них — событие **OnHint**. Остальные при желании вы можете посмотреть во встроенной справе Delphi или в справке [3].

Событие **OnHint** возникает в момент, когда курсор мыши начинает перемещаться над компонентом или разделом меню, в котором определено свойство **Hint**. При этом свойство **Hint** компонента переносится в свойство **Hint** приложения (**Application.Hint**) и в обработчике данного события может отображаться, например, в строке состояния. В разд. 4.9 мы ввели в тестовое приложение, разрабатываемое в данной главе, полосу состояния, причем в зависимости от того, какой компонент находится в фокусе, она было то односекционная, то полосой с тремя панелями. Тогда мы не могли до конца реализовать задуманное: чтобы в третьей панели ото-

бражались подсказки. Теперь мы можем это реализовать. Для этого надо написать следующий код обработчика события **OnHint** компонента **ApplicationEvents**:

```
if (StatusBar1.SimplePanel)
  then StatusBar1.SimpleText:=Application.Hint
else StatusBar1.Panels.Items[2].Text:=Application.Hint;
```

Этот код в зависимости от текущего состояния полосы **StatusBar1** (проверяется по ее свойству **SimplePanel**) отображает текст подсказки **Application.Hint** или в свойстве **SimpleText** простой полосы, или в третьей панели многосекционной полосы. Причем это отображение выполняется независимо от значения свойства **ShowHint** того компонента, над которым перемещается курсор мыши.

Если вы работаете с версией, младше Delphi 5, можно реализовать то же самое, но несколько сложнее. Вам надо ввести функцию обработки события **OnHint** (назовем ее **DisplayHint**), которая может содержать тот же приведенный выше оператор, обеспечивающий отображение текста, переданного компонентом в **Application.Hint**.

Прототип этой функции можно внести в описание класса формы. Кроме того, надо указать приложению на эту функцию как на обработчик события **OnHint**. Это можно сделать, например, задав в обработчике события **OnCreate** формы оператор:

```
Application.OnHint := DisplayHint;
```

В итоге текст вашего модуля может иметь вид:

```
type
  TForm1 = class(TForm)
    StatusBar1: TStatusBar;
    ...
    procedure FormCreate(Sender: TObject);
  private
    procedure DisplayHint(Sender: TObject);
  ...
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.DisplayHint(Sender: TObject);
begin
 if (StatusBar1.SimplePanel)
  then StatusBar1.SimpleText:=Application.Hint
  else StatusBar1.Panels.Items[2].Text:=Application.Hint;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := DisplayHint;
  ...
end;
```

Как видим, в ранних версиях Delphi все это получается более громоздко, чем при использовании компонента **ApplicationEvents**.

4.12 Повторное использование кодов, шаблоны компонентов

Разработка хорошо структурированного приложения, рассмотренная в предыдущих разделах, требует немалых затрат времени. А согласно требованиям Microsoft к приложениям Windows (см. разд. 5.1) меню, пиктограммы, инструментальные панели, диалоги должны быть стандартизованы и привычны пользователю. Это значит, что при разработке какого-то очередного приложения вам придется повторно выполнять практически ту же работу. Обидно! Разработчик программ должен любить себя, беречь свое время и делать все возможное для повторного использования однажды разработанных кодов, компонентов, форм. Некоторые возможности этого, связанные с Депозитарием, уже были рассмотрены ранее в разд. 1.5. Теперь рассмотрим другой путь — сохранение настроенных вами компонентов или их совокупности в качестве шаблонов.

В приложении можно выделить один компонент или группу компонентов и сохранить их в библиотеке как шаблон. Тем самым вы как бы введете в библиотеку подобие своего собственного компонента (правда, настоящие компоненты создаются иначе) и сможете в дальнейшем использовать его в любых своих приложениях.

Рассмотрим создание шаблона на примере приложения, которое проектировалось на протяжении данной главы. Большинство элементов этого приложения пригодятся для любой другой программы, содержащей окно редактирования **RichEdit**. Нажмите клавишу Shift и, не отпуская ее, щелкните на компонентах **RichEdit1**, **ImageList1**, **ActionList1**, **MainMenu1**, **PopupMenu1**, **FontDialog1**, **OpenDialog1**, **SaveDialog1**, **PrintDialog1**, **ToolBar1**, **StatusBar1**, **ApplicationEvents1**. Затем выполните команду Component | Create Component Template.

В открывшемся диалоговом окне Component Template Infomation (рис. 4.14) вы можете задать имя компонента (в верхнем окне редактирования), которое будет появляться на ярлычке подсказки, если пользователь задержит курсор мыши над пиктограммой этого компонента в палитре библиотеки. На рис. 4.14 это имя — «MyRichEditTemplate». В выпадающем списке в средней части окна вы можете выбрать страницу библиотеки визуальных компонентов, на которой хотите разместить пиктограмму компонента. Вы можете также указать новое имя («Мои шаблоны» на рис. 4.14), и тогда в библиотеке визуальных компонентов будет создана новая страница с этим именем. Можно также изменить пиктограмму данного компонента (кнопка Change). По умолчанию будет взята пиктограмма того компонента,

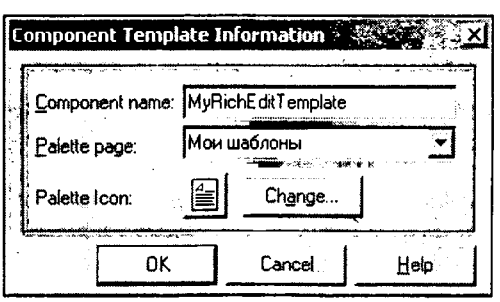


Рис. 4.14
Окно ввода информации о новом шаблоне компонента

который вы первым выделили при включении в группу. Так что в нашем случае было целесообразно начать выделение с компонента **RichEdit1**.

После выполнения всех описанных операций щелкните на кнопке OK.

Ваш шаблон появится в библиотеке. Вы сможете убедиться в этом, посмотрев на указанную вами страницу библиотеки.

Теперь попробуйте создать новое приложение и перенести на форму созданный вами шаблон. Вы увидите, что на форме появятся все компоненты, включенные вами в шаблон, и все обработчики их событий. Только, наверное, имеет смысл немного подредактировать всю группу компонентов. Очевидно, стоит убрать действия, связанные с характеристиками, так как это специфика создававшегося ранее приложения. Имеет смысл убрать и соответствующие этим действиям быстрые кнопки и обработчики событий этих действий. А единственны оператор, который вам придется добавить, чтобы ваш шаблон заработал в новом приложении — объявление глобальной переменной **FName**:

```
var FName: string;
```

После проведенной коррекции выделите опять все компоненты и сохраните шаблон под прежним именем. Новый вариант заменит в библиотеке первоначальный.

Созданный вами шаблон может многократно использоваться при разработке различных приложений. Его применение сэкономит вам много времени. Аналогичным образом вы можете создать себе немало шаблонов компонентов и их групп, кочующих из приложения в приложение. А если в дальнейшем вам перестанет нравиться созданный вами шаблон, вы можете удалить его из библиотеки. Для этого надо выполнить команду Component | Configure Palette. При выполнении этой команды вы увидите диалоговое окно страницы Palette опций среды Delphi. В этом окне вы можете найти и удалить ваш шаблон.

4.13 Формы

4.13.1 Управление формами

Основным элементом любого приложения является форма — контейнер класса **TForm**, в котором размещаются другие визуальные и невизуальные компоненты. Так что рассмотрение методики проектирования было бы неполным без обзора некоторых методов и событий, присущих формам.

Обычно сколько-нибудь сложное приложение содержит несколько форм. Включение в проект новой формы осуществляется командой File | New | Form. Включите и в наше тестовое приложение еще одну форму. Это будет форма, которую пользователь сможет увидеть, выбрав в меню раздел О программе. Поместите на форму метки, в которых опишите назначение вашего приложения, укажите номер его версии, дайте информацию о себе как об авторе, и не забудьте, конечно, сообщить о ваших авторских правах на это блестящее приложение. Добавьте на форму кнопку, щелчком на которой форму можно будет закрыть. В обработчик щелчка на этой кнопке введите единственный оператор

```
Close;
```

Этот оператор закроет форму с помощью ее метода **Close**.

Размер формы, наверное, надо сделать небольшим, так как кроме этой информации на ней ничего не будет.

Назовите форму (свойств **Name**) **FAbout**, чтобы не путаться в дальнейшем в нескольких формах — позднее мы создадим еще одну вспомогательную форму. Сохраните модуль в файле с именем *UAbout*. Это имя мы будем использовать в дальнейшем.

По умолчанию все формы создаются автоматически при запуске приложения, и первая из введенных в приложение форм считается главной. Главная форма отличается от прочих рядом свойств. Во-первых, именно этой форме передается управление в начале выполнения приложения. Во-вторых, закрытие пользователем главной формы означает завершение выполнения приложения. В-третьих, главная форма так же, как и любая другая, может быть спроектирована невидимой, но если все остальные формы зарыты, то главная форма становится в любом случае видимой (иначе пользователь не смог бы продолжать работать с приложением и даже не смог бы его завершить).

Указанные выше условия, принятые по умолчанию (первая форма — главная, все формы создаются автоматически), могут быть изменены. Главной в вашем приложении может быть вовсе не та форма, которая была спроектирована первой. Не стоит также в общем случае все формы делать создаваемыми автоматически. В приложении могут быть предусмотрены формы (например, формы для установки различных опций), которые требуются далеко не в каждом сеансе работы с приложением. Было бы варварским расточительством создавать на всякий случай такие формы автоматически при каждом запуске приложения и занимать под них память.

Изменить принятые по умолчанию условия относительно форм можно в окне опций проекта, которое вызывается командой Project | Options. В открывшемся окне опций проекта надо выбрать страницу Forms, представленную на рис. 4.15.

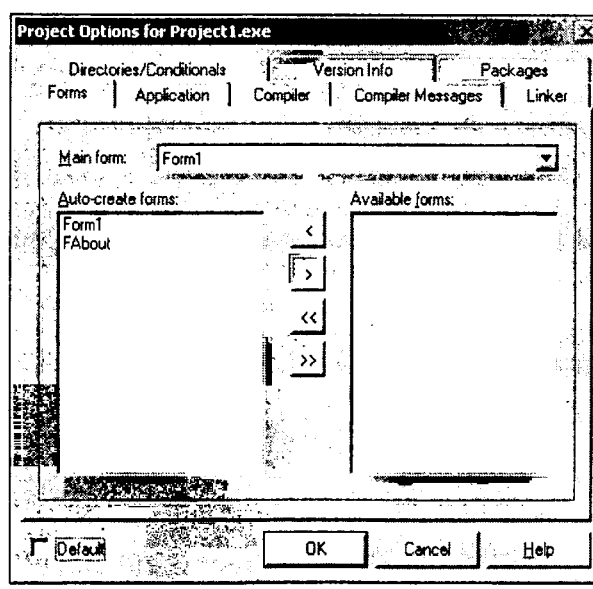


Рис. 4.15
Страница Forms окна опций проекта

В верхнем выпадающем списке Main forms можно выбрать главную форму среди имеющихся в проекте. Пользуясь двумя нижними окнами можно установить, какие формы должны создаваться автоматически, а какие не должны. Например, если надо исключить форму **FAbout** из списка автоматически создаваемых, то надо выделить ее в левом окне (Auto-create forms) и с помощью кнопки со стрелкой, направленной вправо, переместить в правое окно доступных форм (Available forms).

Для каждой автоматически создаваемой формы Delphi добавляет в головной файл программы соответствующий оператор ее создания методом **CreateForm**. Это можно увидеть, если выполнить команду Project | View Source и просмотреть появившийся файл проекта *.dpr*. В нашем примере он будет содержать следующие выполняемые операторы:

```
Application.Initialize;
Application.HelpFile := 'MYHELP.HLP';
Application.CreateForm(TForm1, Form1);
Application.CreateForm(TFAbout, FAbout);
Application.Run;
```

Первый оператор инициирует приложение. Второй задает ссылку на файл справки, который вы создали в разд. 4.10. Третий и четвертый операторы создают соответствующие формы, а последний начинает выполнение приложения. Как видите, каждый оператор использует свойства или методы того объекта **Application**, который был рассмотрен в разд. 4.11.

Для форм, которые были исключены из списка автоматически создаваемых, аналогичный метод **CreateForm** надо выполнить в тот момент, когда форма должна быть создана.

В нужный момент форму можно сделать видимой методами **Show** или **ShowModal**. Последний метод открывает форму как модальную. Это означает, что управление передается этой форме, и пользователь не может передать фокус другой форме данного приложения до тех пор, пока не закроет модальную форму. Более подробное описание модальных форм будет дано позднее в разд. 4.13.2.

Методом **Hide** форму в любой момент можно сделать невидимой. В этот момент в ней возникает событие **onHide**.

Необходимо помнить, что для выполнения методов **CreateForm**, **Show**, **ShowModal**, **Hide** и вообще для обмена любой информацией между формами модули соответствующих форм должны использовать друг друга. Например, если форма в модуле *Unit1* должна управлять формой в модуле *UAbout*, то в оператор **uses** модуля *Unit1* должно быть включено имя второго модуля *UAbout*. А если к тому же форма в модуле *UAbout* должна пользоваться какой-то информацией, содержащейся в модуле *Unit1*, то в оператор **uses** модуля *UAbout* должно быть включено имя первого модуля *Unit1*. Впрочем, проще не включать имена модулей в операторы **uses** вручную, а использовать команду File | Use Unit, которая автоматизирует этот процесс и гарантирует отсутствие циклических ссылок.

Исходя из сказанного, для вызова формы с информацией о программе в нашем приложении надо сделать следующее. Перейдите в Редакторе Кода в модуль главной формы *Unit1* и выполните команду File | Use Unit. В открывшемся диалоговом окне укажите, что хотите соединиться с модулем *UAbout*. Теперь можно написать обработчик действия **AAbout**, соответствующего показу формы с информацией о программе:

```
FAbout.ShowModal;
```

или

```
FAbout.Show;
```

Выполните приложение в обоих этих вариантах, чтбы наглядно увидеть различие между модальной и обычной формами.

Мы реализовали отображение информации о программе, но, честно говоря, не очень грамотно. Вряд ли пользователь будет непрерывно в каждом сеансе работы вызывать форму с информацией о программе, чтобы вспомнить, кто подарил ему такое приложение. Может быть, в каком-то сеансе он ее вызовет. А может, и нет. Так что бессмысленно все время хранить эту форму в памяти, тратя вычислительные ресурсы. Подобная форма должна создаваться только в тот момент, когда она потребовалась пользователю.

Для того чтобы реализовать такой грамотный подход, надо поступить следующим образом. Вызовите окно, показанное на рис. 4.15, и перенесите в нем форму **FAbout** из списка автоматически создаваемых в правое окно доступных форм (выше рассказывалось, как это можно сделать). Тогда оператор создания формы автоматически исключится в головного файла программы. Но его надо включить в обработчик действия **AAbout**, который в нашем примере может иметь вид:

```
procedure TForm1.AAboutExecute(Sender: TObject);
var F: TFAbout;
begin
 Application.CreateForm(TFAbout, F);
 F.ShowModal;
 F.Free;
end;
```

В этом коде вводится локальная переменная **F** типа **TFAbout**. Это тип формы с информацией о схеме. Затем методом **CreateForm** создается объект этой формы. Она предъявляется пользователю методом **ShowModal**, после чего объект уничтожается методом **Free**. Таким образом, объект формы существует, только пока пользователь работает с ним. Примерно по такой же схеме имеет смысл создавать многие диалоговые окна, которые вызываются эпизодически и не в каждом сеансе работы.

Теперь рассмотрим некоторые события, связанные с формами. В момент создания формы возникает событие **OnCreate**. Обработка этого события широко используется для настройки каких-то компонентов формы, создания списков и т.д. Мы уже не раз писали обработчики этого события. Перед тем, как форма станет видимой, в частности, при выполнении методов **Show** или **ShowModal**, возникает событие формы **OnShow**. Обработку события **OnShow** также можно использовать для настройки каких-то компонентов открываемой формы. Отличие от настройки компонентов в момент события **OnCreate** заключается в том, что событие **onCreate** наступает для каждой формы только один раз в момент ее создания, а события **OnShow** наступают каждый раз, когда форма делается видимой. Так что при этом в настройке можно использовать какую-то оперативную информацию, возникающую в процессе выполнения приложения.

Закрыть форму можно методом **Close**. При этом в закрывающейся форме возникает последовательность событий, которые можно обрабатывать. Их назначение — проверить возможность закрытия формы и указать, что именно подразумевается под закрытием формы. Проверка возможности закрытия формы необходима, например, для того, чтобы проанализировать, сохранил ли пользователь

документ, с которым он работал в данной форме и который изменял. Если не сохранил, приложение должно спросить его о необходимости сохранения и, в зависимости от ответа пользователя, сохранить документ, закрыть приложение без сохранения или вообще отменить закрытие.

Рассмотрим последовательность событий, возникающих при закрывании формы.

Первым возникает событие **OnCloseQuery**. В его обработчик передается как **var** (по ссылке) булева переменная **CanClose**, определяющая, должно ли продолжаться закрытие формы. По умолчанию **CanClose** равно **true**, что означает продолжение закрывания. Но если из анализа текущего состояния приложения или из ответа пользователя на запрос о закрытии формы следует, что закрывать ее не надо, параметру **CanClose** должно быть присвоено значение **false**. Тогда последующих событий, связанных с закрытием формы, не будет.

Введем в тестовое приложение, разрабатываемое в этой главе, обработчик события **OnCloseQuery**. В этом приложении пользователь может работать с характеристикой пользователя или с каким-то другим текстовым документом, и закрыть приложение, забыв сохранить то, что он изменил в тексте. Нельзя наказывать пользователя за такую забывчивость. Если в окне **RichEdit1** обнаружился измененный и несохраненный документ, надо предупредить об этом пользователя и спросить, что он хочет делать: сохранить документ, или нет. А может быть, пользователь вообще случайно щелкнул мышью на кнопке или разделе меню, закрывающем приложение. В этом случае надо дать ему возможность продолжать работу с приложением.

Вы уже знаете (см. разд. 4.9), что в окне **RichEdit** имеется свойство **Modified**, которое указывает, был ли изменен пользователем текст в этом окне с момента его последнего сохранения. Всю необходимую поддержку значений этого свойства вы уже обеспечили в разд. 4.9. А для запроса пользователю можно использовать метод **MessageBox**, описанный в разд. 4.11. Тогда обработчик события **OnCloseQuery** может иметь вид:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
                                var CanClose: Boolean);
begin
 if RichEdit1.Modified
  then
  case Application.MessageBox(
              'Документ не был сохранен.'#13'Сохранить?',
              'Подтвердите сохранение',
              MB_YESNOCANCEL+MB_ICONQUESTION) of
    IDYES:    ASaveAsExecute(Sender);
    IDCANCEL: CanClose:=false;
  end;
end;
```

В приведенном операторе вызывается методом **Application.MessageBox** диалоговое окно. Функция **MessageBox** возвращает результат, который указывает на реакцию пользователя в диалоговом окне. Значение **IDYES** возвращается, если пользователь нажал кнопку Да, значение **IDNO** возвращается при нажатии кнопки Нет, значение **IDCANCEL** — при нажатии кнопки Отмена или клавиши Esc. В нашем примере, если пользователь в диалоговом окне с запросом о сохранении ответит Да, то будет вызвана процедура **ASaveAsExecute** — обработчик действия сохранения в файле с заданным именем. Если пользователь нажмет кнопку Отмена

или клавишу Esc, параметр **CanClose** будет равен **false**, и приложение не завершится. Причем этот обработчик сработает при любой попытке пользователя закрыть приложение: нажатии в нем раздела меню Выход, нажатии кнопки системного меню в полосе заголовка окна и т.п. Наконец, если пользователь в окне запроса нажмет кнопку Нет, то сохранения не будет, параметр **CanClose** останется равным **true**, и приложение завершится.

Опробуйте свое модернизированное приложение, и убедитесь, что оно начало проявлять заботу о пользователе.

Теперь рассмотрим дальнейшую последовательность событий при закрывании формы. Если обработчик события **OnCloseQuery** отсутствует или если в его обработчике сохранено значение **true** параметра **CanClose**, то следом наступает событие **OnClose**. В обработчик этого события передается как **var** переменная **Action**, которой можно задавать значения:

caNone	Не закрывать форму. Это позволяет и в обработчике данного события еще отказаться от закрытия формы.
caHide	При этом значении (оно принято по умолчанию) закрыть форму будет означать сделать ее невидимой. Для пользователя она исчезнет с экрана, однако вся хранящаяся в форме информация сохранится. Она может использоваться другими формами или той же самой формой, если она снова будет сделана видимой.
caMinimize	При этом значении закрыть форму будет означать свернуть ее до пиктограммы. Как и в предыдущем случае, вся информация в форме будет сохранена.
caFree	При этом значении закрыть форму будет означать уничтожение формы и освобождение занимаемой ею памяти. Вся информация, содержащаяся в форме, будет уничтожена. Если эта форма в дальнейшем потребуется еще раз, ее надо будет создавать методом **CreateForm**.

Если в обработчике события **OnClose** задано значение **Action**, равное **caFree**, то при освобождении памяти возникает еще одно последнее событие — **OnDestroy**. Оно обычно используется для очистки памяти от тех объектов, которые автоматически не уничтожаются при закрытии приложения. Вы уже не раз использовали обработчики этого события для очистки памяти от ранее созданных объектов.

4.13.2 Модальные формы

Открытие форм как модальных используется в большинстве диалоговых окон. Модальная форма приостанавливает выполнение вызвавшей ее процедуры до тех пор, пока пользователь не закроет эту форму. Модальная форма не позволяет также пользователю переключить фокус курсором мыши на другие формы данного приложения, пока форма не будет закрыта. Т.е. пользователь должен выполнить предложенные ему действия прежде, чем продолжить работу.

Модальной может быть сделана любая форма, если она делается видимой методом **ShowModal**. Если та же самая форма делается видимой методом **Show**, то она не будет модальной.

Поведение модальной формы определяется ее основным свойством **ModalResult**. Это свойство доступно только во время выполнения приложения. При открытии формы методом **ShowModal** сначала свойство **ModalResult** равно нулю. Как только при обработке каких-то событий на форме свойству **ModalResult** будет присвоено положительное значение, модальная форма закроется. А значение ее свойства **ModalResult** можно будет прочитать как результат, возвращаемый методом **ShowModal**. Таким образом программа, вызвавшая модальную форму, может узнать, что сделал пользователь, работая с этой формой, например, на какой кнопке он щелкнул.

В Delphi предопределены некоторые константы, облегчающие трактовку результатов, полученных при закрытии модальной формы:

Численное значение ModalResult	Константа	Пояснение
0	**mrNone**	
1	**mrOk** или **idOK**	закрытие модальной формы нажатием кнопки OK
2	**mrCancel** или **idCancel**	закрытие модальной формы нажатием кнопки Cancel, или методом **Close**, или нажатием кнопки системного меню в полосе заголовка окна
3	**mrAbort** или **idAbort**	закрытие модальной формы нажатием кнопки Abort
4	**mrRetry** или **idRetry**	закрытие модальной формы нажатием кнопки Retry
5	**mrIgnore** или **idIgnore**	закрытие модальной формы нажатием кнопки Ignore
6	**mrYes** или **idYes**	закрытие модальной формы нажатием кнопки Yes
7	**mrNo** или **idNo**	закрытие модальной формы нажатием кнопки No
8	**mrAll**	закрытие модальной формы нажатием кнопки All
9	**mrNoToAll**	закрытие модальной формы нажатием кнопки No To All
10	**mrYesToAll**	закрытие модальной формы нажатием кнопки Yes To All

Все приведенные выше пояснения значений **ModalResult** (кроме значений 0 и 2) носят чисто условный характер. В своем приложении вы вольны трактовать ту или иную величину **ModalResult** и соответствующие константы как вам угодно.

Требуемые значения **ModalResult** можно задавать в обработчиках соответствующих событий в компонентах модальной формы. Однако при использовании кнопок можно обойтись и без подобных обработчиков. Дело в том, что кнопки **But-**

ton имеют свойство **ModalResult**, по умолчанию равное **mrNone**. Для кнопок, расположенных на модальной форме, значение этого свойства можно изменить и тогда не потребуется вводить каких-либо обработчиков событий при щелчке на них. Так что в примере формы с информацией о программе, рассмотренном в разд. 4.13.1, можно было бы не писать обработчик щелчка, а просто задать в свойстве **ModalResult** кнопки любое значение, например, **mrOk**. И форма закрывалась бы при щелчке на этй кнопке.

Давайте разработаем последнее усовершенствование нашего тестового приложения, на котором опробуем работу с модальными формами. Приложение обеспечивает доступ к списочному составу и характеристикам студентов или сотрудников. Такой доступ нельзя давать, кому попало. А тем более нельзя любому позволять изменять характеристики. Так что подобное приложение должно бы обеспечивать вход по паролю. Мы сделаем совсем простой вариант парольного входа. А вы далее своими силами можете его усовершенствовать. Заложим пароли доступа непосредственно в программу. Пусть, например, если пользователь ввел пароль «1», он допускается только к просмотру информации. Если он ввел пароль «2», он может не только просматривать, но и редактировать информацию. А при любом другом пароле пользователь вообще не допускается к работе с приложением.

Реальная форма запроса пароля должна предлагать пользователю ввести свое имя и пароль, сравнивать введенные значения с образцами, хранящимися где-то в системе в зашифрованном виде, при неправильном пароле давать возможность пользователю поправиться. Если пользователь так и не может ввести правильный пароль, форма должна закрыть приложение, не допустив к нему пользователя. При правильном пароле после закрытия формы запроса должна открыться главная форма приложения. Причем, если пользователь допущен только к просмотру информации, в окнах редактирования следует установить свойства **ReadOnly** (только чтение) в **true**.

Все это вы можете сделать позднее самостоятельно. А мы упростим задачу, чтобы не отвлекаться от главного — взаимодействия форм в приложении. Будем использовать всего два пароля, которые непосредственно укажем в соответствующих операторах программы. И не будем давать пользователю возможности исправить введенный пароль.

Добавьте в приложение новую форму. Назовите ее **FPSW** и сохраните ее модуль в файле с именем *UPSW*. Уменьшите размер формы до разумных пределов, поскольку она будет содержать всего одно окно редактирования и одну кнопку. В свойстве **Caption** формы или в отдельной метке напишите «Введите пароль и нажмите Enter». Эта надпись будет служить приглашением пользователю.

Поместите в центре формы окно редактирования **Edit**, в котором пользователь будет вводить пароль. Очистите его свойство **Text**. У окна **Edit** имеется свойство **PasswordChar**. Это символ, который отображается в окне вместо любого символа, вводимого пользователем. По умолчанию значение этого свойства равно **#0** — нулевому символу. Это означает нормальную работу с окном. Но если в это свойство ввести любой другой символ, например, звездочку «*», то именно этот символ будет отображаться в окне. И враг, подсматривающий через плечо пользователя, не сможет увидеть вводимый пароль. Так что измените соответствующим образом значение свойства **PasswordChar** окна **Edit1**.

Добавьте на форму кнопку, напишите в ее надписи «OK» и задайте в ее свойстве **ModalResult** любое значение, отличное от **mrNone** и от **mrCancel**. Например, задайте значение **mrOk**. При щелчке на этой кнопке форма будет закрываться.

На этом проектирование формы запроса пароля закончено. Теперь занесите в модуль главной формы ссылку на модуль *UPSW*. Осталось написать код, показывающий пользователю форму с паролем и анализирующий ответ пользователя. Мы хотим, чтобы пароль запрашивался до того момента, как пользователь увидит главную форму приложения. Так что для вызова формы пароля можно использовать событие **OnShow** главной формы. Обработчик этого события может иметь следующий вид:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 if (FPSW.ShowModal <> mrOk)
   then Close
   else
     begin
       if FPSW.Edit1.Text = '1'
        then RichEdit1.ReadOnly := true
        else if FPSW.Edit1.Text <> '2' then Close;
       FPSW.Free;
     end;
end;
```

Приведенный код анализирует значение свойства **ModalResult** формы запроса пароля. Значение этого свойства возвращает функция **FPSW.ShowModal**. Если результат не равен **mrOk**, значит пользователь закрыл форму не щелчком на ее кнопке OK. Тогда главная форма, а с ней вместе и приложение, закрываются методом **Close**. Затем анализируется значение пароля, занесенного в окно **Edit1** формы **FPSW**. Если пароль не равен «1» или «2», то главная форма закрывается методом **Close**. Если пароль равен «1», то окно **RichEdit1** делается доступным только для чтения. При пароле «1» или «2» выполнение главного приложения продолжается, но предварительно из памяти удаляется методом **Free** объект формы **FPSW**. Сама по себе эта форма в момент ее закрытия не уничтожается, поскольку по умолчанию закрыть форму — значит сделать ее невидимой. Уничтожать форму до этого момента было нельзя, так как при этом уничтожилась бы содержащаяся в ней информация — введенный пароль.

4.14 Некоторые итоги

В данной главе вы освоили методику построения хорошо структурированных приложений, доступных для модернизации и сопровождения. Они содержат все элементы, присущие любым полноценным приложениям: главное и контекстные меню, инструментальные панели, полосу состояния, справочную систему. Вы научились также создавать приложения с несколькими формами и получили немало сведений о работе с окном **RichEdit**.

В процессе изложения материала было описано множество событий. События формы описаны в разд. 4.13.1. А материал по событиям компонентов рассредоточен по разным разделам данной и предшествующих глав. Так что представляется полезным дать сводку рассмотренных событий компонентов и некоторых событий, которые не рассматривались, но полезны при разработке приложений Delphi.

Большинство оконных компонентов имеют следующие события:

Событие	Параметры	Описание
OnChange		Наступает в окнах редактирования, списках и в некоторых других компонентах при изменении состояния (текста) компонента программой или пользователем. Обработчик события может использоваться для определения реакции на это изменение.
OnClick	**Sender: TObject**	Наступает при щелчке мыши на компоненте, при нажатии клавиши пробела, когда кнопка или индикатор в фокусе, при нажатии горячих клавиш или клавиш быстрого доступа. Если компонент связан с объектом действия своим свойством **Action**, то событие наступает не в компоненте, а в объекте действия.
OnDblClick	**Sender: TObject**	Наступает, если пользователь осуществил двойной щелчок: дважды с коротким интервалом нажал и отпустил основную кнопку мыши, когда указатель мыши находился на компоненте. К одному и тому же компоненту нельзя написать обработчики событий **OnClick** и **OnDblClick**, поскольку первый из них всегда перехватит первый из щелчков.
OnEnter	**Sender: TObject**	Наступает в момент получения элементом фокуса (см. разд. 4.7, 4.9). Это событие не наступает при переключениях между формами или между приложениями. При переключениях между элементами, расположенными в разных контейнерах, например, на разных панелях, событие **OnEnter** сначала наступает для контейнера, а потом для содержащегося в нем элемента. Это противоположно последовательности событий **OnExit**, которые при переключении на компонент другого контейнера наступают сначала для компонента, а потом для контейнера.
OnExit	**Sender: TObject**	Наступает в момент потери элементом фокуса, в момент переключения фокуса на другой элемент (см. разд. 4.7, 4.9). Это событие не наступает при переключениях между формами или между приложениями. Значение свойства формы **ActiveControl** (активный компонент) изменяется прежде, чем происходит событие **OnExit**. При переключениях между элементами, расположенными в разных контейнерах, например, на разных панелях, событие **OnExit** сначала наступает для элемента, а потом для содержащего его контейнера. Это противоположно последовательности событий **OnEnter**, которые при переключении из другого контейнера на компонент данного контейнера наступают сначала для контейнера, а потом для компонента.

Событие	Параметры	Описание
OnKey-Down, OnKeyUp	**Sender: Object var Key: Word Shift:TShiftState**	Наступает, если компонент находится в фокусе, при нажатии (**OnKeyDown**) или отпускании (**OnKeyUp**) пользователем любой клавиши (см. разд. 3.2.5, 4.7, 4.9), включая функциональные и вспомогательные, такие, как Shift, Alt и Ctrl. Параметр **Key** определяет нажатую клавишу клавиатуры. Для не алфавитно-цифровых клавиш используются виртуальные коды API Windows. Эти коды определяют клавишу, но не различают символы в верхнем и нижнем регистрах и не различают символы кириллицы и латинские. Параметр **Shift** является множеством, которое может быть пустым или включать следующие элементы: **ssShift** — нажата клавиша Shift, **SsAlt** — нажата клавиша Alt, **SsCtrl** — нажата клавиша Ctrl.
OnKeyPress	**Sender: TObject var Key: Char**	Наступает, если компонент находится в фокусе, при нажатии пользователем клавиши символа. Параметр **Key** в обработчике этого события имеет тип **Char** и соответствует символу нажатой клавиши. При этом различаются символы в верхнем и нижнем регистрах, а также символы кириллицы и латинские. Клавиши, не отражаемые в кодах ASCII (функциональные клавиши и такие, как Shift, Alt, Ctrl), не вызывают этого события. Поэтому нажатие таких комбинаций клавиш, как, например, Shift-A, генерирует только одно событие **OnKeyPress**, при котором параметр **Key** равен «A». Для того чтобы распознавать клавиши, не соответствующие символам, или комбинации клавиш, надо использовать обработчики событий **OnKeyDown** и **OnKeyUp**. Поскольку параметр **Key** передается в обработчик как **var**, его можно изменять, передавая для дальнейшей стандартной обработки другой символ (см. разд. 2.8.5).
OnMouse-Down, OnMouse-Up	**Sender: TObject Button: TMouseButton Shift: TShiftState X, Y: Integer**	Наступают в моменты нажатия (**OnMouseDown**) и отпускания (**OnMouseUp**) пользователем кнопки мыши над компонентом (см. разд. 3.2.5, 4.7, 4.9). Если требуется различная обработка событий в зависимости от того, какая кнопка мыши нажата или какая нажата вспомогательная клавиша, можно анализировать параметры **Button** и **Shift**. Значения параметра **Button** определяют, какая кнопка мыши нажата: **mbLeft** — левая, **mbRight** — правая, **mbMiddle** — средняя. Параметр **Shift** представляет собой множество, содержащее обозначения нажатой кнопки (**ssLeft**, **ssRight**, **ssMiddle** — левая, правая, средняя) и обозначения нажатых одновременно с этим вспомогательных клавиш Shift, Alt, Ctrl (**ssShift**, **ssAlt**, **ssCtrl**). Параметры **X** и **Y** определяют координаты указателя мыши в клиентской области компонента.

Событие	Параметры	Описание
OnMouse-Move	**Sender: TObject** **Shift: TShiftState** **X, Y: Integer**	Наступает при перемещении курсора мыши над компонентом. Параметр **Shift** представляет собой множество, содержащее обозначения нажатой кнопки (**ssLeft**, **ssRight**, **ssMiddle** — левая, правая, средняя) и обозначения нажатых одновременно с этим вспомогательных клавиш Shift, Alt, Ctrl (**ssShift**, **ssAlt**, **ssCtrl**). Параметры **X** и **Y** определяют координаты указателя мыши в клиентской области компонента.

Из приведенных в таблице событий, возможно, некоторый дополнительный комментарий требуется для событий **OnKeyDown** и **OnKeyUp**. Передаваемый в обработчики этих событий параметр **Key** является целым числом, указывающим виртуальный код. В качестве примера в приведенной ниже таблице даны коды некоторых клавиш. Полную таблицу кодов вы можете посмотреть во встроенной справке Delphi или в справке [3].

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по функции ord
F1	112	$70	**VK_F1**	
Enter	13	$0D	**VK_RETURN**	
Shift	16	$10	**VK_SHIFT**	
Ctrl	17	$11	**VK_CONTROL**	
Alt	18	$12	**VK_MENU**	
Esc	27	$1B	**VK_ESCAPE**	
0,)	48	$30		**ord('0')**
1, !	49	$31		**ord('1')**
n, N, т, Т	78	$4E		**ord('N')**
y, Y, н, Н	89	$59		**ord('Y')**

Параметр **Key** определяет именно клавишу, а не символ. Например, один и тот же код соответствует прописному и строчному символам «Y» и «y». Если, как это обычно бывает, в русской клавиатуре этой клавише соответствуют символы кириллицы «Н» и «н», то их код будет тем же самым. Различить прописные и строчные символы или символы латинские и кириллицы невозможно.

Проверять нажатую клавишу можно, сравнивая **Key** с целым десятичным кодом клавиши, приведенным во втором столбце таблицы. Например, реакцию на нажатие пользователем клавиши Enter можно оформить оператором:

```
if (Key = 13) then ... ;
```

Можно сравнивать **Key** и с шестнадцатеричным эквивалентом кода, приведенным в третьем столбце таблицы. Например, приведенный выше оператор можно записать в виде:

```
if (Key = $0D) then ... ;
```

Для клавиш, которым не соответствуют символы, введены также именованные константы, которые облегчают написание программы, поскольку не требуют помнить численные коды клавиш. Например, приведенный выше оператор можно записать в виде:

```
if (Key = VK_RETURN) then ... ;
```

Для клавиш символов и цифр можно производить проверку сравнением с десятичным или шестнадцатеричным кодом, но это не очень удобно, так как трудно помнить коды различных символов. Другой путь — воспользоваться функцией **ord**, определяющей код символа. Коды латинских символов в верхнем регистре совпадают с виртуальными кодами, используемыми в параметре **Key**. Поэтому, например, если вы хотите распознать клавишу, соответствующую символу «Y», вы можете написать:

```
if (Key = ord('Y') then ... ;
```

В подобных операторах можно использовать только латинские символы в верхнем регистре. Если вы напишете **ord('y')** или захотите написать русские символы, соответствующие этой клавише — **ord('Н')** или **ord('н')**, то оператор не сработает.

Для распознавания символов, а не клавиш, следует использовать событие **OnKeyPress**. Соответствующие примеры рассматривались в разд. 2.8.5.

В заключение приведем пример распознавания комбинации клавиш. Пусть вы хотите, например, распознать комбинацию Alt-X. Для этого вы можете написать оператор:

```
if((Key = ord('X'))and (ssAlt in Shift)) then ... ;
```

4.15 Проверьте себя

4.15.1 Вопросы для самопроверки

1. Укажите последовательность действий при создании хорошо структурированного приложения, облегчающего его сопровождение?
2. Почему начинать проектирование целесообразно с формирования списка изображений **ImageList** и зачем надо связывать с этим списком диспетчер **ActionList** прежде, чем начинается формирование в нем списка стандартных действий?
3. Почему при загрузке в **ImageList** изображений для кнопок их надо разделять на несколько отдельных изображений?
4. Надо ли писать обработчики событий **OnExecute** для стандартных действий?
5. Какое свойство компонентов (например, инструментальной панели) надо установить, чтобы всплывали ярлычки с текстами, заданными свойством **Hint**?
6. Для чего в надписях (**Caption**) требуется выделять подчеркнутый символ?
7. Какую роль играет свойство **GroupIndex** действий, разделов меню, кнопок панели **ToolBar** и некоторых других компонентов?
8. В чем различие сносок **K** и **A** в файле тем справки?
9. В чем различие метода **Close** формы и метода **Terminate** объекта **Application**?

10. Какие формы и зачем надо исключать из числа автоматически создаваемых? Как объекты таких форм должны создаваться в приложении?
11. Чем различаются методы формы **Show** и **ShowModal**?
12. Пусть вы вызываете из главной формы другую как модальную. Эта вызываемая форма имеет три кнопки. Как можно узнать в главной форме, какую из них нажал пользователь, работая с модальной формой?

4.15.2 Задачи

1. Замените в некоторых или во всех приложениях, разработанных в предыдущих главах, вызовы функции **ShowMessage** на вызовы **Application.MessageBox**.
2. Введите в некоторые или во все приложения, разработанные в предыдущих главах, списки пиктограмм, списки действий, меню и инструментальные панели.
3. Разработайте к каким-то из приложений, созданных в предыдущих главах, справочные файлы. Введите в приложения разделы меню, вызывающие справки, и обеспечьте контекстно-зависимые темы справок, связанные с отдельными компонентами.

Глава 5

Разработка графического интерфейса пользователя

В этой главе:

- *вы узнаете о требованиях, предъявляемых к интерфейсу приложений Windows, и научитесь удовлетворять эти требования*
- *освоите проектирование окон различного стиля и узнаете, когда какой стиль следует использовать*
- *познакомитесь с компонентами Delphi, позволяющими разнообразить и сделать более удобным интерфейс ваших приложений*
- *научитесь использовать в своих приложениях графику, которая сделает ваши программы красочными и удобными для пользователя*

5.1 Требования к интерфейсу пользователя

5.1.1 Общие сведения

Под графическим интерфейсом пользователя (Graphical User Interface — GUI) подразумевается внешний вид приложения и расположенные в окне управляющие элементы, с которыми работает пользователь.

Delphi предоставляет разработчику приложения широкие возможности быстрого и качественного проектирования графического интерфейса пользователя — различных окон, кнопок, меню и т.д. Так что разработчик может в полной мере проявить свою фантазию. Но полеты фантазии очень полезно ограничивать. Есть определенные принципы построения графического интерфейса пользователя, и пренебрегающий ими обречен на то, что его приложение будет выглядеть чужеродным объектом в среде Windows.

Для пользователя одним из принципиальных преимуществ работы с Windows является то, что большинство имеющихся приложений выглядят и ведут себя сходным образом. После того как вы поработаете с несколькими приложениями, вы обнаружите, что можете заранее почти наверняка сказать, где можно найти ту или иную функцию в программе, которую только что приобрели, или какие быстрые клавиши надо использовать для выполнения тех или иных операций.

Фирма Microsoft предложила спецификации для разработки программного обеспечения Windows, направленные на то, чтобы пользователь не тратил время на освоение нюансов пользовательского интерфейса новой программы, чтобы он смог как можно скорее продуктивно применять ваше приложение. Эти спецификации образуют основу программы логотипа Windows, проводившейся Microsoft. Чтобы вы могли поставить на свой программный продукт штамп «Разработано для Windows ...», ваша программа должна удовлетворять определенным критериям.

Когда вы видите этот логотип на каком-то изделии, аппаратном или программном, вы можете быть уверены, что оно будет работать нормально в среде Windows.

Конечно, вряд ли вы будете очень озабочены приобретением официального права на логотип Windows, если только не разработали сногсшибательную программу широкого применения, которую надеетесь успешно продавать на международном рынке. Но прислушаться к рекомендациям по разработке графического интерфейса пользователя в любом случае полезно. Они основаны на психофизиологических особенностях человека и существенно облегчат жизнь будущим пользователям вашей программы, увеличат производительность их работы.

5.1.2 Цветовое решение приложения

Цвет является мощным средством воздействия на психику человека. Именно поэтому обращаться с ним надо очень осторожно. Неудачное цветовое решение может приводить к быстрому утомлению пользователя, работающего с вашим приложением, к рассеиванию его внимания, к частым ошибкам. Слишком яркий или неподходящий цвет может отвлекать внимание пользователя или вводить его в заблуждение, создавать трудности в работе. А удачно подобранная гамма цветов, осмысленные цветовые акценты снижают утомляемость, сосредоточивают внимание пользователя на выполняемых в данный момент операциях, повышают эффективность работы. С помощью цвета вы можете на что-то намекнуть или привлечь внимание к определенным областям экрана. Цвет может также связываться с различными состояниями объектов.

Надо стремиться использовать ограниченный набор цветов и уделять внимание их правильному сочетанию. Расположение ярких цветов, таких, как красный, на зеленом или черном фоне затрудняет возможность сфокусироваться на них. Не рекомендуется использовать дополнительные цвета. Обычно наиболее приемлемым цветом для фона будет нейтральный цвет, например, светло-серый (используется в большинстве продуктов Microsoft). Помните также, что яркие цвета кажутся выступающими из плоскости экрана, в то время как темные как бы отступают вглубь.

Цвет не должен использоваться в качестве основного средства передачи информации. Можно использовать различные панели, формы, штриховку и другие методики выделения областей экрана. Microsoft даже рекомендует разрабатывать приложение сначала в черно-белом варианте, а уже потом добавлять к нему цвет.

Нельзя также забывать, что восприятие цвета очень индивидуально. А по оценке Microsoft девять процентов взрослого населения вообще страдают нарушениями цветовосприятия. Поэтому не стоит навязывать пользователю свое видение цвета, даже если оно безукоризненно. Надо предоставить пользователю возможность самостоятельной настройки на наиболее приемлемую для него гамму. К тому же, не стоит забывать, что может быть кто-то захочет использовать вашу программу на машине с монохромным монитором.

Посмотрим теперь, как задаются цвета приложения, разрабатываемого в Delphi. Большинство компонентов имеют свойство **Color** (цвет), который вы можете изменять в Инспекторе Объектов при проектировании или программно во время выполнения (если хотите, чтобы цвета в различных режимах работы приложения были разные). Щелкнув на этом свойстве в Инспекторе Объектов, вы можете увидеть в выпадающем списке большой набор предопределенных констант, обозначающих цвета. Все их можно разбить на две группы: статические цвета типа **clBlack** — чер-

ный, **clGreen** — зеленый и т.д., и системные цвета типа **clWindow** — текущий цвет фона окон, **clMenuText** — текущий цвет текста меню и т.д.

Статические цвета вы выбираете сами, и они будут оставаться неизменными при работе приложения на любом компьютере. Это не очень хорошо, поскольку пользователь не сможет адаптировать вид вашего приложения к своим потребностям. При выборе желательной ему цветовой схемы пользователь может руководствоваться самыми разными соображениями: начиная с практических (например, он может хотеть установить черный фон, чтобы экономить энергию батареи), и кончая эстетическими (он может предпочитать, например, шкалу оттенков серого, потому что не различает цвета). Все это он не может делать, если вы задали в приложении статические цвета. Но уж если по каким-то соображениям вам надо их задать, старайтесь использовать базовый набор из 16 цветов. Если вы попытаетесь использовать 256 (или, что еще хуже, 16 миллионов) цветов, это может замедлить работу вашего приложения, или оно будет выглядеть плохо на машине пользователя с 16 цветами. К тому же подумайте (а, как правило, это надо проверить и экспериментально), как будет выглядеть ваше приложение на монохромном дисплее.

Исходя из изложенных соображений, везде, где это имеет смысл, следует использовать для своего приложения палитру системных цветов. Это те цвета, которые устанавливает пользователь при настройке Windows. Когда вы создаете новую форму или размещаете на ней компоненты, Delphi автоматически присваивает им цвета в соответствии со схемой цветов, установленной в Windows. Конечно, вы будете менять эти установки по умолчанию. Но если при этом вы используете соответствующие константы системных цветов, то, когда пользователь изменит цветовую схему оформления экрана Windows, ваше приложение также будет соответственно меняться, и не будет выпадать из общего стиля других приложений.

Хороший стиль программирования

Не злоупотребляйте в приложении яркими цветами. Пестрое приложение — обычно признак дилетантизма разработчика, утомляет пользователя, рассеивает его внимание. Как правило, используйте системные цвета, которые пользователь может перестраивать по своему усмотрению. Из статических цветов обычно имеет смысл использовать только clBlack — черный, clWhite — белый и clRed — красный цвет предупреждения об опасности.

Единству цветового решения отдельных частей экрана способствует также использование свойства **ParentColor**. Если это свойство установлено в **true**, то цвет компонента соответствует цвету содержащего его контейнера или формы. Это обеспечивает единство цветового решения окна и, кроме того, позволяет программно изменять цвет сразу группы компонентов, если вы, например, хотите, чтобы их цвет зависел от текущего режима работы приложения. Для такого группового изменения достаточно изменить только цвет контейнера.

