Юлий Кетков, Александр Кетков

Практика программирования: Visual Basic, C++ Builder, Delphi

САМОУЧИТЕЛЬ

Основы визуального программирования

Общие свойства компонентов визуальных сред

Около 100 готовых к исполнению программ

Содержит исходные тексты всех приведенных программ

Языки программирования в примерах

Юлий Кетков, Александр Кетков

Практика программирования: Visual Basic, C++ Builder, Delphi

Санкт-Петербург

«БХВ-Петербург»

2002

УДК 681.3.06
ББК 32.973.26-018.1
К37

Кетков Ю. Л., Кетков А. Ю.

К37 Практика программирования: Visual Basic, C++ Builder, Delphi. — СПб.: БХВ-Петербург, 2002. — 464 с.: ил.

ISBN 5-94157-191-7

Книга знакомит читателя с системами визуального проектирования Visual Basic, Borland C++ Builder, Delphi. В ней обсуждаются характеристики наиболее часто используемых компонентов (объектов). При этом особое внимание уделяется общности функциональных свойств рассматриваемых компонентов и способов их применения в приложениях, создаваемых в разных системах программирования. В книге содержится множество готовых решений и иллюстраций, которые помогут читателю легко разобраться в изложенном материале.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Анатолий Адаменко*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Петр Науменко*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Татьяна Звертановская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 24.05.02.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 37,41.
Тираж 5000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

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

ISBN 5-94157-191-7

© Кетков Ю. Л., Кетков А. Ю., 2002
© Оформление, издательство "БХВ-Петербург", 2002

СОДЕРЖАНИЕ

Предисловие **1**

Почему мы написали эту книгу 1
Для кого написана эта книга? 2

Глава 1. Краткий обзор **3**

1.1. Немного истории 3
1.2. Типы данных 8
1.2.1. Основные типы данных в системе Visual Basic 8
1.2.2. Основные типы данных в системах BCB и Delphi 12

Глава 2. С чего начать **17**

2.1. Знакомство с IDE 17
2.2. Общие свойства объектов визуальных сред 25
2.3. Главная форма 28
2.4. Прежде чем запустить приложение 30
2.5. Из чего состоит проект пустого приложения 31
2.6. Продолжение работы над проектом 34
2.7. Расстановка компонентов на форме 37
2.8. Главное меню 42
2.8.1. Главное меню среды Visual Basic 6.0 42
2.8.2. Главное меню среды Borland C++ Builder 5.0 52
2.9. Как заставить объекты реагировать на события 57

Глава 3. Обработка строк **61**

3.1. Строковые данные в Borland C++ Builder
и методы их обработки 61
3.2. Отображение строк в поле метки 64
3.3. Работа с объектами типа "Редактируемая строка" 65
3.3.1. Использование компонента *TextBox* в Visual Basic 65
3.3.2. Использование объекта *MaskEdBox* в Visual Basic 75
3.3.3. Специфика компонента *RichTextBox* в Visual Basic 78
3.3.4. Использование компонент типа *TEdit* в средах
Borland C++ Builder и Delphi 80

3.3.5. Специфика работы с объектом *MaskEdit* в средах Borland C++ Builder и Delphi....90
3.4. Визуализация многострочных документов....93
3.4.1. Объект *TextBox* в многострочном режиме....93

Глава 4. Работа со списками.... 95

4.1. Работа с объектом типа *Memo*....95
4.2. Классический список *ListBox* в системе Visual Basic....99
4.3. Классический список *ListBox* в системах Borland C++ Builder и Delphi....102
4.4. Компактный список *ComboBox* в системе Visual Basic....105
4.5. Компактный список *ComboBox* в системах Borland C++ Builder и Delphi....106
4.6. Списки *DriveListBox*, *DirListBox* и *FileListBox* в среде Visual Basic....107
4.7. Списки *DriveComboBox*, *DirectoryListBox* и *FileListBox* в системах Borland C++ Builder и Delphi....109
4.8. Присоединение объектов к строкам списка *ListBox*....112
4.9. Сортировка грибов....113
4.10. Просмотр системных шрифтов....116

Глава 5. Кнопки и меню....127

5.1. Командные и инструментальные кнопки....128
5.2. Специальные кнопки....133
5.3. Специфика обработки событий кнопок....141
5.4. Создание главного меню приложения....144
5.4.1. Работа с *Menu Editor* в среде Visual Basic....145
5.4.2. Пример создания оболочки меню в среде Visual Basic....147
5.4.3. Пример создания оболочки меню в среде Borland C++ Builder....150
5.4.4. Создание всплывающих меню в Visual Basic....152
5.4.5. Создание всплывающих меню в Borland C++ Builer и Delphi....154
5.5. Нестандартное меню....155

Глава 6. Стандартные интерфейсные объекты....161

6.1. Диалоговое окно для ввода текста....161
6.2. Диалоговые окна для вывода сообщений....164
6.2.1. Функция *MsgBox* в системе Visual Basic....164
6.2.2. Вывод сообщений в системах BCB и Delphi....166
6.3. Универсальные и специализированные диалоги....169
6.3.1. Выбор файла в режиме диалога....170

6.3.2. Выбор файла для сохранения данных ... 174
6.3.3. Диалог по поводу выбора шрифта ... 175
6.3.4. Диалог по выбору цвета ... 175
6.3.5. Диалоги по установкам и настройкам принтера ... 176
6.3.6. Диалоговые окна для поиска и замены текста ... 180
6.3.7. Выбор графических файлов ... 186

Глава 7. Работа с файлами ... 187

7.1. Файлы в системе Visual Basic ... 189
7.2. Файлы в среде Borland C++ Builder ... 192
7.3. Файлы в Delphi ... 196

Глава 8. Машинная графика ... 203

8.1. Рисуем? Где? Чем? ... 204
8.2. Что позволяет рисовать система VB? ... 212
8.2.1. Работа с отдельными точками ... 215
8.2.2. Отрезки прямых и прямоугольники ... 216
8.2.3. Окружности, эллипсы, дуги и сектора ... 217
8.2.4. Очистка канвы графического объекта ... 218
8.3. Что позволяют рисовать системы Borland C++ Builder и Delphi? ... 218
8.3.1. Отрезки прямых и ломаные ... 219
8.3.2. Стандартные прямоугольники ... 221
8.3.3. Нестандартные многоугольники ... 228
8.3.4. Кривые второго порядка ... 230
8.3.5. Обмен с графическими файлами ... 230
8.4. Вывод символьных и числовых данных на канве ... 231
8.4.1. Отображение текстовой и числовой информации в среде Visual Basic ... 231
8.4.2. Работа с текстами в Borland C++ Builder и Delphi ... 231
8.5. Специфика работы с графическими объектами ... 232
8.5.1. Графические объекты в системе Visual Basic ... 232
8.5.2. Графические объекты в системах Borland C++ Builder и Delphi ... 237
Объекты типа *TBitmap* ... 238
Объекты типа *TIcon* ... 239
Объекты типа *TMetafile* ... 239
Объекты типа *TPicture* ... 240
Объект типа *Clipboard* ... 241
Экранные и внеэкранные графические объекты ... 241
8.5.3. Копирование растровых изображений ... 245
8.6. Создание монотонно изменяющегося фона ... 250
8.7. Формирование регулярных заливок замкнутых областей ... 252

Глава 9. Обработка календарных дат и временных интервалов 281

9.1. Дата и время в среде Visual Basic 281
9.2. Данные типа *TDateTime* в системах Borland C++ Builder и Delphi 284
9.3. Объекты, связанные с датами и временем 286
9.4. Цифровые часы 288
9.5. Биоритмы 290

Глава 10. Работа с базами данных 303

10.1. Расшифровка схемы таблиц простой БД 306
10.2. Просмотр и редактирование существующей базы данных 314
10.3. Создание базы данных с нуля 321
10.3.1. Основные возможности Database Desktop (DBD) 321
10.3.2. Создание таблицы программным путем 325
10.4. Доступ к полям таблицы и организация запросов 327
10.5. Связь с базами данных в системе Visual Basic 331

Глава 11. Отладка программ и обработка исключений 343

11.1. Отладочные средства Visual Basic 344
11.2. Отладочные средства систем BCB и Delphi 349
11.3. Исключительные ситуации и борьба с ними 355
11.3.1. Реакция на ошибки в системе Visual Basic 355
11.3.2. Обработка исключений в системах Borland C++ Builder и Delphi 359

Глава 12. Редактор условных знаков 363

12.1. Графические команды и система координат 363
12.2. Окно с графической программой 368
12.3. Включение графических команд в описание знака 369
12.3.1. Включение команды *PU* 369
12.3.2. Включение команды *PD* 369
12.3.3. Включение команды *CI* 369
12.3.4. Включение команды *ER* 370
12.3.5. Включение команды *RR* 370
12.3.6. Включение команды *AR* 370
12.3.7. Включение команды *EW* 371
12.3.8. Включение команды *WG* 371
12.3.9. Включение команды *FP* 372
12.3.10. Включение команды *EP* 372
12.3.11. Включение команд *PM0*, *PM1* и *PM2* 373
12.3.12. Включение команды *SP* 373
12.4. Работа с командами главного меню 373

12.4.1. Работа с командами раздела *Таблица* ... 374
12.4.2. Работа с командами раздела *Знак* ... 375
12.4.3. Работа с командами раздела *Подложка* ... 376
12.4.4. Работа с командами раздела *PLT-файл* ... 376
12.4.5. Преобразование векторных описаний в растровые ... 377
12.5. Пошаговое построение знака ... 377
12.6. Набор компонент и их свойства ... 378
12.7. Реализация функций редактора ... 381
12.7.1. Стартовые и финальные операции ... 383
12.7.2. Обработка кнопок инструментальной панели ... 385
12.7.3. Обработка команд раздела *Таблица* ... 389
12.7.4. Обработка команд раздела *Знак* ... 393
12.7.5. Обработка команд раздела *Подложка* ... 406
12.7.6. Обработка команд раздела *PLT-файл* ... 410
12.7.7. Обработка команды *Растр* ... 412
12.7.8. Обработка событий мыши ... 415
12.7.9. Пошаговое построение знака ... 419
12.7.10. Выполнение графических команд ... 421
12.7.11. Разное ... 436

Приложение 1. Глоссарий (компоненты и объекты визуальных сред) ... 441

Visual Basic ... 441
Borland C++ Builder, Delphi ... 442

Приложение 2. Описание дискеты ... 445

Список литературы ... 447

По системе Visual Basic ... 447
По системе Borland C++ Builder ... 448
По системе Delphi ... 448

Предисловие

Почему мы написали эту книгу

Эта книга является логическим продолжением самоучителя по алгоритмическим языкам Бейсик, Си и Паскаль, опубликованного издательством "БХВ-Петербург" в 2001 году. В нашей предыдущей работе основной упор делался на изучение наиболее популярных алгоритмических языков без особой привязки к операционной системе и конкретной среде программирования, хотя большинство примеров были ориентированы на работу под управлением MS-DOS. Основная задача новой книги — научиться создавать приложения в среде Windows 9*x*/NT/ME/2000/XP и овладеть техникой программирования в наиболее распространенных средах визуального программирования. Как и в предыдущей книге, мы стараемся подчеркнуть то, что объединяет различные визуальные среды. Поэтому мы сознательно пошли на некоторый параллелизм в описании функциональных возможностей тех или иных компонент разных систем программирования. Умение проводить аналогии между сходными по назначению структурами и программами поможет вам в дальнейшем быстрее адаптироваться к новым инструментальным средствам, которые неизбежно придут на смену сегодняшнему программному обеспечению.

Мы не ставим перед собой задачу познакомить читателей со всеми возможностями описываемых систем программирования. В рамках начального и даже более продвинутого курса по визуальному программированию это не представляется реальным — слишком велик объем информации и разнообразен круг приложений, в которых могут быть использованы те или иные стандартные средства визуальной среды. Количество компонентов, поставляемых с современными версиями систем, перешагивает границу в 150—200 элементов. А если учесть, что каждый из них может обладать полусотней свойств и обслуживаться несколькими десятками методов, то ни один учебник не в состоянии дать исчерпывающую информацию, сопроводив ее полноценными примерами на каждый случай в жизни. Из всего разнообразия компонент мы выбрали джентльменский набор, который, на наш взгляд, полезен для проектирования любого приложения. С него можно начинать

знакомство с различными свойствами и методами новых технологий. Но ни в коем случае не надо на этом останавливаться. Не делаем мы и особого акцента на выборе и сравнении различных версий визуальных сред. Большинство примеров, приводимых в книге, апробировалось в системах VisualBasic 6.0, Borland C++ Builder 5.0 и Delphi 6.0. Их несложно адаптировать к более ранним или более поздним версиям соответствующих систем программирования.

Несмотря на то, что фундаментом современных визуальных сред являются идеи объектно-ориентированного программирования (ООП) и их реализации в различных алгоритмических языках, мы не требуем от читателей непременного знакомства с ООП. Для большинства приложений, которые вам придется создавать, вполне достаточно умения использовать готовые объекты. Минимум сведений и терминов по ООП, которые будут встречаться в этой книге, занимает не более 1–2 страниц. Первая система визуального программирования Visual Basic, появившаяся в 1991 году, вообще не содержала и намека на такие понятия, как *инкапсуляция* (объединение данных и методов), *классы* (шаблоны с описанием данных и методов), *объекты* (наборы данных и методов, сформированные по шаблонам классов), *наследование* (сохранение фамильных черт в порожденных классах), *полиморфизм* (возможность создавать разные функции с одинаковыми именами). Даже такой тип данных, как *указатели* в Бейсике отсутствовал. Поэтому создание новых VBX компонентов в первых визуальных средах осуществлялось совершенно другими программными средствами, почти недоступными рядовому пользователю.

Для кого написана эта книга?

Для тех, кто изучил один из алгоритмических языков высокого уровня и собирается писать программы, функционирующие под управлением Windows. И, конечно же, хочет научиться создавать профессионально оформленные приложения Windows, пользующиеся спросом на рынке современных программных продуктов (на Западе, как и у нас — встречают по одежке).

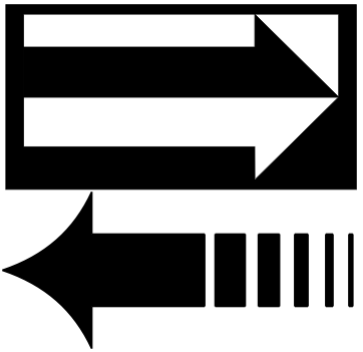
Для тех, кто владеет основами программирования в одной из визуальных сред и хочет познакомиться со спецификой работы в среде, использующей другой алгоритмический язык. Мы считаем, что профессиональный программист не должен замыкаться в рамках одного алгоритмического языка и, тем более, в рамках единственной системы программирования.

Для преподавателей информатики, осваивающих новые программные технологии.

Мы будем крайне признательны читателям за любые предложения по совершенствованию предлагаемых программ и расширению состава демонстрационных примеров.

Глава 1

Краткий обзор



1.1. Немного истории

Появление операционных оболочек типа Windows 3.*x*/95/NT существенно осложнило процесс разработки прикладных программ.

Во-первых, теперь любое приложение должно следовать канонам Windows в плане оформления пользовательского интерфейса, рабочим полем которого уже не является весь экран дисплея, допускавший в рамках MS-DOS прямую адресацию видеопамяти, а перемещаемое по экрану и изменяющее свои размеры типовое окно, снабженное большим количеством элементов управления.

Во-вторых, работа в среде Windows потребовала отказа от традиционной процедурно-ориентированной схемы, в которой последовательность действий приложения жестко запрограммирована. Прикладные программы, выполняющиеся под управлением Windows, организованы в виде набора процедур, откликающихся на адресованные им сообщения. В этом смысле их функционирование повторяет схему работы Windows, что заставляет программиста учитывать довольно большое количество типовых событий и связанных с ними форматов системных сообщений.

Снижению трудоемкости процесса создания Windows-приложений в значительной мере содействовала разработка библиотек типа *Object Windows Library (OWL)*, в которых накапливались как объекты интерфейсного типа, так и классы объектов, упрощающие реализацию типовых процедур обработки данных (графика, работа со списками и таблицами, редактирование текстов и т. п.). Несмотря на очевидный прогресс объектно-ориентированного подхода, традиционная работа с постоянно возрастающим количеством объектов и порождающих их классов требует знания многочисленных деталей — иерархии семейств, параметров и характеристик соответствующих методов и т. п.

Наиболее изящное решение указанных выше проблем, обеспечивающее минимальные затраты на создание Windows-приложений, предлагает технология визуального программирования. Визуальная среда принимает на себя все заботы по оформлению типовых окон — главное окно прикладной программы, диалоговые окна ввода строк и вывода стандартных сообщений. Она предоставляет программисту возможность без каких-либо хлопот встроить в приложение тот или иной объект, приписать ему необходимые свойства, сформировать схему поведения объекта в случае возникновения определенных событий. Прилагательное *визуальный* при этом означает, что основное большинство процедур по заготовке объектов и установке их свойств выполняются по принципу "вижу то, что делаю". Ни одного оператора в своем приложении для этих целей писать не приходится. Указав элемент в палитре компонентов, вы можете с помощью мыши "отбуксировать" объект на нужное место и изменить его размеры. Установка свойств объекта не сложнее выбора строки меню или набора нужного значения в поле ввода. Заботы по отображению установленных объектов и управлению их поведением во время работы приложения система берет на себя.

Впервые, в законченном виде визуальный подход к созданию Windows-приложений был реализован фирмой Microsoft в системе Visual Basic (сокращенно — VB), появившейся в 1992 г. и мгновенно завоевавшей симпатии программистов. В течение двух следующих лет система VB неизменно входила в состав лучших программных продуктов года. Первые три версии VB обновлялись примерно через год. Дольше других задержалась на рынке версия VB 3.0 — последняя полностью интерпретирующая система. Приложения, которые разрабатывались в этой среде, могли работать только в связке с динамически загружаемой библиотекой VBRUN*xx*.DLL (символы *xx* соответствуют номеру версии — 10, 20, 30). Интерпретация Basic-программы существенно замедляла работу приложений по сравнению с аналогичными exe-файлами, изготовленными в других системах компилирующего типа. Только последние версии VB 5.0 и 6.0 наряду с традиционной схемой создания и интерпретации p-кода допускают создание исполняемых exe-файлов. За рубежом Basic-системы пользуются огромной популярностью. По утверждению некоторых специалистов из 5 миллионов программирующих пользователей около 2 миллионов отдают предпочтение Basic. Библиотеки инструментов VB-систем (VBX- и OCX-компоненты) активно расширяются, на их базе появляются новые профессиональные пакеты и наборы инструментальных средств.

Пакет VB распространяется как самостоятельная среда программирования, так и в составе комплекса разработчика Visual Studio. Несколько огорчает решение фирмы Microsoft поставлять современные версии VB без хорошо продуманного встроенного файла помощи. Вместо этого вы должны приобрести и установить автономную справочную систему MSDN (Microsoft Developer Network Library), современная версия которой занимает 3 ком-

пакт-диска. Существует довольно много ситуаций, когда в установленной части справочных материалов не оказывается ответа на возникший вопрос, и тогда вас просят либо установить один из дисков MSDN, либо обратиться за помощью на фирменный сайт. Таким образом, ваш компьютер должен быть подключен к Интернету, и вы будете нести соответствующие расходы за использование каналов связи. Параллельно с автономно функционирующими системами VB фирма Microsoft активно продвигает урезанные версии VBA (Visual Basic for Applications — выпускается с 1993 г.), которые с 1995 г. встраиваются в такие популярные продукты, как MS Word, MS Access, MS Excel. Еще более ограниченное подмножество, известное под названием VBScript, предназначено для создания Web-приложений.

Версия VB 1.0 поставлялась всего на двух 5-дюймовых дискетах. Она могла работать под управлением Windows 3.1 на IBM-совместимых ПК с процессором 80286, оперативной памятью 1 Мбайт и занимала на винчестере всего 2 Мбайта. По мере расширения возможностей VB-систем разрастались и требования к оборудованию ПК. Начиная с версии 4.0, Visual Basic поставляется в одной из трех редакций — учебная (Learning Edition), профессиональная (Professional Edition) и промышленная (Enterprise Edition). Учебная версия Visual Basic позволяет создавать большинство приложений для MS Windows и Windows NT. Она включает все наиболее важные элементы управления, компоненты для работы с таблицами и базами данных. Профессиональная редакция включает в себя все возможности учебной версии, дополнительные компоненты ActiveX, средства разработки интернет-приложений, интегрированную базу данных и конструктор динамических страниц HTML. Промышленная редакция позволяет профессионалам создавать распределенные серверные приложения, работающие не только на IBM-совместимых компьютерах под управлением различных операционных систем.

В рамках современных операционных систем семейства Windows среда VB не выдвигает каких-либо дополнительных требования к объему оперативной памяти. Эти ограничения диктуются, главным образом, операционной системой. Как правило, чем больше оперативная память, тем быстрее работают все приложения. Однако на винчестере VB занимает от 25—35 Мбайт в минимальной установке до 80—140 Мбайт в полной установке. К этому следует добавить порядка 120—200 Мбайт для установки документации, включающей файлы помощи, учебные пособия и справочную библиотеку MSDN.