5.1.3 Шрифты текстов

Шрифт надписей и текстов компонентов Delphi задается свойством **Font**, имеющим множество подсвойств. Кроме того, в компонентах имеется свойство **ParentFont**. Если это свойство установлено в **true**, то шрифт данного компонента берется из свойство **Font** его родительского компонента — панели или формы, на которой

расположен компонент. Использование свойств **ParentFont** и **ParentColor** помогает обеспечить единообразие отображения компонентов в окне приложения.

По умолчанию для всех компонентов Delphi задается имя шрифта **MS Sans Serif** и размер — 8. Константа множества символов **Charset** задается равной **DEFAULT_CHARSET**. Последнее означает, что шрифт выбирается только по его имени и размеру. Если описанный шрифт недоступен в системе, то Windows заменит его другим шрифтом.

Чаще всего эти установки по умолчанию можно не изменять. Конечно, никто не мешает задать для каких-то компонентов другой размер шрифта или атрибуты типа полужирный, курсив и т.д. Но изменять имя шрифта для вашего приложения надо с определенной осторожностью. Дело в том, что шрифт, установленный на вашем компьютере, не обязательно должен иметься и на компьютере пользователя. Поэтому использование какого-то экзотического шрифта может привести к тому, что пользователь, запустив ваше приложение на своем компьютере, увидит вместо русского текста абракадабру на никому не понятном языке. Чтобы избежать таких казусов, вам придется прикладывать к своему приложению еще и файлы использованных шрифтов и пояснять пользователю, как он должен установить их на своем компьютере, если они там отсутствуют. Или вводить автоматическую проверку и установку нужных вам шрифтов в установочную программу вашего приложения.

Использование шрифтов по умолчанию: **System** или **MS Sans Serif**, чаще всего позволяет избежать подобных неприятностей. Впрочем, увы, не всегда. Если вы используете для надписей русские тексты, то при запуске приложения на компьютере с нерусифицированным Windows иногда возможны неприятности. Для подобных случаев все-таки полезно приложить файлы использованных шрифтов к вашей программе. Вы можете при установке вашего приложения узнать, имеется ли на компьютере пользователя нужный шрифт, например, с помощью следующего кода:

```
if (Screen.Fonts.IndexOf('Arial') = -1)
  then ...
```

В этом коде многоточием обозначены действия, которые надо выполнить, если нужного шрифта (в примере — Arial) на компьютере нет. Эти действия могут заключаться в копировании файлов шрифта с установочной дискеты или CD-ROM на компьютер пользователя.

Другой выход из положения — ввести в приложение команду выбора шрифта пользователем. Это позволит ему выбрать подходящий шрифт из имеющихся в его системе. Осуществляется подобный выбор с помощью стандартного диалога, оформленного в виде компонента **FontDialog** (см. разд. 3:2.3).

5.1.4 Меню

Практически любое приложение должно иметь меню, поскольку именно меню дает наиболее удобный доступ к функциям программы. Существует несколько различных типов меню: главное меню с выпадающими списками разделов, каскадные меню, в которых разделу первичного меню ставится в соответствие список подразделов, и всплывающие или контекстные меню, появляющиеся, если пользователь щелкает правой кнопкой мыши на каком-то компоненте.

Все операции по проектированию меню в Delphi подробно рассмотрены в разд. 4.8. А в данном разделе мы обсудим требования, предъявляемые к меню приложений Windows.

Основное требование к меню — их стандартизация. Это требование относится ко многим аспектам меню: месту размещения заголовков меню и их разделов, форме самих заголовков, клавишам быстрого доступа, организации каскадных меню. Цель стандартизации — облегчить пользователю работу с приложением. Надо, чтобы пользователю не приходилось думать, в каком меню и как ему надо открыть или сохранить файл, как ему получить справку, как работать с буфером обмена Clipboard и т.д. Для осуществления всех этих операций у пользователя, поработавшего хотя бы с несколькими приложениями Windows, вырабатывается стойкий автоматизм действий и недопустимо этот автоматизм ломать.

Начнем рассмотрение требований с размещения заголовков меню. Конечно, состав меню зависит от конкретного приложения. Но размещение общепринятых разделов должно быть стандартизированным (см. в гл. 4 рис. 4.1 и 4.6). Все пользователи уже привыкли, что меню Файл размещается слева в полосе главного меню, а, например, раздел Справка — справа.

По возможности стандартным должно быть и расположение разделов в выпадающих меню. Например, раздел Выход (рис. 4.6 а) всегда размещается последним в меню Файл, а раздел информации о версии программы О программе — последним в справочном меню (рис. 4.6 е).

Группы функционально связанных разделов отделяются в выпадающих меню разделителями (рис. 4.6 а, е).

Названия разделов меню должны быть привычными пользователю. Если вы не знаете, как назвать какой-то раздел, не изобретайте свое имя, а попытайтесь найти аналогичный раздел в какой-нибудь русифицированной программе Microsoft для Windows. Названия должны быть краткими и понятными. Не используйте фраз, да и вообще больше двух слов, поскольку это перегружает экран и замедляет выбор пользователя. Названия разделов должны начинаться с заглавной буквы. Применительно к английским названиям разделов существует требование, чтобы каждое слово тоже начиналось с заглавной буквы. Но применительно к русским названиям это правило не применяется.

Названия разделов меню, связанных с вызовом диалоговых окон, должны заканчиваться многоточием (рис. 4.6 а, в), показывающим пользователю, что при выборе этого раздела ему предстоит установить в диалоге еще какие-то параметры.

В каждом названии раздела должен быть выделен подчеркиванием символ, соответствующий клавише быстрого доступа к разделу (клавиша Alt плюс подчеркнутый символ). Хотя вряд ли такими клавишами часто пользуются, но традиция указания таких клавиш незыблема.

Многим разделам могут быть поставлены в соответствие горячие клавиши, позволяющие обратиться к команде данного раздела, даже не заходя в меню. Комбинации таких горячих клавиш должны быть традиционными. Например, команды вырезания, копирования и вставки фрагментов текста практически всегда имеют горячие клавиши Ctrl-X, Ctrl-C и Ctrl-V соответственно. Заданные сочетания клавиш отображаются в заголовках соответствующих разделов меню.

Многие разделы меню желательно снабжать пиктограммами, причем пиктограммы для стандартных разделов должны быть общепринятыми, знакомыми пользователю.

Не все разделы меню имеют смысл в любой момент работы пользователя с приложением. Например, если в приложении не открыт ни один документ, то бессмысленно выполнять команды редактирования в меню Правка. Если в тексте документа ничего не изменялось, то бессмысленным является раздел этого меню Отменить,

отменяющий последнюю команду редактирования. Такие меню и отдельные разделы должны делаться временно недоступными или невидимыми. Это осуществляется заданием значения **false** свойствам раздела **Enabled** или **Visible** соответственно. Различие между недоступными и невидимыми разделами в том, что недоступный раздел виден в меню, но отображается серым цветом, а невидимый раздел просто исчезает из меню, причем нижележащие разделы смыкаются, занимая его место. Выбор того или иного варианта — дело вкуса и удобства работы пользователя. Вероятно, целиком меню лучше делать невидимыми, а отдельные разделы — недоступными. Например, пока ни один документ не открыт, меню Правка можно сделать невидимым, чтобы он не отвлекал внимания пользователя. А раздел Отменить этого меню в соответствующих ситуациях лучше делать недоступным, чтобы пользователь видел, что такой раздел в меню есть и им можно будет воспользоваться в случае ошибки редактирования.

5.2 Окна приложений

5.2.1 Стиль окон приложения

Основным элементом любого приложения является форма — контейнер, в котором размещаются другие визуальные и невизуальные компоненты. С точки зрения пользователя форма — это окно, в котором он работает с приложением.

К внешнему виду окон в Windows предъявляются определенные требования. К счастью, Delphi автоматически обеспечивает стандартный для Windows вид окон вашего приложения. Но вам надо продумать и указать, какие кнопки в полосе системного меню должны быть доступны в том или ином окне, должно ли окно допускать изменение пользователем его размеров, каким должен быть заголовок окна. Все эти характеристики окон обеспечиваются установкой и управлением свойствами формы.

Свойство **BorderStyle** определяет общий вид окна и операции с ним, которые разрешается выполнять пользователю. Это свойство может принимать следующие значения:

bsSizeable	Обычный вид окна Windows с полосой заголовка, с возможностью для пользователя изменять размеры окна с помощью кнопок в полосе заголовка или с помощью мыши, потянув за какой-либо край окна. Это значение **BorderStyle** задается по умолчанию.
bsDialog	Неизменяемое по размерам окно. Типичное окно диалогов.
bsSingle	Окно, размер которого пользователь не может изменить, потянув курсором мыши край окна, но может менять кнопками в полосе заголовка.
bsToolWindow	То же, что **bsSingle**, но с полосой заголовка меньшего размера.
bsSizeToolWin	То же, что **bsSizeable**, но с полосой заголовка меньшего размера и с отсутствием в ней кнопок изменения размера.
bsNone	Без полосы заголовка. Окно не только не допускает изменения размера, но и не позволяет переместить его по экрану.

Свойство **BorderIcons** определяет набор кнопок, которые имеются в полосе заголовка. Множество кнопок задается элементами:

bySistemMenu	кнопка системного меню — это кнопка с крестиком, закрывающая окно
byMinimize	кнопка Свернуть, сворачивает окно до пиктограммы
byMaximize	кнопка Развернуть, разворачивает окно на весь экран
byHelp	кнопка справки

Следует отметить, что не все кнопки могут появляться при любых значениях **BorderStyle**.

На рис. 5.1 представлен вид окон форм во время выполнения при некоторых сочетаниях свойств **BorderStyle** и **BorderIcons.** Для создания диалоговых окон обычно используется стиль заголовка **bsDialog** (формы **Form3** и **Form4** на рис. 5.1), причем в этих окнах можно исключить кнопку системного меню (форма **Form4**), и в этом случае пользователь не может закрыть окно никакими способами, кроме как выполнить какие-то предписанные ему действия на этой форме. При стиле **bsNone** пользователь не может изменить ни размер, ни положение окна на экране. Формы **Form3**, **Form6** и **Form8** внешне различаются только размером полосы заголовка, уменьшенным в двух последних формах. Но между ними есть и принципиальное различие: размер формы **Form6** (стиль заголовка **bsSizeToolWin**) пользователь может изменить, потянув курсором мыши за край окна. Для форм **Form3** и **Form8** это невозможно. Обратите также внимание, что в формах **Form6** и **Form8** задание кнопок свертывания и развертывания окна никак не влияет на его вид: эти кнопки просто не могут появляться в этих стилях полос заголовков окон.

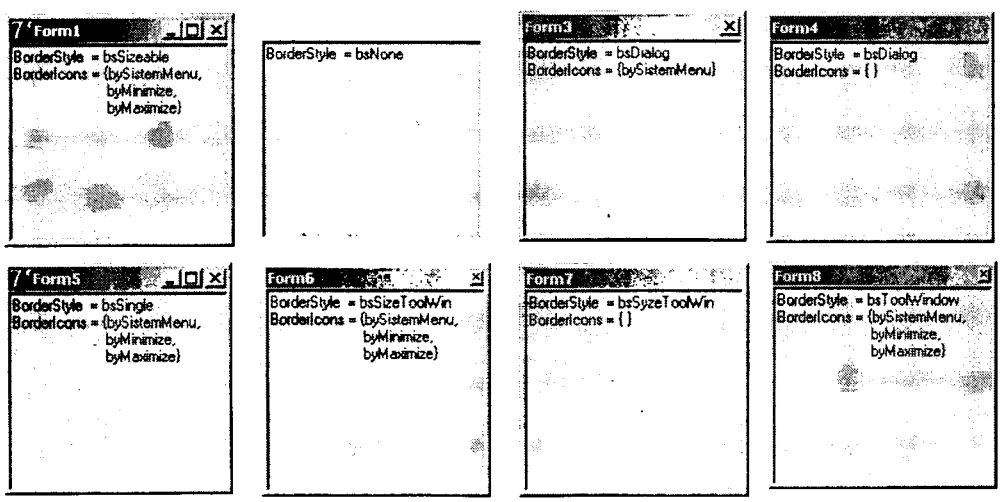


Рис. 5.1
*Формы при разных сочетаниях свойств **BorderStyle** и **BorderIcons***

Хороший стиль программирования

Без особой необходимости не делайте окна приложения с изменяемыми пользователем размерами. При изменении размеров, если не применены специальные приемы, описанные в разд. 5.3, нарушается компоновка окна, и пользователь ничего не выигрывает от своих операций с окном. Окно имеет смысл делать с изменяемыми размерами, только если это позволяет пользователю изменять полезную площадь каких-то расположенных в нем компонентов отображения и редактирования информации: текстов, изображений, списков и т.п.

Хороший стиль программирования

Для основного окна приложения с неизменяемыми размерами наиболее подходящий стиль — BorderStyle = bsSingle с исключением из числа доступных кнопок кнопки Развернуть (BorderIcons.byMaximize = false). Это позволит пользователю сворачивать окно, восстанавливать, но не даст возможности развернуть окно на весь экран или изменить размер окна.

Хороший стиль программирования

Для вторичных диалоговых окон наиболее подходящий стиль — BorderStyle = bsDialog. Можно также использовать BorderStyle = bsSingle, одновременно исключая из числа доступных кнопок кнопку Развернуть (задавая BorderIcons.byMaximize = false). Это позволит пользователю сворачивать диалоговое окно, если оно заслоняет на экране что-то нужное ему, восстанавливать окно, но не даст возможности развернуть окно на весь экран или изменить размер окна.

Предупреждение

Избегайте, как правило, стиля BorderStyle = bsNone. Невозможность переместить окно может создать пользователю трудности, если окно заслонит на экране что-то интересующее пользователя.

Свойство формы **WindowState** определяет вид, в котором окно первоначально предъявляется пользователю при выполнении приложения. Оно может принимать значения:

wsNormal	нормальный вид окна (это значение **WindowState** используется по умолчанию)
wsMinimized	окно свернуто
wsMaximized	окно развернуто на весь экран

Если свойство **WindowState** имеет значение **wsNormal** или пользователь, манипулируя кнопками в полосе заголовка окна, привел окно в это состояние, то положение окна при запуске приложения определяется свойством **Position**, которое может принимать значения:

poDesigned	Первоначальные размеры и положение окна во время выполнения те же, что во время проектирования. Это значение принимается по умолчанию, но обычно его следует изменить.
poScreenCenter, poDesktopCenter	Окно располагается в центре экрана. Размер окна тот, который был спроектирован.
poDefault	Местоположение и размер окна определяет Windows, учитывая размер и разрешение экрана. При последовательных показах окна его положение сдвигается немного вниз и вправо.
poDefaultPosOnly	Местоположение окна определяет Windows. При последовательных показах окна его положение сдвигается немного вниз и вправо. Размер окна — спроектированный.
poDefaultSizeOnly	Размер окна определяет Windows, учитывая размер и разрешение экрана. Положение окна — спроектированное.
poMainFormCenter	Это значение предусмотрено, начиная с Delphi 5. Окно располагается в центре главной формы. Размер окна тот, который был спроектирован. Используется только для вторичных форм. Для главной формы действует так же, как **poScreenCenter**.
poOwnerFormCenter	Это значение предусмотрено, начиная с Delphi 6. Окно располагается в центре формы, указанной как владелец окна в свойстве **Owner**. Размер окна тот, который был спроектирован. Если свойство **Owner** указывает не форму, действует как **poMainFormCenter**.

Хороший стиль программирования

Обычно целесообразно для главной формы приложения задавать значение Position равным poScreenCenter или poDefault. И только в сравнительно редких случаях, когда на экране при выполнении приложения должно определенным образом располагаться несколько окон, имеет смысл оставлять значение poDesigned, принимаемое по умолчанию.

Если выбранное значение свойства **Position** предусматривает выбор размера формы самим Windows по умолчанию, то на этот выбор влияют свойства **PixelsPerInch** и **Scaled**. По умолчанию первое из них задается равным количеству пикселов на дюйм в системе, второе установлено в **false**. Если задать другое число пикселов на дюйм, то свойство **Scaled** автоматически становится равным **true**. В этом случае при запуске приложения размер формы будет изменяться в соответствии с пересчетом заданного числа пикселов на дюйм к реальному числу пикселов на дюйм в системе (но только при разрешающем это значении свойства **Position**).

Одно из основных свойств формы — **FormStyle**. Значение **fsNormal** соответствует окну обычного приложения. Значение **FormStyle** = **fsStayOnTop** делает окно всегда остающимся на экране поверх остальных окон не только данного приложения, но и всех других приложений, в которые может перейти пользователь.

Хороший стиль программирования

Используйте стиль FormStyle = fsStayOnTop для отображения окон сообщений пользователю о каких-то аварийных ситуациях.

5.2.2 Компоновка форм

Каждое окно, которое вы вводите в свое приложение, должно быть тщательно продумано и скомпоновано. Удачная компоновка может стимулировать эффективную работу пользователя, а неудачная — рассеивать внимание, отвлекать, заставлять тратить лишнее время на поиск нужной кнопки или индикатора.

Управляющие элементы и функционально связанные с ними компоненты экрана должны быть зрительно объединены в группы, заголовки которых коротко и четко поясняют их назначение. Такое объединение позволяют осуществлять различные панели. Посмотрите панель **Panel**, расположенную в библиотеке на странице Standard. Поварьируйте ее параметрами **BevelInner**, **BevelOuter**, **BevelWidth**, **BorderStyle**, **BorderWidth**, и вы увидите, какие зрительный эффекты может обеспечить эта панель. Но, пожалуй, чаще всего для цели зрительного объединения функционально связанных компонентов используется панель **GroupBox**, расположенная в библиотеке на странице Standard. Она имеет встроенную рамку с надписью, которая может пояснять назначение расположенных на панели компонентов. Текст надписи задается свойством **Caption**. На рис. 5.2 в качестве примера представлено оформление с помощью **GroupBox** приложения, разработанного ранее в гл. 3 в разд. 3.1.5.4 (рис. 3.3). Думается, что вариант рис. 5.2 удобнее и нагляднее варианта, показанного на рис. 3.3.

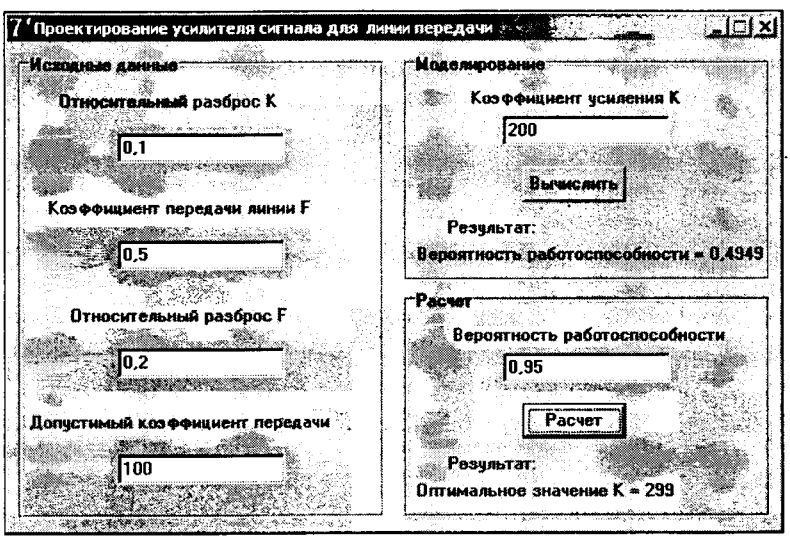


Рис. 5.2
*Использование панелей **GroupBox** в окне приложения, показанного ранее на рис. 3.3*

Внутри панелей, на которых размещаются компоненты, надо продумывать их компоновку, как с точки зрения эстетики, так и с точки зрения визуального отражения их взаимоотношений. Например, если имеется кнопка, которая разворачивает окно списка, то эти два компонента должны быть визуально связаны между собой: размещены на одной панели и в непосредственной близости друг от друга. Если же ваш экран представляет собой случайные скопления кнопок, то именно так он и будет восприниматься. И в следующий раз пользователь не захочет пользоваться вашей программой.

Каждое окно должно иметь некоторую центральную тему, которой подчиняется его композиция. Пользователь должен понимать, для чего предназначено данное окно и что в нем наиболее важно. При этом недопустимо перегружать окно большим числом органов управления, ввода и отображения информации. В окне должно отображаться главное, а все детали и дополнительную информацию можно отнести на вспомогательные окна. Для этого полезно вводить в окно кнопки с надписью «Больше...», многоточие в которой показывает, что при нажатии этой кнопки откроется вспомогательное окно с дополнительной информацией (как показывать пользователю вспомогательные окна см. в разд. 4.13.2).

Помогают также разгрузить окно многостраничные компоненты с закладками. Они дают возможность пользователю легко переключаться между разными по тематике страницами, на каждой из которых имеется необходимый минимум информации. Одним из таких компонентов является **PageControl**, расположенный в библиотеке на странице Win32. Перенесите компонент **PageControl** на форму и поработайте с ним. Чтобы задавать и редактировать страницы этого компонента, надо щелкнуть на нем правой кнопкой мыши. Во всплывшем меню вы можете видеть команды: New Page — создать новую страницу, Next Page — переключиться на следующую страницу, Previous Page — переключиться на предыдущую страницу. Каждая создаваемая вами страница является панелью, на которой можно размещать любые управляющие компоненты, окна редактирования и т.п. Надпись, которая появляется на ярлычке закладки страницы, задается свойством **Caption**. На рис. 5.3 приведен тот же пример, который фигурировал на рис. 5.2 и 3.3, но оформленный с помощью компонента **PageControl**.

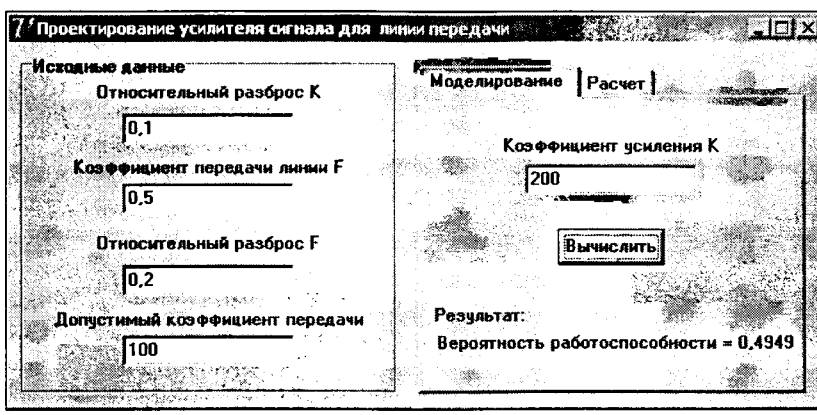


Рис. 5.3
*Использование компонента **PageControl** в окне приложения, показанного ранее на рис. 3.3 и 5.2*

Еще один принцип, которого надо придерживаться при проектировании окон — стилистическое единство всех окон в приложении. Недопустимо, чтобы сходные по функциям органы управления в разных окнах назывались по-разному или размещались в разных местах окон. Все это мешает работе с приложением, отвлекает пользователя, заставляет его думать не о сущности работы, а о том, как приспособиться к тому или иному окну.

Единство стилистических решений важно не только внутри приложения, но и в рамках серии разрабатываемых вами приложений. Это нетрудно обеспечить

с помощью имеющихся в Delphi многочисленных способов повторного использования кодов. Вам достаточно один раз разработать какие-то часто применяемые формы — ввода пароля, запроса или предупреждения пользователя и т.п., включить их в Депозитарий, а затем вы можете использовать их многократно во всех своих проектах.

5.2.3 Последовательность фокусировки элементов

При проектировании приложения важно правильно определить последовательность табуляции оконных компонентов. Под этим понимается последовательность, в которой переключается фокус с компонента на компонент, когда пользователь нажимает клавишу табуляции Tab. Это важно, поскольку в ряде случаев пользователю удобнее работать не с мышью, а с клавиатурой. Пусть, например, вводя данные, пользователь должен указать в отдельных окнах редактирования фамилию, имя и отчество человека. Конечно, набрав фамилию, ему удобнее нажать клавишу Tab и набирать имя, а потом опять, нажав Tab, набирать отчество, чем каждый раз отрываться от клавиатуры, хватать мышь и переключаться в новое окно редактирования.

Свойство формы **ActiveControl**, установленное в процессе проектирования, определяет, какой из размещенных на форме компонентов будет в фокусе в первый момент при выполнении приложения. В процессе выполнения это свойство изменяется и показывает тот компонент, который в данный момент находится в фокусе.

Последовательность табуляции задается свойствами **TabOrder** компонентов. Первоначальная последовательность табуляции определяется просто той последовательностью, в которой размещались управляющие элементы на форме. Первому элементу присваивается значение **TabOrder**, равное 0, второму 1 и т.д. Значение **TabOrder**, равное нулю, означает, что при первом появлении формы на экране в фокусе будет именно этот компонент (если не задано свойство формы **ActiveControl**). Если на форме имеются панели, являющиеся контейнерами, включающими в себя другие компоненты, то последовательность табуляции составляется независимо для каждого компонента-контейнера. Например, для формы будет последовательность, включающая некоторую панель как один компонент. А уже внутри этой панели будет своя последовательность табуляции, определяющая последовательность получения фокуса ее дочерними оконными компонентами.

Каждый управляющий элемент имеет уникальный номер **TabOrder** внутри своего родительского компонента. Поэтому изменение значения **TabOrder** какого-то элемента на значение, уже существующее у другого элемента, приведет к тому, что значения **TabOrder** всех последующих элементов автоматически изменятся, чтобы не допустить дублирования. Если задать элементу значение **TabOrder**, большее, чем число элементов в родительском компоненте, он просто станет последним в последовательности табуляции.

Из-за того, что при изменении **TabOrder** одного компонента могут меняться **TabOrder** других компонентов, устанавливать эти свойства поочередно в отдельных компонентах трудно. Поэтому в среде проектирования Delphi имеется специальная команда Edit | Tab Order, позволяющая в режиме диалога задать последовательность табуляции всех элементов.

Значение свойства **TabOrder** играет роль, только если другое свойство компонента — **TabStop** установлено в **true**. Установка **TabStop** в **false** приводит к тому, что компонент выпадает из последовательности табуляции и ему невозможно передать фокус клавишей Tab (однако передать фокус мышью, конечно, можно).

Имеется и программная возможность переключения фокуса — это метод **SetFocus**. Например, если вы хотите переключить в какой-то момент фокус на окно **Edit2**, вы можете сделать это оператором:

```
Edit2.SetFocus;
```

Выше говорилось, что в приложении с несколькими окнами редактирования, в которые пользователь должен поочередно вводить какие-то данные, ему удобно переключаться между окнами клавишей табуляции. Но еще удобнее пользователю, закончившему ввод в одном окне, нажать клавишу Enter и автоматически перейти к другому окну. Это можно сделать, обрабатывая событие нажатия клавиши **OnKeyDown** (см. разд. 4.14). Например, если после ввода данных в окно **Edit1** пользователю надо переключаться в окно **Edit2**, то обработчик события **OnKeyDown** окна **Edit1** можно сделать следующим:

```
procedure TForm1.Edit1KeyDown(Sender: TObject;
                              var Key: Word; Shift: TShiftState);
begin
 if (Key = VK_RETURN) then Edit2.SetFocus;
end;
```

Единственный оператор этого обработчика проверяет, не является ли клавиша, нажатая пользователем, клавишей Enter. Если это клавиша Enter, то фокус передается окну **Edit2** методом **SetFocus**.

Можно подобные операторы ввести во все окна, обеспечивая требуемую последовательность действий пользователя (впрочем, свобода действий пользователя от этого не ограничивается, поскольку он всегда может вернуться вручную к нужному окну). Однако можно сделать все это более компактно, используя один обработчик для различных окон редактирования и кнопок. Для этого может использоваться метод **FindNextControl**, возвращающий дочерний компонент, следующий в последовательности табуляции. Если компонент, имеющий в данный момент фокус, является последним в последовательности табуляции, то возвращается первый компонент этой последовательности. Метод определен следующим образом:

```
function FindNextControl(CurControl: TWinControl;
         GoForward, CheckTabStop, CheckParent: Boolean): TWinControl;
```

Он находит и возвращает следующий за указанным в параметре **CurControl** дочерний оконный компонент в соответствии с последовательностью табуляции.

Параметр **GoForward** определяет направление поиска. Если он равен **true**, то поиск проводится вперед и возвращается компонент, следующий за **CurControl**. Если же параметр **GoForward** равен **false**, то возвращается предшествующий компонент.

Параметры **CheckTabStop** и **CheckParent** определяют условия поиска. Если **CheckTabStop** равен **true**, то просматриваются только компоненты, в которых свойство **TabStop** установлено в **true**. При **CheckTabStop** равном **false** значение **TabStop** не принимается во внимание. Если параметр **CheckParent** равен **true**, то просматриваются только компоненты, в свойстве **Parent** которых указан данный оконный элемент, т.е. просматриваются только прямые потомки. Если **CheckParent** равен **false**, то просматриваются все, даже косвенные потомки данного элемента.

Таким образом, для решения поставленной нами задачи — автоматической передачи фокуса при нажатии клавиши Enter, вы можете написать единый обработ-

чик событий **OnKeyDown** всех интересующих вас оконных компонентов, содержащий оператор, обеспечивающий передачу фокуса очередному компоненту:

```
if (Key = VK_RETURN)
  then FindNextControl(Sender as TWinControl, true, true,
                       false).SetFocus;
```

5.3 Проектирование окон с изменяемыми размерами

5.3.1 Привязка размеров компонентов к размерам контейнера

В разд. 5.2.1 говорилось, что без необходимости не следует делать окна приложения с изменяемыми размерами. Посмотрите, например, окна, приведенные выше на рис. 5.2 и 5.3. Имеет ли смысл предоставить пользователю возможность изменять размер таких окон? Конечно, нет. Поэтому в приложениях, показанных на этих рисунках, для формы задано значение **BorderStyle = bsSingle** с исключением из числа доступных кнопок кнопки Развернуть. Так что пользователь может только свернуть или закрыть окно.

А теперь посмотрите окно приложения, разработанного в гл. 4 и показанного на рис. 4.1. В таком приложении есть смысл разрешить пользователю изменять размеры окна. Но при этом необходимо, чтобы пропорционально изменялся размер компонента **RichEdit**, в котором пользователь пишет характеристику. Тогда пользователь может увеличить окно, что облегчит ему запись и просмотр большой, развернутой характеристики.

Изменять размеры компонентов при изменении размера окна можно двумя способами. У многих компонентов и, в частности, у панелей, есть свойство **Align** — выравнивание. По умолчанию оно равно **alNone**. Это означает, что никакое выравнивание не осуществляется. Но его можно задать равным **alTop**, или **alBottom**, или **alLeft**, или **alRight**, что будет означать, что компонент должен занимать всю верхнюю, или нижнюю, или левую, или правую часть клиентской области компонента-контейнера. Под клиентской областью понимается вся свободная площадь формы или другого контейнера, в которой могут размещаться включенные в этот контейнер компоненты. Во всех случаях, кроме **Align = alNone**, размеры компонента будут автоматически изменяться при изменении размеров контейнера. Можно также задать свойству **Align** компонента значение **alClient**, что приводит к заполнению компонентом всей свободной клиентской области. Подобное значение имеет смысл задавать, если всю площадь формы или панели должен занимать один компонент, например, многострочное окно редактирования или окно графика.

Вы уже неявно использовали свойство **Align** при проектировании приложения в гл. 4. В нем вы применяли инструментальную панель на основе компонента **ToolBar** и панель состояния **StatusBar**. Инструментальная панель автоматически занимала верхнюю часть формы, и панель состояния — нижнюю. Посмотрев свойство **Align** в этих компонентах, вы увидите, что в **ToolBar** по умолчанию задано значение **alTop**, а в **StatusBar** — **alBottom**. Эти значения обеспечивали автоматическую адаптацию размеров компонентов к размерам формы.

Но свойство **Align** не позволит, например, в приложении рис. 4.1 изменять при изменении окна формы размер окна **RichEdit**. Можно было бы, конечно, прижать

это окно к правому краю формы. Но было бы некрасиво, так как хочется оставить некоторые зазоры между окном и краями формы.

Более гибкое изменение размеров и местоположения компонентов можно получить с помощью свойства **Anchors** (якоря). Оно задает привязку сторон компонента к соответствующим граням контейнера. Свойство **Anchors** представляет собой множество, которое может содержать следующие элементы:

akTop	Верхний край компонента привязан к верхнему краю родительского компонента.
akLeft	Левый край компонента привязан к левому краю родительского компонента.
akRight	Правый край компонента привязан к правому краю родительского компонента.
akBottom	Нижний край компонента привязан к нижнему краю родительского компонента.

По умолчанию привязка осуществляется к левому и верхнему краям родительского компонента. Т.е. по умолчанию свойство **Anchors** равно **[akLeft, akTop]**. Если задать в свойстве **Anchors** привязку к противоположным сторонам родительского компонента, то при изменении размеров родительского компонента будут меняться размеры и данного компонента. В примере рис. 4.1 для окна **RichEdit** имеет смысл задать привязки ко всем сторонам формы: **Anchors = [akLeft, akTop, akRight, akBottom]**. Опробуйте этот вариант. Вы увидите, что размеры окна редактирования изменяются при изменении окна приложения.

Если бы под окном **RichEdit1** были расположены какие-то другие компоненты, например, кнопки, то в них надо было бы отменить привязку к верхнему краю формы и ввести привязку к ее нижнему краю. Тогда при изменении размеров формы эти компоненты смещались бы, выдерживая постоянное расстояние от низа окна. Аналогично, если бы какие-то компоненты были расположены правее окна **RichEdit1**, то в них надо было бы отменить привязку к левому краю формы и ввести привязку к правому краю.

В более сложных случаях, например, если требуется, чтобы какой-то компонент всегда располагался по горизонтали посередине формы, приходится переходить к программному изменению местоположения компонента. Программное изменение расположения и размеров компонентов приходится делать и в ранних версиях Delphi, в которых свойство **Anchors** отсутствует.

Местоположение компонента задается свойствами **Left** и **Top**, характеризующими координаты левого верхнего угла компонента. При этом началом координат считается левый верхний угол клиентской области компонента-контейнера. Горизонтальная координата отсчитывается вправо от этой точки, а вертикальная — вниз.

Размеры компонента определяются свойствами **Width** — ширина и **Height** — высота. Имеется еще два свойства, определяющие размер клиентской области компонента-контейнера: **ClientWidth и ClientHeight**. Для некоторых компонентов они совпадают со свойствами **Width** и **Height**. Но могут и отличаться от них. Например, в форме **ClientHeight** меньше, чем **Height**, за счет бордюра, полосы заголовка, меню и полосы горизонтальной прокрутки, если она имеется.

Если вам надо изменять местоположение и размеры компонентов при изменении размеров формы, соответствующие операторы следует размещать в обработчике события формы **OnResize**. Это событие возникает при любом изменении пользователем размеров окна приложения.

Например, помещенный в обработчике события **OnResize** формы оператор

```
Panel1.Left := (Form1.ClientWidth - Panel1.Width) div 2;
```

обеспечит расположение панели **Panel1** всегда посередине (по горизонтали) формы **Form1**.

5.3.2 Панели с перестраиваемыми границами

В ряде случаев описанного автоматического изменения размеров различных панелей оказывается недостаточно для создания удобного пространства для действий пользователя. Даже при увеличении окна на весь экран какая-то панель приложения может оказаться перегруженной информацией, а другая относительно пустой. В этих случаях полезно предоставить пользователю возможность перемещать границы, разделяющие различные панели, изменяя их относительные размеры. Пример такой возможности можно увидеть в программе Windows «Проводник».

В библиотеке Delphi на странице Additional имеется специальный компонент — **Splitter**, который позволяет легко осуществить это. При работе с ним надо соблюдать определенную последовательность проектирования. Пусть, например, вы хотите установить **Splitter** между двумя панелями **Panel**, первая из которых будет выровнена к какому-то краю клиентской области, а вторая займет всю клиентскую область. Перенесите на форму первую панель и задайте ее свойство **Align = alLeft**. Панель вытянется вдоль левой части окна. Теперь перенесите на форму разделитель **Splitter** и тоже задайте его свойство **Align = alLeft** (впрочем, это значение задано по умолчанию). **Splitter** прижмется к соответствующему краю первой панели. Далее перенесите на форму вторую панель и задайте в ней **Align = alClient**. В результате **Splitter** окажется зажатым между двумя панелями и при запуске приложения он позволит пользователю изменять положение соответствующей границы между этими панелями.

Подобные разделители **Splitter** можно разместить между всеми панелями приложения, дав пользователю полную свободу изменять топологию окна, с которым он работает.

Компонент **Splitter** имеет событие **OnMoved**, которое наступает после конца перемещения границы. В обработчике этого события надо предусмотреть, если необходимо, упорядочение размещения компонентов на панелях, размеры которых изменились: переместить какие-то метки, изменить размеры компонентов и т.д.

Свойство **ResizeStyle** компонента **Splitter** определяет поведение разделителя при перемещении его пользователем. Поэкспериментируйте с ним, чтобы увидеть различие в режимах перемещения разделителя. По умолчанию свойство **Splitter** равно **rsPattern**. Это означает, что пока пользователь тянет курсором мыши границу, сам разделитель не перемещается, и панели тоже остаются прежних размеров. Пожалуй, этот вариант значения **ResizeStyle** наиболее удачен.

Свойство **MinSize** компонента **Splitter** устанавливает минимальный размер в пикселах обеих панелей, между которыми зажат разделитель. Задание такого минимального размера необходимо, чтобы при перемещениях границы панель не сжалась бы до нулевого размера или до такой величины, при которой на ней исчезли бы какие-то необходимые для работы элементы управления. К сожалению, в версиях Delphi, младше Delphi 5, свойство **MinSize** не всегда срабатывает верно. Начиная с Delphi 5, введено новое свойство компонента **Splitter** — **AutoSnap**. Если оно установлено в **true** (по умолчанию), то при перемещении границы воз-

можны те же неприятности, что в младших версиях Delphi. Но если установить **AutoSnap** в **false**, то перемещение границы панелей сверх пределов, при которых размер одной из панелей станет меньше **MinSize**, просто блокируется. Так что можно рекомендовать всегда устанавливать **AutoSnap** в **true**.

Впрочем, и это не решает всех задач, связанных с перемещением границ панелей. Дело в том, что свойство **MinSize** относится к обеим панелям, граница между которыми перемещается, а в ряде случаев желательно раздельно установить различные минимальные размеры одной и другой панели. Это проще сделать, задав в панелях соответствующие значения свойства **Constraints**, о котором будет рассказано в следующем разделе.

5.3.3 Ограничение пределов изменения размеров окон и компонентов

Все рассмотренные ранее методы изменения размеров панелей и компонентов на них имеют общий недостаток: при чрезмерном уменьшении пользователем размеров окна какие-то компоненты могут исчезать из поля зрения. Иногда к некрасивым с точки зрения эстетики результатам приводит и чрезмерное увеличение размеров окна. Хотелось бы иметь средства, ограничивающие пользователя в его манипуляциях с окном и не позволяющие ему чрезмерно уменьшать и увеличивать размеры.

Таким средством является свойство **Constraints**, присущее всем оконным компонентам и позволяющее задавать ограничения на допустимые изменения размеров. Свойство имеет четыре основных подсвойства: **MaxHeight**, **MaxWidth**, **MinHeight** и **MinWidth** — соответственно максимальная высота и ширина, и минимальная высота и ширина. По умолчанию значения всех этих подсвойств равны 0, что означает отсутствие ограничений. Но задание любому из этих свойств положительного значения приводит к соответствующему ограничению размера заданным числом пикселов.

Чтобы какие-то компоненты не исчезали из поля зрения, можно задать им ограничения минимальной высоты и длины. Таким образом можно поддерживать нормальные пропорции отдельных частей окна. Можно задать ограничения на минимальные и максимальные размеры формы, т.е. всего окна. Например, если вы зададите для формы значения **MaxHeight** = 500 и **MaxWidth** = 500, то пользователь не сможет сделать окно большим, чем квадрат 500 x 500. Причем это ограничение будет действовать, даже если пользователь нажмет системную кнопку, разворачивающую окно на весь экран. Окно развернется, но его размеры не превысят заданных. Это иногда полезно делать, чтобы развернутое окно не заслонило какие-то другие нужные пользователю окна.

5.4 Некоторые компоненты интерфейса

В Delphi имеется множество компонентов, которые вы с успехом можете использовать при построении интерфейса пользователя. Рассмотреть их все в рамках данной книги совершенно невозможно. В предыдущих главах и в данной главе многие из них уже рассмотрены. Вы можете найти ссылки на них в предметном указателе в конце книги. Но есть еще ряд компонентов, без которых трудно пред-

ставить себе современный интерфейс пользователя и которые будут использоваться в последующих главах. Рассмотрим эти компоненты. Они показаны на рис. 5.4. Это то же приложение, которое вы создавали в гл. 4 и которое ранее было показано на рис. 4.1. Только в данном случае интерфейс организован с помощью других компонентов. Задание группы осуществляется выпадающим списком **ComboBox**. Далее будет показано, в чем преимущество такого решения. Указание пола на рис. 5.4 реализовано в двух вариантах: с помощью группы радиокнопок **RadioGroup** и с помощью кнопок **SpeedButton**, объединенных панелью **GroupBox**. Конечно, в реальном приложении надо выбрать какой-то один из этих вариантов. Задание года рождения реализовано с помощью компонента **SpinEdit**.

Чтобы реализовать подобный вариант приложения, созданного в гл. 4, надо сделать следующее. Откройте ваш прежний проект и выполните команду File | Save Project As. Сохраните проект под новым именем в том же каталоге, в котором был расположен прежний проект. Далее перейдите в окне Редактора Кода в главный модуль приложения, выполните команду File | Save As и сохраните модуль под новым именем в том же каталоге. Остальные модули приложения можно оставить прежними. Теперь вы сможете, не искажая прежнего приложения, изменять интерфейс, заменяя окна редактирования, которые были в этом приложении, новыми компонентами.

Рис. 5.4
Пример использования различных компонентов интерфейса

5.4.1 Выпадающий список ComboBox

В ряде случаев пользователь, вводя некоторую информацию, должен выбирать из нескольких заранее известных альтернатив. В этих случаях лучше не заставлять пользователя полностью писать ответ, а дать ему возможность выбрать нужный ответ из списка. В примере, рассмотренном в гл. 4, пользователь указывает номер группы, в которой учится студент. Если пользователь вводит имя группы в окно редактирования, как это было сделано в гл. 4, то он может ошибиться, написать имя несуществующей группы, и бедный студент окажется неприкаянным. Да и писать полностью название группы пользователю неудобно. Гораздо проще выбрать группу из списка, поскольку названия групп заранее известны и изменению не подлежат.

Один из списков — компонент **ListBox** уже рассматривался в разд. 3.3.4. Но он не всегда удобен, так как занимает на форме довольно много места. Часто удобнее более компактный выпадающий список **ComboBox**. Список имеет окно редактирования, в котором пользователь может вводить текст, если не хочет производить выбор из списка.

Основное свойство компонента **ComboBox**, содержащее список строк, — **Items**, имеющее рассмотренный в разд. 3.3.4 тип **TStrings**. Заполнить его во время проектирования можно, нажав кнопку с многоточием около этого свойства в окне Инспектора Объектов. А во время выполнения работать с этим свойством можно, пользуясь свойствами и методами класса **TStrings** — **Clear**, **Add** и другими.

Свойство **ItemIndex** показывает индекс строки, выбранной в списке. Индексы, как всегда, отсчитываются от нуля: 0 — первая строка, 1 — вторая и т.д. По умолчанию **ItemIndex** = −1. Это означает, что ни один элемент списка не выбран. Если вы заполнили список во время проектирования, то в последних версиях Delphi можете задать значение **ItemIndex**. В ранних версиях Delphi свойство **ItemIndex** было свойством только времени выполнения. В этих версиях чтобы задать какое-то значение **ItemIndex**, т.е. установить выбор по умолчанию, который будет показан в момент начала работы приложения, надо вводить, например, в обработчик события **OnCreate** формы, оператор вида:

```
ComboBox1.ItemIndex := 0;
```

Если не задать значение **ItemIndex** ни во время проектирования, ни в момент начала выполнения приложения, то в момент запуска приложения пользователь не увидит в окне компонента одно из возможных значений списка и, вероятнее всего, не очень поймет, что с этим окном надо делать. Точнее, в первый момент он увидит текст, заданный во время проектирования в свойстве **Text**. Так что в этом случае этим текстом надо сообщить пользователю, что он должен делать со списком.

Выбор пользователя или введенный им текст можно определить во время выполнения программно по значению свойства **Text**. Если же надо определить индекс выбранного пользователем элемента списка, то можно воспользоваться свойством **ItemIndex** — его значение равно индексу выбранной строки. Но если пользователь в окне списка проводил редактирование данных, то **ItemIndex** = −1. По этому признаку можно определить, что редактирование проводилось.

Стиль изображения компонента **ComboBox** определяется его свойством **Style**. По умолчанию значение свойства **Style** равно **csDropDown**. В этом случае список имеет окно редактирования, позволяющее пользователю вводить или редактировать текст. Но, например, в нашем приложении это значение нас не устроит, так как не надо давать пользователю возможность редактировать наименования групп. Нам больше подойдет значение **csDropDownList**. В этом случае окно редактирования отсутствует, и пользователь может только осуществлять выбор из списка.

Свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted** = **true** новые строки в список добавляются не в конец, а по алфавиту.

Свойство **DropDownCount** указывает число строк, появляющихся в выпадающем списке без возникновения полосы прокрутки.

Замените в вашем приложении окно редактирования, в котором пользователь задает номер группы, списком **ComboBox**. Занесите в свойство **Items** этого компонента список групп. Установите его свойство **Style** равным **csDropDownList**, чтобы запретить ввод несуществующих групп. Задайте **ItemIndex** = 0, а если работаете с ранними весриями Delphi, то задайте **ItemIndex** программно, как показано

выше. В процедуре **APersonSaveExecute**, читающей данные, введенные пользователем, перед запоминанием их в файле, измените оператор, задающий значение **Pers.Dep1**, оператором:

```
Dep1 := ComboBox1.Text;
```

В процедуре **APersonOpenExecute**, читающей данные из файла, замените оператор, читающий значение **Pers.Dep1**, оператором:

```
ComboBox1.ItemIndex := ComboBox1.Items.IndexOf(Dep1);
```

Этот оператор определяет методом **IndexOf** (см. разд. 3.3.3) индекс строки с текстом **Pers.Dep1** в списке и заносит этот индекс в свойство **ItemIndex**.

В процедуре **APersonCharExecute** измените оператор, формирующий информацию о группе в строке, записываемой в **RichEdit1**:

```
S := S + ' группы ' + ComboBox1.Text;
```

Выполните получившееся приложение и проверьте работу со списком **ComboBox**.

В созданном приложении есть один недостаток. Если названия групп изменятся, то приложение придется перекомпилировать, занеся в **ComboBox** новый список. Этот недостаток нетрудно исправить, храня данные в некотором текстовом файле и читая их из него в начале выполнения приложения. Для этого в обработчик события **OnCreate** формы можно вставить операторы:

```
if FileExists('Groups.dat')
 then ComboBox1.Items.LoadFromFile('Groups.dat');
ComboBox1.ItemIndex := 0;
```

В этом коде предполагается, что данные хранятся в текстовом файле с именем *Groups.dat*. И если такой файл имеется в текущем каталоге, он читается в список методом **LoadFromFile** свойства **Items** (см. разд. 3.3.3). Файл *Groups.dat* можно формировать в любом текстовом редакторе. А можете добавить соответствующее действие в ваше приложение. При переходе в режим редактирования можно выполнить оператор

```
RichEdit1.Lines.Assign(ComboBox1.Items);
```

Он занесет в окно **RichEdit1** текущее состояние списка. А после того, как пользователь отредактирует список, изменив какие-то строки, удалив ненужные, добавив новые, надо выполнить операторы:

```
ComboBox1.Items.Assign(RichEdit1.Lines);
ComboBox1.Items.SaveToFile('Groups.dat');
```

Первый из них заносит в список отредактированные данные, а второй сохраняет их в файле. Придумайте сами, как оформить в интерфейсе подобную возможность редактирования списка. Вероятно, стоит предоставлять такую возможность не любому пользователю, а только тому, кто вошел в программу с каким-то специальным паролем (вспомните, что в этом приложении предусмотрен вход по паролю).

5.4.2 Группа радиокнопок RadioGroup

Радиокнопки образуют группы взаимосвязанных индикаторов, из которых обычно может быть выбран только один. Они используются для выбора пользователем одной из нескольких взаимоисключающих альтернатив. Например, если бы

в вашем приложении было немного групп, можно был бы их отображать радиокнопками. Естественно также отображать радиокнопками пол.

Чаще всего радиокнопки создаются с помощью компонента **RadioGroup** — панели группы радиокнопок. Это панель, которая может содержать радиокнопки, регулярно расположенные столбцами и строками. Надпись в левом верхнем углу панели определяется свойством **Caption**. А надписи кнопок и их количество определяются свойством **Items**, имеющим тип **TStrings**. Щелкнув на кнопке с многоточием около этого свойства в окне Инспектора Объектов, вы попадете в редактор списков строк, который уже неоднократно использовали. В нем вы можете занести надписи, которые хотите видеть около кнопок, по одной в строке. Сколько строчек вы запишете — столько и будет кнопок. Например, для отображения пола надо ввести две строки: "м" и "ж".

Кнопки, появившиеся в панели после задания значений **Items**, можно разместить в несколько столбцов (не более 17), задав свойство **Columns**. По умолчанию **Columns** = 1, т.е. кнопки размещаются друг под другом. На рис. 5.4 число столбцов задано равным 2.

Определить, какую из кнопок выбрал пользователь, можно по свойству **ItemIndex**, которое показывает индекс выбранной кнопки. По умолчанию **ItemIndex** = −1, что означает отсутствие выбранной кнопки. Если вы хотите, чтобы в момент начала выполнения приложения какая-то из кнопок была выбрана (это практически всегда необходимо), то надо установить соответствующее значение **ItemIndex** во время проектирования. Если вы используете радиокнопки не для ввода, а для отображения данных, устанавливать значение **ItemIndex** можно программно во время выполнения приложения.

Опробуйте **RadioGroup** в приложении рис. 5.4. Перенесите этот компонент на форму, задайте соответствующие значения его свойств **Caption**, **Items** и **ItemIndex**. Удалите из приложения окно редактирования, в котором ранее пользователь вводил информацию о поле — символы "м" или "ж". Измените в процедуре **APersonSaveExecute** оператор, заносящий в запись **Pers** символ пола:

```
if RadioGroup1.ItemIndex = 0
 then Sex := 'м'
 else Sex := 'ж';
```

В процедуре **APersonOpenExecute** замените оператор, читающий соответствующую информацию:

```
if Sex = 'м'
 then RadioGroup1.ItemIndex := 0
 else RadioGroup1.ItemIndex := 1;
```

В процедуре **APersonCharExecute** измените оператор, формирующий информацию о поле в строке, записываемой в **RichEdit1**:

```
if  RadioGroup1.ItemIndex = 0
 then S := S + 'студент'
 else S := S + 'студентка';
```

Можете выполнить приложение и проверить работу радиокнопок во всех режимах.

Компонент **RadioGroup** очень удобен, но не свободен от некоторых недостатков. Его хорошо использовать, если надписи кнопок имеют примерно одинаковую длину и если число кнопок в каждом столбце (при размещении их в нескольких столбцах) одинаково. Тогда регулярное расположение кнопок оправдано. Но если

желательно нерегулярное расположение кнопок, следует использовать компоненты **RadioButton**, сгруппированные панелью **GroupBox**. Отдельная радиокнопка **RadioButton** особого смысла не имеет. Радиокнопки имеет смысл применять, когда они взаимодействуют друг с другом в группе. Эта группа и объединяется единым контейнером, обычно панелью **GroupBox**.

Рассмотрим свойства радиокнопки **RadioButton**. Свойство **Caption** содержит надпись, появляющуюся около кнопки. Значение свойства **Alignment** определяет, с какой стороны от кнопки появится надпись: **taLeftJustify** — слева, **taRightJustify** — справа (это значение принято по умолчанию). Свойство **Checked** определяет, выбрана ли данная кнопка пользователем, или нет. Поскольку в начале выполнения приложения обычно надо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство **Checked** надо установить в **true** в процессе проектирования. Если вы поэкспериментируете, то заметите, что и во время проектирования можно установить в **true** значение **Checked** только у одной кнопки из группы..

Если вы хотите поработать с кнопками **RadioButton** в вашем приложении, поместите на форму панель **GroupBox** и задайте в ней надпись "Пол". Пометите на панель две кнопки **RadioButton**. Задайте их надписи "м" и "ж". У первой кнопки установите **Checked** = **true**. А рассмотренные ранее операторы, работающие с радиокнопками, надо заменить следующими:

в процедуре **APersonSaveExecute**:

```
if RadioButton1.Checked
 then Sex := 'м'
 else Sex := 'ж';
```

в процедуре **APersonOpenExecute**:

```
if Sex = 'м'
 then RadioButton1.Checked := true
 else RadioButton2.Checked := true;
```

в процедуре **APersonCharExecute**:

```
  if RadioButton1.Checked
   then S := S + 'студент'
   else S := S + 'студентка';
```

Радиокнопки **RadioButton** могут размещаться не только в панели **GroupBox**, но и в любой панели другого типа, а также непосредственно на форме. Группа взаимосвязанных кнопок в этих случаях определяется тем оконным компонентом, который содержит кнопки. В частности, для радиокнопок, размещенных непосредственно на форме, контейнером является сама форма. Таким образом, все кнопки, размещенных непосредственно на форме, работают как единая группа, т.е. только в одной из этих кнопок можно установить значение **Checked** в **true**.

5.4.3 Кнопки SpeedButton

Кнопки **SpeedButton** имеют возможность отображения пиктограмм и могут использоваться как обычные управляющие кнопки или как кнопки с фиксацией нажатого состояния. С похожими кнопками вы уже имели дело при формировании инструментальной панели на основе **ToolBar**.

У **SpeedButton**, как и у других кнопок, имеется свойство **Caption** — надпись. Впрочем, нередко в **SpeedButton** оно оставляется пустым, если вместо надписи используется пиктограмма. Изображение на кнопке задается свойством **Glyph**. Щелчок на кнопке с многоточием около этого свойства в окне Инспектора Объектов вызывает диалог, позволяющий загрузить требуемое изображение. Этот диалог мы рассмотрим позднее в разд. 5.5.1.

Особенностью кнопок **SpeedButton** являются свойства **GroupIndex** (индекс группы), **AllowAllUp** (разрешение отжатого состояния всех кнопок группы) и **Down** (исходное состояние — нажатое). Если **GroupIndex** = 0, то кнопка ведет себя так же, как **Button**. При нажатии пользователем кнопки она погружается, а при отпускании возвращается в нормальное состояние. В этом случае свойства **AllowAllUp** и **Down** не влияют на поведение кнопки.

Если **GroupIndex** > 0 и **AllowAllUp** = **true**, то кнопка при щелчке пользователя на ней погружается и остается в нажатом состоянии. При повторном щелчке пользователя на кнопке она освобождается и переходит в нормальное состояние (именно для того, чтобы освобождение кнопки состоялось, необходимо задать **AllowAllUp** = **true**). Если свойство **Down** во время проектирования установлено равным **true**, то исходное состояние кнопки — нажатое.

Если есть несколько кнопок, имеющих одинаковое ненулевое значение **GroupIndex**, то они образуют группу взаимосвязанных кнопок, из которых нажатой может быть только одна. Если одна кнопка находится в нажатом состоянии и пользователь щелкает на другой, то первая кнопка освобождается, а вторая фиксируется в нажатом состоянии. Поведение нажатой кнопки при щелчке на ней зависти от значения свойства **AllowAllUp**. Если оно равно **true**, то кнопка освободится, поскольку в этом случае возможно состояние, когда все кнопки группы отжаты. Если же **AllowAllUp** равно **false**, то щелчок на нажатой кнопке не приведет к изменению вида кнопки. Впрочем, и в этом случае, как и при любом щелчке на кнопке, возникает событие **OnClick**, которое может быть обработано.

Состояние кнопки во время выполнения можно определить по значению свойства **Down**: если значение равно **true**, то кнопка нажата. Во время события **OnClick** значение **Down** уже равно тому состоянию, которое примет кнопка в результате щелчка на ней.

Можете опробовать кнопки **SpeedButton** в нашем приложении для указания пола вместо рассмотренных выше радиокнопок. Перенесите на форму панель **GroupBox** и задайте на ней надпись "Пол". Поместите на панели две кнопки **SpeedButton**. Мы не будем размещать на них пиктограммы. Задайте просто надписи "м" и "ж". У кнопки с надписью "м" установите в **true** свойство **Down**, чтобы она была нажата. Установите у обеих кнопок значение **AllowAllUp** равным **true** и задайте одинаковое положительное значение **GroupIndex**, например, 1. Ниже приведены операторы, которые вам надо записать для работы с кнопками **SpeedButton**:
в процедуре **APersonSaveExecute**:

```
if SpeedButton1.Down
 then Sex := 'м'
 else Sex := 'ж';
```

в процедуре **APersonOpenExecute**:

```
if Sex = 'м'
 then SpeedButton1.Down := true
 else SpeedButton2.Down := true;
```

в процедуре **APersonCharExecute**:

```
if SpeedButton1.Down
 then S := S + 'студент'
 else S := S + 'студентка';
```

5.4.4 Компонент SpinEdit

Компонент **SpinEdit** расположен в библиотеке на странице Samples. Этот компонент удобно использовать в нашем приложении для ввода года рождения. Он представляет собой окно редактирования, в котором пользователь может вводить целое число, или изменять его кнопками со стрелками.

Основное свойство **Value** позволяет во время проектирования задать значение числа по умолчанию, а во время выполнения — прочитать число, заданное пользователем. Свойства **MinValue** и **MaxValue** задают соответственно минимальное и максимальное значения чисел, свойство **Increment** задает приращение числа при каждом нажатии на кнопку.

Замените в вашем приложении окно, в котором пользователь задавал год рождения, компонентом **SpinEdit**. Задайте значение **Value** равным наиболее ожидаемому году рождения, и установите разумные значения **MinValue** и **MaxValue**. В коде приложения надо изменить операторы, работающие с годом рождения:

в процедуре **APersonSaveExecute**:

```
Year := SpinEdit1.Value;
```

в процедуре **APersonOpenExecute**:

```
SpinEdit1.Value := Year;
```

в процедуре **APersonCharExecute**:

```
S := Edit1.Text + ', ' + IntToStr(SpinEdit1.Value) + ' г.р., ';
```

5.5 Графика

5.5.1 Изображения

5.5.1.1 Компонент Image

Нередко возникает потребность украсить свое приложение какими-то изображениями. Это может быть графическая заставка, являющаяся логотипом вашего приложения. Или это могут быть фотографии студентов или сотрудников при разработке приложения, работающего с базой данных. В первом случае вам потребуется компонент **Image**, расположенный на странице Additional библиотеки компонентов, во втором — его аналог **DBImage**, связанный с данными и расположенный на странице Data Controls.

Начнем знакомство с компонентом **Image**. Откройте новое приложение и перенесите этот компонент на форму. Основное свойство компонента, которое может содержать изображение — **Picture**. Нажмите на кнопку с многоточием около этого свойства, и перед вами откроется окно Picture Editor (рис. 5.5), позволяющее загрузить в свойство **Picture** какой-нибудь графический файл (кнопка Load), а так-

же сохранить открытый файл под новым именем или в новом каталоге. Щелкните на Load, чтобы загрузить графический файл. Перед вами откроется стандартное окно выбора файла изображения. Вы можете найти графические файлы в каталоге *Images*. Он обычно расположен в каталоге *...\program files\Common Files\Borland Shared*.

На рис. 5.5 изображена загрузка файла *...\Images\Splash\16Color\earth.bmp*. После загрузки файла щелкните на OK, и в вашем компоненте **Image** отобразится выбранное вами изображение. Можете запустить ваше приложение и полюбоваться им. Впрочем, вы и так видите картинку, даже не выполняя приложение.

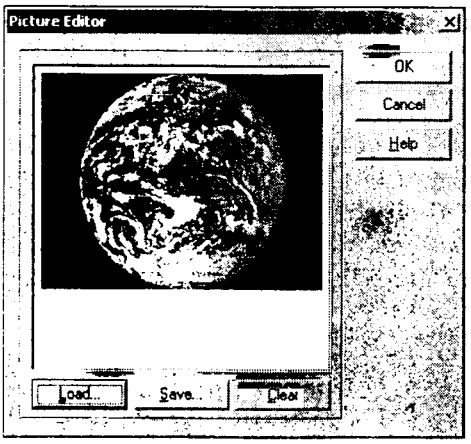


Рис. 5.5
Окно Picture Editor

Когда вы в процессе проектирования загрузили изображение из файла в компонент **Image**, он не просто отображает его, но и сохраняет в приложении. Это дает вам возможность поставлять ваше приложение без отдельного графического файла. Впрочем, как мы увидим позднее, в **Image** можно загружать и внешние графические файлы в процессе выполнения приложения.

Вернемся к рассмотрению свойств компонента **Image**. Если установить свойство **AutoSize** в **true**, то размер компонента **Image** будет автоматически подгоняться под размер помещенной в него картинки. Если же свойство **AutoSize** установлено в **false**, то изображение может не поместиться в компонент или, наоборот, площадь компонента может оказаться много больше площади изображения.

Другое свойство — **Stretch** позволяет подгонять не компонент под размер рисунка, а рисунок под размер компонента. Установите **AutoSize** в **false**, растяните или сожмите размер компонента **Image** и установите **Stretch** в **true**. Вы увидите, что рисунок займет всю площадь компонента, но поскольку вряд ли реально вручную установить размеры **Image** точно пропорциональными размеру рисунка, то изображение исказится. Свойство **Stretch** не действует на изображения пиктограмм, которые не могут изменять своих размеров (см. разд. 5.5.1.2).

Свойство — **Center**, установленное в **true**, центрирует изображение на площади **Image**, если размеры компонента и рисунка не совпадают.

Рассмотрим еще одно свойство — **Transparent** (прозрачность). Если **Transparent** равно **true**, то изображение в **Image** становится прозрачным. Это можно использовать для наложения изображений друг на друга. Поместите на форму второй компо-

нент **Image** и загрузите в него другую картинку. Только постарайтесь взять какую-нибудь мало заполненную, контурную картинку. Можете, например, взять картинку из числа помещаемых обычно на кнопки, например, стрелку (файл *...\program files\common files\borland shared\images\buttons\arrow1r.bmp*). Передвиньте ваши компоненты **Image** так, чтобы они перекрывали друг друга, и в верхнем компоненте установите **Transparent** равным **true**. Вы увидите, что верхняя картинка перестала заслонять нижнюю. Одно из возможных применений этого свойства — наложение на картинку надписей, выполненных в виде битовой матрицы. Эти надписи можно сделать с помощью встроенной в Delphi программы Image Editor.

Вы создали приложение, в котором на форме отображается выбранная вами в процессе проектирования картинка. Вы можете легко превратить его в более интересное приложение, в котором пользователь сможет просматривать и загружать любые графические файлы. Для этого достаточно перенести на форму компонент **OpenPictureDialog**, расположенный в библиотеке на странице Dialogs и вызывающий диалоговое окно открытия и предварительного просмотра изображения, а также кнопку, запускающую просмотр, или меню с единственным разделом Файл.

А теперь вам осталось написать всего один оператор в обработчике щелчка на кнопке или на разделе меню:

```
if OpenPictureDialog1.Execute then
 Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
```

Этот оператор загружает в свойство **Picture** компонента **Image1** файл, выбранный в диалоге пользователем. Выполните свое приложение и проверьте его в работе. Щелкая на кнопке, вы можете выбрать любой графический файл и загрузить его в компонент **Image1**.

В таком приложении есть один недостаток — изображения могут быть разных размеров и их положение на форме или будет несимметричным, или они не будут помещаться в окне. Это легко изменить, заставив форму автоматически настраиваться на размеры изображения. Для этого надо установить в компоненте **Image1** свойство **AutoSize** равным **true**, а приведенный ранее оператор изменить следующим образом:

```
if OpenPictureDialog1.Execute then
 begin
  Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
  Form1.ClientHeight := Image1.Height+10;
  Image1.Top := Form1.ClientRect.Top +
                (Form1.ClientHeight-Image1.Height) div 2;
  Form1.ClientWidth := Image1.Width+10;
  Image1.Left := Form1.ClientRect.Left +
                 (Form1.ClientWidth-Image1.Width) div 2;
 end;
```

В этом коде размеры клиентской области формы устанавливаются несколько больше размеров компонента **Image1**, которые в свою очередь адаптируются к размеру картинки благодаря свойству **AutoSize**.

Запустите теперь ваше приложение, и вы увидите (рис. 5.6), что при различных размерах изображения ваше приложение выглядит отлично.

Рис. 5.6
Адаптация формы к размерам изображения

5.5.1.2 Форматы графических файлов

Теперь следует немного разобраться с форматами графических файлов. Delphi поддерживает три типа файлов — битовые матрицы, пиктограммы и метафайлы. Все три типа файлов хранят изображения; различие заключается лишь в способе их хранения внутри файлов и в средствах доступа к ним. Битовая матрица (файл с расширением *.bmp*) отображает цвет каждого пиксела в изображении. При этом информация хранится таким образом, что любой компьютер может отобразить картинку с разрешающей способностью и количеством цветов, соответствующими его конфигурации.

Пиктограммы (файлы с расширением *.ico*) — это маленькие битовые матрицы. Они повсеместно используются для обозначения значков приложений, в быстрых кнопках, в пунктах меню, в различных списках. Способ хранения изображений в пиктограммах схож с хранением информации в битовых матрицах, но имеются и различия. В частности, пиктограмму невозможно масштабировать, она сохраняет тот размер, в котором была создана.

Метафайлы (Metafiles) хранят не последовательность битов, из которых состоит изображение, а информацию о способе создания картинки. Они хранят последовательности команд рисования, которые и могут быть повторены при воссоздании изображения. Это делает такие файлы, как правило, более компактными, чем битовые матрицы.

Для хранения графических объектов, содержащихся в битовых матрицах, пиктограммах и метафайлах, в Delphi определены соответствующие классы — **TBitmap**, **TIcon** и **TMetafile**. Все они являются производными от абстрактного базового класса графических объектов **TGraphic**. Кроме того, определен класс, являющийся надстройкой над **TBitmap**, **TIcon** и **TMetafile** и способный хранить любой из этих объектов. Это класс **TPicture**, с которым вы уже познакомились в разд. 5.5.1.1 — он соответствует свойству **Picture** компонента **Image**. Класс **TPicture** имеет свойство **Graphic**, которое может содержать и битовые матрицы, и пиктограммы, и метафайлы. Для доступа к графическому объекту можно использовать свойство **TPicture.Graphic**, но если тип графического объекта известен, то можно непосредственно обращаться к свойствам **TPicture.TBitmap**, **TPicture.Icon** или **TPicture.TMetafile**.

Для всех рассмотренных классов определены методы загрузки из файла и сохранения в файл:

```
procedure LoadFromFile(const FileName: string);
procedure SaveToFile(const FileName: string);
```

При этом для классов **TBitmap**, **TIcon** и **TMetafile** формат файла должен соответствовать классу объекта. Объект класса **TPicture** может оперировать с любым форматом.

Для всех рассмотренных классов определены методы присваивания значений объектов:

```
procedure Assign(Source: TPersistent);
```

Однако для классов **TBitmap**, **TIcon** и **TMetafile** присваивать можно только значения однородных объектов: соответственно битовых матриц, пиктограмм, метафайлов. При попытке присвоить значения разнородных объектов генерируется исключение **EConvertError**. Класс **TPicture** — универсальный, ему можно присваивать значения объектов любых из остальных трех классов. А значение **TPicture** можно присваивать только тому объекту, тип которого совпадает с типом объекта, хранящегося в нем.

Приведем пример. Часто в приложениях создается объект типа **TBitmap**, назначение которого — запомнить содержимое графического изображения и затем восстанавливать его, если оно будет испорчено или изменено пользователем. Код, решающий эту задачу, может иметь вид:

```
var Bitmap: TBitmap;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
 Bitmap:= TBitmap.Create;
 Bitmap.LoadFromFile('...');
 Image1.Picture.Assign(Bitmap);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 Bitmap.Free;
end;

procedure SaveClick(Sender: TObject);
begin
 Bitmap.Assign(Image1.Picture);
end;

procedure RestoreClick(Sender: TObject);
begin
 Image1.Picture.Assign(Bitmap);
end;
```

В этом коде сначала объявляется переменная **Bitmap** типа **TBitmap**. Затем в момент создания формы (при событии формы **OnCreate**) в процедуре **TForm1.FormCreate** создается объект **Bitmap** и в него методом **LoadFromFile** загружается изображение из указанного файла. Затем оператор

```
Image1.Picture.Assign(Bitmap);
```

присваивает значение графического объекта свойству **Picture** компонента **Image1**. Изображение тем самым делается видимым пользователю.

Этот оператор можно записать иначе:

```
Image1.Picture.Bitmap.Assign(Bitmap);
```

что даст тот же самый результат.

Если вы создали объект **Bitmap**, то надо не забыть его уничтожить при окончании работы и освободить от него память. Автоматически это не делается. Поэтому надо освобождать память, например, в обработчике события формы **OnDestroy** (процедура **FormDestroy**) методом **Free**:

```
Bitmap.Free;
```

Если надо переписать в **Bitmap** отредактированное пользователем в **Image1** изображение, это можно сделать оператором (процедура **SaveClick**):

```
Bitmap.Assign(Image1.Picture);
```

Если же надо восстановить в **Image1** прежнее изображение, испорченное по каким-то причинам, то это можно сделать оператором (процедура **RestoreClick**):

```
Image1.Picture.Assign(Bitmap);
```

Таким образом, мы видим, что методом **Assign** можно копировать изображение из одного однотипного графического объекта в другой и обратно.

Имеются еще методы загрузки и выгрузки графических объектов в поток и в буфер обмена **Clipboard**, подобные методам работы с файлами. Вы можете посмотреть их во встроенной справке Delphi или в [3].

5.5.2 Канва — холст для рисования

5.5.2.1 Канва и пикселы

Многие компоненты в Delphi имеют свойство **Canvas** (канва, холст), представляющее собой область компонента, на которой можно рисовать или отображать готовые изображения. Это свойство имеют формы, компоненты **Image** и многие другие. Канва содержит свойства и методы, существенно упрощающие графику Delphi. Все сложные взаимодействия с системой спрятаны для пользователя, так что рисовать в Delphi может человек, совершенно не искушенный в машинной графике.

Каждая точка канвы имеет координаты **X** и **Y**. Система координат канвы, как и везде в Delphi, имеет началом левый верхний угол канвы. Координата **X** возрастает при перемещении слева направо, а координата **Y** — при перемещении сверху вниз.

С координатами вы уже имели дело многократно, но пока вас не очень интересовало, что стоит за ними, в каких единицах они измеряются. Координаты измеряются в пикселах. *Пиксел* — это наименьший элемент поверхности рисунка, с которым можно манипулировать. Важнейшее свойство пиксела — его цвет. Для описания цвета используется тип **TColor**. С цветом вы встречаетесь практически в каждом компоненте и знаете, что в Delphi определено множество констант типа **TColor**. Одни из них непосредственно определяют цвета (например, **clBlue** — синий), другие определяют цвета элементов окон, которые могут меняться в зависимости от выбранной пользователем палитры цветов Windows (например, **clBtnFace** — цвет поверхности кнопок).

Рисовать на канве можно разными способами. Первый вариант — рисование по пикселам. Для этого используется свойство канвы **Pixels**. Это свойство представляет собой двумерный массив, который отвечает за цвета канвы. Например, **Canvas.Pixels[10, 20]** соответствует цвету пиксела, 10-го слева и 20-го сверху. С массивом пикселов можно обращаться как с любым свойством: изменять цвет, задавая пикселу новое значение, или определять его цвет по хранящемуся в нем значению. Например, **Canvas.Pixels[10,20] := 0** или **Canvas.Pixels[10,20] := clBlack** — это задание пикселу черного цвета.

Вы, вероятно, можете без особых трудов составить программу, которая рисовала бы по пикселам некоторый график. Но мы не будем на этом останавливаться, так как гораздо эффективнее и более качественно рисовать на канве можно с помощью специальных методов канвы.

5.5.2.2 Рисование с помощью пера Pen

У канвы имеется свойство **Pen** — перо. Это объект, в свою очередь имеющий ряд свойств. Одно из них уже известное вам свойство **Color** — цвет, которым наносится рисунок. Второе свойство — **Width** (ширина линии). Ширина задается в пикселах. По умолчанию ширина равна 1.

Свойство **Style** определяет вид линии. Это свойство может принимать следующие значения:

psSolid	сплошная линия
psDash	штриховая линия
psDot	пунктирная линия
psDashDot	штрих-пунктирная линия
psDashDotDot	линия, чередующая штрих и два пунктира
psClear	отсутствие линии
psInsideFrame	сплошная линия, но при **Width** > 1 допускающая цвета, отличные от палитры Windows

Все стили со штрихами и пунктирами доступны только при **Width** = 1. При большей ширине линии всех стилей рисуются как сплошные.

У канвы имеется свойство **PenPos** типа **TPoint**. Это свойство определяет в координатах канвы текущую позицию пера. Перемещение пера без прорисовки линии, т.е. изменение **PenPos**, производится методом канвы **MoveTo(X,Y)**. Здесь **X** и **Y** — координаты точки, в которую перемещается перо. Эта текущая точка становится исходной, от которой методом **LineTo(X,Y)** можно провести прямую линию в точку с координатами **(X,Y)**. При этом текущая точка перемещается в конечную точку линии, и новый вызов **LineTo** будет проводить линию из этой новой текущей точки.

Перо может рисовать не только прямые линии, но и фигуры. Ниже перечислены некоторые из методов канвы, использующие перо для рисования фигур:

Arc	рисует дугу окружности или эллипса
Chord	рисует замкнутую фигуру, ограниченную дугой окружности или эллипса и хордой
Ellipse	рисует окружность или эллипс

Pie	рисует сектор окружности или эллипса
PolyBezier	рисует кусочную кривую третьего порядка с точным отображением первой и последней точки; число точек должно быть кратно 3
PolyBezierTo	рисует кусочную кривую третьего порядка с точным отображением последней точки; число точек должно быть на 1 меньше числа, кратного 3
Polygon	рисует замкнутую фигуру с кусочно-линейной границей
Polyline	рисует кусочно-линейную кривую
Rectangle	рисует прямоугольник
RoundRect	рисует прямоугольник со скругленными углами

Примеры большинства перечисленных фигур приведены на рис. 5.7.

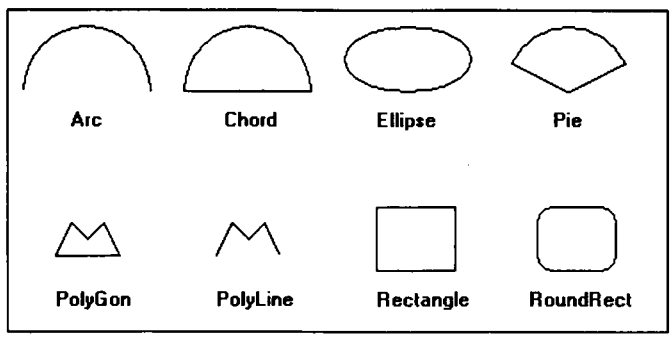


Рис. 5.7
Примеры фигур, нарисованных пером

Ниже приведен текст процедуры, которая рисовала фигуры, показанные на рис. 5.7. Этот текст поможет вам понять методы, осуществляющие рисование фигур. Подробное описание методов рисования вы найдете во встроенной справке Delphi или в справке [3].

```
with Image1.Canvas do
 begin
  Font.Style := [fsBold];
  Arc(10,10,90,90,90,50,10,50);
  TextOut(40,60,'Arc');
  Chord(110,10,190,90,190,50,110,50);
  TextOut(135,60,'Chord');
  Ellipse(210,10,290,50);
  TextOut(230,60,'Ellipse');
  Pie(310,10,390,90,390,30,310,30);
  TextOut(340,60,'Pie');
  PolyGon([Point(30,150),Point(40,130),Point(50,140),
           Point(60,130),Point(70,150)]);
  TextOut(30,170,'PolyGon');
  PolyLine([Point(130,150),Point(140,130),Point(150,140),
            Point(160,130),Point(170,150)]);
  TextOut(130,170,'PolyLine');
  Rectangle(230,120,280,160);
  TextOut(230,170,'Rectangle');
```

```
  RoundRect(330,120,380,160,20,20);
  TextOut(325,170,'RoundRect');
end;
```

Для вывода текста на канву в приведенном примере использован метод **TextOut**, синтаксис которого:

```
TextOut(X,Y,<текст>);
```

Имеется еще несколько методов вывода текста, которые применяются реже. Все методы вывода текста используют свойство канвы **Font** — шрифт, с которым вы уже знакомы. В частности, в приведенном примере с помощью этого свойства установлен жирный шрифт надписей.

5.5.2.3 Brush — кисть

У канвы имеется свойство **Brush** — кисть. Это свойство определяет фон и заполнение замкнутых фигур на канве. **Brush** — это объект, имеющий в свою очередь ряд свойств. Свойство **Color** определяет цвет заполнения. Свойство **Style** определяет шаблон заполнения (штриховку) и может принимать значения:

Значение	Шаблон	Значение	Шаблон
bsSolid		bsCross	
bsClear		bsDiagCross	
bsBDiagonal		bsHorizontal	
bsFDiagonal		bsVertical	

Имеются функции канвы, рисующие заполненные фигуры. Это, например, метод **FillRect**, объявленный как

```
procedure FillRect(const Rect: TRect);
```

Он заполняет заданным стилем прямоугольную область, заданную параметром **Rect**. Этот параметр имеет тип **TRect**. Для его задания проще всего использовать функцию **Rect(X1,Y1,X2,Y2)**, возвращающую структуру **Rect** с координатами углов, заданных параметрами **(X1, Y1)** и **(X2, Y2)**.

Функцию **FillRect** удобно, в частности, использовать для стирания изображения. Например, оператор

```
with Image1 do Canvas.FillRect(Rect(0,0,Width,Height));
```

очищает всю площадь канвы компонента **Image1**.

Кисть участвует в заполнении фигур не только с помощью этой функции. Все перечисленные ранее методы рисования замкнутых фигур тоже заполняют их с помощью кисти. Это относится к методам **Chord**, **Ellipse**, **Pie**, **Polygon** и др.

Имеется еще один метод канвы, связанный с кистью. Это метод **FrameRect**. Он рисует на канве текущей кистью прямоугольную рамку, не закрашивая ее. Синтаксис метода **FrameRect**:

```
procedure FrameRect(const Rect: TRect);
```

Параметр **Rect** определяет позицию и размеры прямоугольной рамки. Толщина рамки — 1 пиксел. Область внутри рамки кистью не заполняется. Метод **FrameRect** отличается от рассмотренного ранее метода **Rectangle** тем, что рамка рисуется цветом кисти (в методе **Rectangle** — цветом пера) и область не закрашивается (в методе **Rectangle** закрашивается).

5.5.2.4 Пример рисования на канве

С помощью рассмотренных методов канвы можно рисовать графики и диаграммы. Но это намного проще делать с помощью рассмотренного в разд. 5.5.3 специального компонента. Так что рисование на канве чаще используется для различных иллюстративных целей. В качестве примера рассмотрим иллюстрацию решения уравнения методом дихотомии. Этот метод вы реализовывали в гл. 2 в разд. 2.8.7.4. Попробуем снабдить его графической иллюстрацией, которая показала бы последовательность поиска корня.

Рассматриваемое приложение приведено на рис. 5.8. Методом дихотомии решается уравнение X * exp(X) − 10 = 0, хотя, конечно, вы можете решать любое другое. В окнах редактирования **Edit1**, **Edit2** и **Edit3** пользователь задает начальный интервал неопределенности и требуемую точность поиска. Кнопка Выполнить инициирует поиск. В метке **Label1** отображается результат поиска, а в компоненте **Image1** — процесс поиска. Утолщенной линией показан график левой части уравнения в заданных пределах. Сплошная горизонтальная линия соответствует нулевому значению, так что точка пересечения графика с этой линией показывает искомый корень. Пунктиром отмечены эксперименты, проводимые в процессе поиска. Цифры указывают номера экспериментов.

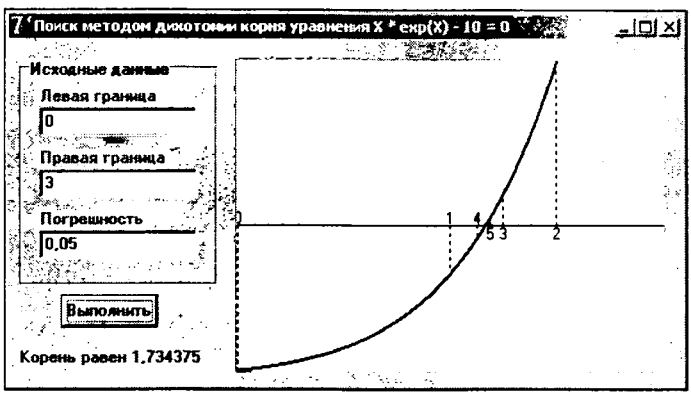


Рис. 5.8
Пример рисования на канве

Ниже приведен листинг этого приложения

```
function F(X: real): real;
// Функция левой части уравнения
begin
 Result := X * exp(X) - 10.;
end;

procedure TForm1.Button1Click(Sender: TObject);
// Поиск методом дихотомии корня уравнения
```

```
var a,  // нижняя граница интервала неопределенности
    b,  // верхняя граница интервала неопределенности
    Dx, // удвоенная допустимая погрешность
    Fa, // значение функции в начале интервала неопределенности
    X,  // текущее приближение корня
    Fx, // значение функции в текущей точке

    // переменные для графика:
    MinF, MaxF, // минимальное и максимальное значения функции
    a0, b0:     // начальный интервал неопределенности
      real;
    PX, PY,                 // координаты в пикселах
    PZero: longint;         // Y-координата нуля в пикселах
    i: integer;             // счетчик циклов
    Arr: array of real;     // динамический массив точек
begin
 // исходные данные для расчета
 a := StrToFloat(Edit1.Text);
 a0 := a;
 b := StrToFloat(Edit2.Text);
 b0 := b;
 Dx := 2. * StrToFloat(Edit3.Text);
 Fa := F(a);
 // занесение в массив точки, соответствующей a0
 SetLength(Arr, 2);
 Arr[0] := a;
 Arr[1] := Fa;
 // задание начальных значений максимума и минимума функции
 MaxF := Fa;
 MinF := Fa;
 // цикл поиска
 repeat
  // новый эксперимент и функция в новой точке
  X := (a + b) / 2;
  Fx := F(X);
  // запоминание точки в массиве
  SetLength(Arr, Length(Arr) + 2);
  Arr[Length(Arr) - 2] := X;
  Arr[Length(Arr) - 1] := Fx;
  // коррекция MaxF и MinF
  if Fx > MaxF then MaxF := Fx;
  if Fx < MinF then MinF := Fx;
  // изменение границ интервала неопределенности
  if (Fa * Fx > 0)
   then a := X
   else b := X;
   // проверка критерия окончания
 until (b - a <= DX);

 // отображение результата
 Label1.Caption := 'Корень равен ' + FloatToStr((a + b) / 2);

 // коррекция MaxF и MinF в случае, если на интервале нет корня
 if MaxF < 0 then MaxF := 0;
 if MinF > 0 then MinF := 0;
 // расчет Y-кординаты нуля
```

```
 PZero := trunc(Image1.Height + MinF * Image1.Height /
                (MaxF - MinF));
 with Image1.Canvas do
 begin
  // очистка площади графика
  FillRect(Rect(0, 0, Image1.Width, Image1.Height));
  // свойства пера для отображения экспериментов
  Pen.Width := 1;
  Pen.Style := psDot;
  // цикл отображения экспериментов
  for i := 1 to Length(Arr) div 2 do
  begin
   // перевод координат в пикселы
   PX := trunc((Arr[2*i-2] - a0) * Image1.Width / (b0 - a0));
   PY := trunc(Image1.Height - (Arr[2*i-1] - MinF) *
               Image1.Height / (MaxF - MinF));
   // рисование линии
   MoveTo(PX, PZero);
   LineTo(PX, PY);
   // запись номера эксперимента
   if (PY > PZero)
    then TextOut(PX - 3, PZero - 12, IntToStr(i-1))
    else TextOut(PX - 3, PZero, IntToStr(i-1));
  end;
  // проведение нулевой линии
  Pen.Style := psSolid;
  MoveTo(0, PZero);
  LineTo(Image1.Width, PZero);
  // начало изображения графика
  Pen.Width := 2;
  MoveTo(0, trunc(Image1.Height - (Arr[1] - MinF) *
                  Image1.Height / (MaxF - MinF)));
  // цикл рисования графика
  for PX:=0 to Image1.Width do
  begin
   X := a0 + PX * (b0 - a0) / Image1.Width;
   PY := trunc(Image1.Height - (F(X) - MinF) * Image1.Height /
               (MaxF - MinF));
   LineTo(PX,PY);
  end;
 end;
end;
```

Функция **F** рассчитывает левую часть уравнения для заданного значения аргумента **X**. Эту функцию вы можете заменить любой другой. Можно также ввести в приложение группу радиокнопок или выпадающий список, с помощью которого пользователь сможет выбирать одну из нескольких функций. Тогда в **F** надо будет предусмотреть переключатель, обеспечивающий вычисление той или иной функции.

В обработчике щелчка на кнопке определен ряд локальных переменных. Первые из них, вплоть до **Fx**, не отличаются от рассмотренных в разд. 2.8.7.4. Остальные переменные введены для организации отображения данных в **Image1**. В переменных **MinF** и **MaxF** будет формироваться соответственно минимальное и максимальное значения функции, полученные в экспериментах. Эти значения понадобятся для выбора масштаба по оси Y. В переменных **a0** и **b0** будут хранить-

ся границы начального интервала неопределенности. Эти значения затем используются для выбора масштаба по оси X. Переменные **PX** и **PY** — координаты точек графика в пикселах. Переменная **PZero** — координата нуля по оси Y в пикселах.

Массив **Arr** — это динамический массив (см. разд. 3.1.3), в котором будут храниться точки экспериментов. Дело в том, что не зная диапазон изменения функции (значения **MinF** и **MaxF**), невозможно определить масштаб отображения данных по оси Y. А значения **MinF** и **MaxF** известны только после проведения всех экспериментов. Так что графическое отображение будет осуществляться после окончания поиска. И чтобы отобразить точки всех экспериментов их надо где-то хранить. Для этого будет использоваться массив **Arr**. Координаты будут храниться в нем попарно: значение X и значение функции в этой точке.

Первые операторы процедуры **Button1Click** читают заданные пользователем параметры расчета, запоминают границы начального интервала неопределенности в переменных **a0** и **b0**, рассчитывают функцию в точке **a** и запоминают эту точку. Далее следует цикл поиска, не отличающийся в основном от рассмотренного в разд. 2.8.7.4. Только на каждой итерации размер массива **Arr** увеличивается на 2, в массив заносятся координаты очередной точки, и корректируются значения **MinF** и **MaxF**.

По окончании поиска идет подготовка к графическому отображению результатов. Сначала проверяются значения **MinF** и **MaxF**. Если **MaxF** < 0 или **MinF** > 0, значит функция на всем интервале неопределенности имеет постоянный знак и корень отсутствует. Соответственно нулевое значение координаты Y лежит вне диапазона изменения функции. Поскольку вертикальные линии, отображающие результаты экспериментов, мы хотим проводить от нулевой горизонтальной линии, то в подобных случаях значение **MaxF** или **MinF** корректируется, чтобы нуль находился в пределах графика. Затем вычисляется значение **PZero** — нулевое значение координаты Y в пикселах. Вспомним, что координата Y нарастает сверху вниз, а нам надо использовать общепринятое в математике направление оси Y снизу вверх. Поскольку мы хотим отображать значения в пределах от **MinF** до **MaxF**, то значению **MaxF** должно соответствовать в пикселах нулевое значение (верх графика), а значению **MinF** — **Image1.Height** (низ графика). Значит, некоторой величине **Y** должна соответствовать координата в пикселах, равная **Image1.Height** – (**Y** – **MinF**) * **Image1.Height** / (**MaxF** – **MinF**). При **Y** = 0 получается значение, равное **Image1.Height** + **MinF** * **Image1.Height** / (**MaxF** – **MinF**), которое запоминается в **PZero**.

После вычисления **PZero** площадь графика очищается методом канвы **FillRect**. Перу канвы **Pen** задается единичная ширина и стиль пунктирной линии. Затем следует цикл по точкам, запомненным в массиве **Arr**. Для каждой точки вычисляются координаты в пикселах **PX** и **PY**. Методом **MoveTo** перо перемещается в нулевую линию с кооридатой X равной **PX**. А затем методом **LineTo** проводится вертикальная прямая в точку с координатой Y равной **PY**.

После того как линия проведена, методом **TextOut** на график заносится номер эксперимента. Место, на которое записывается номер, выбирается ниже или выше нулевой горизонтальной линии в зависимости от того, была ли вертикальная линия проведена вверх или вниз.

По окончании цикла по точкам экспериментов стиль пера задается сплошным и проводится горизонтальная нулевая линия. Затем рисуется график функции линией двойной толщины. Сначала методом **MoveTo** перо перемещается к началу

графика, а затем выполняется цикл, в котором методом **LineTo** проводится линия к очередной точке.

Мы рассмотрели сравнительно простую графическую иллюстрацию. Методы канвы позволяют решать гораздо более сложные задачи. Например, можно построить полноценный графический редактор, с помощью которого пользователь будет рисовать и редактировать изображения. Подобный пример желающие могут посмотреть в книге [1].

5.5.3 Графики и диаграммы в компоненте TeeChart

Теперь рассмотрим компонент **Chart**, который расположен в библиотеке на странице Additional. Этот компонент позволяет строить различные диаграммы и графики, которые выглядят очень эффектно (рис. 5.9). Компонент **Chart** имеет множество свойств, методов, событий, так что если все их рассматривать, то этому пришлось бы посвятить целую главу. Поэтому ограничимся рассмотрением только основных характеристик **Chart**. А с остальными вы можете ознакомиться по встроенной справке Delphi, справке [3], или просто опробовать их, экспериментируя с диаграммами.

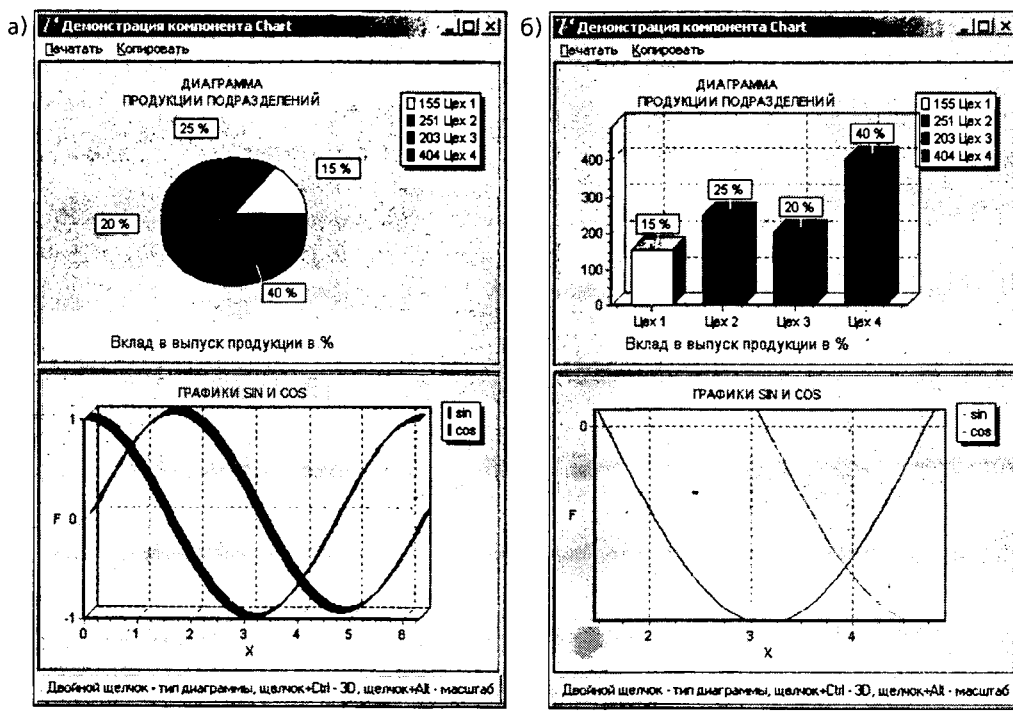


Рис. 5.9

Пример приложения с диаграммами: начальное состояние (а); состояние при изменении типа диаграммы, устранении трехмерности и увеличении фрагмента графика (б)

Компонент **Chart** является контейнером объектов **Series** — наследников класса **TChartSeries**. Каждый такой объект представляет серию данных, характеризующихся определенным стилем отображения: тем или иным графиком или диаграм-

мой. Каждый компонент **Chart** может включать несколько серий. Если вы хотите отображать график, то каждая серия будет соответствовать одной кривой на графике. Если вы хотите отображать диаграммы, то для некоторых видов диаграмм можно наложить друг на друга несколько различных серий, для других (например, для круговых диаграмм) это, вероятно, будет выглядеть некрасиво. Однако и в этом случае вы можете задать для одного компонента **Chart** несколько серий одинаковых данных с разным типом диаграммы. Тогда, делая в каждый момент времени активной одну из них, вы можете предоставить пользователю выбор типа диаграммы, отображающей интересующие его данные.

Давайте, построим тестовое приложение, демонстрирующее некоторые возможности компонента **Chart**. Заодно оно позволит применить на практике ряд рассмотренных ранее приемов проектирования окон с изменяющимися размерами. Вид этого приложения показан на рис. 5.9. В верхнем окне отображается диаграмма выпуска продукции на некотором абстрактном предприятии. Двойной щелчок на диаграмме позволяет выбрать ее вид: круговая (рис. 5.9 а) или вертикальная гистограмма (рис. 5.9 б). В нижнем окне отображаются графики синуса и косинуса.

И на диаграмме, и на графике пользователь может выделить некоторую область изображения и рассмотреть ее детальнее. О том, как это делается, будет сказано позднее. Если пользователь изменит таким образом масштаб, то щелчок при нажатой клавише Alt позволяет его восстановить. Щелчок при нажатой клавише Ctrl позволяет управлять трехмерностью изображение. Например, график на рис. 5.9 а представлен объемным, а на рис. 5.9 б — плоским.

При изменении размеров окна формы изменяются и размеры графиков. Кроме того, между окном диаграммы и графика имеется разделитель, потянув за который пользователь может перемещать границу между этими окнами.

Внизу окна формы имеется панель, в которой указаны доступные пользователю команды. Имеется меню, позволяющее печатать и копировать в буфер обмена диаграмму или график.