Наиболее серьезную конкуренцию продукции фирмы Microsoft составила система визуального программирования Delphi, разработанная в 1995 г. компанией Borland International на базе языка Object Pascal. Объектно-ориентированный подход к созданию компонент был серьезным шагом вперед по сравнению с чрезмерно сложной технологией разработки и модификации VBX-компонент. Дополнительный выигрыш обеспечивался за счет нормальной компиляции, обеспечивающей получение более производительных

программ. Кроме того, запросы по ресурсам у системы Delphi 1.0 были намного скромнее — процессор 80386, оперативная память от 6 до 8 Мбайт, винчестер от 30 Мбайт в минимальной установке до 80 Мбайт в полной. Первые две версии Delphi довольно быстро завоевали симпатии не только вузовских аудиторий, где Pascal пользовался особым уважением, но и среди профессионалов. Это подтолкнуло одно из подразделений фирмы Borland International на перенос визуальной технологии в среду C++. Так, почти одновременно с появлением Delphi 2.0 на рынке появилась первая версия Borland C++ 6Builder. Основу первой версии BCB составила библиотека визуальных компонент VCL (Visual Component Library), перенесенная без изменений из Delphi 2. Интерфейсы сред Delphi и BCB похожи друг на друга как близнецы, да и большая часть BCB была разработана на языке Object Pascal в среде Delphi. Благодаря своему происхождению система BCB оказалась двуязычной. Кроме своего основного языка программирования она позволяет практически без каких-либо доработок использовать формы, объекты и модули, разработанные в среде Delphi. Чтобы еще больше расширить сферу влияния среды BCB, ее авторы в последующих версиях обеспечили возможность использования библиотеки классов MFC (Microsoft Foundation Classes), разработанной фирмой Microsoft и обычно применяемой в связке с системой Visual C++.

Выход версии Delphi 3.0 в 1997 г. лишь чуть-чуть опередил появление BCB 3.0, а в последующие годы соответствующие подразделения фирмы Borland практически одновременно выдали четвертую (начало 1999 г.) и пятую (январь 2000 г.) версии обеих систем. Совсем недавно появилась шестая версия Delphi, тогда как параллельная ей версия BCB несколько задержалась.

По сложившейся традиции фирма Borland International (ее новое название — Inprise Corporation) выпускает каждую версию систем визуального программирования Delphi и BCB в нескольких редакциях, начиная с минимальной конфигурации под названием Standard.

Стандартная конфигурация позволяет создавать 32-разрядные приложения общего назначения, функционирующие под управлением Windows 95/98/NT/Me/2000/XP. Между такими приложениями могут использоваться механизмы обмена данными OLE (Object Linking and Embedding, связывание и внедрение объектов Windows) и COM (Component Object Model, компонентная модель объектов). Они могут получать доступ к некоторым локальным базам данных и готовить отчеты на базе этой информации.

Профессиональная конфигурация (Professional Edition) расширена набором средств по управлению локальными базами данных типа Access, Paradox, dBase. Она поддерживает механизм связи с базами данных *ODBC (Open Database Connectivity)* и включает некоторые объекты типа ActiveX.

Конфигурация "клиент-сервер" (Client/server Suite) расширяет возможности создания сетевых приложений, функционирующих как в памяти одного

компьютера, так и в памяти нескольких компьютеров. Приложение "Клиент" имеет возможность передать запрос на обработку данных приложению "Сервер". Сервер выполняет задание и возвращает результаты клиенту. Такая технология существенно разгружает каналы связи. Кроме того, сервер, находящийся на компьютере с высокой производительностью, может располагать более мощными средствами обработки данных и одновременно обслуживать несколько клиентов. Конфигурация "клиент-сервер" расширена набором драйверов, обеспечивающих поддержку унифицированного языка запросов к базам данных *SQL (Structured Query Language)*, средствами создания интернет-приложений с различными протоколами обмена, возможностью унифицированной обработки многомерных данных (Decision Cube). В ее состав входит сервисный набор InstallShield, упрощающий создание программы инсталляции разработанного приложения.

Самая мощная редакция BCB 5.0 называется Enterprise Edition (возможно, сказывается подражание фирме Microsoft). В эту редакцию дополнительно включены средства защиты разрабатываемых приложений и их данных, объекты и методы для создания многоплатформенных приложений, функционирующих на компьютерах разного типа под управлением соответствующих операционных систем.

Большинство из описанных выше редакций допускает 3 варианта установки, предусматривающие подключение минимальных средств системы (Compact), стандартного (Typical) и полного (Full) наборов функциональных возможностей. Выбор того или иного варианта определяется наличием свободного места на винчестере вашего компьютера. Минимальные требования к оборудованию ПК, предъявляемые различными версиями BCB, приведены в табл. 1.1. Соответствующие версии Delphi мало чем отличаются от аналогичных разработок своих партнеров.

Таблица 1.1. *Минимальные требования конфигурации ПК, предъявляемые различными версиями среды Borland C++ Builder*

Версия BCB	Процессор (минимальный)	Оперативная память: минимальная (рекомендуемая), Мбайт	Винчестер: Compact, Typical, Full, Мбайт	Версия Windows
1.0 Professional	80486	16 (24)	80, N/A, 125	95
1.0 Client/server	80486	24	N/A, N/A, 195	95
3.0 Professional	80486/100	24 (32)	125, 210, 300	95, 98, NT
3.0 Client/server	80486/100	32	N/A, N/A, 300	95, 98, NT
4.0 Professional	80486/100	32 (64)	175, 210, 300	95, 98, NT
4.0 Client/server	Pentium-90	64	N/A, N/A, 300	95, 98, NT

Таблица 1.1 *(окончание)*

Версия BCB	Процессор (минимальный)	Оперативная память: минимальная (рекомендуемая), Мбайт	Винчестер: Compact, Typical, Full, Мбайт	Версия Windows
5.0 Standard	Pentium-90	32 (64)	120, N/A, 185	95-2000, NT
5.0 Professional	Pentium-90	32 (64)	240, N/A, 360	95-2000, NT
5.0 Enterprise	Pentium-166	64	253, N/A, 388	95-2000, NT

Примечание: N/A — информация отсутствует.

1.2. Типы данных

По сравнению со своими предшественниками из MS-DOS визуальные системы программирования располагают довольно широкими возможностями по представлению данных. В этом разделе мы ограничимся краткой справкой о типах данных, используемых в системах Visual Basic, C++ Builder и Delphi. Более подробная информация будет приведена в последующих главах.

1.2.1. Основные типы данных в системе Visual Basic

Типы данных в VB-программах устанавливаются одним из четырех способов:

- явно при непосредственном объявлении тех или иных переменных, например —

```
Dim X As Integer
k As Long
F1 As String
```

- явно с помощью спецсимвола, завершающего имя переменной, например —

```
X%
k!
F1$
```

- неявно — путем определения интервала букв, с которых могут начинаться имена переменных, не объявленные явным образом, например —

```
DefInt I-N
DefStr S,V
DefDbl X-Z
```

- неявно — путем объявления без указания типа (`Dim Alpha, Beta`) или включения в программу без какого-либо объявления. В любом случае такие данные относятся к типу `Variant`.

Перечень служебных слов, определяющих типы данных, и набор соответствующих указаний о значении первого и последнего символа имени приведены в табл. 1.2.

В системе Visual Basic появились однобайтовые целые числа без знака (`Byte`), которые в ряде случаев позволят расположить в оперативной памяти массивы большего размера.

Условно целочисленные данные типа `Currency` ориентированы на обработку денежных сумм. По сути дела операции над этими данными выполняются как над целыми числами, а результат делится на 10 000. Целая часть значений типа `Currency` представляет базовые денежные единицы (рубли, доллары, марки и др.), а дробная — мелкие (копейки, центы, пфенниги и др.).

Basic дольше других алгоритмических языков сопротивлялся введению логических переменных, хотя неявно использовал для этой цели нулевые (`False`) и ненулевые (`True`) целые числа. Теперь логические переменные вступили в свои законные права, и в программах можно использовать обычные логические выражения над логическими операндами.

Для решения задач, использующих календарные даты, предлагаются данные типа `Date`, подробно описанные в *главе 9*.

Строки фиксированной длины объявляются с указанием точного количества символов, составляющих значение строки:

```
Dim S1 As String *20
```

Если присваиваемое значение содержит меньше 20 символов, то оно дополняется пробелами. При попытке присвоить переменной `S1` слишком длинную цепочку символов, лишний хвост будет отрезан. Если в операторе `Dim` указание о максимальной длине отсутствует, то реальным ограничением сверху может стать лишь объем оперативной памяти.

Самым интересным изобретением системы Visual Basic являются универсальные данные типа `Variant`. Переменным этого типа можно присваивать значения любых типов кроме записей. Переменные типа `Variant` хранят информацию о типе и месте расположения текущего значения. Иногда текущее значение находится внутри поля, отведенного под переменную типа `Variant`. Как правило, это относится к значениям числового типа. В случае более сложных переменных, длина которых может изменяться во время работы программы, на поле переменной хранится ссылка на соответствующее значение. Значением переменной типа `Variant` может быть даже массив. Естественно, что операции над такими данными реализуются специальными подпрограммами, к большинству из которых пользователь напрямую не обращается. Более точно, переменные типа `Variant` представляют собой объекты класса данных, представленных связкой (тип, значение). Над этими данными определен довольно большой набор операций, методов и преобразований. И хотя универсальные данные позволяют делать много необычного,

хранение таких значений связано с дополнительными затратами оперативной памяти, да и скорость выполнения операций оставляет желать лучшего.

Переменные типа `Variant` можно объявить и явно с указанием типа:

```
Dim Delta As Variant
```

Данные, представленные последовательностью полей со значениями фиксированного типа и больше известные под названиями структуры (терминология C) или записи (терминология Pascal), относятся к пользовательским типам данных (*user-defined type*). Описание шаблона структуры с именем `Book` в VB-программе может иметь вид:

```
Type Book
  Title As String *40
  Author As String *20
  Year As Integer
  Price As Single
End type
```

После такого описания можно объявить, например, переменные `b1` и `b2`, имеющие структуру типа `Book`, и присвоить соответствующим полям нужные значения:

```
Dim b1 As Book, b2 As Book
b1.Title="Visual Basic 6.0: Разработка приложений"
b1.Author="А. Гарнаев"
b1.Year=2001
b1.Price=135.5
```

Массивы данных любого типа помимо памяти, занимаемой каждым элементом массива, требуют дополнительно 20 байт на общую шапку и по 4 байта на каждое измерение. Например, массив `A`, объявленный как `Dim A(5,12) As Double`, занимает $6 \times 13 \times 8 + 20 + 2 \times 4 = 652$ байта. В табл. 1.2 описаны основные типы данных, принятые в среде Visual Basic 6.0.

Таблица 1.2. *Описание основных типов данных среды Visual Basic 6.0*

Тип данных	Длина, байт	Диапазон допустимых значений	По первой букве	Спец-символ
Целые числа				
`Byte`	1	от 0 до 255	DefByte	
`Integer`	2	от –32 768 до 32 767	DefInt	%
`Long`	4	от –2 147 483 648 до 2 147 483 647	DefLng	&
`Currency` (целые×10-4)	8	от –922337203685477.5808 до +922337203685477.5807	DefCur	@

Таблица 1.2 *(окончание)*

Тип данных	Длина, байт	Диапазон допустимых значений	По первой букве	Спец-символ
Вещественные числа				
`Single`	4	от -3.4×10^{38} до -1.4×10^{-45} и от 1.4×10^{-45} до 3.4×10^{38}	DefSng	!
`Double`	8	от -1.8×10^{308} до -4.9×10^{-324} и от 4.9×10^{-324} до 1.8×10^{308}	DefDbl	#
Строки				
`String` (фикс. длина)	n	n — объявленная длина строки (n <64 k)		$
`String` (перем. длина)	10+m	m — текущая длина строки ($m<2^{31}-1$)	DefStr	$
Логические данные				
`Boolean`	2	`True` или `False`	DefBool	
Календарные даты				
`Date`	8	от 1.01.100 до 31.12.9999	DefDate	
Пользовательский тип данных (структуры, записи)				
`Type`	Сумма длин полей	Длина поля зависит от его типа		
Указатели (ссылки)				
`Object`	4	Ссылка (адрес) на объект	DefObj	
Универсальный тип данных				
`Variant` (число)	16	Любое число из диапазона `Double`	DefVar	
`Variant` (строка)	22+m	m — текущая длина строки	DefVar	

Для явного объявления типа, кроме оператора `Dim`, могут использоваться и другие операторы, например `Private` (при объявлении переменных, недоступных другим модулям), `Public` (при объявлении переменных,

доступных другим модулям), `Static` (при объявлении локальных переменных, сохраняющих свои значения после выхода из процедуры).

Указание о типе данных (`As ...`) можно встретить при объявлении типа констант, типа параметров функций и подпрограмм, типа возвращаемого значения:

```
Const e As Single = 2.718
Function Square(x As Double) As Double
```

Последние версии Visual Basic сделали попытку приблизить Basic к общепринятому правилу других алгоритмических языков — обязательному описанию всех используемых переменных. Если в начале модуля встречается оператор `Option Explicit`, то все переменные данного модуля должны быть объявлены явно. Система может автоматически добавлять указанный оператор, если в окне **Options** (Опции) (вызывается из меню **Tools** (Инструменты)) на вкладке **Editor** (Редактор) (рис. 1.1) помечен режим **Require Variable Declaration** (Требуется Объявление Переменных).

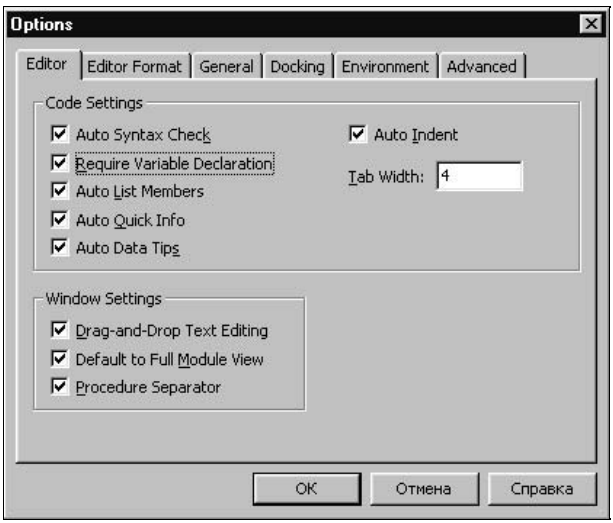


Рис. 1.1. Режим обязательного описания переменных в VB

1.2.2. Основные типы данных в системах BCB и Delphi

В программах на C и Pascal все переменные должны быть объявлены. Причем Pascal предъявляет гораздо более жесткие требования — описание переменных должно предшествовать первому исполняемому оператору программы. Язык C++ более либерален — он позволяет объявлять переменные по ходу программы, но использует специальное соглашение по поводу

области видимости (доступности) таких переменных. Например, переменная `j`, объявленная в цикле, после выхода из цикла становится недоступной:

```
for(int j=0, s=0; j<n; j++)  s=s+a[j];
```

Зоной видимости переменной, объявленной внутри функции C, является блок, в котором такое описание появилось. Это не согласуется с тем, что было реализовано в системе Borland C++ 3.1. Там зона видимости переменной распространялась и на ту часть функции, которая находилась ниже блока. Но в BCB стандарт C++ соблюдается более четко.

Принципиальной новинкой в Delphi является возможность задания начального значения переменной в строке объявления. Раньше Pascal допускал такое присвоение только в разделе `const`, и из книги в книгу кочевали такие несуразицы как *типизированные константы*. На самом деле, это были обычные переменные, но уж слишком долго Pascal не разрешал того, что было принято в большинстве алгоритмических языков. Кстати, Visual Basic пока еще до этого удобного способа не созрел.

По сравнению с системой Visual Basic целочисленные данные в BCB и Delphi в каждом из форматов (1, 2 и 4 байт) допускают по два варианта представления — со знаком и без знака (табл. 1.3). Кроме того, здесь появились целочисленные данные учетверенной длины (8 байт), которые немного отличаются от данных типа `Currency`.

Диапазон вещественных данных по сравнению с Visual Basic здесь расширен за счет 10-байтовых чисел с плавающей точкой, имеющих до 19—20 верных значащих цифр.

Логические значения, похожие на общепринятые `true` или `false`, а также логические операции, в том числе и поразрядные, в C были известны с самого начала. Однако в качестве логических значений выступали данные целочисленного типа или их отдельные биты, а знаки общепринятых логических операций (`OR`, `AND`, `XOR`, `NOT`) были заменены очень специфической символикой (`|`, `&`, `^`, `!`, `~`, `&&`, `||`). Появление данных типа `bool` приблизило C к общепринятой схеме обозначений, хотя и не внесло ничего принципиально нового.

Данные типа `Currency` перекочевали из входного языка VB сначала в систему Delphi, а потом и в BCB. Над денежными единицами в классе `Currency` определены все арифметические операции, свойственные целочисленным данным в языке C:

`+, +=, ++, -, -=, --, *, *=, /, /=, %, %=`

Данные типа `Currency` можно сравнивать (`>, >=, <, <=, ==, !=`). Им можно присваивать любые целочисленные или вещественные значения из допустимого интервала. Для преобразования денежных единиц в строку программа может воспользоваться функцией `AnsiString`:

```
Currency s=s1+s2;
ShowMessage(AnsiString(s));
```

Множества впервые появились в Pascal. В соответствующий класс BCB их идеология перенесена в несколько урезанном виде. Элементами класса `Set` могут быть данные типа `int`, `char` или `enum`, принадлежащие ограниченному интервалу от минимального значения (`min`) до максимального значения (`max`) и содержащие не более 256 элементов. Минимальное значение должно быть не менее 0, а максимальное — не более 255.

Объявить множество можно явно или косвенно:

```
Set <char,'A','Z'> s1;     //множество s1 больших букв от A до Z
typedef Set <int, 10,20> Num10_20;
Num10_20 s2,s3;            //два множества целых чисел от 10 до 20
enum M_K_Key (ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
              ssMiddle, ssDouble); //управляющие клавиши
typedef Set<M_K_Key,ssShift,ssDouble> TShiftState;
TShiftState Shift;         //множество управляющих клавиш
```

После объявления любое множество является пустым и его заполнение возлагается на программу. Подключение элемента из допустимого интервала осуществляется с помощью операции `<<`:

```
s1 << 'A'<< 'C';
s2 << 12 << 10 << 19;
MessageDlg(as,mtInformation,TMsgDlgButton<<mbOk<<mbCancel,0);
```

Операция `>>` используется для исключения элемента из множества:

```
s2 >> 19;
```

С помощью метода `Clear` можно удалить все данные из множества:

```
s1.Clear();
```

Для проверки на принадлежность элемента множеству предназначен метод `Contains`:

```
bool b1=s1.Contains('C');
```

Он возвращает значение `true`, если указанный элемент в данный момент принадлежит множеству.

Над элементами двух множеств определены стандартные операции объединения (операция `+` или `+=`), пересечения (операция `*` или `*=`) и вычитания (операция — или `-=`). Два множества можно сравнивать на совпадение (операция `==`) или несовпадение (операция `!=`).

Данные типа `Variant`, впервые появившиеся в VB, были предназначены для обмена разнотипной информацией. При каждом таком значении хранятся сведения о его типе, а класс `Variant` снабжен сервисными программами по преобразованию данных одного типа в другой. В качестве значений типа `Variant` система BCB допускает использование практически всех числовых данных (`short, int, float, double, Currency, TDateTime`), логических

величин (bool, WordBool), указателей на строки (char *, AnsiString&, wchar_t *) и объекты типа *OLE Automation*. В последнем случае значение типа Variant может использоваться для опроса/установки свойств OLE-объекта или вызова его методов.

В табл. 1.3 дается описание основных типов принятых в средах Delphi 6.0 и Borland C++ Builder.