Начните новый проект. Поместите на форму компонент **Chart** и установите его свойство **Align** равным **alTop**. Компонент займет всю верхнюю часть формы. Перенесите на форму компонент **Panel**. Мы будем использовать эту панель просто для отображения текста о доступных командах. Установите ее свойство **Align** равным **alBottom**, чтобы панель заняла нижнюю часть формы. Далее перенесите на форму компонент **Splitter**, который должен разделять окна диаграммы и графика. Так как граница между этими окнами горизонтальная, установите в компоненте **Splitter** значение свойства **Align** равным **alTop**. Разделитель прижмется снизу к компоненту **Chart1**, занимающему верхнюю часть формы. Теперь перенесите на форму еще один компонент **Chart** и задайте значение свойства **Align** равным **alClient**. Компонент займет все пространство формы межу разделителем и нижней панелью.

Осталось поместить на форму компонент **MainMenu** и создать в нем два меню Печатать и Копировать с двумя разделами в каждом: Диаграмму и График. На этом проектирование интерфейса закончено, и теперь можно заняться его основными компонентами **Chart**.

Выделите верхний компонент **Chart1** и посмотрите его свойства в Инспекторе Объектов. Приведем пояснения некоторых из них.

AllowPanning	Определяет возможность пользователя прокручивать наблюдаемую часть графика во время выполнения, нажимая правую кнопку мыши. Возможные значения: **pmNone** — прокрутка запрещена, **pmHorizontal**, **pmVertical** или **pmBoth** — разрешена соответственно прокрутка только в горизонтальном направлении, только в вертикальном, или в обоих направлениях.
AllowZoom	Позволяет пользователю изменять во время выполнения масштаб изображения, вырезая фрагменты диаграммы или графика курсором мыши (на рис. 5.9 б внизу показан момент просмотра фрагмента графика, целиком представленного на рис. 5.9 а). Если рамка фрагмента рисуется вправо и вниз, то этот фрагмент растягивается на все поле графика. А если рамка рисуется вверх и влево, то восстанавливается исходный масштаб.
Title	Определяет заголовок диаграммы.
Foot	Определяет подпись под диаграммой. По умолчанию отсутствует. Текст подписи определяется подсвойством **Text**.
Frame	Определяет рамку вокруг диаграммы.
Legend	Легенда диаграммы — список обозначений.
MarginLeft, **MarginRight**, **MarginTop**, **MarginBottom**	Значения левого, правого, верхнего и нижнего полей.
BottomAxis, **LeftAxis**, **RightAxis**	Эти свойства определяют характеристики соответственно нижней, левой и правой осей. Задание этих свойств имеет смысл для графиков и некоторых типов диаграмм.
LeftWall, **BottomWall**, **BackWall**	Эти свойства определяют характеристики соответственно левой, нижней и задней граней области трехмерного отображения графика.
SeriesList	Список серий данных, отображаемых в компоненте.
View3d	Разрешает или запрещает трехмерное отображение диаграммы.
View3DOptions	Характеристики трехмерного отображения.
Chart3DPercent	Масштаб трехмерности (для рис. 5.9 это толщина диаграммы и ширина лент графика).

Рядом со многими из перечисленных свойств в Инспекторе Объектов расположены кнопки с многоточием, которые позволяют вызвать ту или иную страницу Редактора Диаграмм — многостраничного окна, позволяющего установить все свойства диаграмм. Вызов Редактора Диаграмм возможен также двойным щелчком на компоненте **Chart** или щелчком на нем правой кнопкой мыши и выбором команды Edit Chart во всплывшем меню.

Сделайте двойной щелчок на верхнем компоненте **Chart**. Вы попадете в окно Редактора Диаграмм (рис. 5.10) на страницу Chart, которая имеет несколько закладок. Прежде всего, вас будет интересовать на ней закладка Series. Щелкните на кнопке Add — добавить серию. Вы попадете в окно (рис. 5.11), в котором вы можете выбрать тип диаграммы или графика. В данном случае выберите Pie — круговую диаграмму. Воспользовавшись закладкой Titles, вы можете задать заголовок диаграммы, закладка Legend позволяет задать параметры отображения легенды диа-

граммы (списка обозначений) или вообще убрать ее с экрана, закладка Panel определяет вид панели, на которой отображается диаграмма, закладка 3D дает вам возможность изменить внешний вид вашей диаграммы: наклон, сдвиг, толщину и т.д.

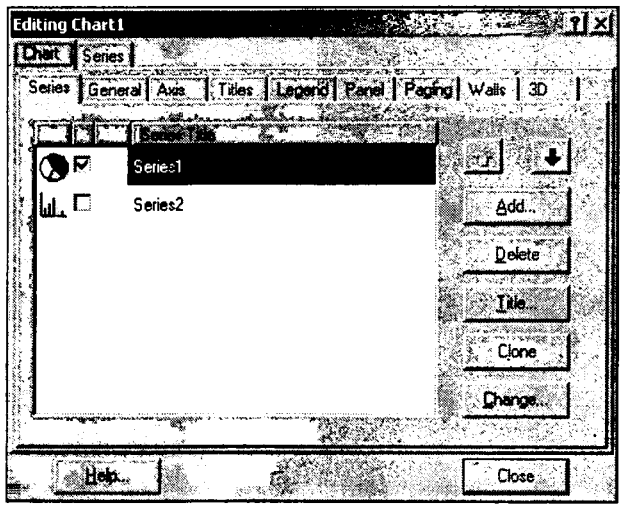


Рис. 5.10
Редактор Диаграмм, страница Chart, закладка Series

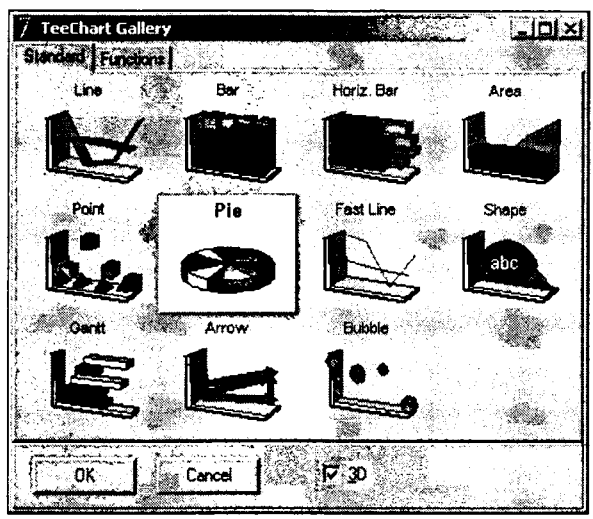


Рис. 5.11
Выбор типа диаграммы в Редакторе Диаграмм

Когда вы работаете с Редактором Диаграмм и выбрали тип диаграммы, в компонентах **Chart** на вашей форме отображается ее вид с занесенными в нее условными данными (см. рис. 5.9). Поэтому вы сразу можете наблюдать результат применения различных опций к вашему приложению, что очень удобно.

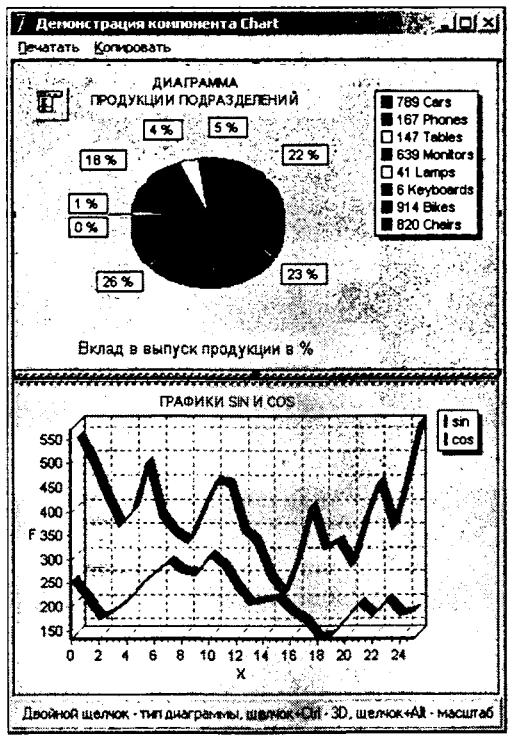


Рис. 5.12
Форма приложения рис. 5.9 с занесенными в нее условными данными

Страница Series, также имеющая ряд закладок, дает вам возможность выбрать дополнительные характеристики отображения серии. В частности, для круговой диаграммы на закладке Format полезно включить опцию Circled Pie, которая обеспечит при любом размере компонента Chart отображение диаграммы в виде круга. На закладке Marks кнопки группы Style определяют, что будет написано на ярлычках, относящихся к отдельным сегментам диаграммы: Value — значение, Percent — проценты, Label — названия данных и т.д. В примере рис. 5.9 включена кнопка Percent, а на закладке General установлен шаблон процентов, обеспечивающий отображение только целых значений.

Итак, введите в верхний компонент **Chart1** серию круговой диаграммы. Нам потребуется еще одна диаграмма с теми же данными, но иного типа. Создать в компоненте **Chart** еще одну тождественную серию можно, нажав на закладке Series страницы Chart кнопку Clone. Затем для этой новой серии нажать кнопку Change (изменить) и выбрать другой тип диаграммы — в нашем случае Bar. Конечно, два разных типа диаграммы на одном рисунке будут выглядеть плохо. Но вы можете выключить индикатор этой новой серии на закладке Series, а потом предоставить пользователю возможность выбрать тот или иной вид отображения диаграммы (ниже будет показано, как это делается).

Выйдите из Редактора Диаграмм, выделите в вашем приложении нижний компонент Chart и повторите для него задание свойств с помощью Редактора Диаграмм. В данном случае вам надо будет задать две независимые серии, так как мы хотим отображать на графике две разные кривые, и выбрать тип диаграммы Line.

Отмечу, что для деловой графики лучше выбирать тип серии не Line, а Fast Line. Этот тип обеспечивает наиболее быстрое построение графиков. Но графики этого типа не дают объемного эффекта (они выглядят так, как показано на рис. 5.6 б). Обычно этот эффект в графиках реальных приложений и не нужен, так как он только мешает восприятию. Но в нашем чисто тестовом приложении выберите все-таки тип Line, чтобы можно было посмотреть график и в объемном виде. Поскольку речь идет о графиках, вы можете воспользоваться закладками Axis и Walls для задания координатных характеристик осей и трехмерных граней графика.

Теперь можно написать код, задающий данные, которые вы хотите отображать. Для тестового приложения давайте зададим в круговой диаграмме просто некоторые константные данные, а в графиках — функции синус и косинус.

Для задания отображаемых значений надо использовать методы серий **Series**. Остановимся только на трех основных методах.

Метод **Clear** очищает серию от занесенных ранее данных.

Метод **Add**:

```
Add(Const AValue:Double; Const ALabel:String; AColor:TColor)
```

позволяет добавить в диаграмму новую точку. Параметр **AValue** соответствует добавляемому значению, параметр **ALabel** — метка, которая будет отображаться на диаграмме и в легенде, **AColor** — цвет. Параметр **ALabel** — не обязательный, его можно задать пустым: ''.

Метод **AddXY**:

```
AddXY(Const AXValue, AYValue: Double; Const ALabel: String;
        AColor: TColor)
```

позволяет добавить новую точку в график функции. Параметры **AXValue** и **AYValue** соответствуют аргументу и функции. Параметры **ALabel** и **AColor** те же, что и в методе **Add**.

Таким образом, в нашем примере можно включить в обработчик события **OnCreate** формы следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
const A1=155;
      A2=251;
      A3=203;
      A4=404;
var i:word;
begin
 with Series1 do
 begin
  Clear;
  Add(A1,'Цех 1',clYellow);
  Add(A2,'Цех 2',clBlue);
  Add(A3,'Цех 3',clRed);
  Add(A4,'Цех 4',clPurple);
 end;
 Series2.Assign(Series1);
 Series2.Active:=false;
 Series3.Clear;
 Series4.Clear;
 for i:=0 to 100 do begin
  Series3.AddXY(0.02*Pi*i,sin(0.02*Pi*i),'',clRed);
```

```
  Series4.AddXY(0.02*Pi*i,cos(0.02*Pi*i),'',clBlue);
 end;
end;
```

Операторы **Clear** в нашем приложении, строго говоря, не нужны. Они понадобились бы, если бы в процессе работы приложения пользователь мог обновлять данные. Тогда без этих операторов повторное выполнение методов **Add** и **AddXY** только добавило бы новые точки, не удалив прежние.

Данные в серию **Series1** заносятся методом **Add**. Затем эти данные методом **Assign** копируются в серию **Series2**. Эта серия делается неактивной заданием ее свойству **Active** значения **false**. Таким образом, в первый момент будет видна только первая серия — круговая диаграмма. Заполнение серий **Series3** и **Series4** и нижнего компонента **Chart1** вряд ли требует комментариев.

Поскольку в компоненте **Chart1** предусмотрены две серии **Series1** и **Series2** разных видов — Pie и Bar, надо написать код, изменяющий по требованию пользователя тип диаграммы. Этот код введите в обработчик событие **On DblClick** компонента **Chart**:

```
Series1.Active:= not Series1.Active;
Series2.Active:= not Series2.Active;
Chart1.View3dOptions.Orthogonal := Series2.Active;
```

Первые два оператора переключают активность серий. Последний оператор в случае, если активной становится серия **Series2**, включает ортогональное отображение диаграммы. Дело в том, что для круговой диаграммы ортогональность автоматически выключается. Так что при смене вида диаграммы надо управлять ортогональностью программно. На рис. 5.9 б вы можете видеть результат переключения пользователя на другой вид диаграммы.

Выше при описании свойств компонента **Chart** описывалось, как пользователь может изменять во время выполнения масштаб изображения. Для этого достаточно вырезать фрагмент диаграммы или графика курсором мыши, обведя его с помощью мыши рамкой, перемещая курсор вправо и вниз. В результате фрагмент растянется на все поле графика. Можете проверить этот механизм, выполнив ваше приложение. Пользователь может также прокручивать наблюдаемую часть графика во время выполнения, нажимая правую кнопку мыши и перемещая курсор. Если, исследовав вырезанный фрагмент, пользователь захочет вернуться к исходному масштабу, ему достаточно нарисовать на графике рамку, перемещая курсор вверх и влево. Но подобный способ возврата к исходному масштабу не очень удобен. Мы собирались обеспечить возвращение к исходному масштабу при нажатии кнопки мыши с одновременным удержанием клавиши Alt. А при нажатии кнопки мыши с одновременным удержанием клавиши Ctrl мы собирались управлять трехмерностью изображения. Для реализации этих возможностей надо написать общий для обоих компонентов **Chart** обработчик события **OnMouseDown**, возникающего при нажатии кнопки мыши (см. разд. 4.14). Ниже приведен код этого обработчика:

```
procedure TForm1.Chart2MouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 if(ssAlt in Shift)
   then (Sender as TChart).UndoZoom
 else if (ssCtrl in Shift)
   then (Sender as TChart).View3d := not (Sender as TChart).View3d;
end;
```

Если при нажатии кнопки мыши нажата клавиша Alt, то для компонента **Chart**, в котором произошло событие, выполняется метод **UndoZoom**, восстанавливающий масштаб. А если нажата клавиша Alt, то в компоненте **Chart** изменяется значение свойства **View3d**, управляющего трехмерностью изображения. Приведенный обработчик события записан в общем случае, без указания конкретного компонента **Chart** (см. описание подобных обработчиков событий в разд. 2.4.7). Это позволяет использовать его для обоих компонентов **Chart**.

Осталось реализовать действия, соответствующие разделам меню. При выборе пользователем в меню Печатать раздела Диаграмму надо выполнить оператор:

```
Chart1.Print;
```

Он обеспечит печать диаграммы. Для печати графика надо выполнить то же метод **Print**, но для компонента **Chart2**.

При выборе пользователем в меню Копировать раздела Диаграмму надо выполнить оператор:

```
Chart1.CopyToClipboardBitmap;
```

Он обеспечит копирование диаграммы в буфер обмена. После этого изображение можно прочитать из буфера обмена в какой-то другой программе, например, в Word. Для копирование графика надо выполнить то же метод **CopyToClipboardBitmap**, но для компонента **Chart2**.

На этом мы закончим знакомство с компонентом **Chart**. Правда, мы рассмотрели только малую часть его возможностей. В частности, не затрагивались многие интересные типы диаграмм, вопросы построения графиков многозначных функций, способы предоставления пользователю возможности вращать диаграммы в любых направлениях, задание функциональных связей между сериями и многое другое. Интересующимся этими возможностями придется изучить их самостоятельно по встроенной справке Delphi или по справкам [3]. В книге [4] компоненту **Chart** уделена целая глава, и там дается развернутая методика построения диаграмм и графиков. А в рамках данного учебника придется ограничиться этими краткими сведениями. Впрочем, и их достаточно, чтобы снабдить многие из разработанных вами приложений графиками и диаграммами.

5.6 Некоторые итоги

Мы завершили рассмотрение вопросов, связанных с разработкой графического интерфейса пользователя. В заключение хотелось бы еще раз обратить внимание на то, что грамотный разработчик должен исходить из принципа: «Пользователь всегда прав». Об этом уже говорилось в разд. 2.12 и 3.4.7, но сейчас посмотрим, как этот принцип должен сказываться на проектировании интерфейса.

Если пользователь путается в кнопках и окнах программы, тратит время на поиск нужного ему раздела меню, не знает, как выполнить какую-то операцию, значит, ваш интерфейс никуда не годится. Проанализируйте, в чем причина затруднений пользователя. Может быть, у вас неудачная компоновка формы, функционально связанные компоненты не объединены панелями, и пользователю приходится искать, где же та кнопка, которая выполнит нужную операцию. Причина затруднений может быть в неудачных надписях на кнопках и панелях, которые не столько помогают пользователю, сколько запутывают его. Подобные недо-

четы часто встречаются в плохо русифицированных зарубежных программах. Возможно, работу пользователя затрудняет плохо организованная система подсказок: отсутствие всплывающих ярлычков, развернутых подсказок в полосе состояния, отсутствие файла справки.

Причиной ошибок может служить также непродуманная последовательность табуляции (см. разд. 5.2.3). Если пользователю для задания каких-то данных приходится метаться по площади окна приложения, щелкая на различных кнопках, списках, индикаторах, то возрастает вероятность, что он промахнется, щелкнет не там, где хотел, и введет неверную информацию, или забудет ввести какие-то безусловно необходимые данные. Так что продуманная последовательность табуляции, возможность переходить от одного управляющего элемента к другому простым нажатием клавиши Tab, а еще лучше — клавиши Enter, является необходимым атрибутом хорошего приложения.

Забота о пользователе должна также проявляться в использовании компонентов, позволяющих минимизировать количество возможных ошибок при вводе информации. Например, рассмотренная в разд. 5.4.1 замена окна редактирования списком при выборе ответа из конечного числа альтернатив, рассмотренная в разд. 5.4.4 замена окна редактирования компонентом **SpinEdit** при вводе целых чисел — все это способствует уменьшению число невольных ошибок при вводе данных. Вспомните также рассмотренные в разд. 2.8.5 и 2.8.6 способы ограничения множества допустимых символов, вводимых в окнах редактирования. Например, в окне ввода фамилии и другой сугубо текстовой информации имеет смысл запретить ввод цифр. А при вводе чисел — запретить ввод букв. Подобные приемы позволяют сократить ошибки, появляющиеся при случайном вводе неправильных символов. Посмотрите также в справке, встроенной в Delphi, или в справке [3] компонент **MaskEdit**, который позволяет обеспечить маскированный ввод данных. Мы не смогли из-за ограничения объема книги его рассмотреть, но он очень помогает в обеспечении безошибочного ввода данных пользователем.

Несмотря на все принятые меры предосторожности, ошибки при вводе данных все равно возможны. Поэтому любое приложение должно перехватывать исключения (см. разд. 2.4.8) и сообщать пользователю о возникших неприятностях, не допуская появления непонятных ему сообщений об ошибках на английском языке. Для того чтобы не отвлекать вас от главного, во многих рассмотренных ранее примерах перехват исключений не использовался. Но в реальных приложениях он необходим.

Необходимо также принять меры против потери пользователем несохраненных результатов своей работы при случайном завершении им приложения. Эти вопросы рассматривались в разд. 4.12.1 при обсуждении события формы **OnCloseQuery**.

Если вы соблюдаете изложенные выше не такие уж сложные правила, то и вы сами, и другие пользователи будут с удовольствием работать с созданными вами приложениями. Ну, а Delphi дает вам неограниченные возможность проявлять свою фантазию и проектировать прекрасные приложения с современным, красивым и удобным интерфейсом для любой сферы деятельности.

5.7 Проверьте себя

5.7.1 Вопросы для самопроверки

1. Как рекомендуется подходить к выбору цветовой гаммы ваших приложений?
2. Какие требования предъявляются к меню приложений?
3. Какие стили, и в каких случаях следует использовать для главного окна приложения?
4. Какие стили целесообразно использовать для вторичных диалоговых окон приложения?
5. Изложите основные особенности проектирования окон с изменяемыми размерами.
6. Какое свойство позволяет разрешить или запретить пользователю воспользоваться окном редактирования в выпадающем списке ComboBox?
7. Как обеспечить нажатое состояние одной из группы радиокнопок?
8. Чем отличается кнопка SpeedButton от кнопки Button? Как обеспечить в кнопке SpeedButton возможность ее залипания в нажатом состоянии?
9. Чем различаются графические форматы файлов .ico и .bmp с точки зрения отображения содержащихся в них изображений?
10. Как можно указать стиль, толщину и цвет линии при рисовании на канве?
11. Чем определяется цвет заполнения фигур, рисуемых на канве?
12. Как можно обеспечить пользователю возможность изменения типа диаграмм, отображающих какие-то данные?

5.7.2 Задачи

1. Спроектируйте такое окно с изменяемыми размерами, содержащее посередине окно редактирования RichEdit и кнопки справа и снизу от него, чтобы при изменении размеров окна пропорционально изменялся размер RichEdit.
2. Сделайте с помощью методов канвы приложение, создающее простейшие графические изображения, как их рисуют дети: домик, человечка, солнышко и т.п.
3. Измените интерфейсы приложений, созданных вами при изучении предшествующих глав этой книги. Добавьте в них новые компоненты, с которыми вы познакомились (панели, выпадающие списки, радиокнопки, графику), словом, сделайте для этих приложений полноценный интерфейс.

Часть 3

Работа с базами данных

Глава 6

Приложения для работы с локальными базами данных

В этой главе:

- *вы узнаете о различных типах баз данных*
- *научитесь создавать собственные базы данных*
- *освоите разработку прикладных программ, работающих с данными*
- *научитесь вводить в таблицы вычисляемые поля данных*
- *изучите способы упорядочивания записей, их фильтрации и поиска*
- *научитесь обеспечивать кэширование результатов редактирования данных*

6.1 Базы данных

6.1.1 Принципы построения баз данных

В наше время почти каждое приложение работает с базами данных. Например, те приложения, работающие с характеристиками сотрудников или студентов, которые вы создавали в гл. 3, 4, 5, естественнее было бы реализовать на основе базы данных. Все намного упростилось бы, и приложение получилось бы более универсальным. В дальнейшем вы создадите при изучении данной главы подобные приложения, и сами в этом убедитесь.

Базы данных (БД) — это информационный фонд отдельных пользователей, фирм и организаций, целых государств и межгосударственных учреждений. Вы можете хранить в базах данных тексты, изображения, звуковые и мультимедийные файлы, можете вести списки предстоящих дел, встреч и т.п. Фирмы хранят в базах данных всю бухгалтерскую и экономическую информацию, сведения о заказчиках, поставщиках, сделках, о наличии товаров на складах и многое другое. На уровне государств функционируют базы данных, содержащие сведения о гражданах, экономических ресурсах, автомашинах — словом, обо всем, что необходимо для функционирования государственных органов управления. Так что современное общество не может существовать без баз данных

Delphi предоставляет очень широкие возможности для работы с любыми типами баз данных. Но прежде, чем обсуждать методику разработки таких приложений, надо рассмотреть основные понятия, связанные с БД.

Мы будем иметь дело только с так называемыми реляционными базами данных. Подавляющее большинство современных баз данных относится именно к этому типу. *Реляционная база данных* — это прежде всего набор таблиц, хотя база данных может также содержать процедуры и ряд других объектов. *Таблицу* можно представлять себе как обычную двумерную таблицу с характеристиками какого-то

множества объектов. Таблица имеет *имя* — идентификатор, по которому на нее можно сослаться. В табл. 6.1 приведен пример фрагмента подобной таблицы с именем Pers, содержащей сведения о списочном составе студентов (учащихся, сотрудников). Такую таблицу вы в дальнейшем создадите сами, и будете использовать в примерах по работе с базами данных.

Таблица 6.1. Пример таблицы данных Pers

Номер	Группа (класс, отдел)	Фамилия, имя, отчество	Оценка по информатике	Дата рождения	Пол	Характеристика	Фотография
Num	Gr	Fam	Mark	Born	Sex	Charact	Photo
1	А1-01	Иванов И.И.	5	17.10.1985	м	...	...
2	А1-02	Петров П.П.	3	01.09.1986	м	...	...
3	А3-01	Сидоров С.С.	4	01.02.1984	м	...	...
4	Абитуриент	Иванова И.И.	1	01.01.1987	ж	...	...
...	...	...	...	...	...	...	...

Столбцы таблицы соответствуют тем или иным характеристикам объектов — *полям*. Каждое поле характеризуется именем и типом хранящихся данных. *Имя поля* — это идентификатор, который используется в различных программах для манипуляции данными. Он строится по тем же правилам, как любой идентификатор, т.е. пишется латинскими буквами, состоит из одного слова и т.д. Таким образом, имя — это не то, что отображается на экране или в отчете в заголовке столбца (это отображение естественно писать по-русски), а идентификатор, соответствующий этому заголовку. Например, для табл. 6.1 введем для последующих ссылок имена полей **Num**, **Gr**, **Fam**, **Mark**, **Born**, **Sex**, **Charact**, **Photo**. Смысл этих полей указан в табл. 6.1. В зависимости от того, какой смысл вы закладываете в таблицу, поле **Gr** может обозначать группу или класс, в котором обучается человек, или отдел, в котором работает сотрудник.

Тип поля характеризует тип хранящихся в поле данных. Это могут быть строки, числа, даты, время, булевы значения, большие тексты (например, характеристики сотрудников), изображения (фотографии сотрудников) и т.п.

Каждая *строка таблицы* соответствует одному из объектов. Она называется *записью* и содержит значения всех полей, характеризующие данный объект.

При построении таблиц БД важно обеспечивать непротиворечивость информации. Обычно это делается введением *ключевых полей* — обеспечивающих уникальность каждой записи. Ключевым может быть одно или несколько полей. В приведенном выше примере можно было бы сделать ключевым полем **Fam**. Но в этом случае нельзя было бы заносить в таблицу сведения о полных однофамильцах, у которых совпадают фамилия, имя и отчество. Поэтому в таблицу введено первое поле **Num** — номер, которое можно сделать ключевым, обеспечивающим уникальность каждой записи.

При работе с таблицей пользователь или программа как бы скользит курсором по записям. В каждый момент времени есть некоторая *текущая запись*, с которой и ведется работа. Записи в таблице БД физически могут располагаться без какого-либо порядка, просто в последовательности их ввода. Но когда данные таблицы предъявляются пользователю, они должны быть упорядочены. Пользователь мо-

жет хотеть просматривать их в алфавитном порядке, или рассортированными по группам (отделам), или по мере нарастания года рождения, по оценкам и т.п. Для упорядочивания данных используется понятие *индекса*. Индекс показывает, в какой последовательности желательно просматривать таблицу. Он является как бы посредником между пользователем и таблицей (см. рис. 6.1).

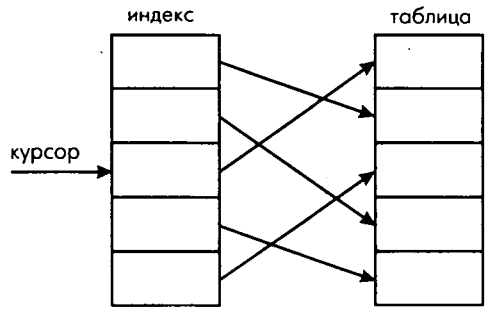


Рис. 6.1
Схема перемещения курсора по индексу

Курсор скользит по индексу, а индекс указывает на ту или иную запись таблицы. Для пользователя таблица выглядит упорядоченной, причем он может сменить индекс, и последовательность просматриваемых записей изменится. Но в действительности это не связано с какой-то перестройкой самой таблицы и с физическим перемещением в ней записей. Меняется только индекс, т.е. последовательность ссылок на записи.

Индексы могут быть *первичными* и *вторичными*. Первичным индексом служат поля, отмеченные при создании БД как ключевые. А вторичные индексы могут создаваться из других полей как в процессе создания самой таблицы данных, так и позднее в процессе работы с ней. Вторичным индексам присваиваются имена — идентификаторы, по которым их можно использовать.

Если индекс включает в себя несколько полей, то упорядочивание таблицы сначала осуществляется по первому полю, а для записей, имеющих одинаковые значения первого поля — по второму и т.д. Например, приведенную выше таблицу данных можно индексировать по полям **Gr**, **Fam**. Тогда записи будут показаны, прежде всего, упорядоченными по полю **Gr** — группе (отделу). А внутри каждой группы они будут упорядочены по алфавиту.

База данных обычно содержит не одну, а множество таблиц. Например, в дополнение к рассмотренной таблице в БД может быть таблица со списком групп (отделов). Пример такой таблицы с именем Gro, которая будет использоваться в дальнейшем, приведен в табл. 6.2. Имена полей этой таблицы, которые в дальнейшем мы будем использовать: **Gr1 и Teach**.

Таблица 6.2. Пример таблицы данных о группах Gro

Группа (класс, отдел)	Преподаватель информатики (руководитель)
Gr1	**Teach**
A1-01	Иванов И.И.
A1-02	Иванов И.И.

Группа (класс, отдел)	Преподаватель информатики (руководитель)
А3-01	Сидоров С.С.
Абитуриент	

Отдельные таблицы, конечно, полезны, но гораздо больше информации можно извлечь именно из совокупности таблиц. Например, пользователю может потребоваться узнать фамилию преподавателя, который преподает какому-то студенту информатику. Для получения ответов на подобные запросы необходимо рассмотрение совокупности связных таблиц. В нашем случае в таблице Pers можно узнать номер группы, а в таблице Gro — преподавателя этой группы.

В связных таблицах обычно одна выступает как *главная*, а другая или несколько других — как вспомогательные, управляемые главной. В этом случае взаимодействие таблиц иллюстрируется рисунком 6.2. Главная и вспомогательная таблицы связываются друг с другом *ключом*. В качестве ключа могут выступать какие-то поля, присутствующие в обеих таблицах. Например, в приведенных ранее таблицах головной может быть таблица Gro, вспомогательной Pers, а связываться они могут по полю **Gr** таблицы Pers и полю **Gr1** таблицы Gro. Курсор скользит по индексу главной таблицы. Каждой записи в главной таблице ключ ставит в соответствие в общем случае множество записей вспомогательной таблицы. Так в нашем примере каждой записи главной таблицы Gro соответствуют те записи вспомогательной таблицы Pers, в которых значение ключевого поля **Gr** (группы) совпадает со значением поля **Gr1** в текущей записи главной таблицы. Иначе говоря, если в текущей записи главной таблицы в поле **Gr1** написано "А1-01", то во вспомогательной таблице Pers выделяются все записи студентов данной группы.

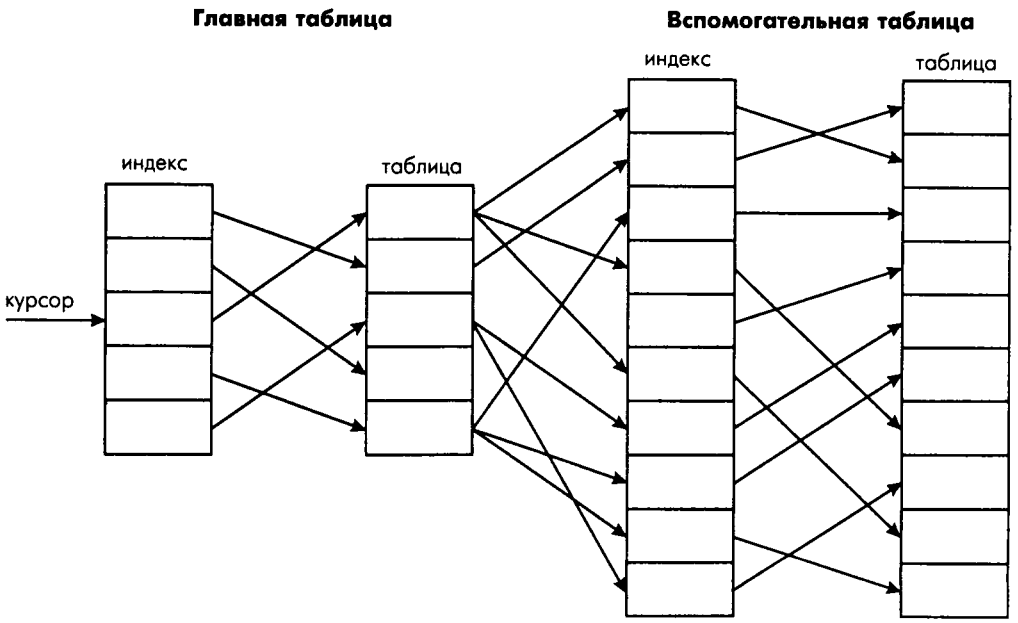


Рис. 9.2
Схема взаимодействия главной и вспомогательной таблицы

Создают БД и обрабатывают запросы к ним *системы управления базами данных — СУБД*. Известно множество СУБД, различающихся своими возможностями или обладающих примерно равными возможностями и конкурирующих друг с другом: Paradox, dBase, Microsoft Access, FoxPro, Oracle, InterBase, Sybase и много других.

Разные СУБД по-разному организуют и хранят базы данных. Например, Paradox и dBase используют для каждой таблицы отдельный файл. В этом случае база данных — это каталог, в котором хранятся файлы таблиц. В Microsoft Access и в InterBase несколько таблиц хранится как один файл. В этом случае база данных — это имя файла с путем доступа к нему. Системы типа клиент/сервер, такие, как серверы Sybase или Microsoft SQL, хранят все данные на отдельном компьютере и общаются с клиентом посредством специального языка, называемого SQL.

Поскольку конкретные свойства баз данных очень разнообразны, пользователю было бы весьма затруднительно работать, если бы он должен был указывать в своем приложении все эти каталоги, файлы, серверы и т.п. Да и приложение часто пришлось бы переделывать при смене, например, структуры каталогов и при переходе с одного компьютера на другой. Чтобы решить эту проблему, используют псевдонимы БД. *Псевдоним* (*alias*) содержит всю информацию, необходимую для обеспечения доступа к БД. Эта информация сообщается только один раз при создании псевдонима. А приложение для связи с БД использует псевдоним. В этом случае приложению безразлично, где физически расположена та или иная БД, а часто безразлична и СУБД, создавшая и обслуживающая эту БД. При смене системы каталогов, сервера и т.п. ничего в приложении переделывать не надо. Достаточно, чтобы администратор базы данных ввел соответствующую информацию в псевдоним.

При работе с БД часто используется *кэширование* всех изменений. Это означает, что все изменения данных, вставка новых записей, удаление существующих записей, т.е. все манипуляции с данными, проводимые пользователем, сначала делаются не в самой БД, а запоминаются в памяти во временной, виртуальной таблице. И только по особой команде после всех проверок правильности вносимых в таблицу данных пользователю предоставляется возможность или зафиксировать все эти изменения в БД, или отказаться от этого и вернуться к тому состоянию, которое было до начала редактирования.

Фиксация изменений в БД осуществляется с помощью *транзакций*. Это совокупность команд, изменяющих БД. На протяжении транзакции пользователь может что-то изменять в данных, но это только видимость. В действительности все изменения сохраняются в памяти. И пользователю предоставляется возможность завершить транзакцию или внесением всех изменения в реальную базу данных, или отказом от этого с возвратом к тому состоянию, которое было до начала транзакции.

6.1.2 Типы баз данных

Для разных задач целесообразно использовать различные модели баз данных, поскольку, конечно, базу данных сведений о сотрудниках какого-то небольшого коллектива и базу данных о каком-нибудь банке, имеющем филиалы во всех концах страны, надо строить по-разному.

Поэтому прежде чем двигаться дальше, необходимо иметь представление о возможных моделях баз данных, поскольку это влияет на построение приложений в Delphi.

В следующих разделах коротко рассмотрены три модели баз данных:

- Автономные.
- Клиент/сервер.
- Многоуровневые распределенные.

6.1.2.1 Автономные базы данных

Автономные БД являются наиболее простыми. Они хранят свои данные в локальной файловой системе на том компьютере, на котором установлены. СУБД, осуществляющая к ним доступ, находится на том же самом компьютере. Сеть не используется. Поэтому разработчику автономной БД не приходится иметь дело с проблемой параллельного доступа, когда два человека пытаются одновременно изменить одну и ту же запись. Такого просто не может быть.

Автономные БД полезны для использования в тех приложениях, которые распространены среди многих пользователей, каждый из которых поддерживает отдельную БД. Это, например, приложения, обрабатывающие документацию небольшого офиса, кадровый состав небольшого предприятия, бухгалтерские документы небольшой бухгалтерии. Каждый пользователь такого приложения манипулирует своими собственными данными на своем компьютере. Пользователю нет необходимости иметь доступ к данным любого другого пользователя, так что отдельная база данных здесь вполне приемлема.

Автономные БД не используются для приложений, работающих с очень большими базами данных или требующих значительной вычислительной мощности для обработки данных, поскольку вся нагрузка ложится на компьютеры пользователей.

6.1.2.2 Базы данных клиент/сервер

Для больших баз данных с множеством пользователей часто используются базы данных на платформе клиент/сервер. В этом случае БД хранится на сервере. Это может быть мощный специализированный компьютер, ориентированный на операции с запросами к БД. Данные, хранящиеся на сервере, доступны одновременно многим клиентам через сеть. Это очень удобно, так как изменения в таких БД видят все пользователи. Например, БД сотрудников крупного учреждения целесообразно делать именно такой, чтобы администраторы отдельных подразделений обращались к ней, а не заводили у себя локальные БД (при этом можно сделать так, чтобы каждый администратор видел только ту информацию, которая относится к его подразделению).

Клиентские приложения дают задание серверу выполнить те или иные операции поиска или обновления БД. При этом возникают (и решаются) проблемы, связанные с возможным одновременным доступом нескольких пользователей к одной и той же информации. Например, при проектировании приложений, работающих с подобными БД, должны быть решены проблемы такого рода: что делать, если пользователь прочитал некоторую запись и, пока он ее просматривает и собирается изменить, другой пользователь меняет или удаляет эту запись.

При использовании платформы клиент/сервер следует принять меры к уменьшению загрузки сети. Например, если запрос пользователя относится всего к одной записи, то было бы недопустимым расточительством передавать через сеть на компьютер пользователя все данные, хранящиеся в таблице. Надо передавать только те данные, которые запросил пользователь. И всю основную обработку данных — так называемую *бизнес-логику* желательно проводить на сервере, минимизи-

зируя информацию, передаваемую по сети, и разгружая компьютеры клиентов от излишних вычислений. Все это осуществляется разработкой так называемых *хранимых на сервере процедур*. Клиент может разрабатывать подобные процедуры и заносить их в серверную БД.

Платформа клиент/сервер удовлетворительно обслуживает системы уровня небольшого предприятия или отдельного подразделения крупного предприятия. Но при создании более крупных систем организация клиент/сервер сталкивается с рядом сложностей. Размещение бизнес-логики как на сервере, так и на компьютерах клиентов мешает созданию единой системы с едиными правилами обработки и представления данных. Много сил уходит на согласование работы различных клиентских приложений, на построение мостов между ними, на устранение дублирования одних и тех же функций различными приложениями.

6.1.2.3 Многоуровневые распределенные базы данных

Основным направлением в настоящее время считается создание многоуровневых распределенных систем обработки данных. Чаще всего используется трехуровневая архитектура. На верхнем уровне расположен удаленный сервер БД. Он обеспечивает хранение и управление данными. На среднем уровне (middlware) располагается *сервер приложений*. Он обеспечивает соединение клиентов с сервером БД и реализует бизнес-логику. На нижнем уровне находятся клиентские приложения. В ряде случаев это могут быть *тонкие клиенты*, обеспечивающие только пользовательский интерфейс, поскольку вся бизнес-логика может располагаться на сервере приложений.

В ряде случаев третий уровень объединяется со вторым: БД располагается на том же компьютере, на котором находится сервер приложений.

Многозвенная система функционирует следующим образом. Пользователь запускает клиентское приложение. Оно соединяется с доступным ему сервером приложений. Затем клиент запрашивает какие-то данные. Этот запрос упаковывается в пакет установленного формата и передается серверу приложений. Там пакет расшифровывается и передается серверу БД, который возвращает затребованные данные. Сервер приложений обрабатывает эти данные согласно заложенной в него бизнес-логике, упаковывает и передает этот пакет клиенту. Клиент распаковывает данные и показывает их пользователю.

Если пользователь изменил данные, то цепочка их передачи на сервер БД выглядит следующим образом. Клиент упаковывает измененные данные и отправляет пакет на сервер приложений. Тот распаковывает их и отправляет на сервер БД. Если все исправления могут быть без осложнений занесены в БД, то на этом все завершается. Но могут возникнуть осложнения: сделанные изменения могут противоречить бизнес-правилам, или изменения одних и тех же данных разными пользователями могут противоречить друг другу. Тогда те записи, которые не могут быть занесены в БД, возвращаются клиенту. Пользователь может исправить их или отказаться от сделанных изменений.

Для обмена информацией сервера приложений с клиентами и серверами БД разработаны различные протоколы и средства. Все активнее применяются для этих целей протоколы, используемые в Интернет. Сама сеть Интернет предоставляет богатейшие возможности по созданию распределенных приложений, с которыми может одновременно работать множество пользователей. Практически все значимые технологии создания и функционирования распределенных приложений доступны в Delphi.

6.1.3 Организация связи с базами данных в Delphi

В первых версиях Delphi основой работы с базами данных являлся Borland Database Engine (BDE) — процессор баз данных фирмы Borland. Не потерял он своего значения и сейчас. Но, начиная с Delphi 5, в библиотеке компонентов стали появляться альтернативные механизмы связи с данными. Особенно много их появилось, начиная с Delphi 6. Тем не менее, начнем рассмотрение организации связи приложений Delphi с базами данных с рассмотрения BDE.

BDE служит посредником между приложением и базами данных. Он предоставляет пользователю единый интерфейс для работы, развязывающий пользователя от конкретной реализации базы данных. Благодаря этому не надо менять приложение при смене реализации базы данных. Приложение Delphi обращается к базе данных через BDE. В этом случае общение с базами данных соответствует схеме, приведенной на рис. 6.3.

Приложение Delphi, когда ему нужно связаться с базой данных, обращается к BDE и сообщает обычно псевдоним базы данных и необходимую таблицу в ней. BDE находит по псевдониму драйвер, подходящий для указанной базы данных. *Драйвер* — это вспомогательная программа, которая понимает, как общаться с базами данных определенного типа. Если в BDE имеется собственный драйвер соответствующей СУБД, то BDE связывается через него с базой данных и с нужной таблицей в ней, обрабатывает запрос пользователя и возвращает в приложение результаты обработки. BDE поддерживает естественный доступ к таким базам данных, как Microsoft Access, FoxPro, Paradox, dBase и ряд других.

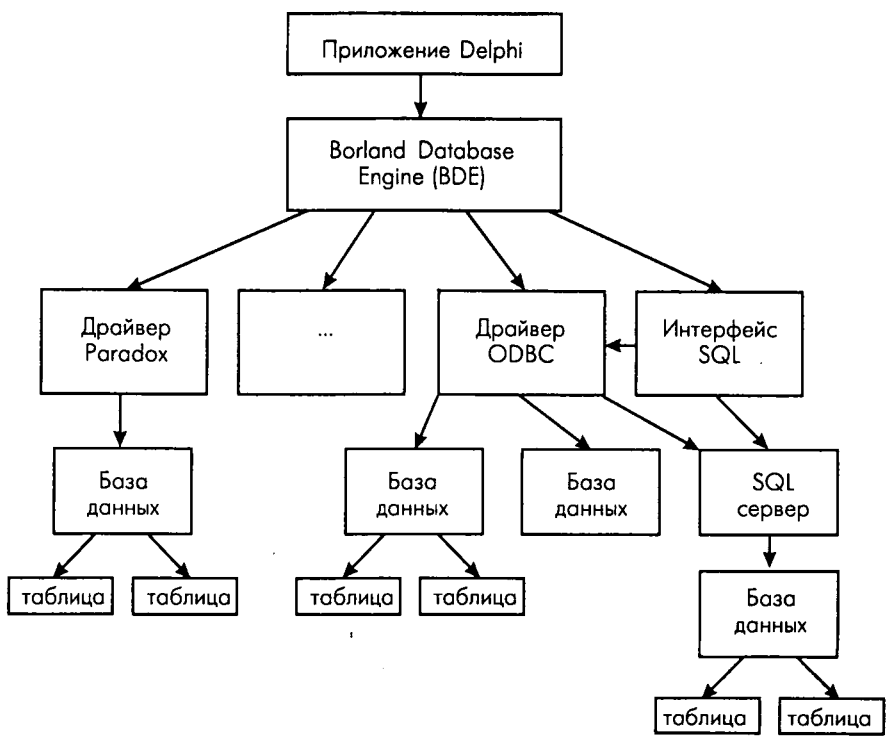


Рис. 6.3
Схема связи приложения Delphi с базами данных

Если собственного драйвера нужной СУБД в BDE нет, то используется драйвер ODBC. ODBC (Open Database Connectivity) — это система, аналогичная по функциям BDE, но разработанная фирмой Microsoft. Поскольку Microsoft включила поддержку ODBC в свои офисные продукты и для ODBC созданы драйверы практически к любым СУБД, фирма Borland включила в BDE драйвер, позволяющий использовать ODBC. Правда, работа через ODBC осуществляется несколько медленнее, чем через собственные драйверы СУБД, включенные в BDE. Но благодаря связи с ODBC масштабируемость Delphi существенно увеличилась, и сейчас из Delphi можно работать с любой сколько-нибудь значительной СУБД.

BDE поддерживает SQL — стандартизованный язык запросов, позволяющий обмениваться данными с SQL-серверами, такими, как Sybase, Microsoft SQL, Oracle, Interbase. Эта возможность используется особенно широко при работе на платформе клиент/сервер и в распределенных приложениях.

Начиная с Delphi 5, была введена альтернативная возможность работы с базами данных, минуя BDE. Это разработанная в Microsoft технология ActiveX Data Objects (ADO). ADO — это пользовательский интерфейс к любым типам данных, включая реляционные и не реляционные базы данных, электронную почту, системные, текстовые и графические файлы. Связь с данными осуществляется посредством так называемой технологии OLE DB.

Использование ADO обеспечивает более эффективную работу с данными. К тому же, снимаются некоторые проблемы с кириллицей, которые в последнее время стали проявляться при работе с BDE. Но главное преимущество ADO — включение этой технологии во все поставки современных версий Windows. Это исключает необходимость ставить BDE на компьютерах клиентов.

Еще один альтернативный доступ к базам данных СУБД Interbase (разработка Borland) был введен в Delphi 5 на основе технологии InterBase Express (IBX). Начиная с Delphi 6, альтернативные возможности доступа к базам данных расширены за счет технологии dbExpress. Это набор драйверов, обеспечивающих доступ к серверам SQL на основе единого интерфейса.

Надо сказать, что разработчики из Borland уже отказались от дальнейшего развития BDE и рекомендуют в распределенных приложениях использовать не BDE, а dbExpress. Тем не менее, в данной главе мы сосредоточимся, прежде всего, на работе с BDE и на основном компоненте набора данных BDE Table. Этот компонент позволяет наиболее просто рассмотреть основы работы с данными. Причем все, что будет рассмотрено, применимо и к другим технологиям.

6.2 Создание базы данных с помощью Database Desktop

Прежде, чем начать строить приложения, работающие с базами данных, надо иметь сами базы данных. Для их создания используются соответствующие СУБД. Но вместе с BDE и Delphi поставляется программа Database Desktop (файл *DBD32.EXE* для 32-разрядных Delphi), которая позволяет создавать таблицы баз данных некоторых СУБД, задавать и изменять их структуру. С помощью этой программы вы можете создать базу данных, включающую описанные в разд. 6.1.1 таблицы Pers и Gro, с которыми мы далее будем работать.

Обычно вызов Database Desktop включен в главное меню Delphi в раздел Tools. При вызове Database Desktop вы увидите окно вида, показанного на рис. 6.4 (если в предыдущем сеансе работы с Database Desktop не была открыта какая-то таблица, то таблицы в середине окна не будет видно, и разделов меню будет поменьше).

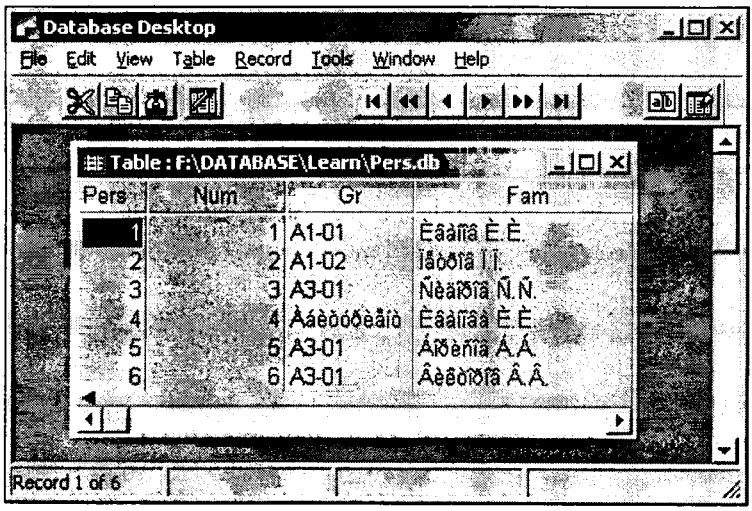


Рис. 6.4
Главное окно программы Database Desktop

Давайте создадим с помощью Database Desktop таблицы данных Pers и Gro, описанные в разд. 6.1.1 (табл. 6.1 и 6.2). Выполните в окне Database Desktop команду File | New. В открывшемся меню выберите раздел Table — создание новой таблицы. Откроется небольшое диалоговое окно. В нем из выпадающего списка вы можете выбрать СУБД, для которой хотите создать таблицу. Можете посмотреть в нем, таблицы каких СУБД могут создаваться с помощью Database Desktop. Выберите Paradox 7. Вы увидите окно, представленное на рис. 6.5. В этом окне вы можете задать структуру таблицы (поля и их типы), создать вторичные индексы, ввести диапазоны допустимых значений полей, значения по умолчанию и ввести много иной полезной информации о создаваемой таблице.

Для каждого поля создаваемой таблицы, прежде всего, указывает имя (Field Name) — идентификатор поля. Он может включать до 25 символов. Затем надо выбрать тип (Type) данных этого поля. Для этого перейдите в раздел Type поля и щелкните правой кнопкой мыши. Появится список доступных типов, из которого вы можете выбрать необходимый вам. В нашем примере таблицы Pers (см. табл. 6.1) для поля **Num** целесообразно выбрать тип **Autoincrement**. Это тип целого числа, автоматически увеличивающегося на 1 для каждой новой записи. Следовательно, поле этого типа обеспечит уникальность каждой записи. Поэтому поле **Num** следует сделать ключевым. Ключевое поле должно быть отмечено символом "*" в последней колонке Key. Для того чтобы поставить или удалить этот символ, надо или сделать двойной щелчок в соответствующей графе информации о поле, или выделить эту графу и нажать клавишу пробела.

Когда заполнена информация об одном поле, можно начинать вводить описание следующего поля. Для этого достаточно нажать клавишу со стрелкой вниз или клавишу табуляции. В полях **Gr**, **Fam** и **Sex** надо задать тип **Alpha** (строка). Для полей этого типа надо в столбцѐ Size указать размер поля — число символов. Для поля **Mark** выберите тип **Short** (короткое целое). Для поля **Born** задайте тип **Date** (дата), для поля **Charact** — **Formatted Memo** (форматированный текст), для поля **Photo** — **Graphic** (графика). Значения полей типов **Formatted Mèmo** и **Graphic** будут храниться в отдельных файлах и могут иметь неограниченный размер. Но для этих полей так же, как для строк, нужно задавать размер в столбце Size. Это число байтов, хранящихся непосредственно в таблице. Практически все равно, какое число задать (например, 10). Но что-то задать надо, иначе Database Desktop сообщит об ошибке.

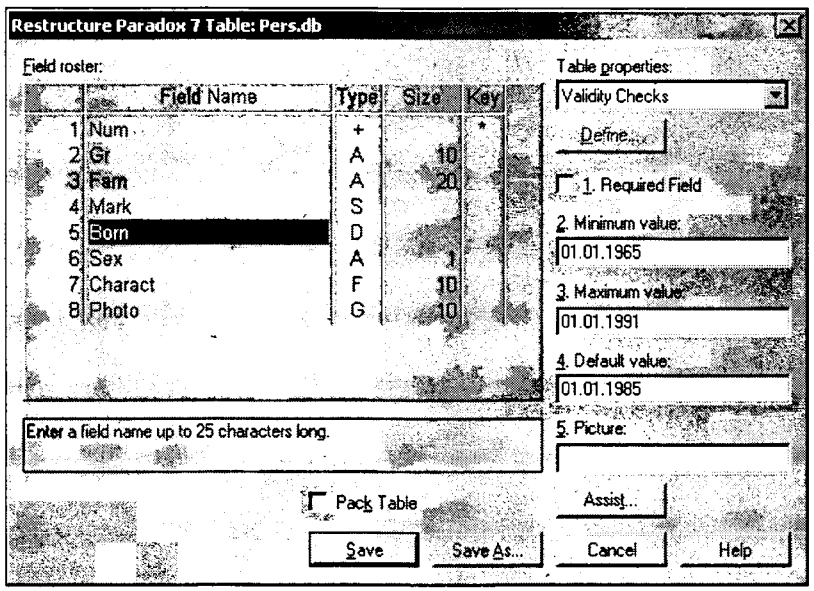


Рис. 6.5
Окно создания структуры таблицы Paradox

Выделив в окне рис. 6.5 то или иное поле, вы можете видеть в правой части окна элементы, позволяющие накладывать на значения поля некоторые ограничения. Индикатор Required Field надо включать для тех полей, значения которых обязательно должны содержаться в каждой записи. Для нашего примера таким полем, очевидно, должно быть поле **Fam**. Окно Default позволяет задать значение поля по умолчанию. Это свойство полезно задавать для полей чисел, дат и некоторых символьных. В нашем примере полезно задать значения по умолчанию для поля **Born** и поля **Sex** (например, "м" — иначе у пользователя могут возникнуть проблемы при вводе информации). Окна Minimum и Maximum позволяют задать минимальное и максимальное значения. В нашем примере полезно указать эти значение для поля **Born** (см. рис. 6.5).

В последующих примерах нам потребуется индексировать таблицу различным образом. Так что вам надо создать в таблице соответствующие вторичные индексы. В дальнейшем вам будут полезны следующие индексы:

Имя индекса	Поля	Пояснение
fio	Fam	Упорядочивание таблицы по алфавиту.
grfio	Gr, Fam	Упорядочивание таблицы по группам, а внутри каждой группы упорядочивание по алфавиту.
B	Born	Упорядочивание таблицы по году рождения.
MarkGr	Mark, Gr	Упорядочивание по оценкам и группам.

Чтобы создать новый вторичный индекс, выберите в выпадающем списке в верхнем правом углу окна рис. 6.5 раздел Secondary Indexes. Нажмите кнопку Define, которая станет доступной. Откроется диалоговое окно, представленное на рис. 6.6. В его левой панели Fields содержится список доступных полей, в правой панели Indexed fields вы можете подобрать и упорядочить список полей, включаемых в индекс. Для переноса поля из левого окна в правое надо выделить интересующее вас поле или группу полей и нажать кнопку со стрелкой вправо. Стрелками Change order (изменить последовательность) можно изменить порядок следования полей в индексе.

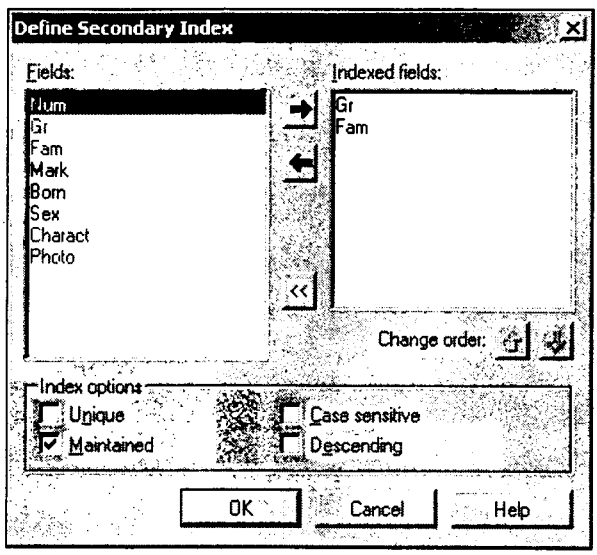


Рис. 6.6
Окно задания вторичного индекса

Панель радиокнопок Index Options (опции индекса) позволяют установить следующие характеристики:

Unique	Установка этой опции не позволяет индексировать таблицу, если в ней находятся дубликаты совокупности включенных в индекс полей. Например, установка этой опции для индекса **fio** не допустила бы наличия в таблице однофамильцев.
Descending	При установке этой опции таблица будет упорядочиваться по степени убывания значений (по умолчанию упорядочивание производится по степени нарастания значений). Например, этот индикатор можно включить для индекса **MarkGr**, чтобы в таблице сначала были видны отличники.

Case Sensitive	При установке этой опции будет приниматься во внимание регистр, в котором введены символы.
Maintained	Если эта опция установлена, то индекс обновляется при каждом изменении в таблице. В противном случае индекс обновляется только в момент связывания с таблицей или передачи в нее запроса. Это несколько замедляет обработку запросов. Поэтому полезно включать эту опцию для обновляемых таблиц. Если таблица используется только для чтения, эту опцию лучше не включать.

После того как индекс сформирован, щелкаете на кнопке OK. Открывается окно, в котором вы задаете имя индекса.

После того как все необходимые данные о структуре таблицы внесены, щелкните на кнопке Save as (сохранить как) и перед вами откроется окно, напоминающее обычный диалог сохранения в файле. От обычного это окно отличается выпадающим списком Alias. Этот список содержит псевдонимы различных баз данных (о них пойдет речь позднее), из которого вы можете выбрать базу данных, в которую будете сохранять свою таблицу. Если вам не надо сохранять таблицу в одной из существующих баз данных, то вы можете воспользоваться обычным списком Сохранить в в верхней части окна. При этом вы с помощью обычной быстрой кнопки можете создать новую папку (каталог). Мы создаем базу данных типа Paradox. Для Paradox база данных — это каталог, в котором сохраняется таблица. Так что создайте с помощью быстрой кнопки, имеющейся в окне диалога, каталог, который будет вашей базой данных, и сохраните в нем сформированную таблицу, задав ей имя Pers. Если вы работаете в сети, то можете сохранить таблицу не на своем, а на другом, серверном компьютере.

Выше было рассмотрено задание структуры таблицы Pers, описанной в разд. 6.1.1 в табл. 6.1. Вам также надо создать таблицу Gro, описанную в тал. 6.2. Она проще: в ней всего два поля **Gr1** и **Teach**. Типы обоих полей — строки. Поле **Gr1** является ключевым. Вторичные индексы создавать не надо. Сохраните эту таблицу в том же каталоге, в котором вы сохранили предыдущую таблицу.

После того как вы создали таблицы, вы можете их открыть командой File | Open. После выполнения этой команды и выбора файла таблицы вы увидите окно вида, представленного ранее на рис. 6.4. Если в таблице уже есть записи, то вы увидите их содержание. Не пугайтесь того, что при этом символы кириллицы будут представлены какой-то абракадаброй, которую вы видите на рис. 6.4. Программа Database Desktop автоматически не настраивается на русский язык. Но при использовании таблицы в приложении все надписи будут выглядеть нормально.

Команда Table | Restructure позволяет изменить структуру таблицы или какие-то ее характеристики. При выполнении этой команды вы попадаете в окно, аналогичное рассмотренному ранее при разработке структуры, и сможете внести в структуру таблицы необходимые изменения, задать дополнительные индексы и т.п.

6.3 Создание и редактирование псевдонимов баз данных

Вы создали базу данных, содержащую две таблицы. Теперь вам надо создать для нее псевдоним (см. разд. 6.1), по которому приложения будут ссылаться на эту БД. Псевдонимы можно просматривать и создавать в Database Desktop, выполнив команду Tools | Alias Manager. Вы увидите диалоговое окно Alias Manager (диспетчера псевдонимов), вид которого представлен на рис. 6.7. Впрочем, вид этого окна существенно зависит от того, псевдоним какой базы данных просматривается. Индикатор Public alias (открытый псевдоним) в верхней части окна показывает, будет ли создаваться открытый, доступный всем псевдоним, или псевдоним проекта. Ниже расположен выпадающий список Database Alias, в котором вы можете выбрать интересующий вас псевдоним из числа уже созданных.

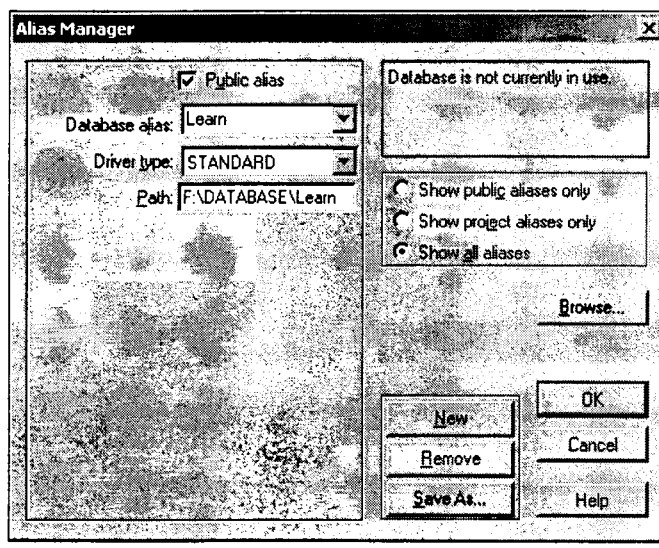


Рис. 6.7
Просмотр псевдонимов баз данных

При выборе псевдонима в списке Database Alias автоматически изменяется тип драйвера в выпадающем списке Driver type и расположенная ниже информация о драйвере. Для драйвера типа STANDARD, который используется для баз данных PARADOX и многих других, эта информация сводится к указанию пути к базе данных в окне Path. Если псевдоним базы данных уже существует, но путь к ней изменился, достаточно изменить информацию в этом окне, щелкнуть на кнопке Save As и согласиться с предложением заменить файл *IDAPI32.CFG*, в котором сохраняется конфигурация BDE. Все приложения, использующие данный псевдоним, даже не почувствуют, что местоположение базы данных изменилось.

Но в нашем случае псевдонима еще нет, и его надо создать. Щелкните для этого на кнопке New (новый). В результате окно несколько преобразуется: вместо кнопки New появится кнопка Keep New — сохранить новый псевдоним, и появится возможность в списке Database Alias не только выбирать существующий псевдоним, но и писать новый.

В нашем примере вам надо открывшемся окне:

- Выбрать драйвер STANDARD в списке Driver type.
- Указать каталог хранения таблиц. В этом может помочь кнопка просмотра Browse.
- В окне Database Alias написать новый псевдоним. Давайте зададим псевдоним **Learn**.
- Щелкнуть на кнопке Keep New, чтобы сохранить введенную информацию, сохранить всю информацию кнопкой Save as и выйти из диалога.

6.4 Обзор компонентов, используемых в BDE для связи с базами данных

Размещение компонентов для работы с базами данных на страницах библиотеки, начиная с Delphi 6, существенно отличается от предшествующих версий. В версиях, младше Delphi 6, компоненты, обеспечивающие доступ к данным через BDE, расположены на странице Data Access. Начиная с Delphi 6, на этой странице остался только компонент **DataSource**, а остальные перенесены на страницу BDE. Компоненты отображения и редактирования данных во всех версиях размещены на странице Data Control.

Каждое приложение, использующее базы данных, обычно имеет, по крайней мере, по одному компоненту следующих трех типов:

- Компоненты — наборы данных (data set), непосредственно связывающиеся с базой данных. Для BDE это такие компоненты, как **Table**, **Query** и ряд других.
- Компонент — источник данных (data source), осуществляющий обмен информацией между компонентами первого типа и компонентами визуализации и управления данными. Таким компонентом является **DataSource**.
- Компоненты визуализации и управления данными, такие, как **DBGrid**, **DBText**, **DBEdit** и множество других.

Связь этих компонентов друг с другом и с базой данных можно представить схемой, приведенной на рис. 6.8.

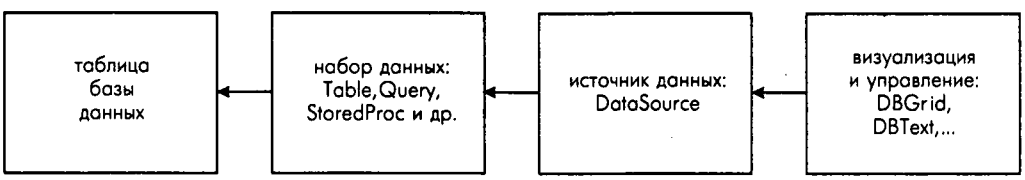


Рис. 6.8
Схема взаимодействия компонентов Delphi с базой данных

Источник данных **DataSource** ссылается на набор данных своим свойством **DataSet**. Компоненты визуализации и управления данными ссылаются на источник данных своим свойством **DataSource**. Кроме того, в тех компонентах, которые отображают данные одного конкретного поля, свойство **DataField** должно указывать поле, с которым связан компонент.

6.5 Установка связей между компонентами и базой данных, навигация по таблице

Давайте построим простейшее приложение, работающее с базой данных. Будем использовать ту таблицу Pers, которую вы создавали ранее в базе данных с псевдонимом **Learn**.

В нашем простейшем приложении мы будем в качестве набора данных использовать компонент **Table**. Откройте новое приложение и перенесите на форму компонент **Table** со страницы библиотеки BDE (или со страницы Data Access, если работаете с версией, младше Delphi 6). Перенесите также на форму со страницы Data Access компонент **DataSource**, который будет являться источником данных. Оба эти компоненты невизуальные, пользователю они будут не видны, так что их можно разместить в любом месте формы. В качестве компонента визуализации данных возьмите компонент **DBGrid** со страницы Data Control. Это визуальный компонент, в котором будут отображаться данные формы. Поэтому растяните его пошире.

Теперь нам надо установить цепочку связей между этими компонентами, показанную выше на рис. 6.8. Главное свойство **DBGrid** и других компонентов визуализации и управления данными — **DataSource**. Выделите на форме компонент **DBGrid1** и щелкните на его свойстве **DataSource** в Инспекторе Объектов. Вы увидите выпадающий список, в котором перечислены все имеющиеся на форме источники данных. В нашем случае имеется только один источник данных — **DataSource1**. Установите его в качестве значения свойства **DataSource**. Далее надо установить связь между источником данных и набором данных. Выделите компонент **DataSource1** и найдите в Инспекторе Объектов его главное свойство — **DataSet**. Щелкните на этом свойстве и из выпадающего списка выберите **Table1** (если бы у вас было несколько компонентов — наборов данных, то все они были бы в этом списке).

Теперь осталось связать компонент **Table1** с необходимой таблицей базы данных. Для этого служат два свойства компонента **Table**: **DatabaseName** и **TableName**. Прежде всего, надо установить свойство **DatabaseName**. В выпадающем списке этого свойства в Инспекторе Объектов вы можете видеть все доступные BDE псевдонимы баз данных. Выберите из этого списка псевдоним **Learn**, который вы сами ввели ранее. Если этого псевдонима там нет, значит вы забыли его создать. В этом случае создайте его с помощью Database Desktop, как описано в разд. 6.3.

После этого можно устанавливать значение свойства **TableName**. В выпадающем списке этого свойства перечислены таблицы, доступные в данной базе данных. Выберите таблицу Pers.

А теперь наступает самый ответственный момент. Вы можете прямо в процессе проектирования соединиться с базой данных. Соединение осуществляется свойством **Active**. По умолчанию оно равно **false**. Установите его в **true**. Если все сделано вами правильно, то вы увидите в поле компонента **DBGrid1** столбцы, соответствующие полям таблицы. Если бы таблица уже содержала записи, то вы увидели бы и их.

Вы можете прямо сейчас, хотя приложение еще не закончено, сохранить проект, запустить его на выполнение и убедиться, что с ним можно работать. Заполните несколько записей в таблице, чтобы можно было в дальнейшем с ней работать (см. рис. 6.9). Можете также убедиться, что в таблице Pers невозможно изменить поле **Num**, поскольку оно изменяется автоматически и доступно только для чтения.

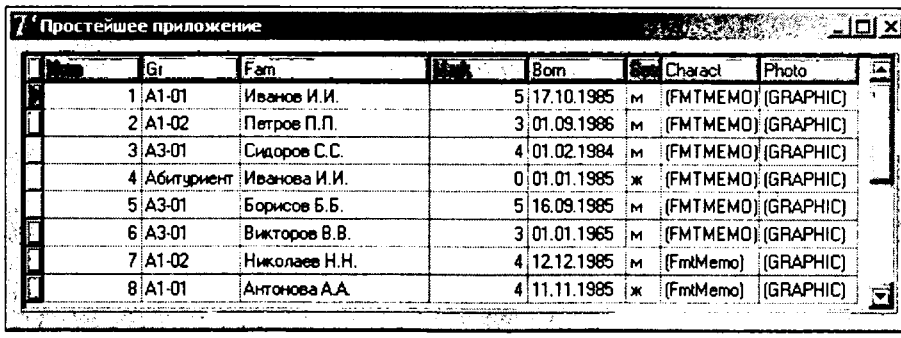


Рис. 6.9
Форма простейшего приложения, построенного на компоненте Table

Закройте приложение. Вы, может быть, с удивлением увидите, что все записи, которые вы старательно заносили в базу данных, не видны на форме. Дело в том, что ИСР Delphi хранит в памяти то состояние таблицы, которое было в момент задания значения **true** свойству **Active**. Если вы установите это свойство в **false**, а затем опять зададите ему значение **true**, то увидите прямо в процессе проектирования все занесенные вами записи.

Следует отметить, что заранее выставлять для набора данных **Active = true** допустимо только в процессе настройки и отладки приложения, работающего с автономными базами данных.

Хороший стиль программирования

В законченном приложении во всех наборах данных сначала должно быть установлено Active = false, затем при событии формы OnCreate эти свойства могут быть установлены в true, а при событии формы OnDestroy эти свойства опять должны быть установлены в false. Это исключит неоправданное поддержание связи с базой данных, которое занимает ресурсы, а при работе в сети мешает доступу к базе данных других пользователей.

Вместо установки значений свойства **Active** лучше вызывать для компонента набора данных **Table** метод **Open**, устанавливающий соединение с БД и задающий значение **Active = true**, и метод **Close**, разрывающий соединение и задающий значение **Active = false**.

Так что давайте грамотно оформим наше первое, пока еще очень несовершенное приложение, работающее с базой данных. Выделите форму, перейдите в Инспекторе Объектов на страницу событий и в обработчик события формы **OnCreate** (см. разд. 4.13.1) вставьте оператор:

```
Table1.Open;
```

Аналогичным образом в обработчик события формы **OnDestroy** или **OnClose** (см. разд. 4.13.1) вставьте оператор:

```
Table1.Close;
```

Теперь в момент создания формы ваше приложение будет соединяться с базой данных, а в момент разрушения формы соединение будет разрываться. В компоненте **Table1** следует установить свойство **Active** в **false**, чтобы не занимать базу

данных в процессе проектирования. Правда, если вы работаете не в сети или если вы единственный пользователь этой базы данных, то это не имеет особого значения. Так что оставьте пока значение **Active = true**, чтобы проще было продолжать проектирование нашего приложения.

Для облегчения работы пользователей полезно добавить в приложение еще один компонент, управляющий работой с таблицей — навигатор **DBNavigator**, расположенный на странице Data Control библиотеки компонентов (см. рис. 6.10 и рассмотренный в следующем разделе рис. 6.12).

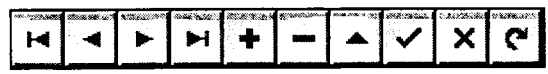


Рис. 6.10
Навигатор DBNavigator

Компонент имеет ряд кнопок, служащих для управления данными. Перечислим их названия и назначение, начиная с левой кнопки:

nbFirst	перемещение к первой записи
nbPrior	перемещение к предыдущей записи
nbNext	перемещение к следующей записи
nbLast	перемещение к последней записи
nbInsert	вставить новую запись перед текущей
nbDelete	удалить текущую запись
nbEdit	редактировать текущую запись
nbPost	послать отредактированную информацию в базу данных
nbCancel	отменить результаты редактирования или добавления новой записи
nbRefresh	очистить буфер, связанный с набором данных

Пользуясь свойством навигатора **VisibleButtons**, можно убрать любые ненужные в данном приложении кнопки. Например, если вы не хотите разрешить пользователю вводить в базу данных новые записи, то можете установить в **false** кнопку **nbInsert**. Если вы хотите вообще запретить редактирование, то можно оставить только кнопки **nbFirst**, **nbPrior**, **nbNext** и **nbLast**, а все остальные убрать (см. приведенный в одном из следующих разделов рис. 6.14).

Отметим еще одно свойство навигатора — **Hints**. Это список строк типа **TStrings**, содержащий тексты всплывающих ярлычков кнопок навигатора. По умолчанию эти тексты, конечно, английские. Щелкнув на кнопке с многоточием около свойства **Hints** в окне Инспектора Объектов, вы можете перевести эти тексты на русский язык. Поскольку навигатором вы, наверняка, будете пользоваться во многих своих приложениях, имеет смысл один раз осуществить подобный перевод и далее сохранить навигатор в виде шаблона (см. разд. 4.12), который можно будет использовать в последующих приложениях.

Чтобы приложение с навигатором работало, надо связать навигатор его свойством **DataSource** с источником данных **DataSource1**. Откомпилируйте приложение, выполните его и посмотрите в работе. Пользуясь последовательно кнопками **nbEdit** и **nbPost**, вы можете редактировать данные. Если вы отредактировали те-

кущую запись, а затем передумали заносить изменения в базу данных, то можете нажать кнопку **nbCancel**, и результаты редактирования будут отменены. Кнопка **nbInsert** позволит вам ввести в базу данных новую запись. Эта запись зафиксируется в таблице после нажатия кнопки **nbPost**.

6.6 Свойства полей

Наше приложение выглядит, конечно, очень плохо. Во-первых, последовательность записей определяется ключевым полем **Num**, а хотелось бы, чтобы записи были расположены по алфавиту или по группам и алфавиту. Первое поле с номерами записей вообще пользователю не нужно, и надо бы, чтобы его не было видно. Шапка таблицы содержит непонятные пользователю имена полей **Num**, **Fam** и т.д., а надо, чтобы были написаны нормальные заголовки по-русски.

Все это можно легко поправить. Начнем с упорядочивания записей. Выделите на форме компонент **Table1**. В Инспекторе Объектов вы увидите среди прочих свойства **IndexName** и **IndexFieldName**. Первое из них содержит выпадающий список индексов, созданных для вашей таблицы. Выберите, например, индекс **fio**, и увидите, что записи окажутся упорядоченными по алфавиту, поскольку этот индекс содержит единственное поле **Fam**. При индексе **grfio** упорядочивание будет по группам, а внутри каждой группы — по алфавиту. Альтернативный вариант индексации предоставляет свойство **IndexFieldName**. В его выпадающем списке просто перечислены предусмотренные в индексах комбинации полей, и вы можете выбрать необходимую, если забыли, что обозначают имена индексов.

Теперь займемся отдельными полями. Для их редактирования служит Редактор Полей. Вызвать его проще всего двойным щелчком на компоненте **Table1**. Сначала вы увидите пустое поле этого редактора (рис. 6.11 а). Щелкните на нем правой кнопкой мыши и из всплывающего меню выберите раздел Add fields (добавить поля). Вы увидите окно, изображенное на рис. 6.11 б, в котором содержится список всех полей таблицы. Выберите из него курсором мыши интересующие вас поля. Если вы при этом будете держать нажатой клавишу Ctrl, то может выделить любую комбинацию полей. Однако имейте в виду, что только к тем полям, которые вы добавите, вы сможете в дальнейшем обращаться. Так что в данном случае вам имеет смысл выделить все поля. Выделив поля, щелкните на OK, и вы вернетесь в основное окно Редактора Полей, но в нем уже будет содержаться список добавленных полей (рис. 6.11 в).

Эти поля будут соответствовать колонкам таблицы. Изменить последовательность их расположения можно, перетащив мышью идентификатор какого-то поля на нужное место. Как мы увидим далее, те поля, которые не должны отображаться в таблице, могут быть сделаны невидимыми.

Выделите в списке какое-то поле и посмотрите его свойства в Инспекторе Объектов. Вы увидите, что каждое поле — это объект, причем его класс зависит от типа поля: **TStringField**, **TSmallintField** и т.п. Все эти классы являются производными от **TField** — базового класса полей. Таким образом, каждое поле является объектом и обладает множеством свойств. Рассмотрим основные из них, которые чаще всего необходимо задавать.

Свойство **Alignment** определяет выравнивание отображаемого текста внутри колонки таблицы: влево, вправо или по центру.

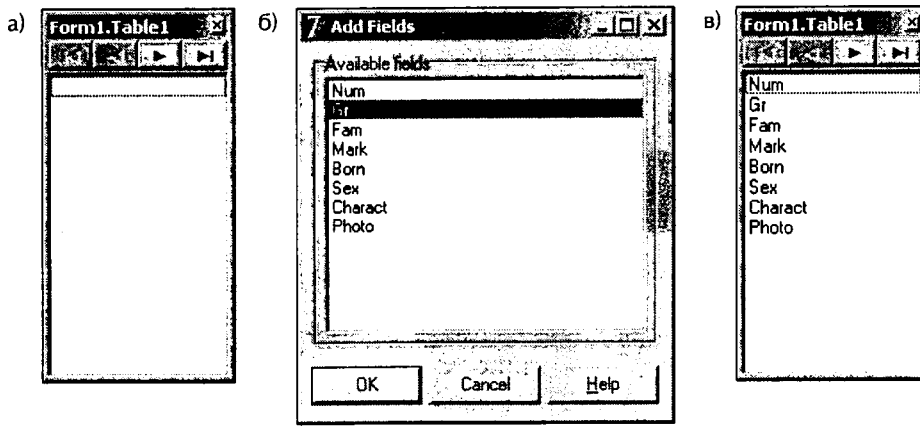


Рис. 6.11
Редактор Полей: исходное состояние (а), окно выбора полей (б), состояние после выбора (в)

Свойство **DisplayLabel** соответствует заголовку столбца данного поля. Например, для поля **Gr** значение **DisplayLabel** можно задать равным "Группа", для **Mark** — "Оценка" и т.д.

Свойство **DisplayWidth** определяет ширину колонки — число символов.

Свойство **ReadOnly**, установленное в **true**, запрещает пользователю вводить в данное поле значения. Свойство **Visible** определяет, будет ли видно пользователю соответствующее поле. В нашем примере, вероятно, можно задать **Visible** = **false** для полей **Num**, **Charact** и **Photo**.

После установки всех необходимых свойств приложение уже приобретает приемлемый вид, который вы видите на рис. 6.12. Правда, на этом рисунке имеется еще панель радиокнопок, позволяющая пользователю упорядочивать записи по своему усмотрению. Ввести такую панель в ваше приложение очень легко.

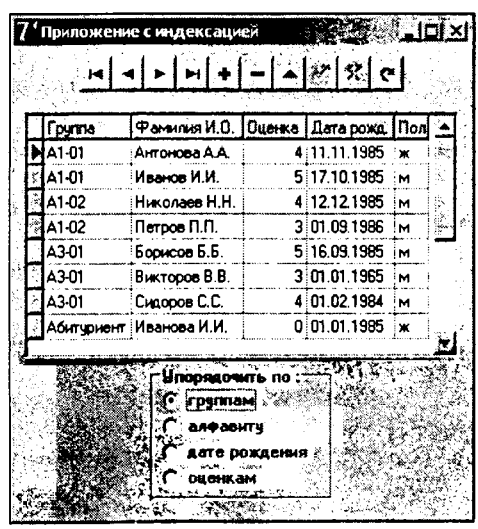


Рис. 6.12
Приложение с установленными свойствами полей и возможностью индексации

Вы уже знаете, что упорядочивание обеспечивается заданием в наборе данных соответствующего значения свойства **IndexName**. Так что при выборе пользователем той или иной радиокнопки надо просто задавать в свойстве **IndexName** имя соответствующего индекса. Перенесите на форму компонент **RadioGroup**, в его свойстве **Caption** напишите "Упорядочить по:", а в свойство **Items** занесите строки:

```
группам
алфавиту
дате рождения
оценкам
```

Задайте значение **ItemIndex** равным 0, и убедиться, что в **Table1** значение **IndexName** равно **GrFio**, т.е. согласовано с первой радиокнопокой. А теперь напишите следующий обработчик события **OnClick** группы радиокнопок **RadioGroup1**:

```
case RadioGroup1.ItemIndex of
 0: Table1.IndexName := 'GrFio';
 1: Table1.IndexName := 'Fio';
 2: Table1.IndexName := 'B';
 3: Table1.IndexName := 'MarkGr';
end;
```

Выполните приложение и убедитесь, что получили возможность упорядочивать данные по своему выбору.

6.7 Некоторые компоненты визуализации и редактирования данных

Компонент **DBGrid**, который мы использовали для отображения данных, удобный, но далеко не единственный и не всегда лучший компонент визуализации. Форма представления данных в нем не всегда удобная: слишком регулярная, без разделителей, не акцентирующая внимания пользователя на каких-то группах полей. Имеется много других компонентов визуализации и управления данными, расположенных на странице Data Control библиотеки компонентов, которые нередко предпочтительнее **DBGrid**.

DBText – аналог обычной метки **Label**, но связанный с данными. Он позволяет отображать данные некоторого поля, но не дает возможности его редактировать. Тип отображаемого поля может быть различным: строка, число, булева величина. Компонент автоматически переводит соответствующие типы в отображаемые символы.

DBEdit – связанный с данными аналог обычного окна редактирования **Edit**. Он позволяет отображать и редактировать данные полей различных типов: строка, число, булева величина. Впрочем, если задать в компоненте **ReadOnly** = **true**, то он, как и **DBText**, будет служить элементом отображения, но несколько более изящным, чем **DBText**.

DBMemo – связанный с данными аналог обычного многострочного окна редактирования **Memo**. Он позволяет отображать и редактировать данные поля типа **Memo**, а также данные любых типов, указанных выше для предыдущих компонентов. Например, в этом компоненте можно отображать и редактировать характеристику сотрудника, если такое поле предусмотрено в таблице.

DBRichEdit — связанный с данными аналог обычного многострочного окна редактирования текста в обогащенном формате RTF. Область применения та же, что для компонента **DBMemo**.

DBImage — связанный с данными аналог обычного компонента **Image**. Компонент позволяет отображать графические поля, например, фотографии сотрудников.

DBComboBox — связанный с данными аналог выпадающего списка **ComboBox**. Компонент позволяет отображать и редактировать поля с ограниченным множеством возможных значений. В нашем примере это может относиться к полям **Gr**, **Mark**, **Sex**. Свойство **Items**, как и в обычном **ComboBox**, определяет строки списка. Компонент **DBComboBox** предоставляет возможность ввода пользователем данных путем выбора, а не написанием полного значения поля. Это сокращает возможность ошибок со стороны пользователя при редактировании данных.

DBRadioGroup — связанный с данными аналог группы радиокнопок **RadioGroup**. Как и **DBComboBox**, этот компонент позволяет отображать и редактировать поля с ограниченным множеством возможных значений. В нашем примере это может относиться к полям **Gr**, **Mark**, **Sex**. Свойство **Items**, как и в обычной группе радиокнопок, определяет число кнопок и надписи около них. Но есть еще свойство **Values**, которое определяет значения полей, соответствующие кнопкам. Таким образом, надписи у кнопок и значения полей в общем случае могут не совпадать друг с другом. Например, при отображении поля **Mark** свойство **Items** может содержать строки "отлично", "хорошо" и т.д., а свойство — **Values** соответствующие значения поля "5", "4" и т.д. Если свойство **Values** не задано, то в качестве значений полей берутся строки из свойства **Items**, т.е. надписи около кнопок.

Все перечисленные компоненты имеют два основных свойства: **DataSource** — источник данных (компонент типа **TDataSource**) и **DataField** — поле, с которым связан компонент.

Характерной особенностью всех этих компонентов, отличающей их от несвязанных с данными аналогичных компонентов, является отсутствие в окне Инспектора Объектов их основных свойств, отображающих содержание: **Caption** в **DBText**, **Text** в **DBEdit**, **Lines** в **DBRichEdit**, **Picture** в **DBImage**, **ItemIndex** в **DBComboBox** и **DBRadioGroup** и т.п. Все эти свойства имеются в компонентах, но они доступны только во время выполнения. Так что программно их можно читать, редактировать, устанавливать, но во время проектирования задать их значения невозможно. Это естественно, так как подобные свойства — это значения соответствующих полей таблицы базы данных.

Опробуем некоторые из этих компонентов. Прежде всего, нам надо внести в приложение компоненты, которые отображали бы фотографию и характеристику. Перенесите на форму компонент **DBRichEdit** для отображения характеристики и **DBImage** для отображения фотографии (см. рис. 6.13). В обоих компонентах установите свойство **DataSource** равным **DataSource1**. Свойство **DataField** в **DBRichEdit1** установите равным **Charact**, а в **DBImage** равным **Photo**.

Для отображения характеристики и фотографии сделанного достаточно. Для формирования характеристики тоже достаточно — пользователь просто сможет писать характеристику в окне **DBRichEdit1**. Более того, вы можете добавить на форму компоненты и инструментальную панель для форматирования текста в окне. Все это не будет отличаться от того, что вы делали в гл. 4. А вот для занесения в окно **DBImage1** изображения из графического файла надо предусмотреть соответствующую кнопку (см. рис. 6.13) и перенести на форму со страницы библиотеки Dialogs компонент **OpenPictureDialog**. Это компонент диалога, обеспечиваю-

щего открытие графического файла. Обработчик щелчка на кнопке ввода фотографии может иметь вид:

```
if OpenPictureDialog1.Execute
 then begin
  Table1.Edit;
  DBImage1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
  Table1.Post;
 end;
```

В приведенном коде использованы методы **Edit** и **Post** набора данных **Table**. Подробнее о методах наборов данных вы узнаете в разд. 6.11. А пока отмечу, что метод **Edit** переводит набор в режим редактирования, а метод **Post** посылает отредактированные данные в БД. Между этими двумя операторами записана загрузка методом **LoadFromFile** в свойство **Picture** компонента **DBImage1** изображения из файла, который выбрал пользователь в диалоге открытия файла **OpenPictureDialog1**. В данном коде можно было бы обойтись без явного использования методов **Edit** и **Post**. Но тогда пользователь перед занесением фотографии должен был бы нажать кнопку навигатора, переводящую набор в режим редактирования, а после занесения фотографии должен был бы нажать кнопку навигатора, пересылающую запись в базу данных. Конечно, это было бы не очень удобно.

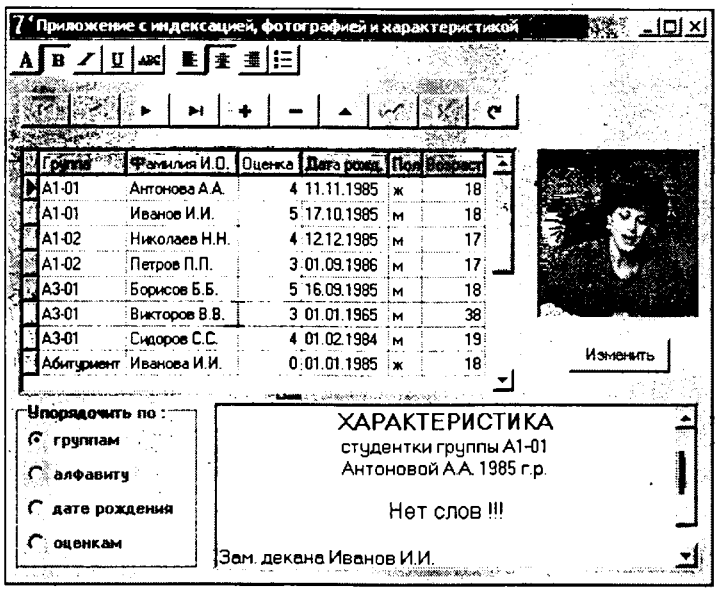


Рис. 6.13
Приложение с отображением фотографии и характеристики

Выполните ваше приложение и заполните поля характеристик и фотографий вашей таблицы данных. С Delphi поставляется несколько изображений, подходящих для имитации фотографий. Одно из них вы видите на рис. 6.13. Соответствующие файлы расположены в каталоге *Program Files \ Common Files \ Borland Shared \ Data*.

Попробуйте и другие компоненты доступа к данным. На рис. 6.14 показано приложение, в котором **DBGrid** используется только для отображения поля **Fam**.

Поле **Gr** отображается в компоненте **DBComboBox**, поля **Born** и **Mark** — в компонентах **DBEdit**, поле **Sex** — в компоненте **DBRadioGroup**. Попробуйте сами создать такое приложение, пользуясь приведенным выше описанием соответствующих компонентов.

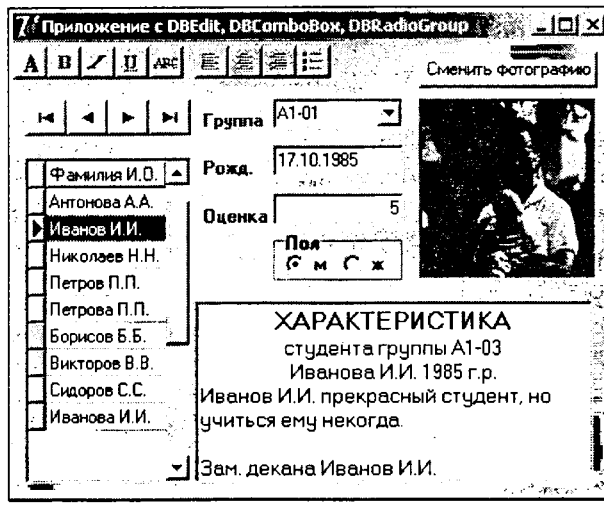


Рис. 6.14
Приложение, использующее компоненты DBEdit и DBRadioGroup

6.8 Вычисляемые поля

Если вы приглядитесь к приведенному ранее рис. 6.13, то увидите, что в таблице имеется столбец Возраст. Но такого поля нет в таблице Pers, с которой работает приложение! Значит, на уровне приложения можно вводить в таблицу дополнительные поля? Да, это так называемые *вычисляемые поля*, не предусмотренные при создании таблицы. Значение подобного поля вычисляется на основании значений других полей записи.

Для того чтобы ввести вычисляемое поле в набор данных, сделайте сначала двойной щелчок на **Table1**, чтобы вызвать Редактор Полей. Щелкните в Редакторе Полей правой кнопкой мыши и во всплывшем меню выберите раздел New (новое поле). Появится окно добавления нового поля, приведенное на рис. 6.15.

В разделе Field properties (свойства поля) вы должны указать имя поля (Name) — в нашем случае назовем это поле **Age**, тип данных (Type) — в нашем случае это Smallint, и для некоторых типов — размер (Size). Размер указывается для строк и других полей неопределенных размеров.

После ввода всех данных проверьте, переключилась ли группа радиокнопок Field type на Calculated (это переключение делается автоматически). Затем щелкните на OK, и вы вернетесь в окно Редактора Полей, причем там появится новое поле **Age**. Задайте для него в Инспекторе Объектов значение **DisplayLabel** равным "Возраст".

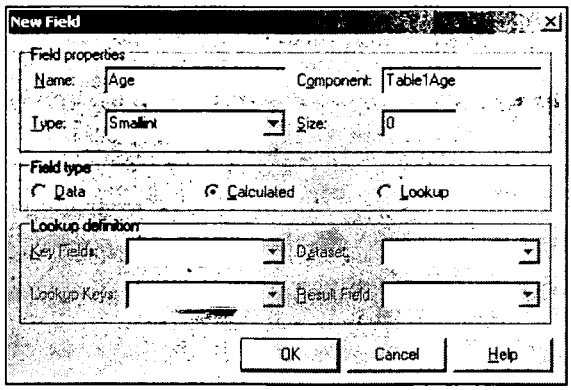


Рис. 6.15
Ввод информации о вычисляемом поле

Вы ввели вычисляемое поле **Age**, но еще не указали программе, как его надо рассчитывать. Чтобы указать процедуру вычислений, выйдите из Редактора Полей, выделите **Table1**, перейдите в Инспекторе Объектов на страницу событий и щелкните на событии **OnCalcFields**. Это событие наступает каждый раз, когда надо обновить значение вычисляемых полей таблицы.

Чтобы вычислить возраст по году рождения вы можете в обработчике этого события написать оператор:

```
Table1Age.Value := YearsBetween(Date, Table1Born.Value);
```

В этом операторе **Table1Age** и **Table1Born** — объекты полей **Age** и **Born** соответственно. Эти имена вы можете видеть в окне Инспектора Объектов, когда работаете с Редактором Полей. Они формируются из имени набора данных ("Table1"), за которым без пробела следует имя поля. Значение поля хранится в свойстве **Value** объекта поля. Так что левая часть приведенного оператора означает, что значению поля **Age** присваивается значение правой части. Значение этой правой части определяется функцией **YearsBetween**, которая возвращает число полных лет, разделяющих две даты. Одна дата — значение поля **Born** (дата рождения). А вторая — текущая дата, возвращаемая функцией **Date**.

Функция **YearsBetween** объявлена в модуле *DateUtils*, который надо подключить соответствующим оператором **uses**:

```
uses DateUtils;
```

Даты и время в Delphi представляются типом **TDateTime**. Это действительное число, целая часть которого содержит число дней, отсчитанное от 0 часов 30 декабря 1899 года, а дробная часть равна части 24-часового дня, т.е. характеризует время и не относится к дате. Так что целая часть разности двух значений типа **TDateTime** дает число дней, разделяющих две даты. Поэтому можно обойтись и без функции **YearsBetween**, определяя возраст оператором:

```
Table1Age.Value := trunc((Date - Table1Born.Value) / 365.25);
```

Тогда и модуль *DateUtils* подключать не надо. Правда, приведенный оператор не полностью учитывает различие между високосными и не високосными годами, исходя из усредненного значения 365.25 дней в году. Но и функция **YearsBetween** делает то же самое.

6.9 Фильтрация данных

Компонент **Table** позволяет не только отображать, редактировать и упорядочивать данные, но и отфильтровывать записи по определенным критериям. Пользователю в нашем примере могло бы захотеться иметь возможность просматривать не всю базу данных, а отдельно записи по той или иной группе, или по совокупности групп, или, например, просмотреть записи студентов, имеющих оценки ниже 4.

Фильтрация может задаваться свойствами **Filter**, **Filtered** и **FilterOptions** компонента **Table**. Свойство **Filtered** включает или выключает использование фильтра. А сам фильтр записывается в свойство **Filter** в виде строки, содержащей определенные ограничения на значения полей. Например, вы можете задать в свойстве **Filter**