Таблица 1.3. *Описание основных типов данных сред Delphi 6.0 и Borland C++ Builder 5.0*

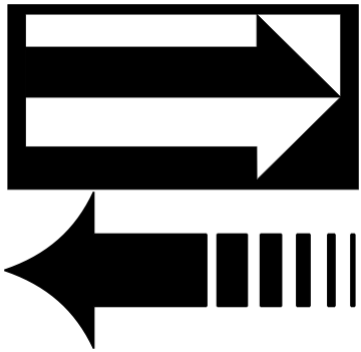
Тип Delphi	Тип BCB	Диапазон	Формат
Целые числа			
ShortInt	signed char	от –128 до 127	8 бит со знаком
Byte	unsigned char	от 0 до 255	8 бит без знака
SmallInt	short	от –32 768 до 32 767	16 бит со знаком
Word	unsigned short	от 0 до 65 567	16 бит без знака
Integer, Longint	integer, long	от –2147483648 до 2147483647	32 бита со знаком
Cardinal	unsigned int	от 0 до 4294967295	32 бита без знака
Int64	int64	от -2^{63} до $2^{63}-1$	64 бита со знаком
Currency	Currency	от –922337203685477.5808 до +922337203685477.5807	64 бита со знаком ($\times 10^{-4}$)
Вещественные числа			
Single, Real	float	от -3.4×10^{38} до -1.4×10^{-45}, 0 и от 1.4×10^{-45} до 3.4×10^{38}	32 бита, пл. зпт.
Double	double	от -1.8×10^{308} до -4.9×10^{-324}, 0 и от 4.9×10^{-324} до 1.8×10^{308}	64 бита, пл. зпт.
Comp	Comp	от -2^{63} до $2^{63}-1$	64 бита, пл.зпт.
Extended	long double	от $-1.1*10^{4932}$ до -3.4×10^{-4932}, 0 и от 3.4×10^{-4932} до 1.1×10^{4932}	80 бит, пл. зпт.
Логические данные			
Boolean	bool	true или false	8 бит
ByteBool	unsigned char	true/false или целое без знака	8 бит
WordBool		true/false или целое без знака	16 бит
LongBool	BOOL	true/false или целое без знака	32 бита

Таблица 1.3 *(окончание)*

Тип Delphi	Тип BCB	Диапазон	Формат
Символьные (строковые) данные			
`Char, AnsiChar`	`char`	8-битовый символ	8 бит
`WideChar`	`wchar_t`	16-битовый символ (Unicode)	16 бит
`String[n]`	`SmallString<n>`	Строка до 255 символов	
`String, AnsiString`	`AnsiString`	Длинная строка до 2 Гбайт	
`PChar`	`unsigned char *`	Указатель на строку	32 бита
Указатель без типа			
`Pointer`	`void *`		32 бита
Календарные даты и время			
`TDateTime`	`TDateTime`		64 бита

Глава 2

С чего начать



2.1. Знакомство с IDE

Интегрированная среда разработки IDE (Integrated Development Environment) — это рабочее место и набор инструментов, участвующих в создании программы. IDE в MS-DOS по сравнению с IDE в Windows выглядит как золушка рядом с принцессой, хотя функциональное назначение у них одно и то же. Более того, вы увидите много знакомых слов в обозначениях разделов и команд главного меню и встретите похожие термины — окно редактора программы, окна отладчика, контекстная помощь и др. Однако в графической среде Windows все это выглядит намного красивее. Идеи визуального программирования пополнили процесс построения программ технологией сборки, напоминающей игру с детским конструктором. Здесь даже прижилась инженерная терминология — *проектирование (конструирование) программы.* А в названиях отдельных окон сплошь и рядом фигурируют дизайнеры, менеджеры и тому подобные профессии.

Очутившись в IDE, вы вправе перемещать составные части среды, изменять их структуру, размеры и цвет, поступая примерно так же, как новосел, обустраивающий свою квартиру. Злоупотреблять этим правом особенно на первых порах не стоит, однако по мере освоения тех или иных возможностей системы вы настроите IDE по своему вкусу. А сначала попробуем разобраться в главных элементах IDE.

Основные элементы визуальной среды появляются на экране дисплея после запуска соответствующей среды программирования (рис. 2.1 и 2.2). Скрытые и вспомогательные окна, количество которых доходит до десятка, могут быть вызваны по мере необходимости.

В разных версиях и редакциях визуальной среды эти картинки могут отличаться от приведенных, да и пользователь вправе изменить расположение и размеры тех или иных элементов. Однако, главное не в этом. Наиболее

важные компоненты, всегда участвующие в конструировании приложения, в рассматриваемых средах имеют функционально сходное назначение и достаточно близки по внешнему виду. В их составе:

- главное меню среды с панелью быстрых кнопок для исполнения наиболее часто используемых команд;
- форма конструируемого приложения;
- хранилище визуальных компонентов, используемых при создании приложения;
- окно компонентов, включенных в состав приложения, и значения их свойств.

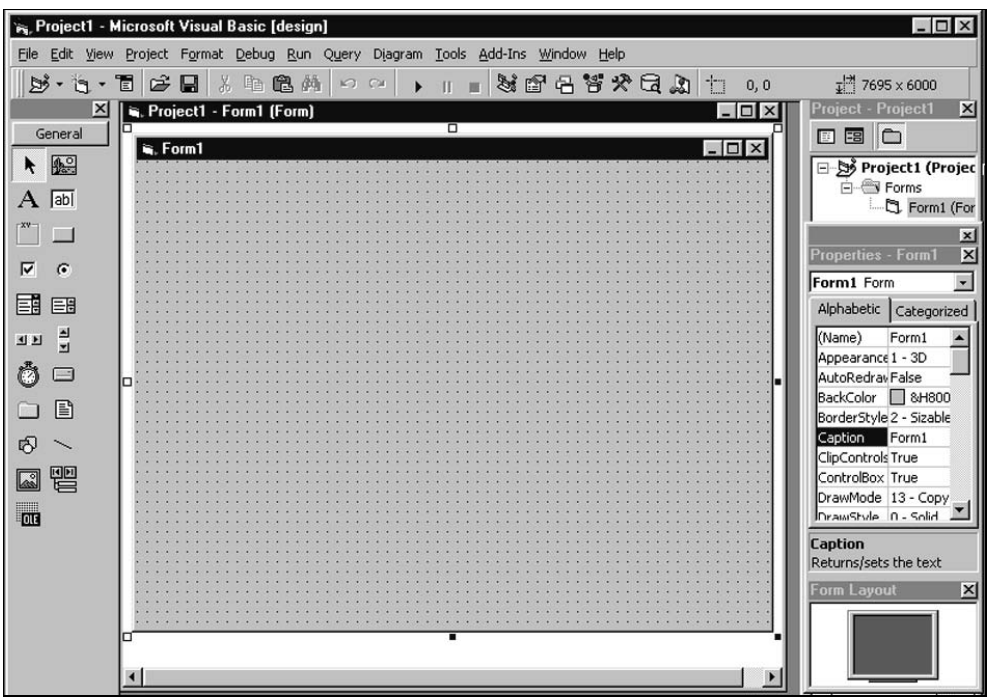


Рис. 2.1. Общий вид визуальной среды VB-6.0

Одним из наиболее важных элементов визуальной среды является окно редактора программы (в большинстве книг его принято называть окном редактора кода). Обычно это окно закрыто формой создаваемого приложения, и появляется оно на переднем плане в тот момент, когда пользователь вызывает его соответствующей командой меню или щелчком мыши сообщает о своем желании набрать текст программы обработки того или иного события. Внешний вид окна редактора программы среды VB приведен на рис. 2.3. Оно немного отличается от аналогичного редактора кода в среде

BCB/Delphi (рис. 2.4) тем, что нагружено двумя выпадающими списками, используемыми для переключения на тот или иной компонент приложения и указания интересующего нас события.

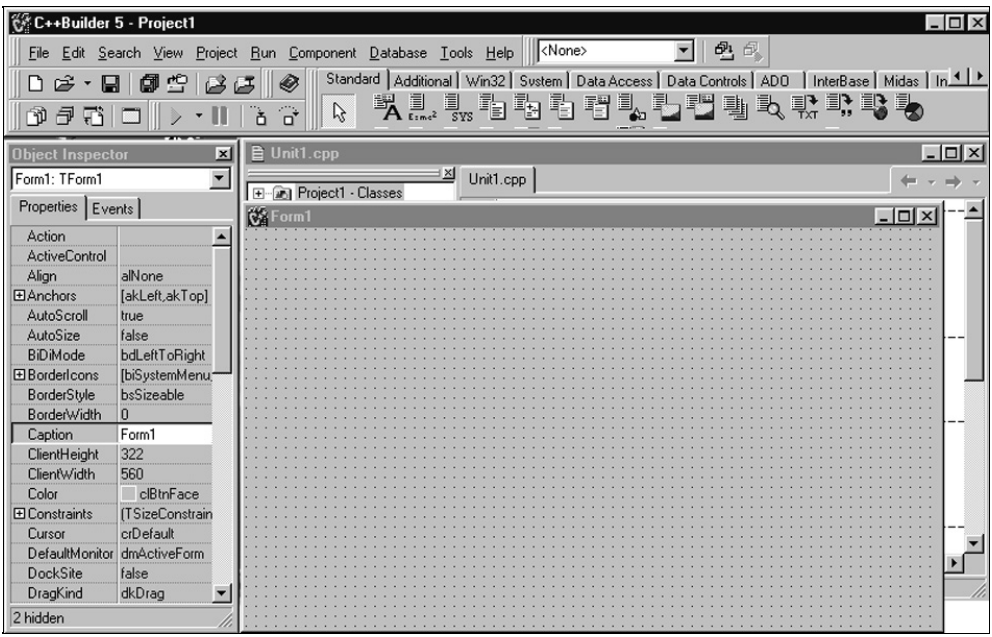


Рис. 2.2. Общий вид визуальной среды BCB/Delphi

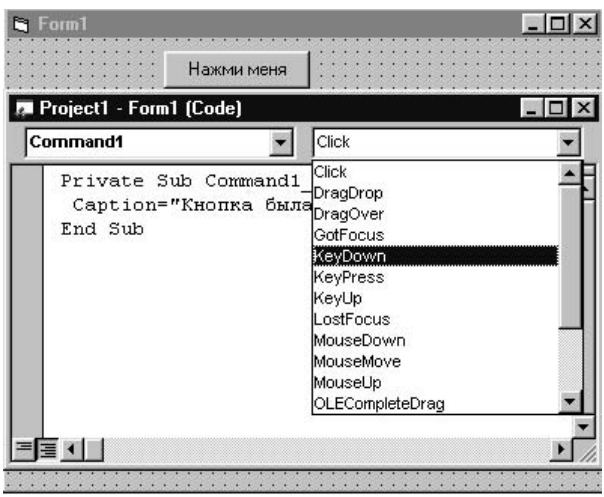


Рис. 2.3. Окно редактора программы в среде VB-6.0

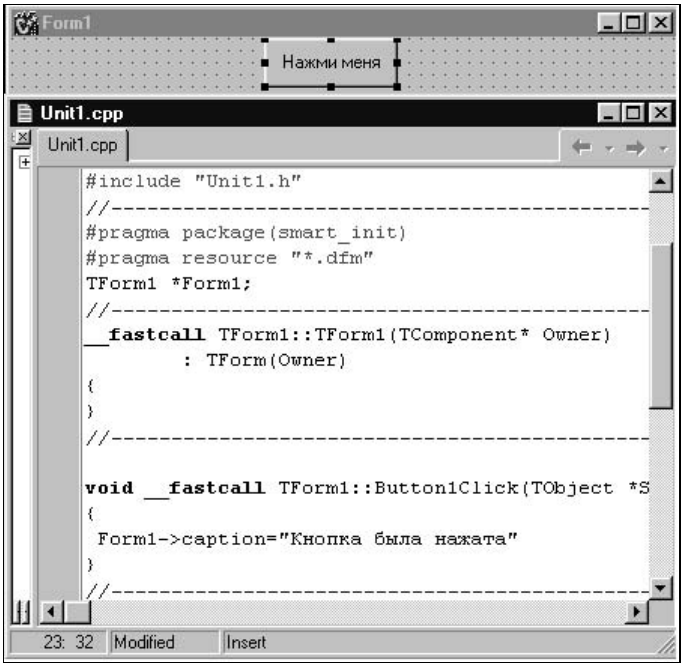


Рис. 2.4. Внешний вид окна редактора программы в среде BCB

Работа в поле редактора программы мало чем отличается от техники набора текстов в таком общеизвестном офисном пакете как MS Word. На поле редактора можно вызвать любой текстовый файл и запомнить отредактированный документ на диске. Здесь можно выделять, вырезать и копировать фрагменты, использовать *Clipboard*. Однако нет необходимости манипулировать параметрами шрифтов. Определенное количество строк в текст программы система вставляет автоматически. Без крайней необходимости не надо такие строки удалять. Может оказаться, что вы набрали текст обработки не интересующего вас события. В этом случае можно удалить набранный вами текст, но не трогайте окаймляющие строки, которые появились автоматически. Получится пустая процедура (функция), которую система удалит при очередном сохранении текста программы.

Одним из приемов работы в визуальной среде является использование готовых компонент и настройка их свойств под нужды нашего приложения. Хранилище компонентов, из которого можно извлечь нужный элемент управления и поместить его на форме нашего будущего приложения, в средах VB, BCB/Delphi устроено по-разному.

Visual Basic предпочитает не загромождать экран пиктограммами всех своих компонент и разместил порядка 20 наиболее употребительных элементов в окне с заголовком **General** (Общие), расположенным в левой части экрана.

Однако общее количество компонент, включенных в профессиональную редакцию VB 6.0, достигает 200. Для подключения нужного объекта из общего хранилища в его видимую часть необходимо войти в позицию главного меню **Project** (Проект) и выполнить команду **Components** (Компоненты). В появившемся диалоговом окне (рис. 2.5) на вкладке **Controls** (Элементы управления) выбрать строку с обозначением интересующего вас элемента или группы элементов (на рисунке выбрана строка с компонентом **Masked Edit**) и щелчком мыши установить галочку в соответствующем квадратике слева. После нажатия кнопки **OK** новый компонент дополнит стандартный набор. Этим набором сможет пользоваться только данное приложение, в противном случае вновь добавляемые к разным приложениям компоненты будут засорять экран. На рис. 2.6 слева расположено стандартное окно **General** (Общие), а справа — оно же, пополненное компонентом **Masked Edit** (Редактирование по маске).

Следует отметить, что авторы VB сохранили вид пиктограмм главных элементов управления такими, какими они появились в первой версии. Это большой плюс в преемственности поколений программного обеспечения.

Авторы Delphi и BCB пошли по другому пути. Все компоненты, зарегистрированные в системе, вынесены на полоску с большим числом вкладок. Каждая из вкладок выдвигает на передний план линейку с пиктограммами компонент соответствующего раздела. В ранних версиях такая компоновка была достаточно наглядной, т. к. количество элементов раздела редко превышало 8–9. Однако в современных версиях общее количество компонент достигает 150–200, и на каждой планке пришлось добавить кнопки горизонтальной прокрутки, чтобы добраться до нужного элемента (рис. 2.7 и 2.8). В отличие от авторов VB создатели более поздних версий BCB пошли на изменение вида пиктограмм, что создает дополнительные неудобства для пользователей.

Перенос любой компоненты из ее хранилища на форму будущего приложения в визуальных средах производится одинаково. Самый быстрый способ заключается в двойном щелчке по пиктограмме выбранной компоненты. После этого аналогичное или эквивалентное изображение появляется в центре нашей формы. Остается только с помощью мыши растянуть элемент до нужных габаритов и "отбуксировать" на предполагаемое место жительства. Второй способ заключается в одинарном щелчке по нужной пиктограмме в хранилище с целью выделить компонент. Затем курсор мыши переводится в нужную точку формы, зажимается клавиша мыши, и изображение элемента растягивается до нужных размеров. Существуют и другие способы размножения элементов, уже выбранных на форму с использованием буфера обмена (Clipboard).

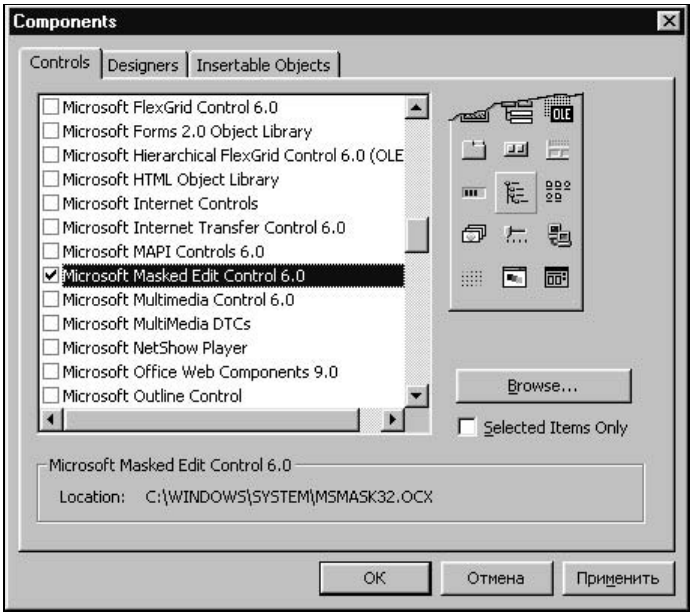


Рис. 2.5. Подключение компонента из общего хранилища к окну **General**

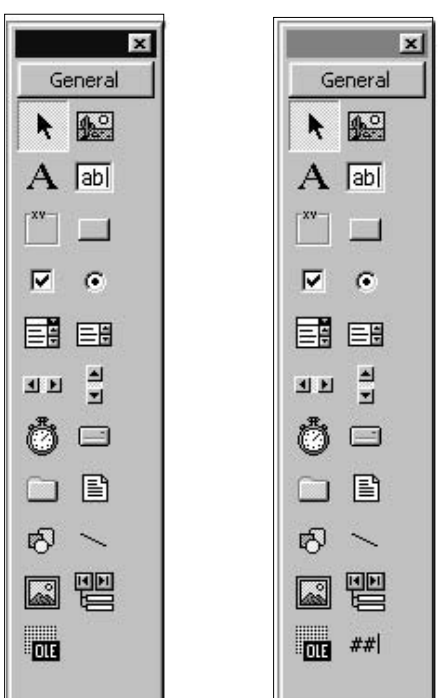


Рис. 2.6. Окна компонент VB

Рис. 2.7. Палитра компонент в среде BCB 5.0

Рис. 2.8. Палитра компонент в среде Delphi 6.0

Визуальная среда автоматически присваивает системные имена формам и объектам, включаемым в состав приложения, — `Form1`, `Form2`, `Button1`, `Button2`, ... Числовой индикатор, присоединяемый системой вслед за стандартным наименованием, отражает порядок появления того или иного объекта данного типа на конкретной форме. Чтобы отличить кнопку `Button1`, размещенную на форме `Form1`, от кнопки с тем же именем, находящейся на форме `Form2`, в программе следует использовать составные имена:

```
Form1.Button1, Form2.Button1 (в системах VB и Delphi)
Form1->Button1, Form2->Button1 (в системе BCB)
```

Если в нашем приложении используется единственная форма, присоединять ее имя к имени компонента не надо, чтобы не создавать себе лишних хлопот по набору длинных имен. Более того, находясь в редакторе программы для формы `Form1`, добавку префикса "`Form2`" к имени компонента следует делать только тогда, когда мы обращаемся к объекту, расположенному на другой форме.

Главным местом для манипуляции с объектами является окно, в котором отображается список свойств объектов, расположенных на форме нашего приложения. В среде VB такое окно имеет заголовок **Properties** (Свойства) (рис. 2.9). Среды BCB и Delphi предлагают для этой цели окно **Object Inspector** (Инспектора объектов — рис. 2.10). Под заголовком каждого окна находится выпадающий список с перечнем объектов, размещенных на нашей форме. Окно **Object Inspector** (Инспектор объектов) помимо свойств вкладка **Properties** (Свойства) содержит вкладку с набором допустимых событий (в среде VB такой список совмещен с редактором программы).

В среде VB список свойств может быть упорядочен по алфавиту (**Alphabetic**) или сгруппирован по категориям (**Categorized**). В средах BCB/Delphi используется только упорядоченный по алфавиту список свойств. Правая половина страницы свойств отображает их текущие значения, которые можно изменять. Если свойство относится к разряду числовых, то новое значение просто набирается.