```
Gr='A1-01'
```

установить свойство **Filtered** в **true**, и увидите, что уже в процессе проектирования в таблице отобразятся только те записи, в которых поле **Gr** имеет значение "A1-01". В условиях сравнения строк можно использовать символ звездочки "*", который как в обычных шаблонах означает: «любое количество любых символов». Например, фильтр

```
Gr='A1*'
```

приведет к отображению всех записей, в которых значение поля **Gr** начинается с "A1". Если цифра после символа "A" означает номер семестра, то тем самым отобразятся данные о студентах первого курса. Но для того, чтобы это сработало, надо, чтобы в опциях, содержащихся в свойстве **FilterOptions**, была выключена опция **foNoPartialCompare**, запрещающая частичное совпадение при сравнении (эта опция выключена по умолчанию). Другая опция в свойстве **FilterOptions** — **foCaseInsensitive** делает сравнение строк нечувствительным к регистру, в котором записано условие фильтра. Если включить эту опцию, то слова "A1-01" и "a1-01" будут считаться идентичными.

При записи условий можно использовать операции отношения =, >, >=, <, <=, <>, а также логические операции **and**, **or** и **not**. Например, вы можете написать фильтр

```
(Gr='A1*') and (Mark < 4)
```

и отобразятся записи студентов первого курса, которые имеют проблемы с информатикой — оценку меньше 4.

В фильтре не разрешается использовать имена вычисляемых полей (например, введенного вами поля **Age**).

Конечно, свойства, определяющие фильтрацию, можно задавать не только в процессе проектирования, но и программно, во время выполнения. Давайте введем такую возможность в наше приложение. Добавьте на форму группу радиокнопок Курс (см. рис. 6.16) и задайте в ее свойстве **Items** строки:

```
Первый курс
Второй курс
Абитуриенты
Все
```

В этой группе радиокнопок пользователь сможет задавать необходимую ему фильтрацию. В начальный момент должна быть выбрана кнопка Все, и фильтра-

Рис. 6.16
Приложение с возможностью фильтрации записей

ция в **Table1** выключена установкой свойства **Filtered** в **false**. В обработчик события **OnClick** группы радиокнопок (ее имя **RadioGroup2**) вставьте код:

```
Table1.Filtered := not (RadioGroup2.ItemIndex = 3);
case RadioGroup2.ItemIndex of
 0: Table1.Filter := 'Gr=''A1*''';
 1: Table1.Filter := 'Gr=''A3*''';
 2: Table1.Filter := 'Gr=''Абитуриент''';
end;
```

Первый оператор включает или выключает фильтрацию в зависимости от того, выбрал ли пользователь кнопку Все (**RadioGroup2.ItemIndex** = 3), или какую-либо другую. А конструкция **case** задает в зависимости от выбора пользователя соответствующее значение свойства **Table1.Filter**.

6.10 Приложения со связанными таблицами

Предыдущие приложения общались с одной таблицей. Теперь посмотрим, как строить приложения с несколькими связанными друг с другом таблицами.

Две таблицы могут быть связаны друг с другом по ключу. Одна из этих связанных таблиц является головной (master), а другая — вспомогательной, детализирующей (detail). Например, мы хотим построить приложение, в котором имеется таблица Gro, содержащая список групп (поле **Gr1**) и какую-то их характеристику

(поле **Teach** — преподаватель по информатике). И хотим, чтобы пользователь, перемещаясь по этой таблице, видел не только характеристику группы, но и список студентов этой группы, т.е. записи из таблицы Pers, в которых значение поля **Gr** совпадает со значением поля **Gr1** в текущей записи первой таблицы. В этом случае головной является таблица Gro, вспомогательной — таблица Pers, а ключом, определяющим их связь, являются поля **Gr1** и **Gr** сответствующих таблиц.

Откройте новое приложение и разместите на нем 2 комплекта компонентов **Table**, **DataSource**, **DBCtrlGrid** и **DBNavigator**. Результат может выглядеть, например, так, как показано на рис. 6.17 (о кнопке Связь таблиц будет сказано позднее). Первый комплект обычным образом настраивается на первую, головную (master) таблицу — Gro. Второй комплект настраивается на вторую вспомогательную таблицу — Pers. Для этой таблицы обязательно должен быть задан индекс, содержащий ключевое поле связи **Gr** (индекс **GrFio**). После того как все это сделано, можете выполнить приложение, чтобы убедиться, что все сделано правильно. Пока вы имеете две несвязанные друг с другом таблицы, по которым можете перемещаться независимо.

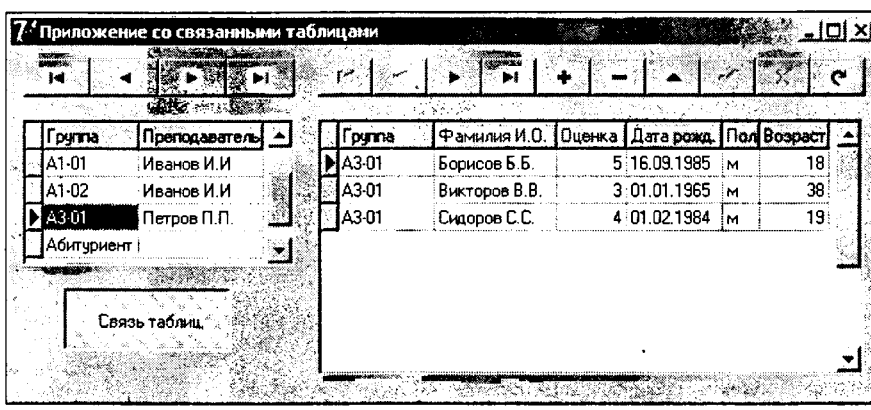


Рис. 6.17
Приложение с двумя связанными таблицами

Теперь свяжите эти таблицы. В компоненте **Table2**, настроенном на вспомогательную таблицу, в свойстве **MasterSource** установите имя источника головной таблицы **DataSource1**. После этого щелкните на свойстве **MasterFields**. Откроется окно редактора связей полей (Field Link Designer). Его вид приведен на рис. 6.18. В нем слева в окне Detail Fields расположены имена полей вспомогательной таблицы, но только тех, по которым таблица индексирована. Именно поэтому надо индексировать таблицу так, чтобы индекс включал ключевое поле связи **Gr**. Справа в окне Master Fields расположены поля головной таблицы. Выделите в одном и другом окне ключевые поля: слева **Gr**, справа **Gr1**. При этом активизируется кнопка Add, и после щелчка на ней ключевые поля переносятся в окно Joined Fields — соединяемые поля. Щелкните на OK. После этого можете проверить наличие связи компонентов **Table** с базой данных **(Active = true)** и запустить приложение.

Вы увидите, что в зависимости от того, какую запись вы выделяете в списке отделов, вам отображается список сотрудников этого отдела. Таким образом, курсор скользит по головной таблице, а вспомогательная таблица отображает только те записи, в которых ключевые поля совпадают с ключевыми полями головной таблицы.

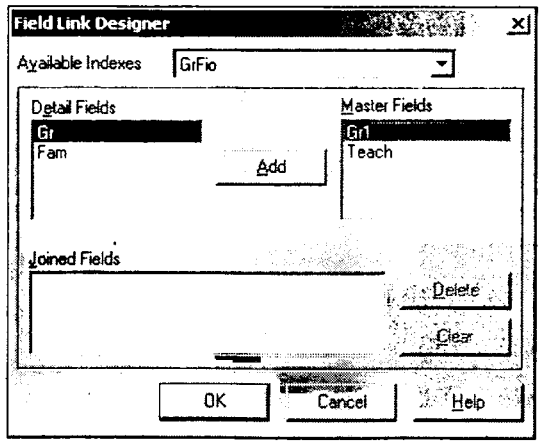


Рис. 6.18
Окно редактора связей полей головной и вспомогательной таблиц

Мы получили связанные таблицы данных. Но хотелось бы иметь возможность по желанию разрывать эту связь, чтобы во второй таблице получить полный список студентов. Введите для этого в приложение кнопку **SpeedButton** (Связь таблиц на рис. 6.17). В нажатом состоянии она будет обеспечивать связь таблиц, а в отжатом состоянии будет разрывать ее. Задайте в кнопке свойство **GroupIndex** = 1 и установите свойство **AllowAllUp** в **true**. Тем самым обеспечивается (см. разд. 5.4.3) возможность фиксации кнопки как в нажатом, так и в отпущенном состоянии. Установите в **true** ее свойство **Down**, чтобы в исходном состоянии она была нажата. А в обработчик щелчка на ней запишите оператор:

```
if SpeedButton1.Down
  then Table2.MasterSource := DataSource1
  else Table2.MasterSource := nil;
```

Если после щелчка кнопка оказывается нажатой, то в свойство **MasterSource** таблицы **Table2** заносится указатель на источник данных **DataSource1**. Тем самым задается связь таблиц. А если кнопка не нажата, то в это свойство заносится нулевой указатель **nil**, т.е. связь таблиц разрывается.

6.11 Программирование работы с базами данных

6.11.1 Состояние набора данных

В рассмотренных ранее простых приложениях практически не требовалось программирование. Все сводилось к размещению на форме компонентов и заданию их свойств. Но более изощренные приложения все-таки требуют программирования и написания различных обработчиков событий.

Начнем рассмотрение вопросов программирования работы с базами данных с основного свойства **State** компонента типа **Table**, определяющего состояние набора данных. Это свойство доступно только во время выполнения и только для чтения. Набор данных может находиться в одном из следующих основных состояний:

dsInactive	Набор данных закрыт, данные недоступны.
dsBrowse	Данные могут наблюдаться, но не могут изменяться. Это состояние по умолчанию после открытия набора данных.
dsEdit	Текущая запись может редактироваться.
dsInsert	Может вставляться новая запись.

Перечисленные состояния могут устанавливаться в приложении во время выполнения, но не непосредственно, а с использованием различных методов.

Метод **Close** закрывает соединение с базой данных, устанавливая свойство **Active** набора данных в **false**. При этом **State** переводится в состояние **dsInactive**.

Метод **Open** открывает соединение с базой данных, устанавливая свойство **Active** набора данных в **true**. При этом **State** переводится в состояние **dsBrowse**.

Метод **Edit** переводит набор данных в состояние **dsEdit**.

Методы **Insert** и **InsertRecord** вставляют новую запись в набор данных и переводят **State** в состояние **dsInsert**.

При программировании работы с базой данных надо следить за тем, чтобы набор данных вовремя был переведен в соответствующее состояние. Например, если вы стали редактировать запись, не переведя предварительно набор данных методом **Edit** в состояние **dsEdit**, то будет сгенерировано исключение **EDatabaseError** с сообщением «Dataset not in edit or insert mode» — «Набор данных не в режиме Edit или Insert». До сих пор со всем этим нам не приходилось сталкиваться только потому, что компоненты **DBGrid** и **DBNavigator** автоматически выполняли все необходимые действия.

6.11.2 Методы навигации

Наборы данных имеют ряд методов, позволяющих осуществлять навигацию — перемещение по таблице:

First	перемещение к первой записи
Last	перемещение к последней записи
Next	перемещение к следующей записи
Prior	перемещение к предыдущей записи
MoveTo(i:integer)	перемещение к концу (при **i**>0) или к началу (при **i**<0) на **i** записей

При перемещениях можно совершить ошибку, выйдя за пределы имеющихся записей. Например, если вы находитесь на первой записи и выполняете метод **Prior**, то вы выйдете за начало таблицы, а если вы находитесь на последней записи и выполняете метод **Next**, то вы окажетесь после последней записи. Чтобы контролировать начало и конец таблицы, существуют два свойства: **EOF** (end of file) — конец данных, и **BOF** (beginning of file) — начало данных. Эти свойства становятся равными **true**, если делается попытка переместить курсор соответственно за пределы последней или первой записи, а также после выполнения методов соответственно **Last** и **First**.

Приведем пример. Пусть в вашем приложении имеется выпадающий список с именем **ComboBox1**, который вы хотите заполнить данными, содержащимися

в полях **Gr1** всех записей таблицы, соединенной с компонентом **Table1**. Это можно сделать следующим кодом:

```
ComboBox1.Clear;
Table1.First;
while not Table1.EOF do
 begin
   ComboBox1.Items.Add(Table1.FieldByName('Gr1').Value);
   Table1.Next;
 end;
```

Первый оператор кода очищает список **ComboBox1**. Второй — устанавливает курсор таблицы на первую запись. Далее следует цикл по всем записям, пока не достигнута последняя, что проверяется выражением **Table1.EOF**. Для каждой записи в список заносится значение поля **Gr1**, после чего методом **Next** курсор перемещается к следующей записи.

6.11.3 Пересылка записи в базу данных

Пока идет редактирование текущей записи, изменения осуществляются в буфере, а не в самой базе данных. Пересылка записи в базу данных производится только при выполнении метода **Post**. Метод может вызываться только тогда, когда набор данных находится в состоянии **dsEdit** или **dsInsert**. Этот метод можно вызывать явно, например, **Table1.Post**. Но, кроме того, он вызывается неявно при любых перемещениях по набору данных, т.е. при перемещении курсора на новую текущую запись, если набор данных находится в состоянии **dsEdit** или **dsInsert**. Отменить изменения, внесенные в запись, можно методом **Cancel**. При выполнении этого метода, если предварительно не был вызван метод **Post**, запись возвращается к состоянию, предшествовавшему редактированию, и набор данных переводится в состояние **dsBrowse**.

Все приложения, которые мы до сих пор создали, страдали очень серьезным недостатком: метод **Post** вызывался в них неявно при перемещениях по базе данных. Пользователь мог по ошибке сделать неверные исправления в какой-то записи и, переместившись на другую запись, невольно занести их в таблицу без какой-то предварительной проверки. Надо принимать меры, чтобы без проверки данных и без подтверждения пользователем в базу данных ничего не заносилось. Это можно сделать, используя множество событий компонента **Table**.

Перед началом выполнения каждого из рассмотренных выше методов и после его выполнения генерируются соответствующие события набора данных **Table**. Например, перед выполнением метода **Post** возникает событие **BeforePost**, а после его окончания — событие **AfterPost**. Аналогичные события **BeforeInsert** и **AfterInsert** сопровождают выполнения метода **Insert** и т.п. Подобные события и надо использовать для проверки данных и получения подтверждения на их изменение.

Один из множества возможных вариантов заключается в использовании события **BeforePost**. В наших примерах, которые вы создали в данной главе, вероятно, имеет смысл проверить, не забыл ли пользователь ввести фамилию. Правда, при создании таблицы вы сделали поле **Fam** обязательным. Но вряд ли имеет смысл дожидаться, пока из базы данных поступит английское сообщение об этой ошибке. Лучше предупредить пользователя заранее и дать ему понятное сообщение. Имеет смысл также проверить правильность заполнения поля **Sex**. Во-первых, пользователь по ошибке может занести в это поле какой-то символ, отличный от "м" и "ж".

А во-вторых, чтобы не было разнобоя в таблице, лучше привести символ этого поля к нижнему регистру, если пользователь ввел его в верхнем регистре. Можно проверить также допустимую дату рождения и сделать ряд других проверок. В результате обработчик события **BeforePost** для наших примеров может иметь вид:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);
var Err: string;
begin
 Table1Sex.Value := AnsiLowerCase(Table1Sex.Value);
 Err := '';
 if (Table1Sex.Value <> 'м') and (Table1Sex.Value <> 'ж')
  then Err := 'Ошибочное значение пола. ';
 if Table1Fam.Value = ''
   then Err := Err + 'Забыли указать фамилию.';
 if Err = ''
  then begin
   if Application.MessageBox(
               'Хотите занести текущую запись в базу данных?',
               'Подтвердите сохранение изменений',
               MB_YESNOCANCEL+MB_ICONQUESTION) <> IDYES
    then begin
     Table1.Cancel;
     Abort;
    end
   end
  else begin
   Application.MessageBox(PChar(Err), 'Исправьте ошибку',
                          MB_OK + MB_ICONEXCLAMATION);
   Abort;
  end;
end;
```

К этому обработчику будет происходить обращение перед выполнением метода **Post**, как бы он ни был вызван: явно или вследствие перемещения по базе данных, если текущая запись была изменена. В обработчике сначала значение поля **Sex** приводится функцией **AnsiLowerCase** к нижнему регистру. Затем проверяется значение этого поля. Если оно не равно "м" или "ж", то в переменную **Err** заносится сообщение об ошибке. Далее проверяется поле **Fam**, и если оно пустое, то в **Err** добавляется сообщение об этой ошибке. Далее проверяется, занесены ли какие-то сообщения в переменную **Err**. Если занесены, то пользователю дается сообщение об ошибочности данных и выполняется функция **Abort**, прерывающая выполнение **Post**. Текущая запись остается в состоянии **dsEdit**, но ошибочные данные в ней не сбрасываются, что позволяет пользователю исправить их.

Если замечаний нет, то у пользователя запрашивается подтверждение изменений в базе данных. Если он ответит отрицательно, то для набора данных выполняется метод **Cancel**, а затем выполняется функция **Abort**. **Cancel** возвращает данные в текущей записи к состоянию, которое было до их редактирования, т.е. уничтожает результаты редактирования.

6.11.4 Кэширование изменений

По умолчанию все изменения в наборе данных, завершаемые методами **Post**, **Insert**, **Delete** и т.п., заносятся в базу данных. Однако возможен и другой режим

работы, при котором все изменения, вносимые пользователем в записи, кэшируются — т.е. сохраняются в памяти локально. В этом случае пользователь работает не с реальными данными, а с их копиями. И только по специальной команде все внесенные пользователем изменения заносятся в базу данных.

Режим кэширования определяется свойством **CachedUpdates** компонента **Table**. По умолчанию это свойство равно **false** и кэширование не производится. Если установить его в **true**, то все изменения набора данных будут кэшироваться. Они передаются в базу данных только после выполнения метода **ApplyUpdates**. Если же после множества изменений выполнить метод **CancelUpdates**, то все изменения, произведенные после последнего выполнения **ApplyUpdates**, отменяются.

Метод **CommitUpdates** очищает буфер кэша, после чего он готов для приема новой порции информации.

Проверить это можно на любом ранее созданном вами приложении, работающем с базами данных. Возьмите, например, за основу приложение, разработанное в разд. 6.7. Но теперь добавьте в него две кнопки: Фиксация и Отмена. Кнопка Фиксация фиксирует все сделанные изменения в базе данных. Кнопка Отмена отменяет сделанные изменения. Вид такого приложения приведен на рис. 6.19. Правда, на этом рисунке вы можете видеть также элементы, связанные с поиском записей, которые будут рассмотрены в следующем разделе.

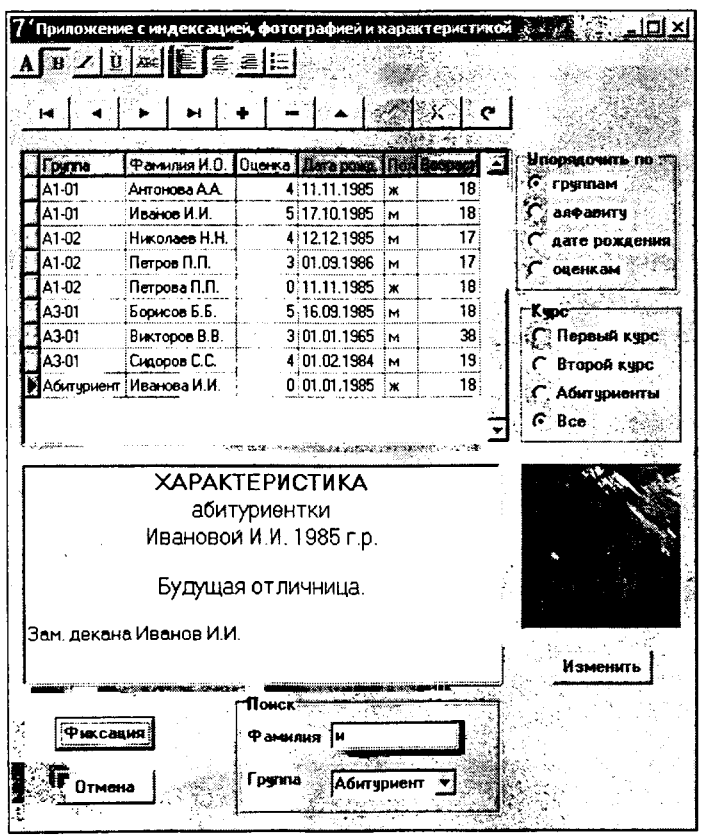


Рис. 6.19

Приложение, демонстрирующее кэширование данных и поиск записей

Свойство **CachedUpdates** компонента **Table1** надо задать равным **true**. В обработчик щелчка на кнопке Отмена вставьте оператор:

```
Table1.CancelUpdates;
```

А в обработчик щелчка на кнопке Фиксация вставьте операторы:

```
Table1.ApplyUpdates;
Table1.CommitUpdates;
```

В обработчик щелчка на группе радиокнопок Упорядочить надо вставить оператор

```
Table1.Close;
```

вначале, перед изменениями значения свойства **IndexName**, и оператор

```
Table1.Open;
```

в конце. Это связано с тем, что в режиме кэширования нельзя изменять свойство **IndexName** при открытом соединении.

Выполните приложение и опробуйте на нем режим кэширования. Вы можете убедиться, что исправления, сделанные в нескольких записях, можно в любой момент отменить, щелкнув на кнопке Отмена. Или принять их все, щелкнув на кнопке Фиксация. Но нетрудно заметить один недостаток нашего приложения. Пользователь может работать с таблицей, исправлять записи, вносить новые данные. Но если он перед закрытием приложения забудет щелкнуть на кнопке Фиксация, то весь его труд пропадет. Нельзя так наказывать пользователя за невнимательность. Давайте сделаем так, чтобы приложение следило за действиями пользователя, и если он закрывает приложение, не зафиксировав в базе данных результаты редактирования, предупреждало бы его об этом и давало возможность сохранить свою работу.

Введите в приложение глобальную переменную **modif**, которая будет фиксировать число отредактированных и несохраненных записей:

```
var modif: integer = 0;
```

Увеличивать значение этой переменной надо в случаях, когда пользователь произвел какие-то изменения в записи. Для этого можно воспользоваться событием **AfterEdit** компонента **Table1**. Это событие возникает после того, как пользователь начал редактирование записи. Так что в обработчик этого события вставьте оператор:

```
Inc(modif);
```

который увеличивает на 1 значение **modif**, фиксируя тем самым факт редактирования записи.

Однако перейдя в режим редактирования, пользователь может раздумать редактирование записи или отменить результаты редактирования соответствующей кнопкой навигатора. В подобных случаях надо уменьшать значение **modif** на 1. Воспользуемся для этого событием **AfterCancel** компонента **Table1**. Это событие возникает, если пользователь отказался от редактирования записи. Так что в обработчик этого события вставьте оператор:

```
Dec(modif);
```

При щелчках пользователя на кнопках Фиксация и Отмена в памяти не остается несохраненных изменений. Так что в обработчики этих событий надо вставить операторы:

```
modif := 0;
```

А теперь можно написать обработчик события **OnCloseQuery** формы:

```
if (modif > 0)
 then
  case Application.MessageBox(
                    'Сохранить изменения в базе данных?',
                    'Подтвердите сохранение изменений',
                    MB_YESNOCANCEL+MB_ICONQUESTION) of
   IDYES:    Table1.ApplyUpdates;
   IDCANCEL: CanClose := false;
   IDNO:     Table1.CancelUpdates;
  end;
```

При наличии не сохраненных изменений пользователю задается вопрос «Сохранить изменения в базе данных?». При положительном ответе изменения сохраняются методом **ApplyUpdates**. При отрицательном — изменения отменяются методом **CancelUpdates**. Если при получении запроса пользователь нажал кнопку Cancel, то приложение не закрывается (**CanClose** задается равным **false**).

И еще одно изменение, которое имеет смысл сделать: если вы написали в приложении обработчик события **BeforePost**, как рассказывалось в разд. 6.11.3, то проверку и сообщение об ошибке в нем следует оставить, а запрос о занесении записи в базу данных надо убрать. Ведь теперь метод **Post** посылает запись не в базу данных, а в кэш. Так что этот запрос становится бессмысленным и только затрудняет работу пользователя.

6.11.5 Поиск записей

Одна из важнейших для пользователя операций с базами данных — поиск записей по некоторому ключу. Например, хотелось бы иметь в приложении окно ускоренного поиска, в котором можно было бы набирать по символам фамилию, и курсор сразу перемещался бы на запись, в которой первые символы фамилии совпадают с введенными пользователем. Еще лучше иметь возможность искать не только по всей таблице, но и по отдельным ее частям, например, по отдельным группам.

Подобная возможность предусмотрена в приложении, вид которого вы могли видеть на рис. 6.19 в разд. 6.11.4. В нем имеется окно Фамилия (**Edit1**), в котором можно вводить искомую фамилию. Имеется также выпадающий список Группа (**ComboBox1**), в котором первая строка имеет текст "Все", а последующие строки содержат названия всех групп. Если пользователь задает в списке первую строку, то поиск ведется по всем группам. А если он выберет в списке любую другую строку, то поиск будет проводиться только по студентам данной группы. Например, на рис. 6.19 в списке выбрана строка "Абитуриенты". Поэтому при вводе в окно символа "и" курсор переместился на Иванову, а не на Иванова, хотя Иванов расположен в списке раньше. Но он не является абитуриентом.

Давайте, реализуем подобные возможности в нашем приложении. Проще всего использовать для поиска метод **Locate**, правда, доступный не во всех версиях Delphi. Этот метод объявлен следующим образом:

```
function Locate(const KeyFields: string;
                const KeyValues: Variant;
                Options: TLocateOptions): Boolean;
```

В качестве первого параметра **KeyFields** передается строка, содержащая список ключевых полей. В качестве второго параметра передается **KeyValues** — массив

ключевых значений. А третий параметр **Options** является множеством опций, элементами которого могут быть **loCaseInsensitive** — нечувствительность поиска к регистру, в котором введены символы, и **loPartialKey** — допустимость частичного совпадения. Метод возвращает **false**, если искомая запись не найдена.

Например, поиск по фамилии, независящий от группы, может осуществляться оператором

```
Table1.Locate('Fam',Edit1.Text,
              [loCaseInsensitive,loPartialKey]);
```

При поиске по нескольким полям можно воспользоваться функцией **VarArrayOf**, которая формирует тип **Variant** из задаваемого ей массива параметров любого типа. Например, поиск по группе и фамилии может быть осуществлен оператором

```
Table1.Locate('Gr;Fam',
                VarArrayOf([ComboBox1.Text, Edit1.Text]),
              [loCaseInsensitive, loPartialKey]);
```

Не стоит углубляться в достаточно тонкий вопрос формирования функцией **VarArrayOf** типа **Variant** из массива параметров. Просто, примите как некоторый формализм, что в вызов функции **VarArrayOf** надо передать в квадратных скобках список искомых значений.

Прежде, чем осуществлять задуманный нами поиск, надо заполнить строки списка **ComboBox1**. Можно, конечно, заполнить его вручную с помощью свойства **Items** компонента **ComboBox1**. Но это было бы неудачным решением, так как при изменении в базе данных списка групп пришлось бы перекомпилировать приложение. Вспомним, что у нас имеется таблица Gro. Она имеет поле **Gr1**, которое содержит список всех групп. Так что достаточно пройти по всем записям таблицы Gro и занести значения ее поля **Gr1** в список **ComboBox1**. Добавьте на форму еще один компонент **Table**, соедините его с таблицей Gro и в обработчик события **OnCreate** формы включите операторы:

```
Table2.Open;
ComboBox1.Clear;
ComboBox1.Items.Add('Все');
Table2.First;
while not Table2.EOF do
 begin
  ComboBox1.Items.Add(Table2.FieldByName('Gr1').Value);
  Table2.Next;
end;
Table2.Close;
ComboBox1.ItemIndex := 0;
```

Первый оператор открывает набор данных **Table2**, второй очищает список **ComboBox1**. Третий оператор заносит в список строку "Все". Впрочем, можно было бы занести эту строку в список вручную с помощью свойства **Items**. Тогда второй и третий оператор были бы не нужны, как и последний оператор, задающий значение **ItemIndex**.

После подготовительных операций курсор таблицы **Table2** устанавливается на первую запись методом **First** (см. этот и другие использованные в коде методы навигации в разд. 6.11.2). Затем в цикле просматриваются все записи таблицы. Значение поля **Gr1** в каждой записи извлекается методом **FieldByName** и заносится

в список **ComboBox1**, после чего методом **Next** осуществляется переход к следующей записи. По окончании цикла (определяется свойством **Eof**) соединение таблицы **Table2** закрывается методом **Close**.

Теперь осталось написать обработчик события **OnChange** окна **Edit1**. Это событие наступает после ввода каждого символа в окно редактирования. Так что по мере ввода фамилии можно позиционировать курсор на запись, наиболее соответствующую введенным символам. Код обработчика события **OnChange** может быть следующим:

```
if ComboBox1.ItemIndex = 0
 then Table1.Locate('Fam', Edit1.Text,
                    [loCaseInsensitive, loPartialKey])
 else Table1.Locate('Gr;Fam',
                    VarArrayOf([ComboBox1.Text, Edit1.Text]),
                    [loCaseInsensitive, loPartialKey]);
```

Выполните усовершенствованное вами приложение, и убедитесь, что поиск записи работает нормально.

6.12 Некоторые итоги

В данной главе вы изучили основы работы с базами данных. Это, собственно, главное предназначение Delphi — обеспечить разработчика инструментарием для построения пользовательского интерфейса к базам данных любого типа. Тут надо подчеркнуть, что Delphi — средство разработки не баз данных, а интерфейса к ним, т.е. программ, обеспечивающих работу пользователя с данными, обработку данных, редактирование и реорганизацию таблиц. Поэтому совершенно неправомерны вопросы, которые иногда задают: «Что лучше: Delphi или FoxPro, Delphi или Microsoft Access» и т.д. Microsoft Access, FoxPro, Oracle и многие другие являются СУБД — системами управления базами данных. СУБД, прежде всего, обеспечивают поддержку внутренних форматов и возможностей БД. Правда, СУБД предоставляют и определенные средства для просмотра и редактирования данных пользователем. Но эти средства, конечно, несопоставимы с тем богатством возможностей, которые дает Delphi. Так что роль Delphi — позволить вам создавать приложения, работающие с любыми базами данных и способные удовлетворить самого взыскательного пользователя.

Рассмотренный в данной главе компонент **Table**, как и рассмотренный в следующей главе компонент **Query**, общаются с данными через BDE (см. разд. 6.1.3). Честно говоря, в настоящее время это не лучший вариант. При работе с некоторыми современными базами данных в BDE могут возникать проблемы с отображением русских текстов. Одним из недостатков BDE является также то, что с вашим приложением сможет работать только тот пользователь, на компьютере которого установлена BDE. Правда, установить BDE не сложно. Проще всего это сделать с тех же дисков, с которых вы устанавливали Delphi на своем компьютере. При установке надо отключить все индикаторы, определяющие, что именно устанавливается, кроме индикатора BDE. Тогда на компьютере установится только BDE. Правда, вам еще надо будет ввести на новом компьютере все используемые вами псевдонимы.

Подобных недостатков лишен доступ к данным посредством разработанной в Microsoft технологии ActiveX Data Objects (ADO). ADO — это пользователь-

ский интерфейс к любым типам данных, включая реляционные базы данных, электронную почту, системные, текстовые и графические файлы. Одним из преимуществ ADO является то, что эта технология встроена в Windows, и поэтому имеется на любом компьютере, на котором установлена не слишком старая версия системы. Создание файла связи ADO — файла *.udl* описано в справке Windows и может несколько различаться для различных версий системы. Получить соответствующую информацию можно, нажав клавишу F1 на рабочем столе компьютера и введя "ADO" в окно поиска информации в справке.

Файл связи вы можете создать для любой имеющейся у вас базы данных. В предыдущих разделах мы работали с базой данных Paradox. СУБД Paradox, конечно, слабая и во многом устаревшая система. Вы можете создать аналогичные таблицы данных, например, в Microsoft Access — СУБД, имеющейся на любом компьютере, работающем под Windows. И можете создать для такой базы данных файл связи.

Если вы, следуя указаниям справки, создали для вашей базы данных файл связи *.udl*, полезно перенести его в папку *.../Program Files/ Common Files/ System/ Ole DB/ Data Links*, в которой обычно расположены подобные файлы. Это облегчит вам в дальнейшем подключение наборов данных.

Теперь вы можете использовать компоненты наборов данных со страницы библиотеки ADO. В частности, вы найдете на этой странице компонент **ADOTable**, являющийся аналогом рассмотренного в данной главе компонента **Table**. Перенесите его на форму. Соединение с базой данных в этом и других компонентах ADO обеспечивается свойством **ConnectionString**. Щелкните в окне Инспектора Объектов на кнопке с многоточием около этого свойства. Вы попадете в окно, показанное на рис. 6.20. Верхняя радиокнопка Use Data Link File позволяет использовать файл связи *.udl*, тот самый, который вы создали средствами Windows. Нижняя радиокнопка Use Connection String позволяет сформировать строку соединения в режиме диалога. Если у вас создан файл *.udl*, то эта кнопка вам не нужна.

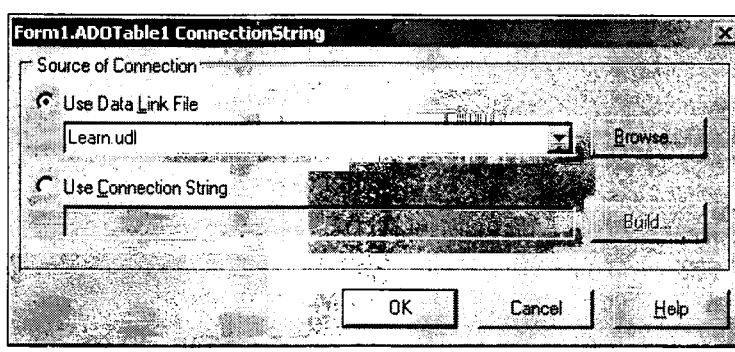


Рис. 6.20
Первое диалоговое окно задания строки соединения

При включенной кнопке Use Data Link File вы можете выбрать файл связи из выпадающего списка (на рис. 6.20 полагается, что вы назвали файл «Learn.udl»). Но в списке имеются только те файлы связи, которые находятся в указанном выше каталоге *Data Links*. Если ваш файл в другом каталоге, найдите его с помощью кнопки Browse.

А дальнейшая работа с **ADOTable** может практически не отличаться от рассмотренной в данной главе работы с **Table**. Так же в свойстве **TableName** задается

имя таблицы, так же подключаются к **ADOTable** источник данных **DataSource** и компоненты отображения данных, так же обеспечивается индексация, фильтрация данных, работа со связанными таблицами. Правда, возможности компонентов, работающих по технологии ADO, во многом шире, чем у компонентов, использующих BDE. Но эти расширенные возможности вам придется, при желании, изучить самостоятельно по встроенным справкам Delphi, по справке [3] или по дополнительной литературе.

6.13 Проверьте себя

6.13.1 Вопросы для самопроверки

1. Зачем при работе с базами данных используются индексы?
2. Как упорядочиваются записи, если индекс содержит несколько полей?
3. Как создать приложение, использующее две таблицы — главную и вспомогательную?
4. Как функционируют многоуровневые распределенные приложения, работающие с данными?
5. Как можно создать вычисляемое поле набора данных?
6. Как организуется фильтрация данных?
7. Зачем можно использовать событие **BeforePost**?
8. Зачем используется кэширование? Какие методы используются в наборе данных **Table** для отмены результатов редактирования, и какие для занесения изменений в базу данных?
9. Как можно осуществить быстрый поиск записи в таблице данных?

6.13.2 Задачи

1. Создайте базу данных с таблицей телефонов ваших друзей и знакомых. Предусмотрите в ней поле, содержащее категорию телефона («Институт», «Дом», «Деловые» и т.п.). Создайте приложение, работающее с этой таблицей в режиме кэширования.
2. Добавьте в приложение, созданное в пункте 1, возможность выбора упорядочивания по алфавиту, по номеру телефона, по категориям.
3. Добавьте в приложение, созданное в пункте 2, окно быстрого поиска записи.
4. Добавьте в базу данных, созданную в пункте 1, таблицу категорий. Создайте приложение, работающее с этой таблицей как головной, и таблицей телефонов как вспомогательной.

Глава 7

Язык SQL и работа с базами данных в сети

В этой главе:

- *вы освоите основные операторы SQL — стандартного языка общения с базами данных*
- *научитесь создавать приложения, использующие запросы SQL*
- *узнаете об основных проблемах многопользовательского доступа к данным*
- *создадите клиентские приложения, реализующие портфельные наборы данных*
- *построите сервер с удаленным модулем данных и научитесь создавать приложения, работающие с ним*

7.1 Основы языка SQL и его использование в приложениях

7.1.1 Общие сведения

Чаще всего общение с базой данных реализуется с помощью языка SQL. В предыдущем рассмотрении о нем не говорилось просто потому, что операторы этого языка выполнялись «за кадром», т.е. незримо для пользователя. Но наибольшую гибкость общения обеспечивает явное использование этого языка. Так что стоит изучить хотя бы основные его операторы.

Язык SQL (Structured Query Language — язык структурированных запросов) был создан в конце 70-ых годов и получил через некоторое время широкое распространение. Он позволяет формировать весьма сложные запросы к базам данных. *Запрос* — это вопрос к базе данных, возвращающий запись или множество записей, удовлетворяющих вопросу. Могут быть также запросы, модифицирующие базу данных: вставляющие, удаляющие, редактирующие записи, индексы, таблицы.

Общие правила синтаксиса SQL очень просты. Язык не чувствителен к регистру, так что, например, рассмотренный ниже оператор **Select** можно писать и **SELECT**, и **Select**, и **select**. Если используется программа из нескольких операторов SQL, то в конце каждого оператора ставится точка с запятой ";". Впрочем, если вы используете всего один оператор, то точка с запятой в конце не обязательна. Комментарий может записываться в стиле C: /*<комментарий>*/, а в некоторых системах и в стиле Pascal: {<комментарий>}. Вот, собственно, и все правила.

При знакомстве с языком SQL полезно сразу пробовать записывать его операторы в различных вариантах, чтобы посмотреть результаты. Проще всего это сделать

с помощью компонента **Query**. В дальнейшем мы его рассмотрим детально. А пока будем использовать просто как инструмент тестирования SQL.

Откройте новое приложение Delphi, перенесите на форму компонент **Query** со страницы библиотеки BDE в Delphi 7–6 или Data Access в более ранних версиях. Установите свойство **DatabaseName** этого компонента равным **Learn** — псевдониму базы данных, с которой вы общались на протяжении всей гл. 6.

Кроме компонента **Query** поместите на форму компонент **DataSource** и в его свойстве **DataSet** задайте **Query1**. Поместите также на форму компонент **DBGrid** и в его свойстве **DataSource** задайте **DataSource1**.

Ваше тестовое приложение для экспериментов с языком SQL готово. Операторы SQL вы можете писать в свойстве **SQL** компонента **Query1**, а чтобы увидеть результаты выполнения написанного оператора, вам надо будет устанавливать значение свойства **Active** компонента **Query1** в **true**. Это надо будет делать после записи каждого нового оператора.

7.1.2 Оператор выбора Select

7.1.2.1 Отбор записей из таблицы

В этом разделе мы познакомимся с наиболее часто используемым оператором SQL — оператором выбора **Select**. Этот оператор возвращает одно или множество значений, которые могут представлять собой значения указанных полей записей, удовлетворяющих указанному условию и упорядоченных по заданному критерию.

Одна из форм этого оператора имеет синтаксис:

```
Select <список имен полей> FROM <таблица>
      WHERE <условие отбора> Order by <список имен полей>;
```

Элементы оператора **WHERE** и **ORDER BY** не являются обязательными. Элемент **WHERE** определяет условие отбора записей: отбираются только те, в которых условие выполняется. Элемент **ORDER BY** определяет упорядочивание возвращаемых записей.

<таблица> — это та таблица базы данных, из которой осуществляется отбор, например, Pers.

Начнем подробное рассмотрение данного оператора со списка полей после ключевого слова **Select**, содержащего имена тех полей таблицы, которые будут возвращены. Имена разделяются запятыми. Например, оператор

```
SELECT Fam, Gr, Sex, Mark FROM Pers
```

указывает, что следует вернуть поля **Fam**, **Gr**, **Sex** и **Mark** из таблицы Pers. Запишите его в свойстве **SQL** компонента **Query1**, установите значение свойства **Active** компонента **Query1** в **true** и посмотрите результаты. Если указать вместо списка полей символ "*" — это будет означать, что требуется вернуть все поля. Например, оператор

```
SELECT * FROM Pers
```

означает выбор всех полей.

В списке могут быть не только сами поля, но и любые выражения от них с арифметическими операциями +, –, *, /. После выражения может записываться псевдоним выражения в форме: **AS <псевдоним>**. В качестве псевдонима может фигурировать любой идентификатор, на который потом можно будет при необходимости

ссылаться. Указанный псевдоним будет при отображении результатов фигурировать в заголовке таблицы.

Приведем пример использования выражения:

```
SELECT Fam, (Mark * 2) AS MARK_10 FROM Pers
```

Этот оператор создает поле **MARK_10**, в котором оценка переводится в десятибалльную, принятую в некоторых странах, по формуле (Mark * 2).

При работе с некоторыми типами баз данных (в частности, с базой данных Paradox) вы можете писать псевдоним по-русски. Например, вы можете изменить предыдущий оператор следующим образом:

```
SELECT Fam AS Фамилия, Mark AS Оценка,
       (Mark * 2) AS Десятибальная FROM Pers
```

В этом случае вы увидите, что в заголовках таблицы появились осмысленные русские наименования:

Фамилия	Оценка	Десятибальная
Иванов	5	10
Петров	2	4
...	...	...

Если вы не вводите псевдоним для выражения, то в заголовке соответствующего столбца будет записано само выражение. Например, если вы напишете оператор

```
SELECT Fam AS Фамилия, Mark AS Оценка, (Mark * 2) FROM Pers
```

то в приведенной выше таблице результатов вместо заголовка "Десятибальная" будет написано "Mark * 2".

Синтаксис ряда систем не требует для введения псевдонима ключевого слова **As**. Например, попробуйте написать приведенный ранее оператор в виде

```
SELECT Fam Фамилия, Mark Оценка, (Mark * 2) Десятибалльная
       FROM Pers
```

и вы получите тот же самый результат, что и при использовании **As**.

Надо иметь в виду, что с русскими текстами в операторах SQL могут возникать неприятности. Некоторые символы могут восприниматься неверно и приводить якобы к синтаксическим ошибкам. Например, часто неверно воспринимается символ "ч". А при работе с некоторыми типами баз данных использование русских псевдонимов вообще невозможно. Впрочем, позднее, при рассмотрении компонента **Query** вы увидите, что эффекта получения русских заголовков можно добиться и другими способами, не вводя русских обозначений в оператор **Select**.

Теперь рассмотрим форму представления условия отбора, задаваемого после ключевого слова **WHERE**. Это условие определяет критерий, по которому отбираются записи. Оператор **Select** отбирает только те записи, в которых заданное условие истинно. Условие может включать имена полей (кроме вычисляемых полей), константы, логические выражения, содержащие арифметические операции, логические операции **and**, **or**, **not** и операции отношения:

=	равно
>	больше
>=	больше или равно
<	меньше
<=	меньше или равно
!= или <>	не равно
Like	наличие заданной последовательности символов
between ... and	диапазон значений
in	соответствие элементу множества

Первые шесть операций очевидны. Например, оператор

```
SELECT Fam, Mark FROM Pers WHERE Sex='ж' and Mark > 3
```

отберет записи, относящиеся к девушкам, имеющим оценку 4 или 5.

Операция **Like** имеет синтаксис:

```
<поле> LIKE '<последовательность символов>'
```

Эта операция применима к полям типа строк и возвращает **true**, если в строке встретился фрагмент, заданный в операции как **<последовательность символов>**. Заданным символам может предшествовать и их может завершать символ процента "%", который означает — любое количество любых символов. Если символ процента не указан, то заданная последовательность символов должна соответствовать только целому слову. Например, оператор

```
Select Gr, Fam FROM Pers Where Gr Like 'A1%'
```

отберет все записи, в которых название группы начинается с "A1". Если первый символ обозначает факультет, а второй символ — номер семестра, то тем самым будет отображен список первокурсников факультета "А". А оператор

```
Select Gr, Fam FROM Pers Where Gr Like '%1-%'
```

отберет записи первокурсников всех факультетов.

Операция **between ... and** имеет синтаксис:

```
<поле> between <значение> and <значение>
```

и задает для указанного поля диапазон отбираемых значений. Например, оператор

```
Select Fam, Mark FROM Pers Where Mark between 4 AND 5
```

отберет записи студентов, сдавших экзамен на 4 и 5.

Операция **In** имеет синтаксис:

```
<поле> in (<множество>)
```

и отбирает записи, в которых значение указанного поля является одним из элементов указанного множества. Например, оператор

```
SELECT Fam, Gr FROM Pers
       WHERE Gr IN('A1-01','A3-01')
```

отберет записи студентов указанных групп, а оператор

```
SELECT Fam, Mark from Pers where Mark in(5,2)
```

отберет записи отличников и двоечников.

Элемент оператора **Select**, начинающийся с ключевых слов **ORDER BY**, определяет упорядочивание (сортировку) записей. После этих ключевых слов следует список полей, определяющих сортировку. Можно указывать только поля, фигурирующие в списке отобранных (в списке после ключевого слова **SELECT**). Причем эти поля могут быть и вычисляемыми.

Если в списке сортировки указано только одно поле, то сортировка производится по умолчанию в порядке нарастания значений этого поля. Например, оператор

```
Select Fam, Born from Pers ORDER BY Born
```

задает упорядочивание возвращаемых значений по нарастанию даты рождения. Если желательно располагать результаты по убыванию значений, то после имени поля добавляется ключевое слово **DESC**:

```
Select Fam, Born from Pers ORDER BY Born DESC
```

Если в списке после **ORDER BY** перечисляется несколько полей, то первое из них — главное и сортировка проводится прежде всего по значениям этого поля. Записи, имеющие одинаковое значение первого поля упорядочиваются по значениям второго поля и т.д. Например, оператор

```
SELECT Gr, Fam FROM Pers ORDER BY Gr, Fam
```

сортирует записи прежде всего по группам (значениям поля **Gr**), а внутри каждой группы — по алфавиту. Оператор

```
SELECT Gr, Fam, Sex FROM Pers ORDER BY Gr, Sex, Fam
```

сортирует записи по группам, полу и алфавиту.

После ключевого слова **Select** в оператор могут вставляться ключевые слова **DISTINCT** или **ALL**. Первое из них означает, что в результирующий набор данных не включаются повторяющиеся записи. Повторяющимися считаются те записи, в которых совпадают значения полей, перечисленных в списке оператора **Select**. Ключевое слово **ALL** означает включение всех записей. Оно подразумевается по умолчанию, так что вставлять его в оператор не имеет смысла. Приведу пример использования **DISTINCT**. Оператор

```
SELECT DISTINCT Gr FROM Pers
```

выдаст список студенческих групп.