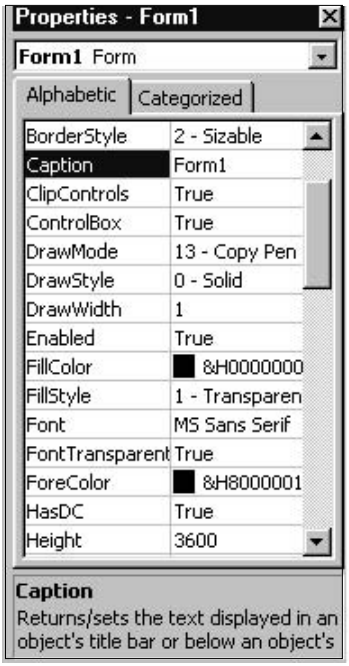


Рис. 2.9. Окно свойств VB

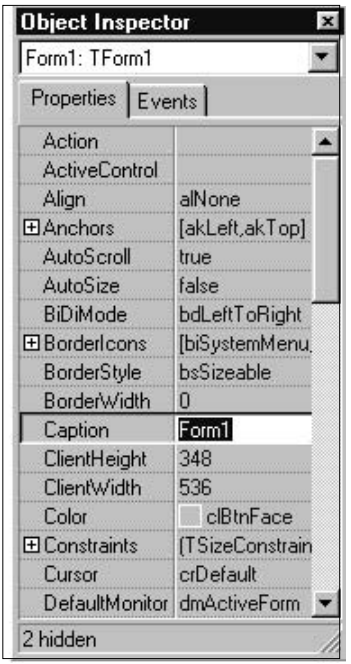


Рис. 2.10. Инспектор объектов BCB

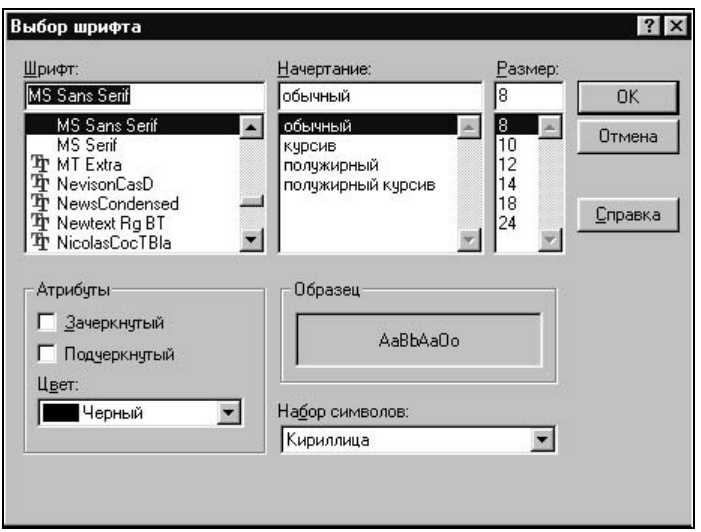


Рис. 2.11. Диалоговое окно для установки параметров шрифта

Большинство свойств может принимать одно из допустимого множества значений, и тогда после щелчка по полю значения открывается выпадающий список, в котором можно выбрать требуемую величину. В средах

BCB/Delphi слева от имени свойства может находиться знак плюс, щелчок по которому открывает расширенный список свойств. Щелчок по правой половине строки, помеченной плюсом, может привести к появлению кнопочки с многоточием. Повторный щелчок по такой кнопке вызывает диалоговое окно, в котором более удобно задать значения нужных характеристик. Примером такого выбора может служить окно установки параметров шрифта (рис. 2.11).

Мы надеемся, что большинство читателей достаточно часто пользовались аналогичным окном при работе в MS Word, так что особых пояснений здесь не требуется.

2.2. Общие свойства объектов визуальных сред

И форма, и размещаемые на ней компоненты являются объектами, т. е. некоторыми наборами данных, которые укомплектованы обслуживающими их функциями и процедурами. По терминологии объектно-ориентированного программирования (ООП) последние принято называть *методами*. Единственное отличие методов от традиционных функций и процедур заключается в способе обращения:

`y=f(x1,x2,…)`	обращение к обычной функции `f` в VB;
`y=object.f(x1,x2,…)`	обращение к методу `f` в VB;
`y:=object.f(x1,x2,…);`	обращение к методу `f` в Delphi;
`y=object→f(x1,x2,…);`	обращение к методу `f` в BCB.

Каждый объект обладает определенным набором свойств, влияющих на поведение объекта и на его внешний вид. В первом приближении эти свойства можно рассматривать как специфические переменные, значения которых можно опросить или изменить. Так как функционально одинаковые свойства у разных объектов имеют одинаковые идентификаторы, то для доступа к нужному свойству приходится применять составные имена:

```
wf:=Form1.Width;      {опрос ширины формы Form1}
Button1.Width:=20;    {установка ширины кнопки Button1}
```

Некоторые методы, к которым пользователь не имеет доступа, отвечают за отображение объекта на экране монитора с учетом тех свойств, которые установлены пользователем или заданы программным путем. Изменения отдельных свойств на стадии проектирования сразу же изменяют внешний вид объекта.

Объекты, размещаемые на форме нашего приложения, можно условно разбить на две группы. Первую группу составляют потенциально видимые объекты, размеры которых поддаются изменению. Во время работы приложения они могут сохранять свою видимость. Ко второй группе относятся объекты,

пиктограмма которых сохраняет фиксированные размеры при любых попытках изменить их габариты с помощью мыши. Во время работы приложения такие объекты не оставляют на форме каких-либо видимых следов.

Объект любой группы обладает свойством `Name`. Мы уже упоминали имена форм (`Form1`, `Form2`) и кнопок (`Button1`, `Button2`), которые система присваивает объектам автоматически в момент их появления в приложении. Как правило, в приложениях целесообразно изменять имена компонент с тем, чтобы программное имя объекта отражало его функциональную направленность. И такое изменение лучше делать сразу, иначе потом придется заниматься поисками в программе тех мест, которые должны быть изменены.

Перечень общих свойств, присущих большинству потенциально видимых объектов, приведен в табл. 2.1. В этой таблице впервые упомянут термин *контейнер*. Для объектов, размещаемых в произвольных местах формы, роль контейнера выполняет сама форма. Это означает, что объект привязан к фиксированному месту формы и любые ее перемещения по экрану автоматически переносятся на "приклеенные" к форме объекты. Некоторые объекты (например, группу логически связанных кнопок) удобно поместить в какой-то внутренний контейнер, который можно было бы перемещать по форме вместе с содержащимися в нем компонентами. В любом случае привязка объекта к контейнеру производится в локальной системе координат контейнера, начало которой расположено в левом верхнем углу. Как и на экране, ось `x` направлена вправо, а ось `y` — вниз. В системах BCB и Delphi координаты измеряются в пикселах, физический размер которых определяется разрешением экрана монитора. Авторы VB остановились на метрических единицах — *твипах* (**twips**), размер которых не зависит от разрешения монитора. В одном дюйме (дюйм = 2.54 см) размещается 1 440 твипов, в одном сантиметре — 567 твипов. При разрешении 800×600 одному пикселу 15-дюймового монитора соответствует примерно 15 твипов. Размер экрана в твипах при этом составляет 12 000×9 000. В среде Visual Basic предусмотрена возможность для перехода к другим единицам измерения. Пересчетом ширины и высоты из твипов заведует свойство `ScaleMode`, значение которого по умолчанию равно 1. При `ScaleMode=3` значения высоты и ширины объекта в пикселах определяются свойствами `ScaleHeight` и `ScaleWidth`. Вы имеете возможность выбрать в качестве единиц длины дюймы (`ScaleMode=5`), миллиметры (`ScaleMode=6`) или сантиметры (`ScaleMode=7`). Однако в любом случае свойства `Height` и `Width` хранятся в твипах. В табл. 2.1 приведены некоторые свойства объектов и их функциональное назначение.

Признак `Tag` (произносится — "тэг") не оказывает никакого влияния ни на внешний вид объекта, ни на его поведение. Этим признаком, который представлен четырехбайтовым целочисленным значением, программа может распоряжаться по своему усмотрению — заносить туда какие-то числа, проверять значение свойства и принимать по этому поводу те или иные решения в зависимости от выбранного вами алгоритма.

Таблица 2.1. *Свойства объектов*

Свойство	Функциональное назначение
`Left`	Смещение левой границы объекта от левой границы контейнера
`Top`	Смещение верхней границы объекта от верхней границы контейнера
`Width`	Ширина объекта
`Height`	Высота объекта
`Tag`	Признак, используемый по усмотрению программиста
`Visible`	Логический признак видимости (`True`) или невидимости (`False`) объекта
`Enabled`	Логический признак доступности (`True`) или недоступности (`False`) объекта

Свойство `Visible` может спрятать потенциально видимый объект так, что на экране его изображение пропадет. Но это не означает, что объект будет бездействовать. Он не потеряет активность и, если такому объекту будет адресовано предусмотренное сообщение, на него последует запрограммированная реакция. В отличие от этого свойство `Enabled` (дословно — разрешен) управляет доступом к объекту. Объект может быть виден, но запрещен (`Enabled=False`). В этом случае вместо четких черных очертаний контура компонента вы увидите бледно-серый силуэт, и сообщение, адресованное запрещенному объекту, не будет передано обработчику соответствующего события.

Современные версии визуальных сред позволяют снабжать каждый компонент динамически появляющейся подсказкой в момент наезда курсором мыши на территорию объекта во время работы приложения. В среде VB достаточно занести в свойство `ToolTipText` нужную подсказку. В средах BCB и Delphi содержанием подсказки является значение свойства `Hint`. Однако для ее демонстрации необходимо установить значение еще одного свойства `ShowHint=True`. На рис. 2.12 показано, как выглядит подсказка в VB (слева) и в Delphi (справа).

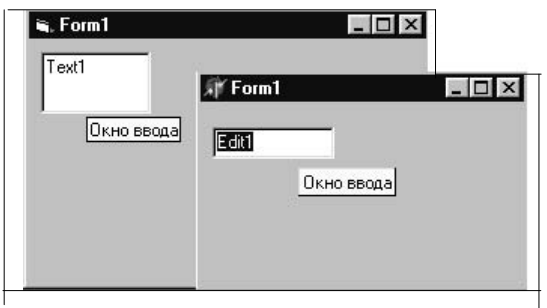


Рис. 2.12. Отображение подсказки во время работы приложения

Многие компоненты Delphi и BCB могут воспользоваться некоторыми свойствами того контейнера, на территории которого они находятся. Хотя контейнер выступает в роли родителя (свойство `Parent`) содержащихся в нем объектов, мы будем пользоваться термином *опекун*. Причиной такого отступления от точного перевода является нежелание смешивать идею опекунства с понятиями родительского класса и порожденного им класса, принятыми в объектно-ориентированном программировании. В последнем случае имеет место родство "по крови" и дочерние объекты могут унаследовать данные и методы своего родителя. В случае *опекунства* временные квартиранты имеют право на использование шрифта, цвета, разрешения на демонстрацию подсказки и свойства объемности. Для этого достаточно присвоить значение `True` таким свойствам как `ParentFont`, `ParentColor`, `ParentShowHint` и/или `ParentCtl3D`. Последним свойством приходится пользоваться редко, т. к. в большинстве современных систем программирования все объекты выглядят объемными, хотя это сопряжено с несколько большими затратами по ресурсам. Смена шрифта опекуна обходится гораздо дешевле, чем изменение аналогичного свойства у нескольких объектов, попавших в контейнер.

Свойства `TabOrder` и `TabStop` присущи объектам, которые могут быть активизированы с помощью клавиши <Tab> или сочетанием клавиш <Shift>+<Tab>. При создании приложения каждый такой объект получает порядковый номер `TabOrder`, определяемый очередностью извлечения объектов из хранилища (нумерация ведется от 0). Если в режиме проектирования нажимать клавишу <Tab>, происходит автоматическое выделение очередного объекта в порядке возрастания их номеров. Комбинация <Shift>+<Tab> выполняет эту же процедуру в обратном порядке. Для того чтобы исключить тот или иной объект из очереди активизируемых компонент, достаточно присвоить свойству `TabStop` значения `False` (по умолчанию оно равно `True`). Во время работы приложения действия, вызываемые нажатием клавиши <Tab> или сочетания клавиш <Shift>+<Tab>, эквивалентны последовательной передаче фокуса очередному объекту. Для кнопок такое выделение можно заметить по пунктирному прямоугольнику вокруг надписи, для окон ввода — по появлению мигающего курсора.

2.3. Главная форма

В простейшем Windows-приложении вполне достаточно одной формы — той самой, которая появляется при запуске визуальной среды и имеет заголовок `Form1` (рис. 2.13).

У нее уже есть минимальный набор элементов управления — строка заголовка, на которой расположены название формы и четыре управляющих кнопки. Самая левая кнопка с характерным для каждой визуальной среды рисунком позволяет во время работы нашей программы вызвать системное меню с типовым набором команд по минимизации, максимизации и свер-

тыванию окна, по прекращению работы нашего приложения или переключению на другую программу. В правом верхнем углу расположены три стандартные кнопки, предназначенные для свертки, минимизации и максимизации окна приложения. Наконец, форма снабжена особой наружной рамкой, позволяющей во время выполнения нашего приложения изменять размеры окна и перемещать его по экрану.

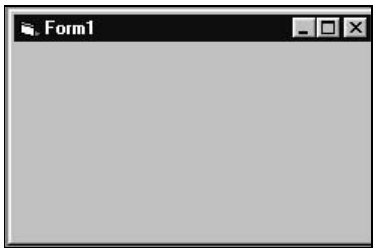


Рис. 2.13. Форма нового приложения

Рабочее поле формы, расположенное под его заголовком, принято называть областью клиента (в средах BCB и Delphi для этой области используется термин `ClientArea`). А в списке свойств формы вы можете обнаружить такие характеристики, как высота (`ClientHeight`) и ширина (`ClientWidth`) области клиента, ее положение (`ClientRect` — структура типа `TRect`, хранящая координаты двух противоположных углов). Эти свойства могут оказаться полезными, если во время работы приложения пользователь изменяет размеры окна, и программа должна произвести разумную перестройку содержимого контейнера.

Иногда на форме приходится размещать так много элементов, что для их отображения не хватает площади экрана. В этом случае задают разумные значения свойств ширины (`Width`) и высоты (`Height`), чтобы форма не вылезала за пределы экрана, и устанавливают размеры области клиента, превышающие видимую часть формы. Эти размеры могут быть намного больше габаритов экрана. Теперь достаточно задать свойство `AutoScroll=True`, и на форме появятся линейки прокрутки, обеспечивающие доступ к невидимым участкам области клиента.

Изначально заголовок формы повторяет значение его свойства `Name`. На самом деле, содержание заголовка не зависит от имени формы и определяется значением свойства `Caption`. Так как заголовок формы определяет лицо будущего приложения, изменение свойств формы можно начать со свойства `Caption`. Заходим в окно **Properties** (Свойства), выбираем строку `Caption` и в поле значения вводим свой текст. Переключение регистров позволяет набрать заголовок приложения на русском языке, и состоять он может из нескольких слов. Самое первое приложение, которое мы собираемся создать, не будет производить никаких действий. И даже не будет высвечивать

хрестоматийную фразу "Hello, World!". Поэтому мы присвоим заголовку значение "Пустое приложение".

Свойство Font на саму форму особого влияния не оказывает, однако его установками могут пользоваться те элементы управления, размещаемые на форме, которые обладают свойством наследовать характеристики родительского шрифта (ParentFont=True). Поэтому вместо многочисленных установок в таблицах свойств каждого компонента проще произвести этот выбор в форме.

В момент запуска приложения главное окно может появиться либо в нормальном виде, либо занимать весь экран, либо в виде иконки, которую еще предстоит развернуть. Выбор начального варианта определяется характеристикой WindowState, которая может принимать одно из значений wsNormal (в VB — Normal), wsMaximized (в VB — Maximized) или wsMinimized (в VB — Minimized). По умолчанию размеры формы стартовавшего приложения совпадают с теми, которые она имела на стадии проектирования (WindowsState=wsNormal).

2.4. Прежде чем запустить приложение

Пустое приложение после смены заголовка формы готово к обработке (компиляции или интерпретации), но прежде, чем выполнить команду **Run** (Выполнить), необходимо сохранить все, что может потребоваться при последующих сеансах работы. Усвойте следующее правило и никогда не отступайте от него: *все файлы, связанные с новым приложением, надо запоминать в отдельной папке.* По умолчанию система всегда присваивает файлам приложения одинаковые названия:

- VB — Project1.vbp, Form1.frm;
- Delphi — Project1.dpr, Unit1.dfm, Unit1.pas;
- BCB — Project1.bpr, Project1.cpp, Unit1.dfm, Unit1.cpp.

Поэтому всегда сохраняется вероятность затирания новым приложением файлов предыдущей программы.

Как только на экран вызвана форма нового VB-приложения, вы должны зайти в меню **File** (Файл) и выполнить команду **Save Project1 As** (Запомнить проект под новым именем). В появившемся диалоговом окне (рис. 2.14) вы можете создать новую папку и отправить туда свою, пока еще пустую заготовку.

Аналогичная операция в средах BCB и Delphi выполняется по команде **File | Save Project As**. Появляющееся диалоговое окно мало чем отличается от приведенного выше (рис. 2.15).

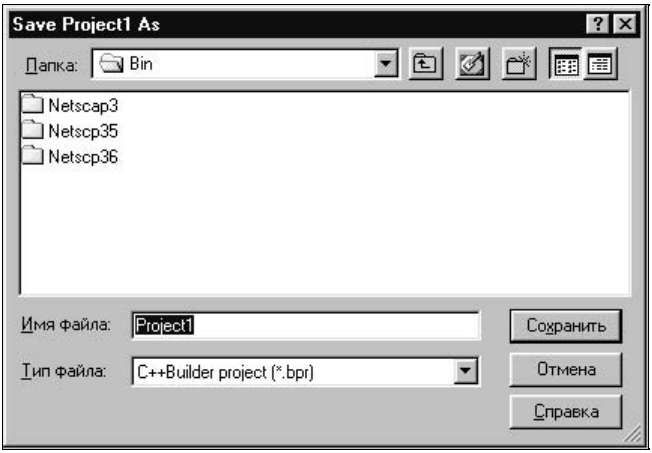


Рис. 2.14. Диалоговое окно для сохранения формы нового VB-проекта

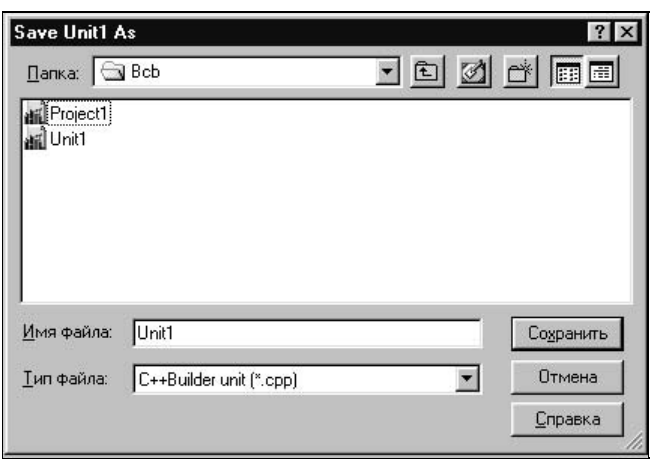


Рис. 2.15. Диалоговое окно для сохранения нового BCB-проекта

После того, как файлы проекта попали в отдельный каталог, вы можете вызывать проект из этого каталога и после внесения изменений в структуру формы или в текст программы обращаться к команде **File | Save** (Файл | Сохранить) или **File | Save All** (Файл | Сохранить все). Если вы забудете сохранить изменения, то система выдаст соответствующий запрос, прежде чем вы ее покинете.

2.5. Из чего состоит проект пустого приложения

После запоминания файлов пустого приложения, к составу которых мы не приложили никаких усилий, остается только выполнить команду **Run→Run**.

То есть — запустить приложение. Так как в нем не предусмотрено никаких действий, то остается только убедиться в соблюдении формой стандартных требований Windows. Попробуйте подвигать окно приложения по экрану, изменить его размеры, убедитесь в работоспособности каждой кнопки. После завершения работы приложения стоит заглянуть в каждый из каталогов, чтобы познакомиться с составом файлов, изготовленных каждой визуальной средой.

Приложение 2.1 (VB)

Меньше всего файлов — всего три — изготовила система Visual Basic.

Первый из них Form1.frm содержит символьное описание формы и текстов программ приложения. Структура описания формы и ее начинки напоминает вложенные блоки языка Pascal — используются известные операторные скобки `BEGIN` и `END`. В каждой строке описания содержится идентификатор свойства и его значение, если оно было изменено на стадии проектирования. Текст программ в точности повторяет содержимое окна редактора программ, каким вы наблюдали его на стадии проектирования приложения.

Второй файл Project1.vbp принято называть файлом *проекта* (vbp — сокращение от Visual Basic Project). Заниматься изучением каждой его строки великого смысла не имеет, но их назначение — управление различными режимами работы интерпретатора или компилятора. Третий файл Project1.vbw является небольшой добавкой к файлу проекта.

После заявлений о том, что версии 5 и 6 систем Visual Basic умеют компилировать, вызывает удивление отсутствие exe-файла. Дело в том, что основной режим работы систем VB — интерпретация. Правда, довольно быстрая. На первой стадии система изготавливает промежуточный *p-code* (пи-код), который не является готовой к исполнению машинной программой. На втором этапе происходит интерпретация этого пи-кода. Чтобы заставить VB-систему изготовить exe-файл, необходимо выполнить команду **File | Make Project1.exe**. После этого в нашем каталоге появится четвертый файл и чуть-чуть изменится содержимое файла проекта. Полный список файлов пустого приложения, изготовленных VB-системой, и их длины приведены в табл. 2.2.

Таблица 2.2. *Файлы пустого проекта, создаваемые средой Visual Basic*

Имя файла	Размер	Содержание файла
`Form1.frm`	496	Файл с описанием формы и текстами программ
`Project1.exe`	16.384	Выполняемый файл приложения
`Project1.vbp`	582	Файл проекта
`Project1.vbw`	44	Дополнение к файлу проекта

Не стоит надеяться, что exe-файл будет работать на другом компьютере. На самом деле, для обеспечения его работы в составе Windows должна присутствовать динамическая библиотека MSVBVM60.DLL, входящая в систему Visual Basic и физически расположенная в каталоге c:\Windows\System. Ее объем 1 409 024 байт, и этот файл может понадобиться на другом компьютере, чтобы там заработало приложение Project1.exe. Для переноса файлов более серьезного приложения на другой компьютер может понадобиться и не одна дискета.