7.1.2.2 Совокупные характеристики

Оператор **Select** позволяет возвращать не только множество значений полей, но и некоторые совокупные (агрегированные) характеристики, подсчитанные по всем или по указанным записям таблицы. Одна из функций, возвращающих такие совокупные характеристики, **count(<условие>)** — количество записей в таблице, удовлетворяющих заданным условиям. Например, оператор

```
SELECT count(*) FROM Pers
```

подсчитает полное количество записей в таблице Pers. А оператор

```
SELECT count(*)FROM Pers WHERE Gr='A1-01'
```

выдаст число записей студентов группы A1-01.

Оператор, использующий ключевое слово **DISTINCT** (уникальный), выдаст число неповторяющихся значений в указанном поле. Например, оператор

```
SELECT count(DISTINCT Gr) FROM Pers
```

вернет число различных групп, упомянутых в поле **Gr** таблицы Pers.

Функции **min(<поле>)**, **max(<поле>)**, **avg(<поле>)**, **sum(<поле>)** возвращают соответственно минимальное, максимальное, среднее и суммарное значения указанного поля. Например, оператор

```
SELECT min(Born), max(Born), avg(Born) FROM Pers
```

вернет минимальное, максимальное и среднее значение даты рождения, а оператор

```
SELECT min(Born), max(Born), avg(Born) FROM Pers WHERE Gr = 'A1-01'
```

выдаст вам аналогичные данные, но относящиеся к возрасту студентов группы A1-01.

При использовании суммарных характеристик надо учитывать, что в списке возвращаемых значений после ключевого слова **SELECT** могут фигурировать или поля (в том числе вычисляемые), или совокупные характеристики, но не могут фигурировать и те, и другие (без указания на группирование данных, о чем будет сказано ниже). Это очевидно, так как оператор может возвращать или множество значений полей записей, или суммарные характеристики по таблице, но не может возвращать эти несовместимые друг с другом данные. Поэтому нельзя, например, записать оператор

```
SELECT Fam, max(Born) FROM Pers
```

в котором мы пытаемся определить фамилию самого молодого студента. Впрочем, эту задачу можно решить с помощью вложенных запросов, которые рассмотрены ниже.

Смешение в одном операторе полей и совокупных характеристик возможно, если использовать группировку записей, задаваемую ключевыми словами **GROUP BY**. После этих ключевых слов перечисляются все поля, входящие в список **SELECT**. В этом случае смысл совокупных характеристик изменяется: они проводят вычисления не по всем записям таблицы, а по тем, которые соответствуют одинаковым значениям указанных полей. Например, оператор

```
SELECT Gr Группа, count(*) Всего_студентов FROM Pers
       GROUP BY Gr
```

вернет таблицу, в которой будет 2 столбца — столбец Группа с названиями групп, и столбец Всего_студентов, в котором будет отображено число студентов в каждой группе.

Результаты, возвращаемые оператором **Select**, можно использовать в другом операторе **Select**. Причем это относится и к операторам, возвращающим совокупные характеристики, и к операторам, возвращающим множество значений. Например, выше нам не удалось узнать фамилию самого молодого студента. Теперь это можно сделать с помощью вложенных запросов:

```
SELECT Fam, Born FROM Pers
       WHERE Born=(SELECT max(Born) FROM Pers)
```

В этой конструкции второй вложенный оператор **SELECT max(Born) FROM Pers** возвращает максимальную дату рождения, которая используется в элементе **WHERE** основного оператора **Select** для поиска студента, чья дата рождения совпадает с максимальной.

7.1.3 Операции с записями

В этом разделе вы уже не сможете проверять операторы SQL с помощью вашего тестового приложения. Позднее будет показано, как можно использовать подобные операторы в приложениях.

Вставка новой записи в таблицу осуществляется оператором **Insert**, который может иметь вид:

```
INSERT INTO <имя таблицы> (<список полей>) VALUES (<список значений>)
```

В списке перечисляются только те поля, значения которых известны. Остальные могут пропускаться. Для пропущенных полей значения берутся по умолчанию (если значения по умолчанию заданы) или поля остаются пустыми.

Например:

```
INSERT INTO Pers (Gr, Fam, Sex) VALUES ('A1-01', 'Иванов', 'м')
```

Другая форма оператора **Insert** использует множество значений, возвращаемых оператором **Select**. Этот оператор может выбирать записи из какой-то другой таблицы и вставлять их в данную. Синтаксис этой формы **Insert**:

```
INSERT INTO <имя таблицы> <оператор Select>
```

Пусть, например, вы хотите создать таблицу Pers2 отличников, аналогичную по структуре таблице Pers, и хотите заполнить ее соответствующими записями из таблицы Pers. Создать дубль существующей таблицы легко с помощью Database Desktop. Для этого достаточно открыть исходную таблицу, выполнить команду Table | Restructure и сохранить ее (кнопка Save As) под новым именем. Далее надо удалить все записи из созданной таблицы (это будет рассмотрено чуть позже). А заполнить таблицу (назовем ее Pers2) записями отличников можно оператором:

```
INSERT INTO Pers2 SELECT * FROM Pers WHERE Mark = 5
```

Приведенную форму оператора **Insert** можно использовать для копирования всех данных одной таблицы в другую, причем эти таблицы могут быть созданы разными СУБД.

Редактирование записей осуществляется оператором **Update**:

```
UPDATE <имя таблицы> SET <список вида <поле>=<выражение>>
                     WHERE <условие>
```

Наличие в этом операторе условия позволяет редактировать не только одну запись, но сразу множество их. Например, если группу А1-01 перевели на другой факультет и она называется теперь Б1-01, то исправление всех записей в таблице можно сделать одним оператором:

```
UPDATE Pers SET Gr = 'Б1-01' WHERE Gr = 'A1-01'
```

Удаление записей осуществляется оператором **Delete**:

```
DELETE FROM <имя таблицы> WHERE <условие>
```

Наличие в операторе условия позволяет удалять не только одну, но сразу множество записей. Например, если студенты второго курса (групп, названия которых начинаются с "А3") закончили изучение Delphi, то удалить из таблицы все записи студентов этих групп можно оператором:

```
DELETE FROM Pers WHERE Gr = 'A3*'
```

А оператор без условия:

```
DELETE FROM Pers
```

удалит из таблицы все записи.

7.2 Компонент Query

7.2.1 Table или Query — что лучше?

Мы рассмотрели основные операторы SQL. Теперь посмотрим, как этот язык можно использовать в приложениях. Для этого существует компонент набора данных класса **TQuery**. Он имеет большинство свойств и методов, совпадающих с **Table**, и компонент **Query** может во многих случаях включаться в приложения вместо **Table**. Дополнительные преимущества **Query** — возможность формировать запросы на языке SQL.

Поскольку в ряде случаев компоненты **Table** и **Query** в приложениях взаимозаменяемы, то возникает естественный вопрос — какой же из них использовать?

При работе с локальными базами данных чаще используется **Table**. С его помощью проще не только просматривать таблицу базы данных, но и модифицировать записи, удалять их, вставлять новые записи. Но при работе с серверными базами данных компонент **Table** становится мало эффективным. В этом случае он создает на компьютере пользователя временную копию серверной базы данных и работает с этой копией. Естественно, что подобная процедура требует больших ресурсов и существенно загружает сеть.

Этот недостаток отсутствует в компоненте **Query**. Если запрос SQL сводится к просмотру таблицы (запрос **Select**), то результат этого запроса (а не сама исходная таблица) помещается во временном файле на компьютере пользователя. Правда, в отличие от набора данных, создаваемого **Table**, это таблица только для чтения и не допускает каких-то изменений. Впрочем, это ограничение можно обойти, и в дальнейшем будет показано, как это можно делать. Если же запрос SQL связан с какими-то изменениями содержания таблицы, то никаких временных таблиц не создается. BDE передает запрос на сервер, там он обрабатывается, и в приложение возвращается информация о том, успешно ли завершена соответствующая операция. Благодаря такой организации работы эффективность **Query** при работе в сети становится много выше, чем эффективность **Table**. К тому же язык SQL позволяет формулировать сложные запросы, которые не всегда можно реализовать в **Table**.

С другой стороны, при работе с локальными базами данных эффективность **Query** заметно ниже эффективности **Table**. Замедление вычислений получается весьма ощутимым.

Исходя из этого краткого обзора возможностей **Table** и **Query**, можно заключить, что в серверных приложениях обычно целесообразнее использовать компонент **Query**, а при работе с локальными базами данных — компонент **Table**.

7.2.2 Просмотр данных с помощью Query

Чтобы ознакомиться с **Query**, откройте в Delphi новое приложение и поместите на форму компоненты **Query**, **DataSource**, **DBGrid**. В свойстве **DataSet** компонента **DataSource1** задайте **Query1**, а в свойстве **DataSource** компонента **DBGrid1**

задайте **DataSource1**. Таким образом, вы создали обычную цепочку: набор данных (**Query1**), источник данных (**DataSource1**), компонент визуализации и управления данными (**DBGrid1**). А теперь займемся интересующим нас компонентом **Query**.

В свойстве **DatabaseName** компонента **Query** надо задать, как это делается и для компонента **Table**, базу данных, с которой будет осуществляться связь. База данных задается выбором из выпадающего списка псевдонимов, или указанием полного пути к каталогу или файлу (в зависимости от используемой СУБД).

Предупреждение

Не устанавливайте свойство DataSource — как вы увидите позднее, это свойство имеет отношение к приложениям с несколькими связанными таблицами и в других случаях не устанавливается.

Основное свойство компонента **Query** — **SQL**, имеющее тип **TString**. Это список строк, содержащих запросы SQL. В процессе проектирования приложения обычно необходимо, как будет показано ниже, сформировать в этом свойстве некоторый предварительный запрос SQL, который показал бы, с какой таблицей или таблицами будет проводиться работа. Но далее во время выполнения приложения свойство **SQL** может формироваться программно методами, обычными для класса **TString**: **Clear** — очистка, **Add** — добавление строки и т.д.

Свойства **TableName**, которое было в компоненте **Table**, в **Query** нет, т.к. таблица, с которой ведется работа, будет указываться в запросах SQL. Поэтому, прежде всего, надо занести в свойство **SQL** запрос, содержащий имя таблицы, с которой вы хотите работать.

Предупреждение

Прежде, чем начинать детальную настройку компонента Query, надо сформировать в его свойстве SQL запрос, в котором указывается таблица. Пока такой запрос в SQL отсутствует, дальнейшая настройка Query невозможна. Запрос, заносимый в SQL в начале проектирования, носит чисто служебный характер. В дальнейшем вы можете его программно заменить на любой другой запрос.

Запрос, заносимый в **SQL** в начале проектирования, может иметь, например, следующий вид:

```
Select * from pers
```

После этого система поймет, с какой таблицей будет проводиться работа, и можно будет настроить поля в **Query**. Если работа будет проводиться с несколькими таблицами, вы можете все их указать в запросе. Например:

```
Select * from pers, gro
```

После того как соответствующий запрос написан, можете установить свойство **Active** компонента **Query** в **true**. Если все выполнено правильно, то вы увидите в компоненте **DBGrid1** информацию из запрошенных таблиц.

Как и в компоненте **Table**, подобное соединение с базой данных во время проектирования можно делать только при изучении компонента **Query** и в процессе от-

ладки приложения. А в законченном приложении для открытия и закрытия соединения с базой данных надо использовать соответствующие методы.

Метод **Close** закрывает соединение с базой данных, переводя свойство **Active** в **false**. Этот метод надо выполнять перед изменением каких-то свойств, влияющих на выполнение запроса или на отображение данных. Например, при программном изменении запроса в свойстве **SQL** надо сначала методом **Close** закрыть соединение, связанное с прежним запросом, а потом уже формировать новый запрос.

Метод **Open** открывает соединение с базой данных и выполняет запрос, содержащийся в свойстве SQL. Но этот метод применим только в том случае, если запрос сводится к оператору **Select**. Если же запрос содержит другой оператор, например, **Update**, то при выполнении **Open** будет генерироваться исключение **EDatabaseError**. Для осуществления любого другого запроса, кроме **Select**, используется метод **ExecSQL**.

Наличие двух методов выполнения запроса — **Open** и **ExecSQL** ставит вопрос: «Как же поступить, если запрос сформирован пользователем или программой и его характер неизвестен?». Неизвестный запрос, сформированный в некоторой переменной **ssql** типа строки, можно выполнить с помощью следующего кода:

```
try
 Query1.Close;
 Query1.SQL.Clear;
 Query1.SQL.Add(ssql);
 Query1.Open;
 ...
except
 on EDatabaseError do Query1.ExecSQL;
end;
```

В блоке **try** (см. разд. 2.4.8) осуществляется попытка выполнить запрос с помощью метода **Open**. А если эта попытка завершается генерацией исключения, т.е. если запрос состоит из оператора, отличного от **Select**, то генерируется исключение **EDatabaseError**, которое перехватывается в разделе **except**, и выполняется метод **ExecSQL**.

Для управления отображением данных в **Query**, как и в компоненте **Table**, имеется уже известный вам Редактор Полей (Field Editor). Вызвать его можно или двойным щелчком на **Query**, или щелчком правой кнопки мыши на **Query** и выбором Fields Editor из всплывающего меню. С Редактором Полей вы уже знакомы (см. разд. 6.6). В нем вы можете добавить имена получаемых полей (щелчок правой кнопкой мыши и выбор раздела меню Add), задать заголовки полей, отличающиеся от их имен, сделать какие-то поля невидимыми (**Visible**), не редактируемыми (**ReadOnly**), задать формат отображения чисел, создать вычисляемые поля и многое другое.

Давайте попробуем использовать **Query** в приложении, подобном тому, которое создавали с помощью **Table**. Например, в приложении, рассмотренном в разд. 6.7 и приведенном на рис. 6.13. Проще всего это сделать так. Откройте прежний проект. Выполните команду File | Save Project As и сохраните проект в новой папке или под новым именем. Затем выполните команду File | Save As и сохраните модуль в новой папке или с новым именем. Удалите с формы компонент **Table1** и поместите вместо него компонент **Query**. В компоненте **DataSource1** сошлитесь в свойстве **DataSet** на **Query1**.

Теперь можно заняться настройкой компонента **Query**. В его свойстве **DataBaseName** сошлитесь на вашу базу данных, а в свойство **SQL** внесите запрос:

```
Select * from pers
```

После этого можете попробовать установить в **true** свойство **Active**. Если все выполнено правильно, то вы увидите в компонентах **DBGrid1**, **DBImage1** и **DBRichEdit1** информацию из базы данных.

С помощью Редактора Полей русифицируйте заголовки полей, введите вычисляемое поле возраста, словом, настройте поля компонента **Query** так же, как вы делали это ранее для **Table**. Теперь в коде приложения с помощью контекстной замены (команда Search | Replace) замените везде, кроме предложения **uses**, "Table" на "Query". И закомментируйте код в обработчике **RadioGroup1Click**, так как упорядочивание данных в **Query** отличается от упорядочивания в **Table**.

Выполните ваше приложение. Если вы сделали все, как описано выше, то оно будет работать. Но если вы попробуете в нем не только просматривать, но и редактировать данные, вас постигнет неудача. Да и в навигаторе все кнопки, относящиеся к редактированию, будут недоступными. Дело в том, что компонент **Query** с запросом **Select** формирует таблицу только для чтения. Впрочем, в таком простом приложении, как наше, это можно исправить. Достаточно установить в компоненте **Query** свойство **RequestLive** в **true**. Это позволяет возвращать как результат запроса изменяемый, «живой» набор данных, вместо таблицы только для чтения. Точнее, установка **RequestLive** в **true** делает попытку вернуть «живой» набор данных. Успешной эта попытка будет только при соблюдении ряда условий, в частности:

- набор данных формируется обращением только к одной таблице
- набор данных не упорядочен (в запросе не используется **Order by**)
- в наборе данных не используются совокупные характеристики типа **Sum**, **Count** и др.
- набор данных не кэшируется (свойство **CachedUpdates** равно **false**)

В нашем примере все эти условия соблюдаются. Так что можете установить **RequestLive** в **true**, сделать доступными все кнопки навигатора и выполнить приложение. При этом вы убедитесь, что можете редактировать данные, удалять записи, вставлять новые записи.

Теперь давайте реализуем код обработчика щелчка на группе радиокнопок **RadioGroup1**, обеспечивающий упорядочивание данных по выбору пользователя. В **Query** нет свойства **IndexName**, с помощью которого мы индексировали данные в **Table**. Такое свойство не требуется, так как упорядочивание данных можно проводить с помощью элемента **Order by** оператора **Select**.

Занесите в свойство **Items** компонента **RadioGroup1** строки:

```
без упорядочивания
по группам
по алфавиту
по дате рождения
по оценкам и группам
```

А в обработчик щелчка на группе радиокнопок поместите код:

```
Query1.Close;
Query1.SQL.Clear;
case RadioGroup1.ItemIndex of
```

```
 0: Query1.SQL.Add('Select * from Pers');
 1: Query1.SQL.Add('Select * from Pers Order By Gr, Fam');
 2: Query1.SQL.Add('Select * from Pers Order By Fam');
 3: Query1.SQL.Add('Select * from Pers Order By Born, Fam');
 4: Query1.SQL.Add(
          'Select * from Pers Order By Mark DESC, Gr, Fam');
end;
Query1.Open;
```

Первый оператор этого кода закрывает методом **Close** соединение с базой данных. Второй оператор очищает свойство **SQL**, удаляя из него прежний запрос. Затем в зависимости от выбора пользователя в этом свойстве формируется новый запрос, обеспечивающий требуемое упорядочивание данных. Последний оператор соединяется с базой данных, передавая в нее новый запрос.

Выполнив приложение, вы увидите, что редактирование данных возможно только в случае, если данные не упорядочены. В остальных случаях редактирование недопустимо. Поэтому, чтобы не смущать пользователя возможностью сменить фотографию тогда, когда это недопустимо, полезно добавить в приведенный выше код оператор:

```
Button1.Enabled := (RadioGroup1.ItemIndex = 0);
```

Он сделает недоступной кнопку Изменить в ситуациях, когда редактирование данных невозможно. А более кардинальное решение проблемы редактирования будет рассмотрено в следующем разделе.

На примере рассмотренного приложения вы могли увидеть, что замена набора данных **Table** на **Query** осуществляется, хотя и не автоматически, но достаточно просто. Большинство методов у этих компонентов совпадает. Большинство свойств, в частности, обеспечивающих фильтрацию данных, тоже одинаково. Но богатство языка SQL позволяет строить сложные запросы к базам данных и обеспечивает большие возможности, чем компонент **Table**.

7.2.3 Редактирование данных и кэширование изменений

Приложение, созданное вами в разд. 7.2.2, имеет два существенных недостатка. Прежде всего — невозможность редактирования данных при упорядочивании записей. Другой недостаток связан с тем, что результаты редактирования немедленно передаются в базу данных. А нередко удобнее было бы кэшировать результаты редактирования (хранить их временно в памяти), и только после того, как все изменения и проверки сделаны, переслать их в базу данных или, по решению пользователя, отменить все сделанные исправления.

Режим кэширования осуществляется так же, как и для компонента **Table** (см. разд. 6.11.4): устанавливается в **true** свойство **CachedUpdates** компонента **Query**, и применяются методы **ApplyUpdates** для записи изменений в базу данных, метод **CancelUpdates** для отмены изменений и метод **CommitUpdates** для очистки буфера кэша. Так что задайте соответствующее значение свойству **CachedUpdates**, и добавьте в приложение две кнопки. В первой задайте надпись Фиксация и в обработчик щелчка на ней вставьте операторы:

```
Query1.ApplyUpdates;
Query1.CommitUpdates;
```

А в другой задайте надпись Отменить и в обработчик щелчка вставьте оператор:

```
Query1.CancelUpdates;
```

Но при **CachedUpdates = true** перестанет действовать, как было сказано в разд. 7.2.2, свойство **RequestLive**. Так что для осуществления возможности редактирования надо еще перенести на форму компонент **UpdateSQL**. Этот компонент, расположенный на странице библиотеки BDE (в версиях, младше Delphi 6, на странице Data Access), позволяет модифицировать наборы данных, открытые в режиме только для чтения. Таким образом, он решит все стоящие перед нами задачи. Установите в компоненте **Query1** свойство **UpdateObject** равным имени введенного компонента **UpdateSQL1**. Это имя вы можете выбрать из выпадающего списка в свойстве **UpdateObject**.

Теперь можно проводить настройку компонента **UpdateSQL**. Но сначала давайте рассмотрим проблемы, возникающие при одновременной работе нескольких пользователей с одной и той же базой данных.

До сих пор мы предполагали, что все приложения, созданные в данной главе и в гл. 6, работают с локальной базой данных. Впрочем, без каких-либо проблем на одном компьютере может выполняться одновременно несколько приложений, работающих с одной и той же базой данных. В этом случае реализуется многопользовательский режим доступа к данным в пределах одного компьютера. Однако по тому же принципу можно создавать приложения, работающие в сети. Возможности полноценной работы с удаленными данными будут рассмотрены в разд. 7.4. Но и все разработанные нами до сих пор приложения могут работать в сети. Только в этом случае надо несколько иначе соединяться с базой данных.

Если, как и раньше, заносить в свойство **DatabaseName** компонентов **Table** и **Query** псевдонимы баз данных, то возможен ряд недоразумений. Пусть, например, вы запускаете приложение, расположенное на сервере. И пусть в нем имеется ссылка на псевдоним базы данных, расположенной также на сервере. Тогда, если на компьютере пользователя зарегистрировано такое же имя псевдонима, но относящиеся к базе данных на его компьютере, то пользователь будет работать с этой базой данных, а не с базой данных, расположенной на сервере. Если же на компьютере клиента нет псевдонима, используемого в приложении, запускаемом на сервере, то пользователь получит сообщение о ненайденном псевдониме, и соединения с базой данных не будет.

Чтобы не возникало подобной путаницы, вы можете при создании псевдонима на своем компьютере (см. разд. 6.3) указать путь (Path) с включением в него имени серверного компьютера. Например:

```
\\mycomp2\c\database\learn
```

В этом примере подразумевается, что имя сервера "mycomp2", а база данных расположена на нем в каталоге *c:\database\learn*. Как видите, имя компьютера предваряется двумя обратными слэшами и после имени диска отсутствует двоеточие. Впрочем, форма записи пути на удаленном компьютере может зависеть от типа используемой сети.

Если вы таким образом создали на своем компьютере псевдоним, и на компьютере сервера имеется псевдоним с тем же именем, то запустив со своего компьютера приложение, расположенное на вашем компьютере или на компьютере сервера, вы будете работать именно с удаленной базой данных.

При работе в сети, когда к одной и той же базе данных одновременно может обращаться несколько пользователей, может возникать множество проблем. Например, вы прочитали какую-то запись из таблицы и редактируете ее. Но, пока вы редактируете, другой пользователь может изменить эту запись или вообще удалить ее. Что тогда будет при попытке сохранить ваши изменения в этой уже отредактированной или вообще удаленной записи?

Эта и множество других аналогичных проблем разрешаются с помощью механизма транзакций (transaction). *Транзакция* — это групповая операция, связанная с передачей сообщения. С точки зрения приложения транзакция — это группа логически связанных операторов SQL, причем только успешное выполнение всех операторов должно приводить к изменению данных сервером. Пока все операторы транзакции не выполнены, сохраняется возможность отменить их и не фиксировать результаты в базе данных. Подобный механизм необходим для надежной работы с удаленным сервером и в многопользовательском режиме. В тех приложениях, которые вы создавали до сих пор, транзакции тоже использовались, но неявно. Каждый вызов метода **Post**, каждое изменение таблицы автоматически начинало и заканчивало транзакцию. Такие небольшие транзакции непродуктивно нагружают сеть. Режим кэширования более эффективен — множество исправлений пересылаются в базу данных одной транзакцией при выполнении метода **ApplyUpdates**. Но и описанные выше проблемы многопользовательского доступа возникают при этом чаще. Так как пользователь может достаточно долго редактировать записи, прежде, чем решит переслать их в базу данных, возрастает опасность, что за это время кто-то другой изменит или удалит некоторые из них. Значит, должен быть механизм, позволяющий определить, как найти в базе данных ту запись, которую данный пользователь хочет занести, но которая, может быть, за это время изменена другим пользователем.

Этот механизм реализуется свойствами компонента **UpdateSQL1**: **DeleteSQL**, **InsertSQL** и **ModifySQL**. Они содержат соответственно запросы, которые должны выполняться при удалении, вставке или модификации записи. Эти запросы можно записать обычным образом, вручную. А можно воспользоваться редактором UpdateSQL, который вызывается двойным щелчком на **UpdateSQL** или при выделенном **UpdateSQL** вызывается из всплывающего меню. Окно этого редактора имеет вид, представленный на рис. 7.1. Первая страница Options Редактора UpdateSQL, представленная на рисунке, содержит два окна Key Fields и Update Fields.

В левом окне Key Fields надо выделить поля, по которым программа будет распознавать модифицируемую или удаляемую запись. Можно выделить все поля, что гарантирует максимально возможную надежность распознавания, но обычно следует ограничиться выбором нескольких наиболее важных полей. При работах с очень большими таблицами это сэкономит время выполнения запроса. Ну а в нашем случае выделение всех полей было бы просто ошибочным. Во-первых, такие поля, как **Charact** и **Photo** вообще не могут участвовать в идентификации записи, так как, естественно, нет возможности сравнивать фотографии и тексты характеристик. Во-вторых, если во время редактирования записи одним пользователем другой пользователь изменил что-то в этой же записи (например, поменял значение поля **Mark** — оценки), то приложение первого пользователя не сможет найти запись в базе данных. Так что разумно в левом окне оставить только одно выделенное поле — **Num**. Это ключевое поле, которое однозначно идентифицирует запись и которое ни один пользователь не может изменить.

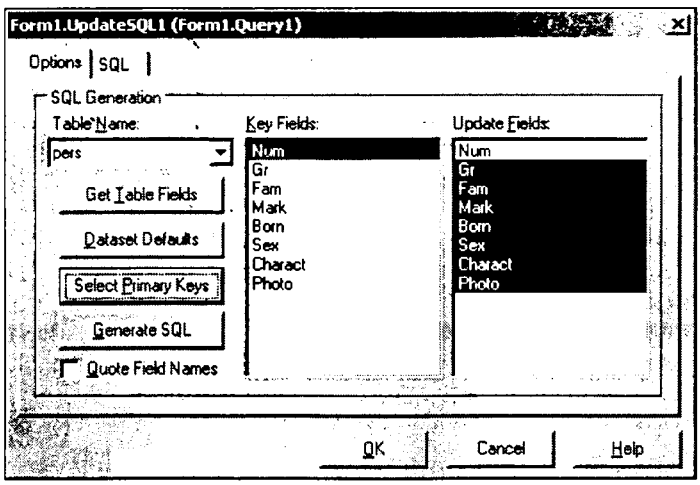


Рис. 7.1
Окно редактора UpdateSQL

В правом окне Update Fields надо выделить поля, значения которых будут задаваться при модификации или вставке записи. В нашем примере можно выделить все поля, кроме поля **Num**. Значение этого поля определяется базой данных и не может изменяться пользователем.

После того как вы выделили требуемые поля, нажмите кнопку Generate SQL. Вам будет показана вторая страница SQL редактора **UpdateSQL**, на которой вы можете просмотреть сгенерированные запросы для модификации (Modify), вставки (Insert) и удаления (Delete) записи. После щелчка на OK эти запросы перенесутся в свойства **DeleteSQL**, **InsertSQL** и **ModifySQL**, где вы их можете дополнительно отредактировать.

При выделении полей, показанном на рис. 7.1, запрос **ModifySQL** будет иметь вид:

```
update pers
set
  Gr = :Gr,
  Fam = :Fam,
  Mark = :Mark,
  Born = :Born,
  Sex = :Sex,
  Charact = :Charact,
  Photo = :Photo
where
  Num = :OLD_Num
```

В нем в разделе **Set** указана установка всех полей, кроме **Num**, в значения, имеющиеся в текущей записи приложения. В этот раздел включаются те поля, которые вы выделили в окне Update Fields редактора UpdateSQL. В разделе **Where** содержатся условия, по которым идентифицируется модифицируемая запись. В этих условиях используются параметры с именами, тождественными именам полей, но с префиксом **OLD_**. Эти параметры — прежние значения соответствующих полей, которые были получены компонентом до модификации записи. В условия **Where** включены те поля, которые вы выделили в окне Key Fields редактора **UpdateSQL**. В нашем случае идентификация проводится только по полю **Num**.

Запросы в свойствах **DeleteSQL** и **InsertSQL** строятся по такому же принципу.

Приложение, практически, завершено. Запустите его, и увидите, что оно стало работать так же, как работало ранее с компонентом **Table**. Можно редактировать записи, удалять, вставлять новые записи, управлять кэшированием. Впрочем, нетрудно заметить один недостаток нашего приложения. Пользователь может работать с таблицей, исправлять записи, вносить новые данные. Но если он перед закрытием приложения забудет щелкнуть на кнопке Фиксация, то весь его труд пропадет. Так что полезно ввести механизм, обеспечивающий в этом случае выдачу запроса пользователю о сохранении результатов редактирования. Такой механизм был рассмотрен в разд. 6.11.4.

В приложении имеется еще один недостаток. Вы создали приложение, обеспечивающее многопользовательский режим работы с данными. Так что надо бы ввести в него еще одну кнопку — Обновить, щелчок на которой позволит получить текущее состояние базы данных с учетом возможных результатов работы других пользователей. В ряде компонентов наборов данных, в частности, в **Table**, имеется метод **Refresh**, который обновляет данные. Имеется он и в **Query**, но не работает в режиме кэширования. Так что обновить данные можно, закрыв и вновь открыв соединение:

```
Query1.Close;
Query1.Open;
```

Подобные операторы можно написать в обработчике щелчка на кнопке Обновить. Но если в приложении имеются изменения данных, которые пользователь не сохранил щелчком на кнопке Фиксация, то они пропадут. Так что желательно предупредить об этом пользователя и дать ему возможность сохранить результаты редактирования. Это может быть сделано аналогично тому, как реализовывался в разд. 6.11.4 подобный запрос при закрытии приложения. Надеюсь, вы легко справитесь с этой задачей сами.

7.2.4 Работа со связанными таблицами

Для построения приложений, содержащих связанные друг с другом таблицы, используется свойство **DataSource** компонента **Query**. Рассмотрим, как это делается, на том же примере, который использовался в гл. 6 (разд. 6.10) при рассмотрении **Table**. Пусть мы хотим построить приложение, включающее в себя таблицу Gro, содержащую список групп (поле **Gr1**), в качестве головной таблицы, и таблицу студентов Pers, содержащую в поле **Gr** название группы, в которой обучается студент. Мы хотим, чтобы при выборе записи в таблице Gro в таблице Pers отбирались только записи, относящиеся к выбранной группе.

Откройте новое приложение. Перенесите на форму компоненты **Query1**, **DataSource1**, **DBGrid1** и соедините их обычной цепочкой: в **DBGrid1** задайте свойство **DataSource** равным **DataSource1**, а в **DataSource1** задайте свойство **DataSet** равным **Query1**. Компонент **Query1** настройте на таблицу Gro. Для этого установите соответствующее значение свойства **DatabaseName**, а в свойстве **SQL** напишите оператор

```
Select * from Gro
```

Установите свойство **Active** в **true** и убедитесь, что все работает нормально: в **DBGrid1** должны отобразиться данные таблицы Gro.

Создайте другую аналогичную цепочку, перенеся на форму компоненты **Query2**, **DataSource2**, **DBGrid2**, и свяжите ее с таблицей Pers запросом

```
Select * from Pers
```

в компоненте **Query2**. Установите свойство **Active** компонента **Query2** в **true**, и в **DBGrid2** должны отобразиться данные таблицы Pers.

Вы можете запустить приложение и убедиться, что оно работает, но таблицы независимы. Теперь давайте свяжем эти таблицы. Делается это следующим образом. Измените текст запроса в свойстве **SQL** вспомогательного набора данных **Query2** на

```
Select * from Pers where (Gr=:Gr1)
```

В этом запросе вы указываете условие отбора: значение поля **Gr** должно быть равно значению поля **Gr1** другого набора данных. Для указания этого набора данных надо в свойстве **DataSource** компонента **Query2** сослаться на **DataSource1** — источник данных, связанный с таблицей Gro.

Запустите приложение и вы увидите, что при перемещении по первой таблице во второй отображаются только те записи, которые относятся к группе, указанной в текущей записи первой таблицы.

7.3 Клиентские наборы данных

7.3.1 Общие сведения

Рассмотрим теперь клиентские наборы данных. Их особенность заключается в том, что данные хранятся в памяти и могут сохраняться в файле на диске и читаться из этого файла. Таким образом, клиентские наборы данных могут выступать как автономные основанные на файлах наборы данных. Другая немаловажная функция клиентских наборов — создание так называемой «портфельной» базы данных, в которую первоначально записываются данные с сервера или какой-то иной базы данных, а затем вся работа проходит с файлом на компьютере клиента. Пользователь может изменять эти данные, причем при закрытии клиентского набора все изменения запоминаются в файле. А при открытии набора данные из файла записываются в клиентский набор. Таким образом, «портфельная» база данных — альтернатива кэширования. Преимуществом является то, что изменения могут производиться не обязательно в одном сеансе работы, а в нескольких. И когда пользователь решит, что изменения заслуживают пересылки в базу данных, он может занести в нее все изменения, сделанные на протяжении этих сеансов.

Клиентские наборы данных реализуются компонентом **ClientDataSet**, который в Delphi 5 расположен в библиотеке на странице *Midas*, а начиная с Delphi 6 переместился на главную страницу *Data Access*. Этот компонент имеет общего прародителя с компонентами **Table** и **Query**. Поэтому у них много общего. Особенно близок **ClientDataSet** к **Table**. Так что все, что было рассмотрено в гл. 6 по упорядочиванию данных с помощью индексов, фильтрации, созданию вычисляемых полей, работе со связанными таблицами в равной степени относится и к клиентским наборам данных.

7.3.2 Наборы данных, основанные на файлах

Основное свойство клиентского набора данных — **FileName**. Это имя файла (может указываться с путем), в котором хранится база данных. Свойство **FileName** задается, если компонент должен всегда читать и записывать данные одного определенного файла. Если файл, указанный в **FileName**, существует, то при каждом открытии набора данных в него будут загружаться данные из этого файла, а при каждом закрытии набора данных в него будут записываться данные из набора.

Можно также осуществлять чтение данных из файла методом **LoadFromFile**:

```
procedure LoadFromFile(const FileName: string = '');
```

Если параметр **FileName** — пустая строка или вообще не указан, то чтение производится из файла, заданного свойством **FileName**. Таким образом, оператор

```
ClientDataSet1.LoadFromFile;
```

обеспечит чтение данных из того же файла, из которого производится чтение при открытии набора. Отличие в том, что это чтение можно произвести в любой момент, а не только в момент открытия набора. А введя в приложение диалог открытия файла, можно с помощью **LoadFromFile** организовать чтение из любого выбранного пользователем файла таблицы.

Метод **SaveToFile** позволяет в любой момент сохранить данные в файле. Он объявлен следующим образом:

```
procedure SaveToFile(const FileName: string = '';
                     Format: TDataPacketFormat=dfBinary);
```

Параметр **FileName** аналогичен рассмотренному для метода **LoadFromFile**. А параметр **Format** указывает формат файла. Обычно его целесообразно задавать равным **dfXML**.

Аналогом кэширования, позволяющего при необходимости отменить совокупность операций редактирования данных, является свойство **SavePoint**. Оно является целым числом, указывающим текущее состояние набора данных. Если вы хотите в какой-то момент запомнить текущее состояние набора данных, выполните оператор

```
var MySavePoint: integer;
...
MySavePoint := ClientDataSet1.SavePoint;
```

Тогда, если впоследствии вы захотите отменить все результаты редактирования и вернуться к прежнему состоянию набора данных, достаточно будет выполнить оператор

```
ClientDataSet1.SavePoint := MySavePoint;
```

А если вы хотите зафиксировать проведенные операции редактирования, но оставить за собой возможность отменить последующее редактирование, надо выполнить оператор

```
MySavePoint := ClientDataSet1.SavePoint;
```

Упорядочивание записей при их отображении происходит так же, как в **Table**, с помощью индексов. В клиентских наборах данных также имеются свойства **IndexName** и **IndexFieldName**. Но, в отличие от компонента **Table**, в котором возможные значения этих свойств выбираются из числа предопределенных при формировании таблицы, в данном случае этих предопределенных значений нет.

Значение свойства **IndexFieldName** может задаваться как во время проектирования, так и во время выполнения в виде строки, перечисляющей поля с разделяющими их точками с запятой. Например, значение "Gr;Fam" соответствует упорядочиванию по названию группы, а внутри каждой группы — в алфавитной последовательности фамилий. Но иногда, как вы увидите в разд. 7.3.3, требуется сформировать индекс, на который можно было бы ссылаться по имени. Это имя задается в свойстве **IndexName**. Но прежде, чем ссылаться на имя индекса, его надо определить. Это делается с помощью свойства **IndexDefs**. Щелчок на кнопке с многоточием около этого свойства в Инспекторе Объектов вызывает окно формирования коллекций объектов. Нажав в нем кнопку New, вы создадите новый объект индекса и увидите в Инспекторе Объектов его свойства. Свойство **Name** определяет имя индекса, на которое в дальнейшем можно ссылаться. В свойстве **Fields** записывается строка перечисления полей, аналогичная описанной выше для свойства **IndexFieldName**.

После того как вы заполнили свойства **Name** и **Fields** индекса, можете выйти из редактора индексов и посмотреть свойство **IndexName** компонента клиентского набора. Около этого свойства в Инспекторе Объектов появится выпадающий список, из которого вы можете выбрать имя введенного вами индекса.

Фильтрация записей производится с помощью свойств **Filter** и **Filtered** так же, как описано для компонента **Table** в разд. 6.9. Но для клиентских наборов данных существенно расширен перечень операций, которые можно использовать при фильтрации. Посмотрите их во встроенной справке Delphi или в справке [3].

7.3.3 Совокупные характеристики

Отличительной особенностью клиентских наборов данных является возможность расчета совокупных характеристик по всем записям таблицы или по некоторому множеству записей. Под совокупными характеристиками понимается число записей, сумма значений какого-то поля, среднее значение поля и т.п.

Задается поле, содержащее совокупную характеристику, в Редакторе Полей, вызываемом, как и во всех других наборах данных, двойным щелчком на компоненте **ClientDataSet**. Щелкнув правой кнопкой в этом окне и выбрав в контекстном меню раздел New Field, вы откроете окно задания нового поля. Это окно отличается от аналогичного окна компонента **Table** (рис. 6.15) двумя дополнительными радиокнопками: InternalCalc и Aggregate. Кнопка Aggregate позволяет определить поле совокупной характеристики. Для этого поля достаточно задать имя Name, а тип поля автоматически установится равным Aggregate.

После того как вы щелкнете на OK в этом окне, в Редакторе Полей появится введенное поле, а в Инспекторе Объектов вы сможете увидеть свойства объекта этого поля. Основное свойство, задающее значения поля, — **Expression**. В нем вы должны записать выражение, задающее значение поля. Помимо обычных арифметических операций, скобок, имен полей и констант, выражение может содержать следующие функции:

Sum	Сумма значений числового поля или арифметического выражения.
Avg	Среднее значение числового поля, или поля даты и времени, или арифметического выражения.
Count	Число непустых значений указанного поля или выражения. Функция **Count(*)** возвращает общее число записей, независимо от значений полей.

Min	Возвращает минимальное значение числового поля, строкового поля, поля дат и времени или соответствующего выражения.
Max	Возвращает максимальное значение числового поля, строкового поля, поля дат и времени или соответствующего выражения.

Например, выражение

```
Sum(Mark) / Count(Mark)
```

вычислит среднюю оценку студентов, то же, что вычислит и более простое выражение

```
Avg(Mark)
```

В записи выражения нельзя смешивать совокупные характеристики и значения полей, так как это принципиально различные объекты: совокупная характеристика — это одно значение, а поле — это множество значений.

Помимо задания выражения, вычисляющего характеристику, надо установить в **true** свойство **Active** данного поля. В этом случае значение поля будет вычисляться, но только если значение свойства **AggregatesActive** компонента клиентского набора будет установлено в **true**.

Выражения совокупных характеристик могут вычисляться по всем записям таблицы, или по некоторой их совокупности. Если вы хотите вычислять по совокупности записей, в которых какое-то поле имеет определенное значение, то набор данных должен быть индексирован так, чтобы интересующее вас поле входило в этот индекс. Причем, этот индекс должен быть активным, т.е. на него должно ссылаться свойство **IndexName** компонента.

Создание индекса было описано в разд. 7.3.2. А ссылка на индекс из поля совокупной характеристики осуществляется следующим образом. В свойстве поля **IndexName** надо задать индекс, на который опирается вычисление суммарной характеристики. А в свойстве **GroupingLevel** надо задать число первых полей в индексе, совпадение которых выделяет группу записей, по которой вычисляется характеристика. Пусть, например, задан текущий индекс вида "Gr;Mark", индексирующий набор данных по группам, а внутри каждой группы — по оценкам. Тогда, если сослаться на этот индекс в поле совокупной характеристики выражением **Count(*)** и задать **GroupingLevel** = 1, то поле покажет число студентов группы, соответствующей текущей записи. А если задать **GroupingLevel** = 2, то поле покажет число студентов, у которых значения полей группы и оценки совпадают с этими полями в текущей записи.

Значение **GroupingLevel** = 0 соответствует подсчету характеристик по всем записям набора данных, независимо от текущего индекса.

7.3.4 Портфельные наборы данных

Если клиентский набор данных должен представлять собой «портфельную» копию какого-то другого набора данных, для связи с этим другим набором в приложение должен быть введен компонент-провайдер, обеспечивающий связь с этим набором данных. Провайдер может размещаться в том же приложении, в котором находится набор данных, или может быть частью удаленного сервера. В качестве провайдера может выступать компонент **DataSetProvider** со страницы Data Access. В этом случае ссылка на этот провайдер должна содержаться в свойстве **ProviderName** компонента **ClientDataSet**.

Рассмотрим некоторые свойства компонента **DataSetProvider**. Основным свойством является **DataSet** — набор данных, являющийся сервером для этого провайдера. В качестве него могут выступать любые наборы данных: **Table**, клиентские наборы данных и т.п. Провайдер упаковывает послание от этого набора данных в пакет, который может воспринимать клиентский набор данных. Через провайдер осуществляется и обратная передача данных от клиента серверному набору.

Если клиентский набор данных открывается методом **Open** и файла, задаваемого свойством **FileName**, нет или это свойство не задано, то клиентский набор воздействует через провайдер на набор-источник, открывает его, если он был закрыт, и считывает через провайдер данные источника.

Данные содержатся в свойстве только для чтения **Data**. У клиентского набора данных имеется аналогичное свойство **Data**, но его значение можно задавать во время выполнения. Таким образом, если в какой-то момент надо передать данные из серверного набора с помощью провайдера **DataSetProvider1** в клиентский набор **ClientDataSet1**, надо выполнить оператор

```
ClientDataSet1.Data := DataSetProvider1.Data;
```

Так осуществляется передача данных от набора-источника к набору-клиенту. Обратная передача от клиента к источнику осуществляется методом **ApplyUpdates** клиентского набора данных:

```
function ApplyUpdates(MaxErrors: Integer): Integer;
```

Параметр **MaxErrors** определяет максимальное число ошибок передачи, при превышении которого передача прекращается. Если задать **MaxErrors** = −1, количество ошибок не ограничено. Метод возвращает число ошибок, произошедших при передаче.

7.3.5 Тестовый пример

Давайте построим тестовый пример, иллюстрирующий многое из того, о чем говорилось в предыдущих разделах. Этот тестовый пример показан во время выполнения на рис. 7.2.

Верхняя таблица отображает серверный набор данных, созданный из таблицы Pers базы данных **Learn** компонентом **Table**. Эта таблица только для чтения введена в приложение просто для того, чтобы наблюдать состояние базы данных. Нижняя таблица отображает клиентский набор данных, созданный компонентом **ClientDataSet**. Это основанный на файле набор данных, который сначала воспринимает данные из серверного набора, а затем становится самостоятельным. Любые исправления в нем запоминаются в файле и читаются при очередном сеансе работы с приложением. Навигатор, который вы видите на форме, относится к этому клиентскому набору данных.

Кнопка В базу данных пересылает изменения, сделанные в текущем сеансе в клиентском наборе, в серверную базу данных. Кнопка В файл фиксирует изменения в файле. Кнопка Отменить отменяет все исправления, сделанные в клиентском наборе с начала сеанса или с последнего выполнения команды В файл. Кнопка Обновить пересылает текущее состояние таблицы базы данных в клиентский набор.

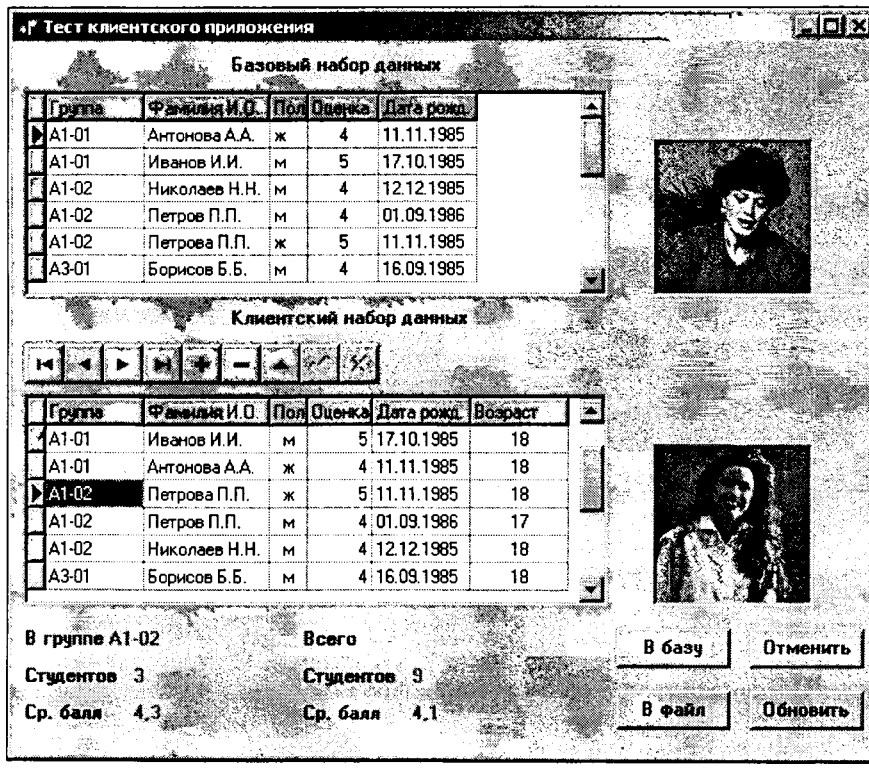


Рис. 7.2
Приложение с клиентским набором данных во время выполнения

Внизу окна расположены метки, демонстрирующие расчет совокупных характеристик. Слева отображается число студентов и средний балл в группе, соответствующей текущей записи клиентского набора данных. А справа отображается общее число студентов и средний балл по всем группам (извиняюсь за малое число студентов, которое вы видите на рис. 7.2, но, сознаюсь, мне было лень набивать ради этого примера большую таблицу).

Рассмотрим реализацию подобного приложения. Его форма показана на рис. 7.3. Последовательность ее формирования следующая.

Начните новое приложение и сразу сохраните его, чтобы далее не путаться с каталогами. Перенесите на форму компоненты **Table1**, **DataSource1** и **DBGrid1** и соедините их обычной цепочкой друг с другом и с таблицей Pers базы данных **Learn**. Можете с помощью Редактора Полей оформить отображение полей в таблице. Впрочем, это не относится к сути приложения, так что делать это не обязательно. Установите в **Table1** свойство **ReadOnly** в **true**, чтобы редактировать базу данных можно было только из клиентского набора. В свойстве **IndexName** задайте индекс **grfio**, чтобы данные были упорядочены по группам и алфавиту. Добавьте на форму компонент **DBImage1** и настройте его на поле **Photo**.