Приложение 2.2 (Delphi)

Список файлов пустого приложения, изготовленных системой Delphi 6.0, приведен в табл. 2.3. Количество файлов достаточно велико. Можно порадоваться небольшому размеру exe-файла. Но так же, как и в предыдущем случае, на другом компьютере полученная программа работать не будет без поддержки соответствующих динамических библиотек, входящих в состав Delphi (например, без библиотеки *vcl40.bpl* объемом 1 884 136 байт). Файл конфигурации системы позволяет продолжить работу с проектом, восстановив все установки, которые были сделаны вами в предыдущий раз.

Таблица 2.3. *Файлы пустого проекта, создаваемые средой Delphi*

Имя файла	Размер	Содержание файла
`Project1.cfg`	444	Файл конфигурации Delphi-системы
`Project1.dof`	1 109	Файл опций проекта
`Project1.dpr`	188	Файл проекта
`Project1.exe`	12 800	Выполняемый файл приложения
`Project1.res`	876	Файл ресурсов
`Unit1.dcu`	2 867	Промежуточный результат компиляции
`Unit1.dfm`	264	Файл с описанием формы
`Unit1.pas`	298	Модуль приложения

Приложение 2.3 (BCB)

Аналогичный экскурс в каталог, хранящий продукцию BCB, запротоколирован в табл. 2.4. Количество файлов увеличилось, т. к. каждая программа на C++ сначала транслируется в объектный модуль. Кроме того, здесь присутствует заголовочный файл с описанием всех функций модуля, других подключаемых файлов, классов и глобальных констант. Поражает своими размерами файл с расширением tds, содержащий информацию для отладки. Его можно убить, не переносить в другие каталоги или на соседний компьютер. Но во время отладки сложных программ он окажет вам неоценимую помощь.

Таблица 2.4. *Файлы пустого проекта, создаваемые средой Borland C++ Builder*

Имя файла	Размер	Содержание файла
`Project1.bpr`	2 684	Файл проекта
`Project1.cpp`	710	Модуль проекта
`Project1.exe`	23 040	Выполняемый файл приложения
`Project1.obj`	15 717	Объектный модуль файла `Project1.cpp`
`Project1.res`	876	Файл ресурсов
`Project1.tds`	1 769 472	Файл с отладочной информацией
`Unit1.cpp`	523	Модуль приложения
`Unit1.dfm`	340	Файл с описанием формы
`Unit1.h`	750	Заголовочный файл проекта
`Unit1.obj`	36 330	Объектный модуль файла `Unit1.cpp`

Размер файлов, приведенных в табл. 2.2—2.4, является ориентировочным. Он зависит от установки многочисленных параметров, влияющих на работу системы, — включаются ли в состав exe-файла процедуры и функции из динамически загружаемых библиотек, требуется ли оптимизация по объему программы и/или скорости ее работы, создается ли отладочная информация и т. п. Например, попытка создать exe-файлы, способные работать на других компьютерах, где отсутствуют соответствующие dll- и bpl-файлы, увеличивает размер программы пустого приложения до 200—300 Кбайт.

Программистам довольно часто приходится переносить проекты из одного каталога в другой. Все ли файлы из перечисленных в табл. 2.2—2.4 следует копировать? Естественно, что exe-файлы из этого списка исключаются, т. к. в новом каталоге можно заново повторить компиляцию приложения. В среде Delphi не переносится файл с расширением dcu. В среде BCB не копируются объектные модули и файл с отладочной информацией. Все это создается заново при очередной компиляции.

Для повторного запуска визуальной среды с одновременным вызовом нужных частей проекта достаточно дважды щелкнуть по файлу проекта.

2.6. Продолжение работы над проектом

Открываем пустое приложение. Так как мы продолжаем работу над проектом, который сохранили в предыдущем сеансе, то его имя и путь к соответствующему каталогу система запомнила. В VB имя такого проекта находится в одной из последних строк меню **File** (Файл), а в BCB и Delphi — в специальном окне, которое раскрывается по команде **File | Reopen**. Это избавляет

нас от лишней работы по выбору пути и набору имени открываемого проекта. Конечно, вызвать тот или иной проект можно, заглянув в соответствующий каталог и щелкнув по файлу проекта.

Цель модификации нашего первого приложения — заставить его выполнять хотя бы какую-нибудь видимую работу. Например — реагировать на щелчки мышью в области клиента формы. А чтобы мы эту реакцию заметили, пусть приложение считает количество щелчков и отображает текущее значение счетчика в заголовке формы.

Щелчок мышью по области клиента во время выполнения приложения породит событие `OnClick` (в VB — `Click`), которое операционная система адресует нашей форме. И для того чтобы форма отреагировала на поступившее сообщение, к нашей программе необходимо подключить обработчик соответствующего события.

Приложение 2.4 (VB)

В среде VB на поле редактора кодов раскрываем правый список, в котором хранится набор всех событий, адресуемых форме, и выбираем строку `Click`. После появления заготовки, содержащей заголовок и оператор конца подпрограммы, набираем в ее теле три следующие оператора (они выделены жирным шрифтом):

```
Private Sub Form_Click()
  Static Count
  Count=Count+1
  Form1.Caption=Count
End Sub
```

Переменная `Count` объявлена статической, так как должна сохранять свое новое значение при повторном входе в подпрограмму. Пусть вас не смущает, что в левой части последнего оператора присваивания находится символьная переменная, а в правой — числовое значение. Необходимые преобразования типов система берет на себя.

Приложение 2.5 (BCB)

В среде BCB обработчик события подключается похожим способом. В окне **Object Inspector** (Инспектор объектов) переходим на вкладку **Events** (События), находим строку `OnClick` и щелкаем по ее правой части. В ней появится название функции `FormClick`, а на поле редактора кодов — соответствующая заготовка. В тело функции мы внесем аналогичные три строки:

```
void __fastcall Tform1::FormClick(Tobject *Sender)
{
```

```
  static int count;
  count++;
  Form1->Caption=count;
}
```

Приложение 2.6 (Delphi)