Разместите в нижней части формы компоненты **ClientDataSet1**, **DataSource2** и **DBGrid2**. В **DBGrid2** обычным образом сошлитесь в свойстве **DataSource** на **DataSource2**, а в свойстве **DataSet** компонента **DataSource2** сошлитесь на **ClientDataSet1**. Перенесите на форму **DBNavigator** и свяжите его с **DataSource2**.

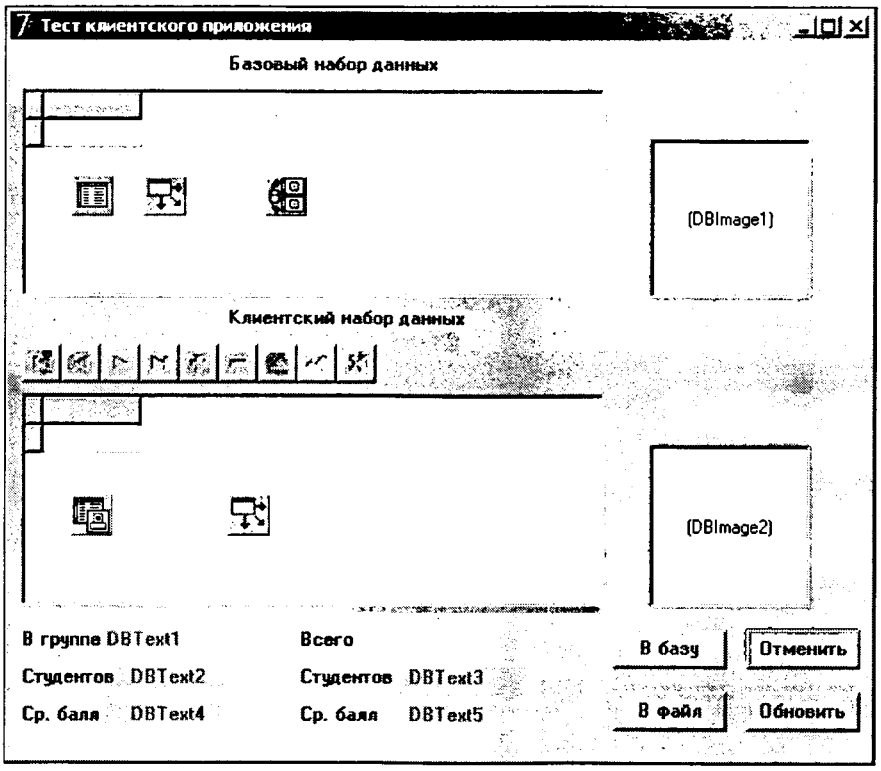


Рис. 7.3
Форма тестового приложения

Поскольку мы хотим создать набор данных, основанный на файле, в свойстве **FileName** компонента **ClientDataSet1** надо бы задать имя файла, желательно, со стандартным расширением *.xml*, в котором должны храниться данные клиентского набора. Задать его можно в Инспекторе Объектов. При этом для выбора каталога можно воспользоваться диалогом, вызываемым кнопкой с многоточием около этого свойства. Но тут надо быть осторожным. Если вы занесете в приложение имя файла с путем, то ваше приложение нельзя будет перенести на другой компьютер с другими каталогами. Даже если вы зададите имя файла без указания каталога, Delphi автоматически добавит путь к имени файла. И вы опять получите непереносимый проект. Поэтому лучше не задавать свойство **FileName** во время проектирования. Сделаем это позднее программными средствами.

Мы построили два набора данных — серверный и клиентский. Для связи этих наборов перенесите на форму компонент **DataSetProvider**. В его свойстве **DataSet** сошлитесь на **Table1**. А в свойстве **ProviderName** компонента **ClientDataSet1** сошлитесь на **DataSetProvider1**. Таким образом, вы установили связь между клиентским набором данных и сервером.

Теперь давайте посмотрим контекстное меню компонента **ClientDataSet**, всплывающее при щелчке на нем правой кнопкой мыши. Раздел CreateDataSet позволит вам создать набор данных. Поскольку **ClientDataSet1** связан провайдером с **Table1**, набор полей, который вы увидите в нижней таблице вашей формы, повторит набор **Table1**. Раздел контекстного меню Assign Local Data вызывает диало-

говое окно, в котором вы можете выбрать из списка набор, данные которого вы хотите скопировать в свой клиентский набор. В нашем примере в списке будет единственная строка — "Table1". Когда вы выберете ее, в вашем клиентском наборе не только появятся поля таблицы, но и занесутся все записи таблицы **Table1**. Раздел контекстного меню Load from MyBase table вызывает диалог открытия файла, в котором вы можете выбрать ранее созданный файл таблицы, данные из которой вы хотите загрузить в свой клиентский набор.

Если вы загрузили одним из описанных выше способов данные в набор, вам становятся доступными в контекстном меню разделы Save to MyBase XML table, Save to MyBase XML UTF8 table, Save to binary MyBase file. Эти разделы позволяют прямо во время проектирования создать файл таблицы данных соответствующего формата (или выбрать уже имеющийся файл) и сохранить в нем данные, загруженные в компонент **ClientDataSet**. Раздел меню Clear Data позволяет удалить данные из набора.

Пользуясь описанными разделами контекстного меню, можно прямо в процессе проектирования создать файлы XML таблиц MyBase, с которыми впоследствии будут работать приложения. Но в нашем примере вы можете ничего этого не делать, поскольку все необходимые операции приложение будет осуществлять во время выполнения. К тому же, нам еще надо сформировать поля совокупных характеристик, а при необходимости можно отредактировать свойства остальных полей. Все это делается Редактором Полей, вызываемым двойным щелчком на **ClientDataSet**. Введите обычным образом в набор объекты полей и отредактируйте их свойства — введите русские надписи и т.д.

Прежде, чем создавать поля совокупных характеристик, надо создать в наборе **ClientDataSet1** индекс, который будет нужен не только для упорядочивания данных, но и для их группирования при расчете совокупной характеристики. Это делается, как было описано в разд. 7.3.2, с помощью свойства **IndexDefs**. Щелчок на кнопке с многоточием около этого свойства в Инспекторе Объектов вызовет окно формирования индексов. Нажав в нем кнопку New, вы создадите новый объект индекса. В окне Инспектора Объектов задайте его свойство **Name** равным "grmark", а в свойстве **Fields** запишите строку "Gr;Mark;Fam". Это обеспечит упорядочивание данных по группам, внутри каждой группы — по оценке, а среди получивших равные оценки — по алфавиту. В свойство **DescFields** занесите имя поля **Mark**, по которому упорядочивание должно быть в убывающей последовательности. Это обеспечит приоритет отличников перед двоечниками. После формирования индекса надо в компоненте **ClientDataSet1** установить свойство **IndexName** равным введенному вами индексу **grmark**.

Теперь можно приступить к заданию полей, отображающих совокупные характеристики. Эта процедура была описана в разд. 7.3.3. Создайте поле для отображения числа студентов по группам. Назовите его **Count**. В свойстве этого поля **Expression** задайте строку "Count(*)" — расчет числа записей. В свойстве **GroupingLevel** задайте значение 1, а в свойстве **IndexName** — "grmark". Тем самым вы указали, что при расчете значения данного поля записи будут группироваться индексом **grmark**, и при выделении группы будет учитываться совпадение значений только одного первого поля (первым в индексе **grmark** указано поле **Gr**).

Создайте еще одно поле совокупной характеристики для отображения общего числа студентов. Назовите его **CountAll**. В свойстве этого поля **Expression** задайте строку "Count(Gr)". Это обеспечит расчет числа записей, в которых поле **Gr** не пустое. Свойство **GroupingLevel** оставьте равным нулю по умолчанию, а свойство

IndexName — пустым. Тем самым вы указали, что данное поле рассчитывается не по группе записей, а по всей таблице данных.

В описании поля **CountAll** вместо строки "Count(Gr)" можно было бы записать "Count(*)" — общее число записей. Различие этих выражений проявится, если вы внесете в таблицу записи, в которых поле **Gr** не заполнено (например, исключенных из учебного заведения, или только выразивших намерение поступить в него). При выражении "Count(Gr)" такие записи учитываться не будут, а при выражении "Count(*)" — будут.

Введите еще два поля совокупных характеристик: **AveMarkGr** и **AveMarkAll**. Задайте в обоих полях выражение "Avg(Mark)". Кроме того, в поле **AveMarkGr** в свойстве **GroupingLevel** задайте значение 1, а в свойстве **IndexName** — "grmark". Чтобы при расчете среднего балла результат отображался с точностью до десятых долей, задайте в обоих полях в свойстве **Precision** значение 2 — две значащих цифры. Иначе результат будет отображаться с точностью до 15 цифр, что, пожалуй, многовато.

Не забудьте во всех полях совокупных характеристик установить в **true** свойство **Active**, а также установить в **true** свойство **AggregatesActive** компонента клиентского набора.

Вам осталось разместить на поле компонент **DBImage2**, связав его с источником данных **DataSource2** и полем **Photo**, а также метки, отображающие совокупные характеристики. В примере на рис. 7.2 и 7.3 надписи "В группе", "Всего", "Студентов", "Ср. балл" обеспечены обычными метками **Label**, а отображение названия группы ("A1-02" на рис. 7.2), чисел студентов и средних баллов обеспечивается компонентами **DBText**, настроенными на поля **Gr**, **Count**, **CountAll**, **AveMarkGr**, **AveMarkAll**.

Теперь добавьте на форму кнопки В базу данных (ее имя в примере **BToDataBase**), В файл (имя **BToFile**), Отменить (имя **BRollback**), Обновить (имя **BRefresh**). Далее можно приступать к программированию. Код раздела **implementation** тестового приложения следующий:

```
var MySavePoint: integer;
    Modified: boolean;

procedure TForm1.FormCreate(Sender: TObject);
begin
 Table1.Open;
 ClientDataSet1.FileName := 'XML1.xml';
 ClientDataSet1.Open;
 ClientDataSet1.SaveToFile('', dfXML);
 MySavePoint := ClientDataSet1.SavePoint;
end;

procedure TForm1.BToDataBaseClick(Sender: TObject);
begin
 ClientDataSet1.ApplyUpdates(-1);
 Table1.Refresh;
 MySavePoint := ClientDataSet1.SavePoint;
 Modified := false;
end;

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

```
 ClientDataSet1.SavePoint := MySavePoint;
 Modified := false;
end;

procedure TForm1.BToFileClick(Sender: TObject);
begin
 MySavePoint := ClientDataSet1.SavePoint;
 Modified := false;
 //ClientDataSet1.SaveToFile('', dfXML);
end;

procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
 Modified := true;
end;

procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
 if Modified then
  case Application.MessageBox(
               'Данные были изменены. Сохранить их?',
               'Подтвердите сохранение изменений',
               MB_YESNOCANCEL+MB_ICONQUESTION) of
    IDCancel: CanClose := false;
    IDNo: ClientDataSet1.SavePoint := MySavePoint;
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Table1.Close;
 ClientDataSet1.Close;
end;

procedure TForm1.BRefreshClick(Sender: TObject);
begin
 if Modified then
  if Application.MessageBox(
               'В клиентском наборе имеются' +
               #13'несохраненные результаты редактирования.' +
               #13'Они будут стерты. Вы согласны на это?',
               'Подтвердите обновление данных, или откажитесь',
               MB_YESNO+MB_ICONQUESTION) = IDNo
    then exit;
 ClientDataSet1.FileName := '';
 ClientDataSet1.Close;
 ClientDataSet1.Open;
 ClientDataSet1.FileName := 'XML1.xml';
 ClientDataSet1.SaveToFile('', dfXML);
 MySavePoint := ClientDataSet1.SavePoint;
 Modified := false;
end;

procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
```

```
 ClientDataSet1Age.Value :=
          trunc((Date - ClientDataSet1Born.Value) / 365.25);
end;

end.
```

Рассмотрим этот код. В модуле вводится глобальная переменная **MySavePoint**, которая будет использоваться для запоминания состояния набора в момент начала серии незафиксированных операций редактирования. Глобальная булева переменная **Modified** будет показывать, имеются ли незафиксированные операции редактирования. Это потребуется нам при закрытии формы и при обновлении данных.

Процедура **FormCreate** является обработчиком события **OnCreate**, наступающего при создании формы приложения. Первый оператор открывает серверный набор данных **Table1**. Второй оператор задает имя файла клиентской базы данных. Поскольку имя задается без пути к нему, подразумевается файл в рабочем каталоге программы. Третий оператор открывает клиентский набор данных. Этот оператор срабатывает следующим образом. Если файл клиентской базы данных имеется в рабочем каталоге, данные из него загружаются в набор **ClientDataSet1**. Если же файла нет, то выполняется обращение к серверному набору данных **Table1** и информация загружается из него.

Следующий оператор сохраняет в файле данные из клиентского набора. Если такого файла не было, он создается этим оператором. Собственно, только для этого случая нужен данный оператор. Так что его можно было бы выполнять, только проверив отсутствие файла на диске. Последний оператор запоминает текущее состояние набора данных в переменной **MySavePoint**.

Следующая процедура **BToDataBaseClick** срабатывает при щелчке на кнопке В базу данных. Первый оператор передает данные в серверный набор методом **ApplyUpdates** без ограничения числа ошибок. Далее обновляется набор **Table1**, чтобы внесенные изменения отобразились в таблице приложения. В переменной **MySavePoint** запоминается текущее состояние набора данных, а в переменную **Modified** заносится **false**, чтобы констатировать, что незафиксированных изменений в клиентском наборе нет.

Процедура **BRollbackClick** срабатывает при щелчке на кнопке Отменить. В этой процедуре восстанавливается запомненное состояние набора данных, что приводит к отмене всех результатов редактирования. В переменную **Modified** заносится **false**, свидетельствуя об отсутствии незафиксированных изменений.

Процедура **BToFileClick** срабатывает при щелчке на кнопке В файл. Она просто заносит в переменную **MySavePoint** текущее состояние набора данных, поскольку сохранение в файле все равно произойдет автоматически при закрытии приложения. Впрочем, если учесть, что в процессе последующей работы возможны какие-то сбои компьютера, можно было бы выполнить в этой процедуре метод **SaveToFile** (закомментирован в приведенном листинге).

Процедура **ClientDataSet1AfterPost** является обработчиком события **AfterPost** набора **ClientDataSet1**. Это событие наступает после выполнения метода **Post** — передачи в набор каких-то результатов редактирования. В этом обработчике переменной **Modified** задается значение **true**, что свидетельствует об операции редактирования, еще не зафиксированной в файле или в серверной базе данных. Дело в том, что метод **Post** заносит данные не в базу, а в ее копию в памяти. И если не зафиксировать эти данные, они так и не попадет в базу. А фиксировать сразу нельзя — вдруг пользователь их отменит?

Процедура **FormCloseQuery** является обработчиком события формы **OnCloseQuery**. В этой процедуре проверяется по значению переменной **Modified**, есть ли незафиксированные изменения. Если есть, пользователю задается вопрос о необходимости их сохранения. Если пользователь ответил отрицательно, база данных приводится в состояние, запомненное в переменной **MySavePoint**, т.е. последние изменения данных отменяются и именно это состояние будет запомнено в файле. Если пользователь в диалоговом окне запроса нажал кнопку Отменить, закрытие приложения отменяется заданием параметру **CanClose** значения **false**. Если же пользователь согласился с необходимостью сохранить результаты редактирования, ничего не делается и текущее состояние набора не изменяется. В результате текущее состояние набора данных будет сохранено в файле.

Процедура **FormClose** является обработчиком события формы **OnClose**. В этом обработчике закрываются оба набора данных. При закрытии клиентского набора его текущее состояние сохраняется в файле.

Процедура **BRefreshClick** срабатывает при щелчке на кнопке Обновить. Считывание данных из базового набора сотрет незафиксированные результаты редактирования, если такие имеются в приложении. Поэтому, прежде всего, проверяется наличие незафиксированных изменений. Если они имеются, пользователю выдается сообщение, поясняющее возникшую ситуацию и запрашивающее разрешение на обновление. Если пользователь нажмет в диалоговом окне кнопку Нет, обновление прервется.

Если обновление должно проводиться, то следующий оператор процедуры обнуляет значение свойства **FileName** клиентского набора данных. Затем следует оператор, который закрывает методом **Close** соединение с базой данных. При этом сохранение данных в файле не происходит, так как имя файла в **FileName** не указано. Затем соединение с базой данных открывается методом **Open**. Так как имя файла по-прежнему не указано, клиентский набор загружается данными с основной базы данных. Последующие операторы восстанавливают имя файла в свойстве **FileName**, запоминают данные в файле, запоминают текущее состояние изменением значения **MySavePoint** и заносят **false** в переменную **Modified**, свидетельствуя об отсутствии незафиксированных изменений.

Процедура **ClientDataSet1CalcFields** обеспечивает расчет вычисляемого поля, отображающего возраст (см. разд. 6.8).

Мы разобрали все процедуры тестового приложения. Выполните его и проверьте работу в самых разных режимах. В частности, попробуйте удалять средствами Windows файл клиентского набора данных, чтобы убедится, что и в этом случае приложение работает нормально и этот файл восстанавливается.

7.4 Построение сервера с удаленным модулем данных

В разд. 7.3.5 был рассмотрен пример клиентского приложения. Однако в нем для простоты в качестве базового использовался набор данных, включенный в это же приложение. Конечно, реальные клиентские приложения работают, как правило, в распределенных системах, в которых база данных может быть расположена на удаленном компьютере, и с ней должно работать множество клиентских приложений.

Работа с удаленными данными в Delphi может быть организована различными способами. Мы рассмотрим, пожалуй, наиболее универсальный вариант — технологию DCOM, встроенную в Windows и поэтому не требующую развертывания какого-то дополнительного программного обеспечения.

Рассмотрим основные составляющие механизма построения многоуровневых распределенных приложений, работающих с данными. Приложение в общем случае состоит из следующих элементов:

Удаленные модули данных	Серверы, связанные с удаленными базами данных и обеспечивающие доступ клиентов к специальным компонентам — поставщикам информации.
Поставщики информации	Объекты, размещенные на удаленных модулях данных и возвращающие клиенту нужную информацию — результаты вычислений или наборы данных. Выполняют функции сервера приложений.
Компоненты связи	Обеспечивают связь между отдельными составляющими системы.
Клиентские приложения	Приложения, обрабатывающие получаемые наборы данных и посылающие запросы на удаленные модули данных.

Работа всех этих элементов организуется следующим образом. Клиентское приложение соединяется с сервером, содержащим удаленный модуль данных и расположенным на серверном компьютере. Если сервер не был запущен, то клиентское приложение автоматически запускает его на выполнение. После этого сервер будет постоянно располагаться в памяти, обслуживая клиентов.

Далее в процессе работы клиентское приложение запрашивает у сервера приложений (поставщика информации) какой-то набор данных. Сервер приложений обращается к соответствующей базе данных, получает требуемый набор, упаковывает его и посылает клиентской программе. Клиентская программа распаковывает набор, создает его локальную копию и с помощью обычных компонентов отображения и редактирования данных предоставляет его пользователю.

Если пользователь вносит в данные (локальную копию) какие-то изменения, клиентская программа посылает пакет сделанных изменений серверному приложению. Сервер приложений распаковывает его и формирует транзакцию для сервера данных. Если никаких проблем не возникает, то клиентской программе возвращается измененный набор данных. Если же при изменении данных возникли какие-то недоразумения, то сервер приложений формирует набор ошибочных данных и возвращает его клиенту. Пользователь, работающий с клиентской программой, должен определить, как разрешить возникшие проблемы.

Начнем создание такого многоуровневого приложения с построения сервера, содержащего удаленный модуль данных. Начните новое приложение. Создавшаяся форма будет представлять собой окно сервера. Никаких особых операций с сервером выполняться не будет. Так что можете ограничиться заданием какого-то заголовка окна и, если хотите, добавить на форму кнопку Закрыть и написать в обработчике щелчка на ней оператор **Close**. Свойство формы **BorderStyle** целесообразно задать равным **bsSingle**, поскольку изменять размер окна пользователю не имеет смысла, а свойство **WindowState** целесообразно задать равным **wsMinimized**, так как с окном сервера пользователю работать не придется и удобнее, если сервер будет выполняться в свернутом виде. Сохраните проект под именем *RMDLearn*.

Теперь надо добавить в сервер удаленный модуль данных. Выполните команду File | New | Other и на странице Multitier (многоуровневые приложения) выберите пиктограмму Remote Data Module (удаленный модуль данных). Появится диалоговое окно, показанное на рис. 7.3. Не будем вдаваться в тонкости технологии реализации нашего модуля данных Вам достаточно написать в окошке CoClass Name произвольное имя интерфейса создаваемого модуля данных. Занесите, например, "clRMDLearn", как показано на рис. 7.4, и щелкните на кнопке OK. В вашем приложении появится окно модуля данных, похожее на окно обычной формы. Но пользователь не увидит этой формы. Она служит только для того, чтобы на нее можно было поместить компоненты, обеспечивающие связь с базой данных.

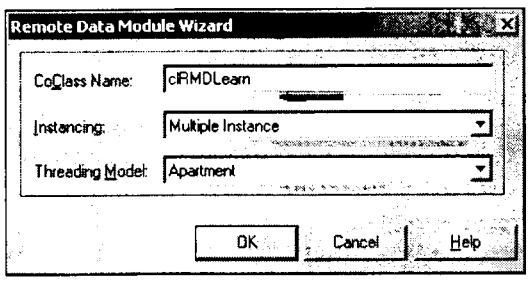


Рис. 7.4
Окно задания свойств удаленного модуля данных

В качестве компонента набора данных будем использовать простейший — компонент **Table**. Перенесите этот компонент в модуль данных и соедините его обычным образом с таблицей Pers базы **Learn**. Установите в **true** свойство **Active**, чтобы сервер при его запуске сразу соединялся с таблицей данных. Такую связь можно установить заранее, так как сервер будет постоянно находиться в памяти, обслуживая запросы клиентов.

Теперь надо перенести в модуль компонент-провайдер, который будет играть роль серверного приложения, описанного выше. Это компонент **DataSetProvider**. Начиная с Delphi 6, он расположен в библиотеке на странице Data Access, а ранее располагался на странице MIDAS.

Основное свойство компонента — **DataSet**. Оно определяет набор данных, с которым связан поставщик данных. В нашем примере в этом свойстве надо указать набор данных **Table1**. Булево свойство **Constraints** определяет, будут ли передаваться клиенту ограничения, заложенные в таблицу данных. Если **Constraints = true**, то эти ограничения передаются, и клиентское приложение может локально проверять, удовлетворяют ли им новые или редактируемые данные. Это понизит возможность ошибок при обратной пересылке данных на сервер и фиксации их в базе данных.

Свойство **UpdateMode** определяет, по каким полям провайдер идентифицирует запись при ее фиксации в наборе данных после редактирования. Возможные значения **UpdateMode**:

upWhereAll	идентификация по начальным значениям всех полей
upWhereChanged	идентификация по ключевым полям и по начальным значениям измененных полей
upWhereKeyOnly	идентификация только по ключевым полям

В нашем примере, очевидно, надо выбрать вариант **upWhereKeyOnly**.

Проектирование сервера баз данных закончено. Сохраните ваш проект. Вы создали простейший сервер баз данных, не выполняющий никаких операций, кроме связи с указанной таблицей указанной базы данных. Теперь надо зарегистрировать его в системе как сервер данных. Для того чтобы зарегистрировать его на том компьютере, на котором вы работаете, надо выполнить на нем приложение сервера с параметром командной строки "/regserver".

Из среды разработки Delphi это можно сделать следующим образом. Выполните команду Run | Parameters и в отрывшемся окне на странице Local занесите в окошко Parameters значение "/regserver". После этого надо закрыть данное диалоговое окно и запустить серверное приложение на выполнение. При этом никакой реакции вы не увидите, никаких окон на экране не появится, но Windows зарегистрирует ваш сервер, внеся в свой реестр соответствующие записи.

При последующих выполнениях сервера надо удалить из командной строки параметр "/regserver ". А если вы в дальнейшем захотите удалить информацию о регистрации вашего сервера из системы, вам надо будет выполнить ваше серверное приложение с параметром командной строки "/unregserver".

Теперь можно проектировать клиентское приложение. Построим его на основе того клиентского приложения, которое вы создали в разд. 7.3.5 (рис. 7.2 и 7.3). Откройте то приложение, сохраните его проект (File | Save Project As) под новым именем и сохраните под новым именем (File | Save As) файл модуля. В новом приложении нам не потребуется набор данных на основе **Table**, так как мы будем работать с удаленным модулем данных. Поэтому удалите все компоненты верхней части формы (**Table1**, **DataSource1**, **DBGrid1**, **DBImage1**), оставив только **DataSetProvider** — он пригодится нам в новом приложении. Из кода уберите операторы, работающие с **Table1** в процедурах **FormCreate**, **BToDataBaseClick**, **FormClose**.

Как видите, изменения в приложении минимальны. Нам осталось только обеспечить соединение с удаленным модулем данных. Перенесите на форму компонент **DCOMConnection** со страницы DataSnap. Этот компонент обеспечит связь клиента с удаленным модулем данных по технологии DCOM.

Для указания сервера воспользуйтесь свойством **ServerName**. Выпадающий список около этого свойства в Инспекторе Объектов содержит имена классов зарегистрированных серверов DCOM. В нем должен быть и сервер, который вы создали и зарегистрировали. Впрочем, он будет там, если сервер расположен на том же компьютере, на котором проектируется клиентское приложение. Если сервер расположен на другом компьютере, надо задать имя этого компьютера в свойстве **ComputerName**.

Установка в **true** свойства **Connected** обеспечивает соединение с сервером. Даже во время проектирования установка этого свойства обеспечит запуск сервера. Если он находится на том же компьютере, вы увидите, что сервер появится в полосе задач. При задании **Connected = false** сервер завершит выполнение. Впрочем, нам не требуется сейчас задавать **Connected = true**. Это свойство установится программно во время выполнения.

Снабжать данными клиентский набор, который у нас основан на компоненте **ClientDataSet**, будет **DataSetProvider**. Компоненты **ClientDataSet1**, **DCOMConnection1** и **DataSetProvider1** должны быть связаны друг с другом и с сервером следующим образом. Компонент соединения **DCOMConnection1**, как уже говорилось, связывается с сервером. Набор данных **ClientDataSet1** связывается своим свойством **RemoteServer** с **DCOMConnection1**, а свойством **ProviderName**

с провайдером **DataSetProvider1**. Компонент **DataSetProvider1** своим свойством **DataSet** связывается с набором данных **ClientDataSet1**. Впрочем, это просто дублирует связь между провайдером и набором данных, уже заданную свойством **ProviderName** набора данных.

Разработка клиентского приложения, использующего удаленный модуль данных, завершена. Можете выполнить ваше приложение и опробовать его в работе. Чтобы почувствовать возможность параллельной работы нескольких клиентов, можно запустить средствами Windows несколько экземпляров приложения. Если сервер расположен на том же компьютере, вы увидите, что он выполняется до тех пор, пока выполняется хотя бы одно клиентское приложение.

7.5 Некоторые итоги

Данная глава завершает изучение языка Pascal и основ программирования в Delphi. Вы многому научились, если, конечно, серьезно изучали материал, для себя, а не для галочки. И, возможно, считаете себя специалистом, способным решить любые задачи, которые перед вами возникнут. Частично вы правы — материал данной книги действительно позволит вам разрабатывать приложения для решения множества прикладных задач самого различного характера. А изучение языка Pascal создает прекрасный задел для освоения любых других языков программирования, если это вам потребуется. Но, увы, материала данной книги мало для того, чтобы можно было считать себя специалистом, даже начинающим. Слишком многое, о чем хотелось бы рассказать, в книгу не вошло. Так что, рискуя вас разочаровать, я все же перечислю для тех, кто хотел бы знать больше, что еще имеет смысл изучить.

В описании языка Object Pascal пропущен очень важный раздел — работа с сообщениями Windows. Вся работа Windows основана на сообщениях. Реакция на многие сообщения в компонентах Delphi заложена изначально. И когда вы пишете обработчики событий компонентов, вы в действительности пишете обработчики событий Windows. Но имеется возможность генерировать собственные сообщения и перехватывать любые сообщения, не предусмотренные в компонентах Delphi. Это одна из возможностей общения ваших приложений друг с другом. Имеется также ряд других технологий, позволяющих различным приложениям общаться друг с другом. Одно из первых мест среди них занимает OLE. С помощью этой технологии вы можете, например, управлять из своего приложения офисными приложениями Windows: Word, Excel, почтовыми программами. Можете даже встраивать эти всем известные программы в свое приложение. Впрочем, другие технологии, такие, как COM и DCOM (частично вы использовали эту технологию в разд. 7.4, но без каких-либо пояснений) позволяют делать не меньше. Представляете, какие широкие возможности открывают перед вами подобные технологии? Так что, безусловно, стоит их изучить.

Большинство современных технологий, обеспечивающих взаимодействие приложений, основано на понятии интерфейса. В данной книге об интерфейсах (не надо путать их с интерфейсной частью модулей или с графическим интерфейсом пользователя) даже не упоминалось. Но если вы хотите осознанно использовать современные технологии, с этим понятием надо бы ознакомиться.

В разд. 3.5 вы научились работать с классами и создавать собственные классы. Но при этом совершенно не рассматривался вопрос создания своих собственных компонентов. А такая возможность имеется. Вы можете создать свои компоненты

(кнопки, окна редактирования и т.п.), включить их в Delphi и использовать в своих разработках. Можете передавать эти компоненты друзьям или заказчикам. Многие занимаются этим, и вы можете влиться в их ряды.

Имеется также возможность разрабатывать так называемые объекты ActiveX. Создав подобный объект, вы или кто угодно другой может использовать их в программах, написанных на любом языке программирования. В частности, вы можете вызывать их из таких программ, как Word, Excel и других. Тем самым вы как бы встраиваете собственные разработки в эти используемые всеми программы Windows. Есть и другой способ решения подобных задач — разработка библиотек DLL, которые также могут использоваться вашими или любыми иными программами. Delphi предусматривает возможность реализации всех этих возможностей.

Совершенно не затронуты в книге вопросы создания приложений для работы в Интернет. Delphi предоставляет множество средств для разработки подобных приложений, которые будут по мере расширения в нашей стране доступа к Интернет занимать все более важное место среди программных продуктов. Так что освоить хотя бы начала этих технологий необходимо, чтобы не отставать от жизни.

Множество интереснейших задач можно решить, используя возможности, заложенные в механизмах Windows. Можно создавать непрямоугольные формы и окна, можно делать надписи, наклоненные под любыми углами, можно регистрировать в Windows свои приложения или свои расширения файлов, можно создавать приложения-невидимки, можно перехватывать все нажатия клавиш или кнопок мыши и трактовать их по-своему, можно управлять хранителями экрана и обоями рабочего стола, выключать компьютер, перезагружать систему — словом, если вы освоите функции, заложенные в API Windows, вы получите почти неограниченную власть над компьютером и механизмами Windows.

Пожалуй, на этом я остановлюсь, чтобы не подавить читателя объемом знаний, которых ему не хватает, чтобы с полным правом считать себя специалистом. Всему, что я перечислил, научиться не так уж сложно. Это, пожалуй, намного проще, чем то, что вы уже освоили. Тем, кто хочет знать больше, посвящен последний раздел данной книги.

7.6 Проверьте себя

7.6.1 Вопросы для самопроверки

1. Почему в компоненте **Query** отсутствуют свойства, позволяющие упорядочить отображение данных, и чем можно заменить эти свойства?
2. В чем ошибка следующего оператора SQL:

```
SELECT Fam, max(Mark) FROM Pers
```

3. Для чего используется элемент **GROUP BY** оператора **Select**?
4. Какие проблемы могут возникать при многопользовательском доступе к базе данных?
5. В каких случаях и почему компонент **Query** лучше, чем **Table**?
6. Какие преимущества у клиентских наборов данных?
7. В каком отношении портфельный набор данных лучше, чем кэширование?

7.6.2 Задачи

1. Напишите запрос SQL, выбирающий из таблицы **Pers** записи отличников. Проверьте результат с помощью компонента **Query**.
2. Напишите запрос SQL, выбирающий из таблицы **Pers** записи студентов, которые учатся ниже среднего (оценка ниже средней по всем группам). Проверьте результат с помощью компонента **Query**.
3. Напишите запрос SQL, выбирающий из таблицы **Pers** запись самого молодого студента. Проверьте результат с помощью компонента **Query**.
4. Напишите запрос SQL к таблицам **Pers** и **Gro**, отображающий группу, фамилию, оценку и фамилию преподавателя (избегайте дублирования результатов с помощью элемента **FROM**). Проверьте результат с помощью компонента **Query**.
5. Дополните клиентское приложение, созданное в разд. 7.3.5 и 7.4, возможностью выбора пользователем способа упорядочивания данных.
6. Дополните клиентское приложение, созданное в разд. 7.3.5 и 7.4, возможностью выбора пользователем способа фильтрации данных по курсам.

Тем, кто хочет знать больше

Материал, который вы изучили, позволил вам освоить основы программирования на языке Pascal и основы создания прикладных программ для Windows. Но это только основы (см. разд. 7.5). Для тех, кто хочет существенно пополнить свои знания, ниже приводятся сведения по дополнительным источникам информации, написанным автором, и список сайтов, на которых вы можете получить ответы практически на любые вопросы.

1. **Архангельский А. Я. Программирование в Delphi 7 — М: ЗАО «Издательство БИНОМ», 2003**

Книга содержит методику разработки в Delphi приложений различного назначения. Она позволит вам намного глубже изучить Delphi и начать действительно профессионально работать с этой системой. Материал книги рассчитан на использование разных версий Delphi — от 4 по 7.

Рассмотрены такие особенности Delphi, как создание кросс-платформенных приложений, технологии доступа к данным ADO, InterBase Express, dbExpress, компоненты-серверы COM, технологии распределенных приложений COM, CORBA, MIDAS. Дается методика построения прикладных программ, реализующих текстовые и графические редакторы, мультипликацию и мультимедиа, работу с базами данных, построение отчетов, приложений для Интернет, распределенных приложений, клиентов и серверов. Справочная часть книги содержит материалы по языку Object Pascal, функциям Delphi и API Windows, компонентам и классам Delphi, их свойствам, методам и событиям.

2. **Архангельский А. Я. Delphi 7. Справочное пособие. — М: ЗАО «Издательство БИНОМ», 2003**

Это справочное пособие содержит подробные справочные сведения по языку Object Pascal и по Delphi 7. В ней дается полное описание языка (в данной книге некоторые важные его аспекты не были рассмотрены — см. разд. 7.5). Приводится полный список функций Object Pascal и Delphi, причем по большинству функций дается развернутое описание и примеры. Даются справочные сведения по многим компонентам Delphi, их свойствам, методам, событиям. Однако в отличие от книги [1] это именно справочное пособие, т.е. в нем отсутствует последовательное рассмотрение методики проектирования. Так что, руководствуясь приводимыми в пособии справочными сведениями и многочисленными примерами, вам надо самостоятельно продумывать методику разработки приложений различного назначения. Следует отметить также, что основной материал пособия содержится в электронном виде в источнике [3].

3. **Серия справочных файлов «Русские справки по Delphi»**

Серия справок — это программный продукт, призванный оказать вам поддержку в процессе проектирования. Справки встраиваются в среду Delphi в дополнение к англоязычной справке и в процессе проектирования при нажатии клавиши F1

вам предлагаются на выбор темы английских или русских справок. Русские справки — это не перевод с английского, а, скорее, электронный вариант некоторых материалов книг [1], [2] и [4]. Так что они могут быть полезны не только тем, кто испытывает определенные сложности с английским языком, но и всем пользователям Delphi, поскольку содержат иначе построенное и скомпонованное изложение справочных данных, иные примеры, в них устранен ряд ошибок англоязычных справок (надеюсь, не добавлено собственных ошибок).

Достоинство справок по сравнению с книгами в том, что они обеспечивают оперативную помощь в среде разработки Delphi, облегчают поиск нужной информации (в книгах это делать значительно сложнее), позволяют легко переносить примеры в свой проект. Да и стоят справки намного дешевле, чем книги. Но, конечно, справки не заменяют книг, хотя и содержат много материала, не поместившегося в книги.

Справки распространяются через Интернет по адресу: http://lab18.ipu.rssi.ru/help2. Там вы найдете условия распространения, включающие бесплатную поддержку — несколько раз в год выходят дополнения к справкам, которые распространяются бесплатно тем, кто приобрел начальную версию. Возможна корпоративная поставка справок учебным заведениям для постановки их в дисплейных классах.

Распространение через Интернет не означает, что вы обязательно должны иметь доступ в Интернет с домашнего компьютера и иметь собственный адрес e-mail. Достаточно, если доступ в Интернет и e-mail есть у вас на работе или у кого-то из ваших друзей и знакомых. Вы можете воспользоваться этими возможностями, а затем на дискетах перенести файлы на свой компьютер.

4. Архангельский А. Я. Приемы программирования в Delphi. Версии 5–7. — М: ЗАО «Издательство БИНОМ», 2003

Книга рассчитана на читателей, освоивших Delphi, например, по данной книге или по книге [1] и желающих расширить свои знания и возможности проектирования. Охвачен широкий круг вопросов, начиная с типовых решений традиционных вычислительных задач (обработка массивов, векторы, матрицы, решение систем линейных и нелинейных уравнений), и кончая приемами программирования при взаимодействии с механизмами Windows и с приложениями Microsoft Office. Рассматривается обработка документов различных видов. Детально обсуждаются вопросы построения графиков и диаграмм, создание и использование DLL. Излагается методика работы с удаленными модулями данных.

Книга содержит много модулей процедур и функций с подробными пояснениями, а также законченные приложения, которые можно включать в свои проекты. Даются ответы на множество вопросов, часто задаваемых пользователями Delphi на конференциях и форумах.

5. Сайты в Интернет

В Интернет имеется чрезвычайно много сайтов, посвященных Delphi и содержащих статьи, собрания FAQ (часто задаваемые вопросы), программы. Ниже дается очень краткий список некоторых из них. Я отдал предпочтение в этом списке тем сайтам, которые содержат каталоги ссылок на другие источники информации в Интернет. Впрочем, во многих случаях для получения ответа на возникший у вас вопрос лучше всего пользоваться адресом **www.ya.ru**. Это поисковая система Ян-

декс, в которой надо только удачно сформулировать вопрос и наверняка получишь ответ, имеющийся на одном из сайтов, посвященных Delphi.

delci.h1.ru — сайт автора данной книги, информация о вышедших и готовящихся к изданию книгах, форум.

lab18.ipu.rssi.ru/help2 — сайт информации о сериях русских справок, регистрация, получение, форум.

www.borland.ru — сайт представительства Borland в России, информация о программных продуктах, технологиях, о проводимых семинарах и т.п.

www.delphikingdom.com — виртуальный клуб программистов «Королевство Delphi», круглый стол с участием прекрасных специалистов, каталог ссылок и многое другое.

http://rusdoc.df.ru/index.shtml, **http://rusdoc.df.ru/reviews/programming/delphi** — большое и интересное собрание статей на русском языке по Delphi.

www.delphiplus.org — статьи, ссылки на сайты, посвященные Delphi, большая подборка FAQ.

delphi.mastak.ru — «Мастера Delphi», форум, собрания FAQ, статьи, программы, ссылки на сайты, посвященные Delphi; все на высоком уровне.

forum.vingrad.ru — очень хороший форум программистов.

http://list.mail.ru, **list.mail.ru/10355/1/0_1_0_1.html** — очень хороший каталог сайтов и FAQ по Delphi.

http://blackman.wp-club.net — «Курс борьбы с Delphi», статьи, большая подборка FAQ.

http://alex-co.narod.ru — статьи, FAQ.

www.borland.xportal.ru — «Borland X Portal», Delphi и C++Builder (правда, больше C++Builder), форум, FAQ, скачиваемые ресурсы.

www.softodrom.ru — программы, статьи, форум.

http://delphi.chertenok.ru — статьи, форум, FAQ.

www.download.ru — множество программ и исходных текстов.

listsoft.ru — программы.

Приложение

Некоторые сообщения об ошибках и замечания компилятора Delphi

';' not allowed before 'ELSE'
«Точка с запятой недопустима перед 'else'»

Сообщение об ошибке. Перед **else** в операторе **if...then...else** поставлена точка с запятой, что не допускается. Удалите эту точку с запятой.

File not found: '...'
«Не найден файл '...'»

Сообщение об ошибке. Компилятор не нашел указанный файл модуля, или файл откомпилированного модуля (*.dcu*), объектного файла, файла ресурсов.

'...' expected but '...' found
«Ожидалось '...', но найдено '...'»

Сообщение о синтаксической ошибке. По правилам синтаксиса ожидался какой-то один синтаксический элемент, а компилятор обнаружил другой. Проверьте, не ошиблись ли вы в синтаксисе. Возможно, в оператор попал ошибочный символ. Нередко причина не в данном операторе, а в предыдущем: например, вы забыли поставить точку с запятой в конце предыдущего оператора.

Circular unit reference to 'Unit...'
«Циклические ссылки на модуль 'Unit...'»

См. разд. 2.3.3.

Could not compile used unit '...'
«Невозможно компилировать модуль '...'»

Результат серьезных ошибок, исключающих возможность компиляции и выполнения приложения. Посмотрите и исправьте ошибки, на которые указывают сообщения компилятора, предшествующие данному.

For loop control variable must be simple local variable
«В качестве управляющие переменной цикла for надо использовать простую локальную переменную»

Замечание компилятора. Для управляющей переменной цикла **for** (см. разд. 2.8.7.2) надо использовать простую переменную (например, не поле записи). Рекомендуется также использовать локальную, а не глобальную переменную, так как это способствует генерации более эффективного кода и может предотвратить ошибки, связанные с использованием одной глобальной переменной в разных циклах.

FOR-Loop variable '...' may be undefined after loop
«Управляющая переменная '...' цикла for может быть не определена после цикла»

Замечание компилятора. В коде используется значение управляющей переменной цикла **for** (см. разд. 2.8.7.2) после окончания цикла. Но это значение не определено, и никто не гарантирует, чему оно будет равно. Несмотря на то, что это только замечание компилятора, игнорировать его нельзя. Иначе в программе возникнут ошибки вычислений.

Incompatible types
«Несовместимые типы»

Сообщение об ошибке, связанной обычно с отличием типа аргумента, передаваемого в функцию или процедуру, от типа, указанного в объявлении функции или процедуры. Посмотрите объявление соответствующей функции или процедуры и приведите типы аргументов в соответствие с объявлением.

Incompatible types: '...' and '...'
«Несовместимые типы: '...' и '...'»

Сообщение об ошибке, связанной с несовместимыми типами данных. Это может быть результатом записи неправильного оператора присваивания, когда тип его левой части младше типа правой части (например, целой переменной присваивается действительное значение) или вообще несовместим с ним (например, строке присваивается численное значение). Аналогичные ошибки могут возникать при передаче аргумента недопустимого типа в функцию или процедуру. Так что проверьте типы данных, используемые в ошибочном операторе, и исправьте ошибку. См. разд. 2.4.2.

Left side cannot be assigned to
«Левой части нельзя присвоить значение»

Сообщение об ошибке. В качестве левой части оператора присваивания указана величина, допускающая только чтение. Обычно способом устранения этой ошибки является введение промежуточной переменной, которой присваивается требуемое значение.

Missing operator or semicolon
«Ошибочный оператор или точка с запятой»

Сообщение об ошибке. Обычно свидетельствует о том, что вы забыли поставить точку с запятой в конце предыдущего оператора.

Ordinal type required
«Требуется порядковый тип»

Сообщение об ошибке. Часто относится к оператору **case** (см. разд. 2.8.5) или аналогичным операторам, в которых выражение должно иметь порядковый тип (см. об этих типах в разд. 2.5).

Return value of function '...' might be undefined
«Возвращаемое значение функции '...' может быть не определено»

Замечание компилятора. Надо проверить, действительно ли возвращаемое значение может быть не определено. Иногда во всех случаях значение определено, но компилятор этого просто «не знает». Соответствующий пример вы найдете в разд. 3.1.5.3. В этих и только в этих случаях замечание можно игнорировать.

The project already contains a form or module named Form...
«Проект уже содержит форму или модуль с именем Form...»

Сообщение об ошибке. См. разд. 1.5.

Type '...' needs finalization — not allowed in file type
«Тип '...' требует критерия окончания — недопустим как тип файла»

Сообщение об ошибке при объявлении файловой переменной типизированного файла. Обычно свидетельствует о том, что компилятор не знает, сколько места надо выделить под элемент файла или о том, что это место может быть различным в разных элементах, что недопустимо для типизированного файла. Например, тип длинной строки надо заменить на тип короткой строки или на массив символов — см. разд. 3.4.5.

Undeclared identifier: '...'
«Необъявленный идентификатор '...'»

Сообщение об ошибке. Возможные причин: ошибка в написании идентификатора, забыли ввести объявление переменной или функции, вызвали библиотечную функцию, но модуль, содержащий ее объявление, к проекту не подключен. Если дело в вызове библиотечной функции, то посмотрите во встроенной справке, в каком модуле функция объявлена, и подключите предложением **uses** этот модуль. См., например, разд. 2.4.2.

Unsatisfied forward or external declaration: '<имя_класса.идентификатор>'
«Неудовлетворенное предварительное или внешнее объявление <имя_класса.идентификатор>»

Сообщение об ошибке. Было объявление функции или процедуры с указанным идентификатором, но не найдено ее реализации. Обычно следствие того, что вы объявили в классе какой-то метод, или поместили в интерфейсе модуля объявление, но забыли реализовать его. Или ошиблись в написании идентификатора в объявлении или в реализации.

Unterminated string
«Незавершенная строка»

Сообщение об ошибке. Обычно ошибка заключается в том, что в операторе имеется строковая константа, заключенная в одинарные кавычки, но вы забыли поставить одну из них. Поскольку в окне Редактора Кода строковые константы выделяются цветом, вы легко найдете, где должна стоять эта забытая вами кавычка.

Value assigned to '...' never used
«Значение, присвоенное '...' нигде не используется»

Замечание компилятора, которое можно игнорировать, если пока вы просто не успели написать код, использующий значение, присвоенное указанной переменной. Но на всякий случай проверьте — может быть, это сообщение свидетельствует об ошибке в программе.

Variable '...' is declared but never used in '...'
«Переменная '...' объявлена, но нигде не используется в '...'»

Замечание компилятора, которое можно игнорировать, если переменная введена для какого-то будущего использования. См. разд. 2.4.2.

Variable '...' might not have been initialized
«Возможно, переменная '...' не инициализирована»

Предупреждение компилятора, которое нельзя игнорировать. См. разд. 2.4.1.

Предметный указатель

Учебное издание

Архангельский Алексей Яковлевич

Язык Pascal и основы программирования в Delphi

Оформление обложки *И.Ю. Буровой*
Компьютерная верстка *С.В. Лычагина*

Подписано в печать 25.05.2004. Формат $70x100/_{16}$. Усл. печ. л. 40,3
Гарнитура Школьная. Бумага газетная. Печать офсетная
Тираж 3500 экз. Заказ № 2585

Издательство «Бином-Пресс», 2004 г.
170026, Тверь, Комсомольский просп., 12

При участии ПФ «Сашко»

Отпечатано с готовых диапозитивов во ФГУП ИПК
«Ульяновский Дом печати». 432980, г. Ульяновск, ул. Гончарова, 14