В среде Delphi подключение обработчика события `OnClick` делается почти так же, как и в BCB. Разница только в оформлении процедуры обработчика события. В связи с отсутствием в Delphi аналога статических переменных, счетчик `count` приходится объявить как глобальную переменную, вынеся ее описание в раздел `var` модуля `Unit1`:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
logs;
type
  TForm1 = class(TForm)
    procedure FormClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
  count:integer;
implementation
{$R *.DFM}
procedure TForm1.FormClick(Sender: TObject);
begin
  inc(count);
  Form1.Caption:=IntToStr(count);
end;
end.
```

Попробуем разобраться со спецификой в текстах программ обработчиков события.

Во-первых, мы решили использовать заголовок формы для отображения результата работы программы. Это не запрещено, и мы превратили заголовок в табло результатов только потому, что не захотели загромождать форму другими компонентами. В обработчиках для присвоения значения этому свойству задействованы составные имена `Form1.Caption` и `Form1->Caption` (в BCB переменная `Form1` является указателем на объект, поэтому в составном имени использована стрелка, а не точка). На самом деле, в любом из этих обработчиков вместо составного имени свойства можно было бы сохранить только идентификатор `Caption`. По умолчанию добавку `Form1` система приклеит сама. Но как только в нашем приложении появится несколько объектов, обладающих свойством `Caption`, то для их точной идентификации имя объекта придется обязательно добавлять.

Во-вторых, в заголовке BCB-обработчика система вставила служебное слово `fastcall` (дословно — быстрый вызов). Обычно, в программах на Си параметры функций передаются через стек, и на это расходуются лишние такты. При быстром вызове несколько первых аргументов передаются через машинные регистры.

Добавка `TForm1` перед именами функции в BCB и процедуры Delphi означает, что эти программы представляют собой методы класса `TForm1` и обращаться к ним, если это понадобится, следует особым образом.

В BCB и Delphi обработчику передается указатель на объект, явившийся причиной возникновения события (`Sender` — отправитель). В большинстве случаев этот аргумент не используется. Однако в приложении могут оказаться два объекта, порожденные одним и тем же классом. И тогда придется разбираться, кто из них выступил в роли отправителя.

Перед тем как запустить модифицированное приложение, не забудьте сохранить все файлы в уже существующем каталоге. Такая предосторожность не повредит, т. к. приложение может так завесить компьютер, что для его реанимации придется произвести перезагрузку. Если бы мы не запомнили файлы перед стартом приложения, то вся работа по его модификации пошла бы насмарку и все пришлось бы повторять заново. Старайтесь чаще запоминать текущее состояние своих файлов.

2.7. Расстановка компонентов на форме

Пунктирная сетка, нарисованная на форме во время проектирования приложения, играет вспомогательную роль. Ее узлы можно использовать для грубого выравнивания размеров и положения управляющих элементов на форме. Однако добиться высокой точности таким способом вряд ли удастся. Конечно, можно прибегнуть к заданию числовых значений координат точек привязки (свойства `Left` и `Top` каждого объекта) и их габаритов (свойства `Width` и `Height`). Но это очень утомительное занятие. В каждой визуальной

среде предусмотрены команды главного меню и специальные диалоговые окна, упрощающие компоновку объектов на форме или в каком-либо другом контейнере.

Продемонстрируем эти возможности в среде VB. Разместим на форме три объекта типа окон ввода и умышленно сделаем произвольными их размеры и положение (рис. 2.16). Обратите внимание, что объект, помещенный на форму последним, остается выделенным — вокруг его границы расположены восемь характерных точек. За любую из них можно "ухватиться" мышкой и потянуть в соответствующую сторону для изменения размеров объекта.

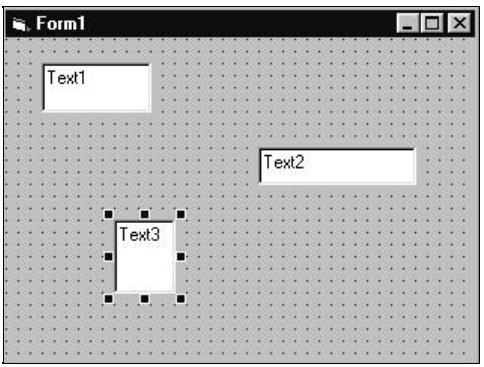


Рис. 2.16. Несколько объектов разной формы

Теперь зажмем клавишу <Shift> и щелчком мыши выделим объекты `Text1` и `Text2`. Результат отражен на рис. 2.17. Обратите внимание на то, что объект, выделенный последним (`Text2`), окаймлен более жирными точками. Именно под него будут подстраиваться размеры и положения других выделенных объектов. Заходим в пункт главного меню **Format** (Формат) (его команды показаны на рис. 2.18).

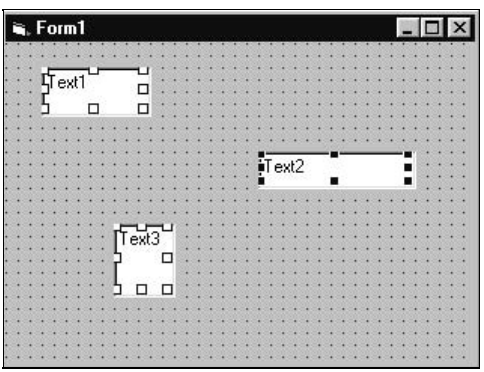


Рис. 2.17. Результат выделения нескольких объектов

Команда **Align** (Выравнивание) переведет нас в подменю, управляющее выравниванием выделенных компонент (см. рис. 2.18).

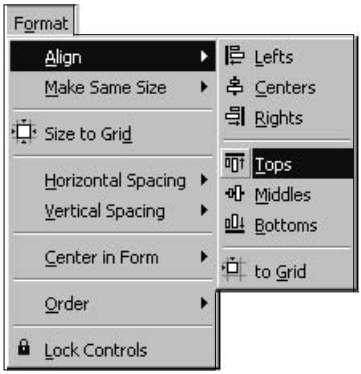


Рис. 2.18. Меню выравнивания границ

Верхняя часть подменю позволяет произвести выравнивание по вертикали, сделав у всех объектов одинаковой границу слева (**Lefts**), справа (**Rights**) или совместив их центры на одной линии (**Centers**). Следующий фрагмент подменю регулирует аналогичное выравнивание по горизонтали. У всех объектов может быть сделана на одном уровне верхняя граница (**Tops**), нижняя граница (**Bottoms**) или средняя линия (**Middles**). И в том, и в другом случае выделенные объекты будут подстраиваться под границы компонента `Text3`. В нашем случае операция **Align** (Выравнивание) имеет смысл только для выравнивания объектов по вертикали. В случае выравнивания горизонтальных границ объекты `Text1` и `Text3` наползут друг на друга. При этом ничего не пропадет, мы можем зацепить любой из них и передвинуть в другое место. Результат выравнивания границ по вертикали отражен на рис. 2.19.

Команда **Make Same Size** (Сделать одинаковый размер) позволяет подогнать габариты объектов `Text1` и `Text3` под размеры окна `Text2`. Причем допускается как раздельное выравнивание — только по ширине (**Width**) или высоте (**Height**) объектов, так и по обоим параметрам (**Both**) одновременно (рис. 2.20).

Команда **Size to Grid** (Задать размер согласно сетке) устанавливает границы выделенных объектов по ближайшим линиям пунктирной сетки. Для группы объектов таких, например, как клавиши калькулятора важно установить не только одинаковые размеры, но и выдержать нужные интервалы между объектами по горизонтали и/или по вертикали. В этом случае помогут команды **Horizontal Spacing** (Горизонтальный зазор) и **Vertical Spacing** (Вертикальный зазор). В каждом из появляющихся подменю имеется возможность сделать интервалы между выделенными элементами равными (**Make Equal**), увеличить (**Increase**) или уменьшить (**Decrease**) текущие интервалы на один шаг сетки, сделать интервалы нулевыми (**Remove**). В последнем случае границы объектов сольются.

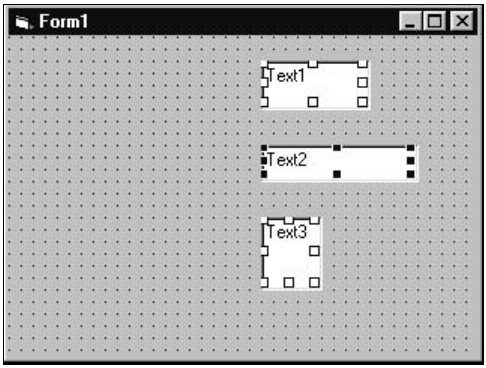


Рис. 2.19. Результат выравнивания левых границ

Команда **Center in Form** (По центру формы) позволяет расположить центры выделенных объектов либо на одной вертикали, либо на одной горизонтали, проходящей через центр формы.

К команде **Order** (Упорядочить) прибегают в тех случаях, когда несколько объектов перекрывают друг друга. Тогда кого-то можно выдвинуть вперед (**Bring to Front**) или отправить на задний план (**Send to Back**).

Команда **Lock Controls** (Закрепить) закрепляет позиции выделенных элементов, после чего их перетаскивание с помощью мыши становится невозможным.

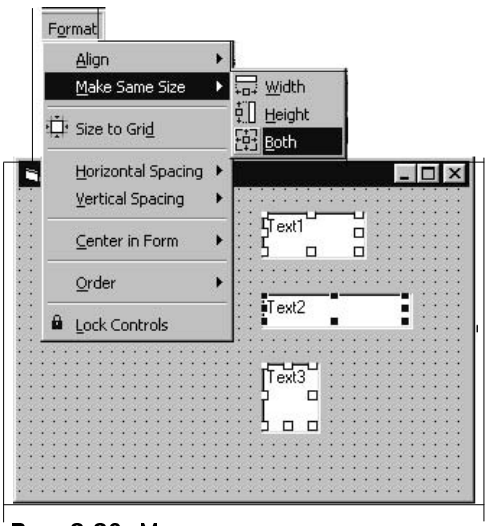


Рис. 2.20. Меню выравнивания размеров

В системах BCB и Delphi аналогичные команды по выравниванию размеров и положения вынесены в меню **Edit** (Правка) (рис. 2.21). Обратите внимание, что в отличие от VB, здесь нет четко выраженного ведущего элемента,

под который подстраиваются остальные. Однако в визуальных средах фирмы Borland ведущим элементом всегда является первый выделенный.

При выбранном на рис. 2.21 расположении элементов произвести выравнивание границ (команда **Align**) не удастся, т. к. один из элементов неизбежно наедет на другой. Если чуть-чуть сместить окно `Edit2` вниз, то появится возможность подровнять левые или правые границы выделенных элементов.

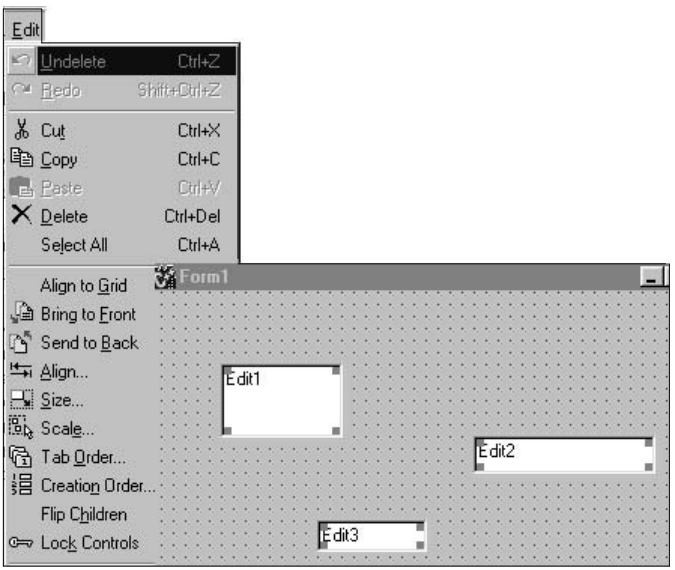


Рис. 2.21. Средства размещения объектов в среде BCB

Выполнение команды **Align** (Выравнивание) приводит к появлению диалогового окна (рис. 2.22). Слова, как видите, почти те же самые, что и в системе VB. В пояснении нуждается только новая команда **Scale** (Масштабировать), с помощью которой можно произвести масштабирование выделенных объектов. В появляющемся диалоговом окне вы можете изменить масштабирующий коэффициент, равный по умолчанию 100%.

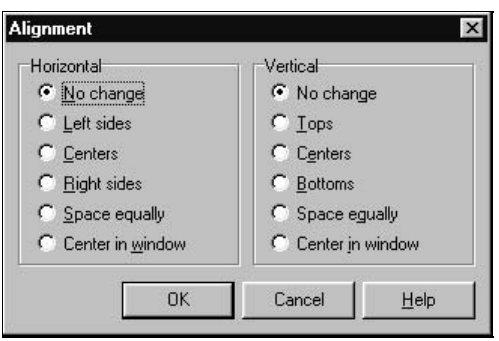


Рис. 2.22. Средства выравнивания границ

2.8. Главное меню

По ходу изложения материала этой главы в той или иной мере были затронуты некоторые команды главного меню. На первых порах нам придется использовать достаточно ограниченные возможности визуальных сред. Однако меню, это то место, которое посещается очень часто. Поэтому всегда полезно иметь компактную информацию о том, что мы можем сделать с помощью той или иной команды соответствующего раздела главного меню.

2.8.1. Главное меню среды Visual Basic 6.0

Меню **File** (Файл). В большинстве систем программирования здесь сосредоточены команды по извлечению **Open** (Открыть) ранее сохраненных наборов данных или запоминанию текущих файлов на диске **Save** (Сохранить). Команды типа **New** (Новый) обычно подготавливают среду к началу разработки нового приложения или подключению к текущему приложению новых форм, наборов данных и т. п. С помощью команд типа **Print** (Печать) можно вывести на принтер текст текущей программы, выделенного фрагмента или формы. Наконец, во всех известных авторам системах раздел **File** (Файл) не обходится без команды **Exit** (Выход), выполняемой с целью завершения сеанса работы. Раздел **File** (Файл) в системе VB в этом смысле особой оригинальностью не отличается.

- Команда **New Project** (Новый проект) открывает диалоговое окно, в котором можно выбрать тип вновь создаваемого проекта. Вкладка **New** (Новый) предлагает вам более десятка различных вариантов — стандартное приложение (**Standard EXE**), набор данных (**Data Project**), различные управляющие элементы и динамически загружаемые библиотеки. Среди этого разнообразия, как правило, выбирается новое стандартное приложение или одна из ранее разработанных программ. Имена трех-четырех последних проектов помещаются непосредственно в меню **File** (Файл). На вкладке **Recent** (Недавний) вы обнаружите имена всех проектов, которые появлялись с момента установки визуальной среды.
- Команда **Open Project** (Открыть проект) используется для вызова существующего проекта.
- Команда **Add Project** (Добавить проект) позволяет открыть новый проект, не закрывая текущий. Это позволяет в течение одного сеанса работать с группой проектов. Преимущество такого режима работы заключается в более эффективном переключении с одного проекта на другой.
- Команда **Remove Project** (Удалить проект) закрывает текущий проект.
- Команды **Save Project** (Сохранить проект) и **Save Project As** (Сохранить проект под новым именем) обеспечивают запоминание текущего проекта под старым или новым именем. Аналогичное запоминание форм выпол-

няется с помощью команд **Save Form** (Сохранить форму) и **Save Form As** (Сохранить форму под новым именем).

- Команда **Print** (Печать) позволяет выдать на принтер текущую форму или содержимое окна редактора программы в зависимости от того, какое из окон активизировано. Настройки принтера производятся в типовом окне Windows после выполнения команды **Print Setup** (Установки печати).
- Команда **Make...** (Построить...) — вместо многоточия вы увидите имя текущего проекта — используется для построения exe-файла. Если открыта группа проектов, то аналогичная команда строит exe-файлы для каждого проекта группы.
- Меню **Edit** (Правка). Здесь, как правило, сосредоточены команды, обеспечивающие внесение изменений в текущий проект или позволяющие отказаться от ошибочных модификаций.
- Команды **Undo** (Отменить операцию) и **Redo** (Вернуться к отмененной операции) обеспечивают отказ от очередного исправления или возврат к только что отвергнутому изменению.
- Операции по вырезанию (**Cut**), копированию (**Copy**), вставке (**Paste**), специальной вставке (**Paste Link**), удалению (**Delete**), выделению всего (**Select All**) характерны для любого текстового редактора. Такими же естественными являются команды поиска (**Find**, **FindNext**) и замены (**Replace**).
- Команды **Indent** (Сдвинуть фрагмент вправо) и **Outdent** (Сдвинуть фрагмент влево) сдвигают выделенный фрагмент вправо или влево.
- С помощью команды **Insert File** (Вставить файл) в текущую позицию на поле редактора программы можно вставить выбранный вами файл.
- Команда **List Properties/Methods** (Список свойств/методов) используется для ускорения набора текста программы. Если вы обратитесь к ней после того, как поставили точку вслед за именем объекта, то появится диалоговое окно со списком свойств и методов, приписанных данному объекту. Среди них вы можете выбрать нужную строку и быстро вставить ее в свой текст.
- Команда **List Constants** (Список констант) тоже используется для ускорения набора и одновременно играет роль подсказки. К ней обращаются при наборе оператора присваивания, в левой части которого находится имя свойства. По команде **List Constants** (Список констант) высвечивается список с допустимыми значениями данного свойства.
- Команда **Quick Info** (Краткая информация) выдает краткую информацию по вызову функции, процедуры или метода, на чьем имени находится курсор. Аналогичную подсказку по аргументам вызываемой процедуры высвечивает команда **Parameter Info** (Информация о параметрах).

- Если вы не помните точное написание служебного слова или хотите ускорить его набор, воспользуйтесь командой **Complete Word** (Полное слово) после набора первой буквы слова. Вам предложат список подходящих слов, из которых можно выбрать нужное слово.
- Команда **Go To Row** (Перейти к строке) используется при работе с таблицами — основной структурой информации в базах данных. Строка таблицы (**Row**) является аналогом записи, а сама таблица может рассматриваться как массив строк. Описываемая команда обеспечивает навигацию по таблице, позволяя переходить к первой (**First**), последней (**Last**), следующей (**Next**), предыдущей (**Previous**) записи или к записи с заданным номером. В открывающемся подменю встречается еще одна команда — **New** (Новый), с помощью которой формируется новая запись.

Перемещения по тексту большой программы и возврат в ту или иную точку представляет собой довольно утомительное занятие, несмотря на наличие клавиш <Page Down> и <Page Up>. В современных системах для работы с большими текстовыми документами принято использовать закладки — *Bookmarks*. С каждой такой закладкой связывается небольшое число, обычно от 1 до 9, и система запоминает координаты курсора в точке, где создается очередная закладка. Потом по этому номеру вы можете быстро вернуться на нужное место. Закладки можно отменять и переназначать. Похожие возможности реализованы в команде **Bookmarks** (Закладки). Открывающееся подменю позволяет создавать или отменять закладку в текущей позиции курсора (**Toggle Bookmark**), переходить к следующей (**Next Bookmark**) или предыдущей (**Previous Bookmark**) закладке, отменять все закладки (**Clear Bookmarks**).

Меню **View** (Вид). Это меню относится к разряду часто посещаемых разделов главного меню, и его основное назначение — показать или скрыть одно из служебных окон среды, в которых отображается та или иная информация. Для перехода в нужное окно раздел **View** (Вид) предлагает следующие команды.

- **Code** (Код) — переход в окно редактора текущей программы.
- **Object** (Объект) — переход в окно текущей формы.
- **Definition** (Определение) — переход в окне редактора программы на определение переменной или процедуры, на имени которой находится курсор.
- **Last Position** (Последняя позиция) — возврат в поле редактора программы на место, где было сделано последнее изменение. Редактор кода хранит информацию о позициях последних 8 изменений — нечто вроде стека, по которому производится возврат.
- **Object Browser** (Браузер объектов) — переход в окно просмотра объектов. Здесь располагается полный каталог всего, чем можно пользоваться при создании программы — список типов, классов, констант, методов,

свойств и событий. Для удобства поиска имеется возможность отобрать один из нескольких разделов этой коллекции — весь набор данных (**All Libraries**), объекты, использованные в текущем проекте (**Project1**, например), объекты среды VB или VBA, объекты, используемые на стадии выполнения (**VBRUN**).

- **Immediate Window** (Окно немедленных вычислений) — переход в окно немедленных вычислений. Здесь можно выполнить некоторые операторы над текущими данными приостановленной программы. Сюда же попадает отладочная информация, отражающая ход выполнения программы.
- **Locals Window** (Окно локальных переменных) — в этом окне отображаются значения локальных переменных функций и подпрограмм во время отладки программы.
- **Watch Window** (Окно контролируемых выражений) — в этом окне отображаются значения выражений (в частном случае — переменных), которые вы решили контролировать в процессе отладки программы.
- **Call Stack** (Вызов стека) — в этом окне находятся имена подпрограмм и функций, вызванных в процессе работы к моменту попадания в точку просмотра (нечто вроде стека возвратов).
- **Project Explorer** (Просмотр проекта) — переход в окно, в котором отображается дерево структуры проекта.
- **Properties Window** (Окно свойств) — переход в окно свойств объектов, включенных в состав текущего проекта. В это окно приходится заходить довольно часто на стадии проектирования приложения для установки значений нужных свойств. Обычно это окно постоянно присутствует на экране, но если его закрыли, то можно воспользоваться указанной командой или нажать клавишу <F4>.
- **Form Layout Window** (Окно расположения формы) — окно, демонстрирующее расположение формы на экране во время выполнения программы. На наш взгляд особого смысла в существовании этого окна и, уж тем более, его присутствии на поле разработки нет. Все вопросы, связанные с размещением и размерами окна приложения во время его появления на экране монитора после старта, решаются с помощью свойств, описанных в разделе 2.3. В других визуальных средах такое украшательство посчитали излишеством.
- **Property Pages** (Страница свойств) — диалоговое окно, в котором отображаются свойства выбранного объекта.
- Команды **Table** (Таблица), **Zoom** (Увеличение/Уменьшение) и **Show Panes** (Показать страницы) предназначены для просмотра таблиц и результатов, полученных при извлечении и обработке документов базы данных. Более подробно мы их рассматривать не будем.

- **Toolbox** (Инструменты) — с помощью этой команды можно сделать активным закрытое окно **General** (Общие) с набором основных компонент. Обычно такая операция производится с помощью быстрой кнопки, расположенной под главным меню.
- **Data View Window** (Окно просмотра данных) — с помощью этой команды к текущему проекту можно подключить базу данных.
- **Color Palette** (Цветовая палитра) — переход в окно цветовой палитры, с помощью которого можно изменить цвет фона выделенного объекта или формы. Клавиша **Default** (По умолчанию) восстанавливает цвет, принятый в системе по умолчанию.
- **Toolbars** (Панель инструментов) — с помощью этой команды на экране можно разместить одну из небольших линеек с набором быстрых кнопок, наиболее подходящих для выполнения той или иной группы операций. Внешний вид этих линеек для команд **Debug** (Отладка), **Edit** (Правка), и **Form Editor** (Редактор форм) приведен на рис. 2.23. Команда **Standard** (Стандарт) включает или отключает линейку с системными инструментами, расположенную под главным меню. По команде **Customize** (По выбору) вы можете создать собственную инструментальную линейку.

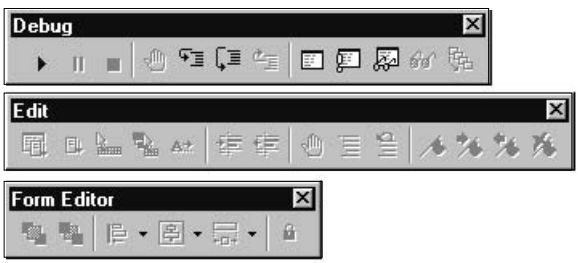


Рис. 2.23. Специализированные линейки инструментов

- Команда **Visual Component Manager** (Менеджер визуальных компонентов) вызывает диалоговое окно с аналогичным названием (рис. 2.24). В нем можно получить информацию о компонентах, входящих в состав других визуальных систем, входящих в состав Visual Studio.

Меню **Project** (Проект). В этот раздел главного меню входит группа команд по дополнению текущего проекта новыми или существующими элементами — формами, модулями, файлами и т. п.

- Команда **Remove** (Удалить) удаляет выбранный элемент из проекта.
- Команда **References** (Ссылки) открывает диалоговое окно, в котором сформирован список всех приложений, способных участвовать в OLE-обмене. Этим можно воспользоваться для установления связи с программой, которая может изготовить объект, внедряемый в наше приложение.

- О команде **Components** (Компоненты) мы уже упоминали в *разд. 2.1*, когда потребовалось пополнить панель **General** (Общие) объектом **Masked Edit** (Редактирование по маске).
- Команда **Properties** (Свойства) позволяет изменить некоторые свойства текущего проекта — имя, тип, режим компиляции, способ оптимизации и т. п. На первых порах заглядывать в это окно не стоит.

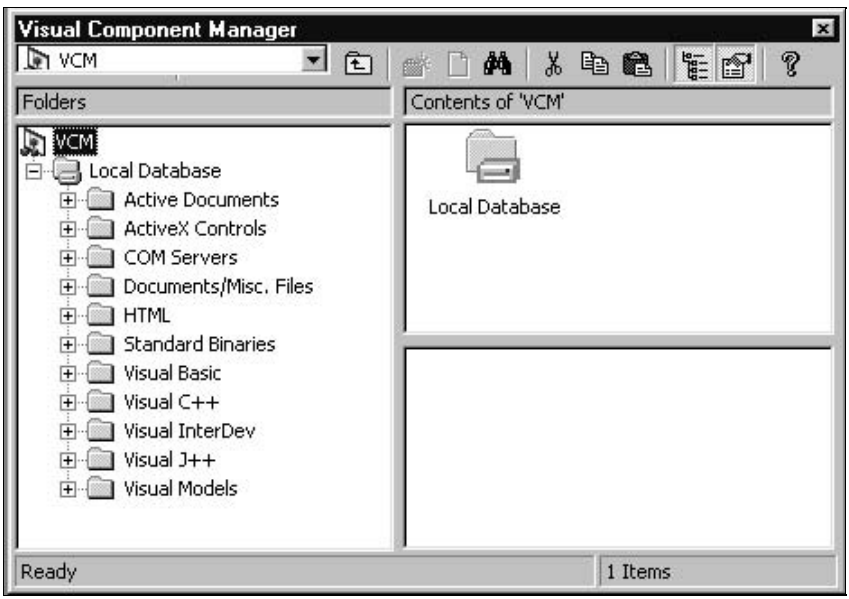


Рис. 2.24. Менеджер визуальных компонент

Меню **Format** (Формат). Со всеми командами этой вкладки мы познакомились в *разд. 2.7*.

Меню **Debug** (Отладка). Без этого раздела не обходится ни одно приложение, т. к. первая заповедь программиста гласит — "в любой отлаженной программе есть, по крайней мере, одна ошибка". Ошибки синтаксиса система заметит и без вмешательства человека. Но ошибки алгоритма, нарушения логики и схемы вычислений, выход за пределы массива, неверное количество повторений циклов и другие малоприятные ситуации никто кроме вас не обнаружит. И самую существенную помощь в поиске и устранении ошибок оказывают команды раздела **Debug** (Отладка).

- Команды **Step Into** (Шаг внутрь) и **Step Over** (Шаг поверх) позволяют организовать пошаговое выполнение программы. При очередном нажатии клавиш <F8> или <Shift>+<F8> система выполняет очередную строку программы. Разница между этими двумя режимами прокрутки заключается в том, что команда **Step Into** (Шаг внутрь) продолжает пошаговое

выполнение и в вызываемых функциях и подпрограммах. Тогда как по команде **Step Over** (Шаг поверх) вызываемая функция или подпрограмма выполняются в автоматическом режиме. Имеет смысл заходить в вызываемые процедуры только тогда, когда вы не уверены в правильности их работы.

- Команда **Step Out** (Шаг наружу) поможет в том случае, когда мы не хотим проталкивать строки процедуры в пошаговом режиме, а предпочитаем завершить работу процедуры на автомате. После выхода из процедуры пошаговое выполнение программы продолжится.

- Выполнение всей программы строка за строкой не всегда имеет смысл, т. к. в каких-то частях программы мы уже уверены. И такие участки следовало бы проскакивать на автомате, останавливаясь только в определенных точках и простукивая подозрительные фрагменты. Для этой цели отдельные строки программы могут быть помечены как точки останова — *Breakpoints.* Когда мы выполняем команду **Toggle Breakpoint** (Переключение точки останова) (этому эквивалентно нажатию клавиши <F9>), текущая строка либо окрашивается в другой цвет (обычно, красный), либо восстанавливает свой обычный окрас. Все окрашенные таким образом строки являются точками останова, перед их выполнением работа приложения приостанавливается для совершения каких-либо вспомогательных операций — просмотра значений переменных или выражений, перехода в пошаговый режим работы и т. д. Одной из точек останова может быть текущая строка, если мы запускаем приложение не командой **Run** (Выполнить), а командой **Go To Cursor** (Перейти к курсору). С помощью команды **Clear All Breakpoints** (Отменить все точки останова) производится отмена всех точек останова.

- К числу отладочных приемов, установившихся во многих системах программирования, относится наблюдение за значениями отдельных переменных и выражений. Список таких переменных и выражений заносится в специальное окно с помощью команды **Add Watch** (Добавить контролируемое выражение). В момент останова программы мы можем заглянуть в окно **Watch** (Контролируемые выражения) и сравнить полученные и ожидавшиеся значения. Команда **Edit Watch** (Редактирование списка контролируемых выражений) позволяет изменить список просматриваемых величин, пополнив его, изменив или удалив существующие выражения. Команда **Quick Watch** (Быстрый контроль) обеспечивает быстрый просмотр выражения, которое не было предварительно включено в контролируемый список.

  Для просмотра значений переменных в момент останова программы современные системы программирования добавили очень удобную деталь — когда курсор мыши подводится к имени переменной, появляется этикетка со значением этой переменной (рис. 2.25).

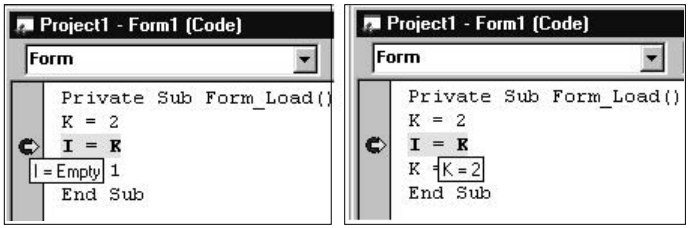


Рис. 2.25. Просмотр значений переменных по указанию курсором

- Команда **Set Next Statement** (Установить следующий оператор) позволяет продолжить выполнение прерванной программы не с точки останова, а с любого оператора программы. Для этого надо щелкнуть по указанной команде раздела **Debug** (Отладка) и перевести курсор на нужную строку. С нее и продолжится выполнение программы после нажатия одной из клавиш <F9> (автоматическое продолжение), <F8> или <Shift>+<F8> (пошаговое выполнение).
- Команда **Show Next Statement** (Показать следующий оператор) показывает оператор, который будет выполняться после текущего оператора. Может быть, в этом действии и есть какая-то сермяжная правда, но, кажется, проще нажать клавишу <F8> и перейти к следующему оператору. За нашу суммарную практику программирования (а она уже перевалила за полвека), ни одному из авторов не приходилось пользоваться подобной услугой.

Меню **Run** (Выполнить). Содержит всего 5 команд, с помощью которых можно запустить приложение (**Start** или **Start With Full Compile** — Старт с полной перекомпиляцией), приостановить (**Break**) или прервать (**End**) работу приложения, осуществить повторный перезапуск (**Restart**). Для запуска приложения часто используют клавишу <F5> или кнопку быстрого запуска. При заходе в меню **Run** (Выполнить) в точке останова вместо команды **Start** (Запустить) появляется команда **Continue** (Продолжить). К команде **Start With Full Compile** (Старт с полной перекомпиляцией) приходится прибегать в ситуациях, когда работа системы или приложения становится непредсказуемой. Такие трудности возникают и в других системах программирования (в частности, в BCB и Delphi) и кроме полной перекомпиляции проекта ничего не помогает. Всегда помните первую заповедь программиста, звучавшую выше. Ни одна система не свободна от ошибок и способы их проявления непредсказуемы.

Два следующие раздела главного меню — **Query** (Запрос) и **Diagram** (Диаграмма) мы предпочитаем пропустить, т. к. они относятся к процедурам обработки документов баз данных (в нашей книге эти вопросы детально не рассматриваются).

Меню **Tools** (Инструменты). Дословный перевод названия раздела — **Tools** (Инструменты). Однако из команд, близких по смыслу к этому заголовку,

относится только команда **Menu Editior** (Редактор меню), вызывающая диалоговый редактор меню, включаемого в состав текущего приложения. Более подробно работа по подключению главного и всплывающего меню рассматривается в *главе 5*.

- Команда **Add Procedure** (Добавить процедуру) позволяет включит в список процедур, расположенный в окне редактора программ, новую процедуру. В появляющемся диалоговом окне задаются имя новой процедуры (**Name**), ее тип (**Type**), режим доступа (**Scope**) и способ использования локальных переменных (рис. 2.26). Дополнительные атрибуты новой процедуры устанавливаются с помощью команды **Procedure Attributes** (Атрибуты процедуры).

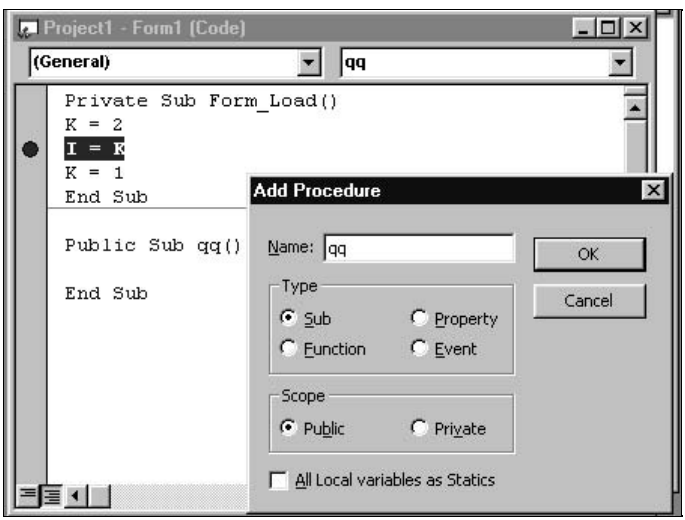


Рис. 2.26. Добавление новой процедуры

- К числу изредка используемых команд раздела **Tools** (Инструменты) относится команда **Options** (Параметры). Такого рода команда встречается в большинстве пакетов программ и ее основное назначение — произвести настройку параметров, влияющих на работу системы. Она открывает диалоговое окно с большим количеством вкладок (рис. 2.27). Вы можете попробовать что-то изменить, но для возможного отката не забудьте записать предшествующее значение того или иного параметра. А еще лучше — сохраните картинки каждой из шести вкладок окна **Options** (Параметры). Сделать это довольно просто. Вы открываете очередную вкладку и нажимаете клавишу <Print Screen>. После этого запускается стандартный графический редактор **Paint** и в меню **Edit** (Правка) выполняется команда **Insert** (Вставить). Из появившегося экрана вырезается фрагмент с вкладкой окна и запоминается в формате bmp. Потом эти картинки помогут восстановить ситуацию.

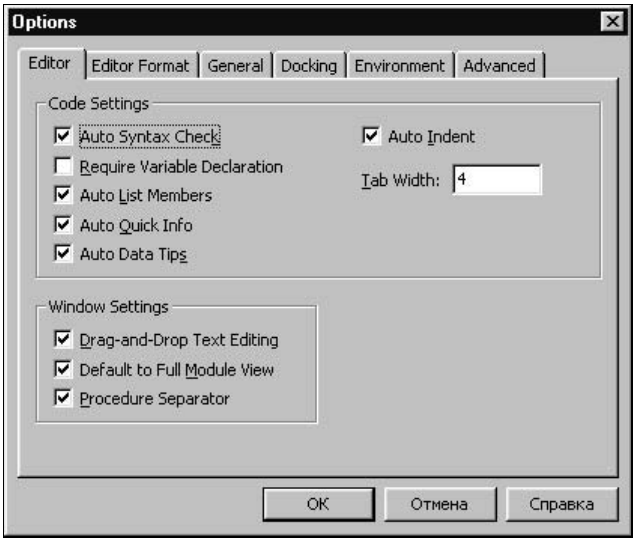


Рис. 2.27. Окно настройки параметров среды VB

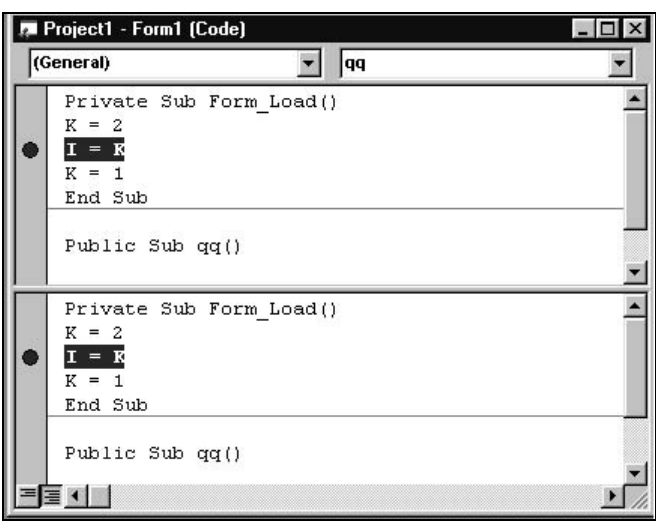


Рис. 2.28. Расщепление окна редактора программы

Меню **Add-Ins** (Добавления) предназначено для включения в состав своего раздела строк по запуску различного рода сервисных программ-утилит. На первых порах это вам не понадобится.

Меню **Window** (Окно). В этом разделе находятся команды, используемые для реорганизации размещения открытых окон. Команда **Split** (Расщепить) расщепляет активное окно редактора программы на две половинки (рис. 2.28),

что позволяет видеть одновременно два фрагмента программы, копировать выделенные блоки из одного экрана в другой.

- Команда **Tile** (Черепица) располагает окна как черепицы на крыше — рядом друг с другом. Команды меню позволяют организовать расположение "черепиц" по горизонтали (**Tile Horizontally**) или вертикали (**Tile Vertically**). Команда **Cascade** (Каскад) располагает окна как карты в колоде, оставляя видимыми заголовки окон. Команда **Arrange Icons** (Упорядочить иконки) располагает пиктограммы закрытых окон в нижней части экрана. В нижней части раздела Windows хранится список всех окон, открытых пользователем.

Меню **Help** (Справка). В этот раздел вам придется заглядывать довольно часто, т. к. по мере изучения новых возможностей системы количество вопросов будет только возрастать. В меню **Help** (Справка) можно попасть стандартным способом, щелкнув мышкой по его заголовку, или по нажатию клавиши <F1>, когда вы что-то выделили или установили курсор на непонятном термине. Команда **Contents** (Содержание) открывает оглавление справочной системы, в котором можно выбрать подходящий раздел и продолжить поиск, сузив объем информации. По команде **Index** (Указатель) открывается список ключевых слов, среди которых может оказаться нужное или похожее. Наконец, можно прибегнуть к услугам команды **Search** (Поиск). Что касается выхода на техническую поддержку (**Technical Support**) в режиме on-line или на один из сайтов фирмы Microsoft, то консультацию вы получите бесплатно, но за использование каналов связи платить придется.

2.8.2. Главное меню среды Borland C++ Builder 5.0

Визуальные среды BCB и Delphi имеют настолько близкий интерфейс, что структура и состав команд главного меню у них почти одинаковы. Конечно, от версии и от редакции системы кое-что зависит, но основные средства меню достаточно консервативны. Так как мы не ставим своей целью рассказать обо всем, то отдельные элементы главного меню останутся за кадром.

Меню **File** (Файл) по традиции предоставляет возможность начать работу с новым приложением (**New Application**) или выбрать тип нового проекта в специальном диалоговом окне (**New**). С помощью команды **Open Project** (Открыть проект) мы можем выбрать любой существующий проект, побродив по дискам и каталогам. Дополнительные удобства при выборе не очень давнего проекта предоставляет команда **Reopen** (Открыть снова), открывающая список с именами последних побывавших в работе проектов.

- Команда **Open** (Открыть) позволяет открыть и вызвать в поле редактора не только любой файл текущего проекта, но и любой файл того или иного типа. В открывающем диалоговом окне вы можете выбрать любое расширение и поискать нужный файл в любом каталоге.

- Команды группы **Save** (Сохранить) позволяют сохранить активный файл, находящийся на верхней вкладке окна редактора программ, либо под его родным именем, либо сменив имя по команде **Save As** (Сохранить как). При запоминании проекта всегда имеется возможность заменить стандартное имя Project1 на более осмысленное (**Save Project As**). Наконец, при изменении в ряде файлов проекта проще выполнить команду **Save All** (Сохранить все) или нажать соответствующую быструю кнопку в полоске инструментов.
- Команды **Close** (Закрыть) и **Close All** (Закрыть все) закрывают либо активный файл, либо все файлы проекта.
- С помощью команд **New Form** (Новая форма) и **New Frame** (Новая рамка) мы можем присоединить к текущему проекту новую форму. В первом случае она будет создана автоматически при старте приложения. Во втором случае дополнительные усилия придется приложить для активизации присоединенной формы.
- Команда **Include Unit Hdr** (Включить заголовочный файл) подключает к текущему модулю заголовочный файл другого модуля, чтобы получить доступ к разрешенным процедурам и данным другой программы.

В меню **Edit** (Правка) вы встретите стандартные операции по модификации текстов программ и состава компонент, расположенных на форме, — **Copy** (Копировать), **Cut** (Вырезать), **Paste** (Вставить), **Delete** (Удалить), **Select All** (Выделить все). Как и в любом редакторе имеется возможность отказа от сделанных исправлений (**Undelete**) и отмены неверно произведенного отката (**Redo**).

В состав этого раздела включены все операции по выравниванию позиций выделенных объектов и унификации их размеров (об этом подробно рассказано в *разд. 2.7*).

- Команда **Flip Children** (Перестроить дочерние окна) позволяет перестроить последовательность расположения дочерних окон, выдвигая на передний план очередное окно.

Меню **Search** (Найти) наряду с обычными процедурами поиска (**Find**) и замены (**Replace**) позволяет производить поиск не только по активному файлу, но и по всем файлам проекта (**Find in Files**). Еще одна разновидность поиска предоставляется командой **Incremental Search** (Пошаговый поиск), которая производит поиск по мере того, как вы набираете поисковый образ. Может оказаться, что нужный результат будет получен при наборе всего части слова.

Переход к строке с заданным номером (**Goto Line Number**) может облегчить путешествие по длинной программе, хотя механизм закладок (Bookmarks), реализованный в VB, гораздо удобнее.

Упомянем еще об одной команде, которой даже опытные программисты пользуются крайне редко. Речь идет о переходе по адресу (**Goto Address**),

который вы узнали в момент аварийного останова выполнявшейся программы. Дело в том, что установить соответствие между адресом ошибочной команды и номером строки исходной программы без помощи системы вам не удастся. Команда позволяет перейти от адреса к той строке программы, в которой возникла аварийная ситуация.

Команды раздела **View** (Вид) в основном предназначены для перехода в то или иное окно, которое в данный момент оказалось невидимым. Среди таких окон наряду с формами (**Forms**) и программами (**Units**) текущего проекта на экране можно проявить Менеджер Проекта (**Project Manager**), Инспектор Объектов (**Object Inspector**), окна отладки (**Debug Windows**), окно команд выравнивания положения и размеров объектов (**Alignment Palette**) и другие. Команда **Toggle Form/Unit** (Переключение Форма/Модуль) или функциональная клавиша <F12> позволяют переключаться с формы на программу, окна которых имеют обыкновение загораживать друг друга.

Меню **Project** (Проект) относится к разряду часто посещаемых. Наиболее важными операциями, выполняемыми командами этого раздела, являются:

- подключение к проекту новых программных или заголовочных файлов и библиотек (**Add To Project**);
- исключение из проекта ненужных разделов (**Remove From Project**);
- компиляция отдельных модулей (**Compile Unit**) или измененных частей проекта (**Make Project**);
- полная перекомпиляция проекта (**Build Project**).

К последней мере прибегают тогда, когда вы получаете абсолютно необъяснимые сообщения об ошибках, которых в вашей программе заведомо нет (имеет место стандартная ситуация — ошибаются и системы программирования).

Наконец, очень важной строкой в разделе **Project** (Проект) является команда **Options** (Параметры), вызывающая окно еще более устрашающего вида, чем в среде VB (рис. 2.29). Оно содержит 14 вкладок, на которых можно установить или изменить текущие значения параметров, управляющих работой системы. Делать это надо осторожно, но без изменения некоторых параметров не обойтись. К числу таких установок относится режим создания приложения — отладочный (**Full Debug**) или завершающий (**Release**), с включением в состав exe-файла процедур из динамических библиотек или использование последних во время работы программы (**Use Dynamic DLL**), назначение путей к личным библиотекам и включаемым файлам и т. п. Набив некоторые синяки и шишки на первых порах, вы обязательно овладеете этими секретами. Не забывайте только запоминать ранее действующие установки. Вообще говоря, годится и такой метод — вы создаете каталог с полной копией своего приложения. Так как система запоминает проект вместе с файлом конфигурации системы (в BCB — файл с расширением dsk,

в Delphi — с расширением cfg), то в каждом из каталогов будут сохранены все установки, и вернуться к предыдущему варианту будет не сложно.

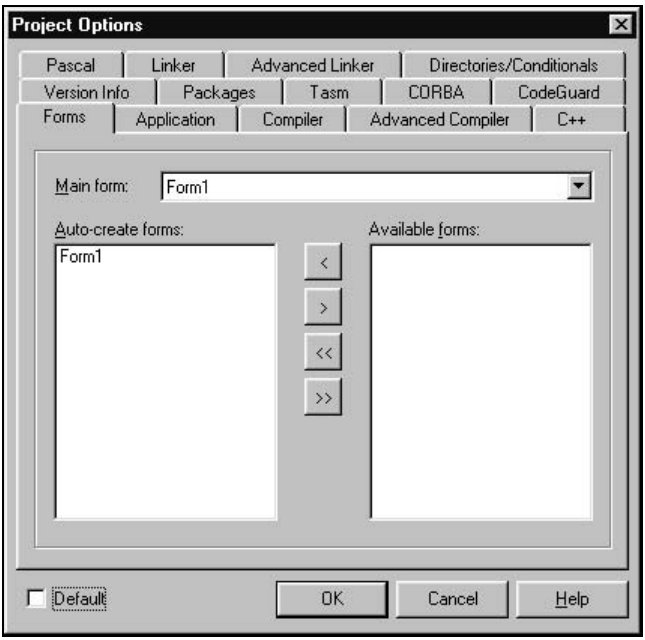


Рис. 2.29. Окно настройки параметров среды BCB

В меню **Run** (Выполнить) чаще всего эксплуатируется команда **Run** (Выполнить), выполняющая компиляцию и стартующая приложение. Аналогом этой команды является нажатие клавиши <F9> или щелчок по кнопке со стартовым треугольником. Если приложение ведет себя странным образом (например, молча имитирует бурную деятельность или выдает ахинею), то его работу можно приостановить командой **Program Pause** (Пауза). Для повторного запуска программы, во время работы которой произошло какое-то ЧП, перед командой **Run** (Выполнить) рекомендуется произвести восстановление нормальных условий для нового старта — выполнить команду **Program Reset** (Восстановление программы).

В меню **Run** (Выполнить) сосредоточены все отладочные средства, позволяющие назначить точки останова (**Run To Cursor**, **Add Breakpoint**), заказать визуальный контроль тех или иных переменных и выражений (**Add Watch**), перейти на пошаговое выполнение программы с заходом в вызываемые функции (**Trace Into**) или с выполнением их в автоматическом режиме (**Step Over**). Пройдя несколько шагов в вызванной процедуре, можно выполнить ее оставшуюся часть на автомате (**Run Until Return**). В момент останова программы вы можете просматривать значения любых доступных переменных, подведя курсор мыши к их имени. Команда **Inspect** (Вставить) вызывает

диалоговое окно, в котором можно просматривать как значения отдельных переменных, так и элементов массивов. При этом имеется возможность управлять форматом просматриваемых величин. Команда **Evaluate/Modify** (Вычислить/Модифицировать) уже давно присутствует в составе отладочных средств ряда систем программирования. С ее помощью в момент останова выполнявшейся программы можно вычислить значение нужного выражения и заменить текущее значение переменной более правильным по вашему мнению. А затем продолжить выполнение программы с учетом сделанных изменений.

Такие разделы главного меню, как **Component** (Компонент), **Database** (База данных) и **Tools** (Инструменты), пригодятся вам на более профессиональном уровне. В них предусмотрены средства для разработки своих компонент, подключения новых компонент, разработанных другими авторами, мощные инструменты для общения с базами данных (**SQL Monitor**), многочисленные окна для настройки более сотни параметров, управляющих работой визуальной среды. Одно из таких окон со свойствами редактора программ приведено на рис. 2.30.

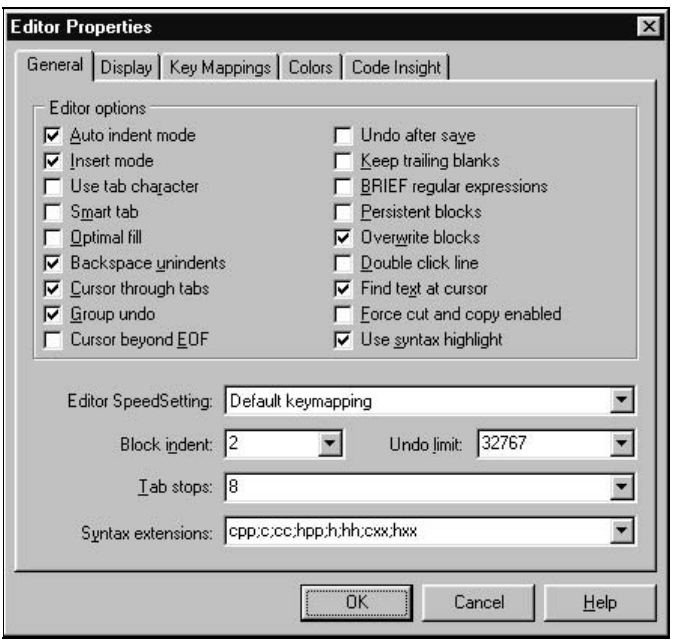


Рис. 2.30. Окно, вызываемое командой **Editor Options**

Кроме раздела **Tools** (Инструменты) вы встретите термин **Repository** (Репозитарий) в меню **Project** (Проект) (команда **Add To Repository**). Это — персональный склад различных заготовок (частично или полностью завершенных форм, диалоговых окон, проектов, модулей данных и т. п.), куда вы можете складывать все, что сможет пригодиться в дальнейшей работе.

Последний раздел главного меню напоминает службу спасения, к которой можно обратиться при нажатии клавиши <F1> или поплутав по заголовкам содержания (**Contents**) и списку ключевых слов (**Index**). Последние версии BCB и Delphi приглашают всех желающих на свои многочисленные сайты (Borland Home Page, Borland Community Page, C++ Builder Home Page, C++ Builder Developer Support, C++ Builder Direct). Использование справочной службы мало чем отличает продукцию фирмы Borland от аналогичных файлов помощи в других системах.

2.9. Как заставить объекты реагировать на события

Большинство объектов, размещенных на форме, пребывает в состоянии спячки до тех пор, пока к кому-нибудь не поступит определенное сообщение. Сообщение, адресованное конкретному объекту, является следствием какого-либо *события* (Event), возникающего по инициативе пользователя, служебной или прикладной программы.

События, инициированные пользователем, могут возникнуть в результате взаимодействия человека с клавиатурой или кнопками мыши. К другим событиям, инициатором которых является операционная система, относятся истечение заданного интервала времени (таймер), создание и уничтожение различных окон (форм), активизация или утрата активности такими компонентами, как кнопки, окна вывода и т. п. Менее приятными событиями, возникающими в результате алгоритмических ошибок, являются различные *аварийные ситуации* (Exception) — деление на 0, переполнение разрядной сетки, обращение к библиотечным функциям с недопустимыми значениями аргументов и т. п.

Событие, инициировавшее сообщение тому или иному объекту, может остаться незамеченным, если для обработки этого сообщения в приложении не предусмотрена специальная процедура — обработчик события. Иногда объект может "заметить" пришедшее сообщение, но не сделать ничего полезного. Например, вы щелкнули по кнопке, она изобразила движение вниз, на ее поверхности появился признак активизации (пунктирный прямоугольник), но ничего больше это событие не вызвало. Причина тому — пользователь решил не реагировать на событие, которое не имеет отношения к его приложению.

Что нужно сделать, чтобы приложение выполнило какую-то полезную работу в момент поступления того или иного сообщения? В первую очередь, вы должны предупредить визуальную среду о желании подключить к своему приложению обработчик выбранного события.

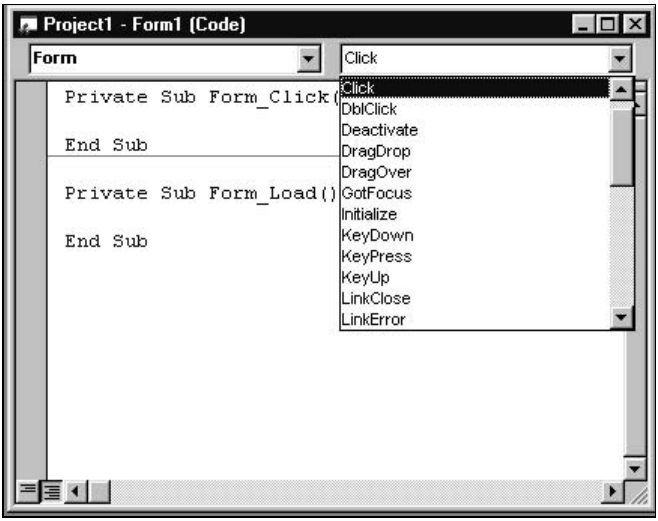


Рис. 2.31. Подготовка к набору обработчика события `Click` в среде VB

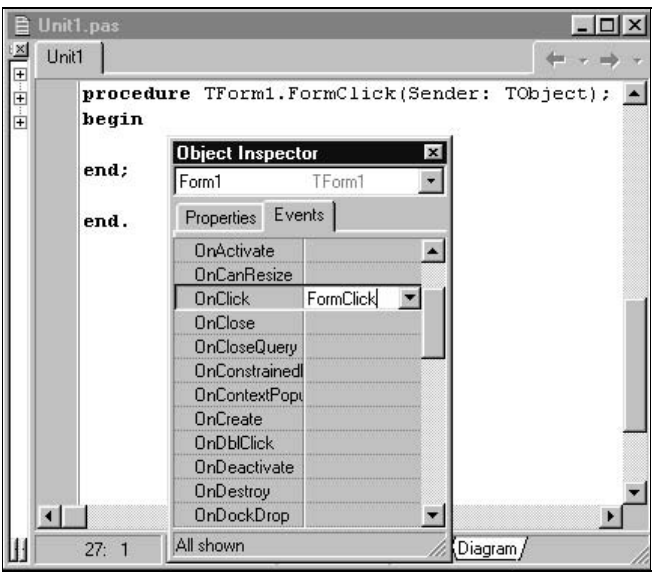


Рис. 2.32. Подготовка к набору обработчика события `OnClick` в среде Delphi

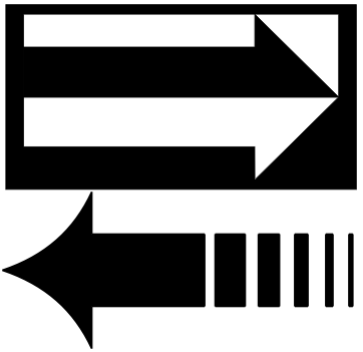
В среде VB, находясь в окне редактора программы, вы должны заглянуть в правый список событий, на которые может реагировать выбранный в левом списке компонент. Переместите выделенный прямоугольник на нужное событие и щелкните кнопкой мыши. В поле редактора немедленно появится заготовка для обработчика этого события, включающая заголовок процедуры с перечнем параметров, характерных для данного сообщения. На рис. 2.31

приведен вид экрана в момент начала создания обработчика события `Click` (щелчок мышью), адресованного форме. Аналогичная операция в среде Delphi и BCB выполняется с помощью вкладки **Events** (События) в окне Инспектора объектов (**Object Inspector**) — рис. 2.32. Вас не должна смущать небольшая разница в названии процедуры обработчика одного и того же события (`Click` и `OnClick`) и наличие дополнительного параметра в передаваемом сообщении. В средах Delphi и BCB любое сообщение включает параметр `Sender`, обозначающий объект, инициировавший появление события. Иногда на форме могут располагаться несколько однотипных объектов, а обработчик события — один на всех. В этой ситуации полезно знать, кто из объектов подал сигнал.

К плюсам визуальных сред относится то, что программист не обязан помнить список и тип всех параметров, сопровождающих конкретное сообщение. Автоматически включаемая заготовка пустой процедуры содержит эту информацию, а обращение к разделу **Help** (Справка) поможет вам разобраться с каждым из полученных аргументов.

Глава 3

Обработка строк



3.1. Строковые данные в Borland C++ Builder и методы их обработки

Строки Pascal и Basic долгое время были предметом зависти программистов, работавших на С. Мало того, что вместо одной строковой переменной в С приходилось заводить массив типа `char`. Необходимо было самому заботиться о том, чтобы в конце программно сформированного предложения находился байт с нулевым кодом. Надо было не забывать о подключении заголовочного файла `string.h` и вместо абсолютно прозрачного сравнения типа `if(str1>str2)` набирать менее вразумительное `if(strcomp(s1,s2)>0)`. С появлением класса `AnsiString` эти заботы канули в лету, однако появились новые, т. к. длинные строки потянули за собой порядка 40 новых функций (методов).

Объекты класса `AnsiString`, в первом приближении, можно рассматривать как обычные переменные символьного типа. Их можно складывать (операция конкатенации) и сравнивать, используя естественные операции отношения. С помощью различного рода конструкторов класса такие переменные можно объявлять и присваивать им различные начальные значения:

```
AnsiString as1 = AnsiString();    // пустая строка
AnsiString as2 = AnsiString(s1);  // строка s2 — копия строки s1
AnsiString as = AnsiString(150);  // строка — целое число
AnsiString as = AnsiString(3.14); // строка — вещественное число
char ch[5]={'A','B','C','D','E'}; // обычный массив в стиле Си
AnsiString s3 = AnsiString(ch);   // строка из 5 символов "ABCDE"
AnsiString s4 = AnsiString(ch,3); // строка из 3 символов "ABC"
```

По способу своего хранения длинные строки BCB напоминают обычные строки языка Pascal (`string`). Однако на указание длины здесь выделен не 1,

а 4 байта, что позволяет работать со строками, на длину которых накладывает ограничение лишь размер оперативной памяти. Интересной новинкой в данных типа `AnsiString` является возможность хранения и обработки двухбайтовых символов таблицы UNICODE (в help-материалах им соответствует аббревиатура MBCS — Multi-Byte Character Set (Многобайтный набор символов). В связи с тем, что на эту кодировку отечественные пользователи еще не перешли, мы ограничимся методами, обслуживающими однобайтовую кодировку символов.

Объявление длинных строк можно совместить с их инициализацией, напоминающей аналогичную операцию с массивами типа `char`:

```
AnsiString s1, s2 = "ABCDEFGH";
```

Длинную строку можно заполнить заданным символом с помощью метода `StringOfChar`, первый аргумент которого задает код символа-заполнителя, а второй — количество повторений:

```
s1.StringOfChar('=',20);
```

Подобно Паскалю BCB-программа имеет доступ к любому символу длинной строки:

```
s1=s2[5];
```

Индексы длинной строки отсчитываются от 1, поэтому в результате приведенного выше оператора присваивания в переменной `s1` окажется буква "E". С помощью метода `Length` можно определить количество символов в ANSI-строке:

```
int k=as.Length();
```

Нулевая длина соответствует пустой, не инициализированной строке. Узнать об этом можно и с помощью метода `IsEmpty`, выдающего значение `true`, если анализируемая строка пуста:

```
bool b1=s1.IsEmpty();
```

Указатель на последний значащий символ можно получить, не обращаясь к определению длины:

```
char *k;
k=s1.AnsiLastChar();
```

Три метода `TrimLeft`, `TrimRight` и `Trim` позволяют отсечь либо только лидирующие (слева), либо только хвостовые (справа), либо и головные и хвостовые пробелы.

Для отсечения `k`-символов с конца строки (среди них могут находиться и не только пробелы) можно воспользоваться методом `SetLength`:

```
s1.SetLength(3);
```

Аргумент этой функции устанавливает новую длину, которая не может быть больше числа символов, находившихся в `s1`.

Для добавления новых символов к концу строки используется операция конкатенации:

```
AnsiString s1="Привет!";
s1=s1+" Вася";
s1 += " и Петя";
```

В строке типа `AnsiString` можно выделить (метод `SubString`) или удалить (метод `Delete`) подстроку заданной длины `d`, начинающуюся символом с индексом `j`:

```
s2=s1.SubString(j,d);
s1.Delete(j,d);
```

Вставка подстроки `s2` в строку `s1` после символа `s1[j]` осуществляется следующим образом:

```
s1.Insert(s2,j);
```

Для поиска индекса первого вхождения строки `s2` в строку `s1` используется метод `AnsiPos` или `Pos`:

```
AnsiString s1="Abcdefgh", s2="cd",s3="cb";
int k=s1.AnsiPos(s2);          // результат k=3
int j=Pos(s1,s3);              // результат j=0
```

Над данными типа `AnsiString` можно выполнять любые сравнения, используя стандартные операции отношения — `>`, `>=`, `<`, `<=`, `==`, `!=`. Обратите внимание, что методы типа `AnsiCompare` и `AnsiCompareIC` могут привести к неожиданным результатам из-за использования других кодировочных таблиц:

```
AnsiString s1="a",s2="A";
int k=s1.AnsiCompare(s2);
```

Этот пример дает `k=-1`, что свидетельствует о соотношении `s1<s2`. Метод `AnsiCompareIC` игнорирует разницу между кодами больших и малых букв.

В классе `AnsiString` предусмотрено довольно много различных преобразований типов данных:

```
int v = s.ToInt();           // перевод строки в целое число
int v = s.IntDef(int d);     // то же самое, при ошибке в данных
                             // выдает значение d по умолчанию
double v = s.ToDouble();     // перевод строки в вещественное число
s1 = InToHex(100,5);         // (10)→(16), результат = "00064"
s1 = CurrToStr(1.5);         // результат = "1,5"
s1 = AnsiString(v);          // перевод в строку данных любого типа
s1 = FloatToStrF(1993.14159265,формат,n1,n2)
```

В последнем преобразовании значения первого аргумента в строку участвуют форматный указатель, количество верных значащих цифр (`n1`) и общее

количество цифр в дробной части (n2). Результаты этого преобразования на конкретных данных приведены в табл. 3.1.

Таблица 3.1. *Результат преобразования числа с плавающей точкой в строку в зависимости от формата*

Формат	n1	n2	Строка результата
ffGeneral	6	10	1993,14
ffExponent	6	10	1,99314E+3
ffFixed	6	10	1993,1400000000
ffNumber	6	10	1993,1400000000
ffCurrency	6	10	1993,1400000000р

Одним из наиболее часто используемых методов преобразования ANSI-строк в обычные строки С, заканчивающиеся нулевым байтом, является процедура c_str(). Она применяется в тех случаях, когда вызываемая функция в качестве аргумента ожидает указатель на массив типа char:

```
printf("%s",s.c_str());
```

С помощью методов LowerCase и UpperCase производится замена кодов латинских букв на коды малых или больших букв.

Одно из преимуществ данных типа AnsiString по сравнению с прежними массивами С заключается в том, за выделением памяти под их хранение система следит сама. И как только новое значение изменяет длину строки, ей выделяется новое место. Конечно, это замедляет работу программы, но предохраняет ее от ошибок, связанных с записью за пределы массива.

3.2. Отображение строк в поле метки

Объекты типа Label используются для вывода различных подписей, располагаемых рядом с другими объектами, нуждающимися в пояснениях. Подпись, которая отображается в поле метки, должна быть присвоена ее заголовку — свойству Caption. Такое присвоение можно выполнить как на стадии проектирования программы, так и во время работы приложения:

```
Label1.Caption = "Фамилия"          ' VB
Label1.Caption := 'Фамилия';          { Delphi}
Label1->Caption = "Фамилия";          // BCB
```

Надпись может быть выполнена одним из шрифтов, установленных в операционной системе. Ее размер и дополнительные характеристики шрифта управляются комплексным свойством Font, щелчок по которому выводит на экран знакомое многим диалоговое окно (см. рис. 2.11).

Поле метки имеет ограниченные размеры, установленные либо до, либо во время выполнения программы. Может оказаться, что установленная ширина

окна не позволяет разместить нужную подпись. Для выхода из такого положения предлагаются два варианта. Либо ширина и высота поля метки автоматически подстраиваются под габариты размещаемого заголовка в пределах допустимого формой пространства. Такой режим устанавливается свойством `AutoSize=True`. Либо при фиксированной ширине окна подбирается такая его высота, которая позволяет расположить подпись в две или более строк. В этом случае при `AutoSize=False` потребуется изменить значение свойства `WordWrap` на `True`.

В отличие от VB, где метка обладает свойством `BorderStyle`, в средах BCB и Delphi поле метки не выделяется какой-либо границей и ее фон сливается с фоном формы, если не предпринимать какие-либо специальные меры. Однако фон метки можно изменить, задав то или иное значение свойства `Color` и изменив значение свойства прозрачности (`Transparent=False`). В среде Visual Basic цвет фона метки и цвет символов устанавливаются свойствами `BackColor` и `ForeColor`.

Текст в поле метки по умолчанию прижимается к левой границе компонента, однако его можно расположить симметрично относительно центра поля или прижать к правой границе. Выбор той или иной возможности достигается изменением значения свойства `Alignment`. В среде VB используются мнемонические константы `LeftJustify`, `Center` и `RightJustify`, в средах BCB/Delphi — `taLeftJustify`, `taCenter` и `taRightJustify`.

3.3. Работа с объектами типа "Редактируемая строка"

Объекты типа "Редактируемая строка" предназначены для ввода и/или вывода символьных строк с предоставлением некоторых возможностей, характерных для текстовых редакторов. В среде VB к объектам этой группы относятся компоненты `TextBox` и `MaskEdBox`. В средах BCB/Delphi близкими возможностями располагают объекты типа `TEdit` и `TMaskEdit`. В отличие от поля метки указанные объекты являются активными — в среде Windows принято говорить, что таким объектам может быть передан фокус. Это означает, что сообщения, поступающие с клавиатуры, адресуются объекту, находящемуся в фокусе.

3.3.1. Использование компонента *TextBox* в Visual Basic

Основное свойство `Text` можно рассматривать как переменную символьного типа, значение которой отражается в окне компонента `TextBox`. Как только программа заносит в эту переменную какое-либо сообщение, оно мгновенно

появляется в соответствующем окне. Если компонент активизирован, то в окне мерцает вертикальная черточка — курсор, и пользователь может изменить содержимое окна. Например, набрать новый текст или модифицировать прежнее содержимое, используя типичные для текстового редактора приемы, — перемещения вдоль текста, выделение фрагмента, удаление или копирование фрагмента, замена символов, обмен с Clipboard (Буфер обмена) и т. п.

Передача фокуса окну ввода с именем `Input1` производится путем обращения к методу `SetFocus`:

```
Input1.SetFocus()
```

При запуске приложения фокус передается окну ввода автоматически, если это окно было первым помещено на форму. В остальных случаях программа должна активизировать окно ввода в тот момент, когда ей необходимо обратиться к пользователю. Если приложение находится в режиме ожидания какого-либо события, пользователь с помощью мыши может активизировать то или иное окно ввода.

Как программа может определить, набрал ли пользователь новое сообщение или модифицировал прежнее содержимое окна? Для этой цели необходимо предусмотреть один или несколько обработчиков событий, адресованных окну ввода. Среди многочисленных событий, на которые может реагировать объект `TextBox`, наиболее полезными являются следующие:

- `GotFocus()` — событие, возникающее в момент активизации окна;
- `LostFocus()` — событие, возникающее в момент потери фокуса;
- `KeyDown(KeyCode,Shift)` — событие, возникающее в момент движения нажимаемой клавиши вниз;
- `KeyPress(KeyAscii)` — событие, возникающее при удержании нажатой клавиши;
- `KeyUp(KeyCode,Shift)` — событие, возникающее при отпускании нажатой клавиши;
- `Change()` — событие, возникающее при изменении, добавлении или удалении очередного символа в поле ввода.

В событиях, связанных с клавиатурой, обработчик получает код ASCII (`KeyAscii`) или клавиатурный код (`KeyCode`) нажатой клавиши и числовой код `Shift` из диапазона [0,7], определяющий сочетание управляющих клавиш <Alt>, <Ctrl> и <Shift>, нажатых одновременно с клавишей. Не все программисты четко разделяют такие понятия, как *ASCII-код*, *скан-код* и *клавиатурный код*. Кодировочные таблицы операционных систем, именуемые кодовыми страницами (например, 866-я страница в MS-DOS или 1251-я страница в Windows), включают однобайтовые коды от 0 до 255, соответствующие всем отображаемым символам клавиатуры и некоторым управляющим символам (перевод строки, возврат каретки, табуляция и т. п.). Однако

в таблицах ASCII вы не найдете коды таких клавиш, как <F1>, <F2>, <Shift>, <Insert> и др. Для работы с полным набором клавиш в программах низкого уровня обычно используются скан-код (scan-code), представляющий собой порядковый номер клавиши на клавиатуре (нумерация клавиш ведется по рядам слева направо и сверху вниз). Таблицы скан-кодов были придуманы фирмой IBM давно, когда количество клавиш и их расположение на клавиатуре было несколько иным (функциональные клавиши, например, располагались не в верхнем ряду, а были расположены в две колонки слева, цифровой клавиатуры тоже сначала не было). Поэтому не надо сравнивать последовательность скан-кодов с расположением клавиш на современных клавиатурах, единства среди которых сегодня нет ни по числу клавиш, ни по их расположению. В программах более высокого уровня пришлось отказаться от только однобайтового представления кода, считываемого с клавиатуры. Вы, наверное, помните, что при нажатии некоторых клавиш программы чтения их кодов сначала выдают нулевое значение, и только при повторном обращении к соответствующей процедуре вы получите отличный от нуля код, который не является кодом ASCII. В среде Windows принято использовать так называемые *виртуальные коды клавиатуры* (Virtual Key — виртуальные клавиши). В системе Visual Basic для обозначения кодов виртуальных клавиш используется мнемонические константы, большая часть которых приведена в табл. 3.2.

Таблица 3.2. *Мнемонические константы, применяемые в среде Visual Basic, для обозначения кодов*

Константы VB	KeyCode	Пояснение
`VbKeyBack`	8	Клавиша <Backspace>
`VbKeyTab`	9	Клавиша <Tab> (совпадает с кодом ASCII)
`VbKeyReturn`	13	Клавиша <Enter>
`VbKeyShift`	16	Клавиша <Shift>
`VbKeyControl`	17	Клавиша <Ctrl>
`VbKeyEscape`	27	Клавиша <Esc> (совпадает с кодом ASCII)
`VbKeySpace`	32	Пробел (совпадает с кодом ASCII)
`VbKeyPageUp`	33	Клавиша <Page Up>
`VbKeyPageDown`	34	Клавиша <Page Down>
`VbKeyEnd`	35	Клавиша <End>
`VbKeyHome`	36	Клавиша <Home>
`VbKeyLeft`	37	Стрелка влево (←)
`VbKeyUp`	38	Стрелка вверх (↑)

Таблица 3.2 *(окончание)*

Константы VB	KeyCode	Пояснение
VbKeyRight	39	Стрелка вправо (→)
VbKeyDown	40	Стрелка вниз (↓)
VbKeyInsert	45	Клавиша <Insert>
VbKeyDelete	46	Клавиша <Delete>
VbKey0 — VbKey9	48–57	Коды цифр совпадают с их ASCII-кодами
VbKeyA — VbKeyz	65–90	Коды букв совпадают с их ASCII-кодами
VbKeyF1 — VbKeyF12	112–123	Коды функциональных клавиш <F1> — <F12>

Во многих случаях приходится учитывать не только клавиатурный код нажатой клавиши, но и ситуацию, в которой была нажата клавиша. Для того чтобы различать коды больших и малых букв, клавиша может нажиматься в комбинации с управляющей клавишей регистра (<Shift>). Разобраться в сигналах, поступающих в событиях клавиатуры, вам поможет приводимый ниже пример.

Приложение 3.1 (VB). Анализ кодов клавиатуры

Для его реализации вы должны поместить на форме четыре метки — `Label1`, `Label2`, `Label3` и `Label4`. Установите следующие заголовки в этих метках:

```
Label1.Caption = Код ASCII
Label2.Caption = Клавиатурный код
Label3.Caption = Комбинация управляющих клавиш
Label4.Caption = Символ
```

Справа от каждой метки поместите окна `Text1`, `Text2`, `Text3` и `Text4`, защищенные от ввода (свойство `Locked=True`). Форма первого приложения может иметь вид, приведенный на рис. 3.1.

Включите в текст программы обработчики следующих событий:

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
  Text2.Text = KeyCode
  msg$ = ""
  If Shift And 1 Then msg$ = "SHIFT+" & msg$
  If Shift And 2 Then msg$ = "CTRL+" & msg$
  If Shift And 4 Then msg$ = "ALT+" & msg$
  If Len(msg$) > 0 Then msg$ = Left(msg$, Len(msg$) - 1)
```

```
  Text3.Text = msg$
End Sub

Private Sub Text1_KeyPress(KeyAscii As Integer)
  Text1.Text = KeyAscii
  Text4.Text = Chr$(KeyAscii)
End Sub

Private Sub Text1_KeyUp(KeyCode As Integer, Shift As Integer)
  Text1.Text = ""
  Text2.Text = ""
  Text3.Text = ""
  Text4.Text = ""
End Sub
```

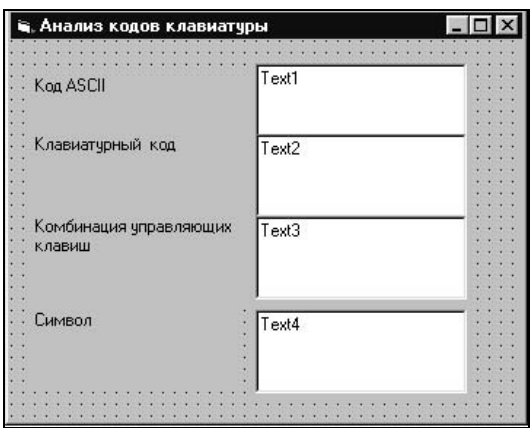


Рис. 3.1. Форма приложения для анализа кодов клавиатуры

При нажатии любой клавиши первым возникает событие `KeyDown`. Его обработчик заносит клавиатурный код (значение параметра `KeyCode`) в окно `Text2`. О преобразовании числового значения кода в символьную строку (свойство `Text` имеет тип `String`) позаботится система. Анализ сочетания управляющих клавиш, нажатых одновременно с исследуемой клавишей, сводится к проверке разрядов логической шкалы состояния клавиатуры, представленной значением параметра `Shift`. В этой шкале единица в нулевом бите соответствует нажатию клавиши <Shift>, единица в первом бите — нажатию клавиши <Ctrl>, единица во втором бите — нажатию клавиши <Alt>. По каждому единичному биту к символьной переменной `msg$` добавляется наименование соответствующей клавиши. Если значение `msg$`

оказалось не пустым, то последний символ (лишний знак +) устраняется и сформированная комбинация заносится в окно Text3. Обратите внимание на новую особенность входного языка VB — вместо знака + для обозначения операции конкатенации (сцепления текстов) здесь предпочитают использовать амперсанд (&).

Вторым по очереди генерируется событие KeyPress, обработчику которого передается ASCII-код нажатой клавиши. Обработчик заносит этот код в окно Text1, а соответствующий символ — в окно Text4.

Последнее событие KeyUp, возникающее при возврате клавиши в исходное состояние, используется для сброса содержимого всех окон. Поэтому результат работы программы можно наблюдать только в тот момент, когда исследуемая клавиша удерживается в сочетании с управляющими клавишами.

Приведенный пример демонстрирует, что обработчики событий клавиатуры могут определить, нажал ли пользователь обусловленную клавишу. Например, нажатие клавиши <Enter> с кодом vbKeyReturn=13 может быть воспринято как признак окончания набора очередного сообщения. Воспользуемся таким соглашением для демонстрации техники приема символьного сообщения, набираемого пользователем.

Приложение 3.2 (VB). Прием и подтверждение приема сообщения

Поместим на форме окно TextBox с нестандартным именем Input1 и компонент Label1, в свойстве которого будет отображаться текст принятого сообщения. Перед тем, как начать набор текста обработчика события KeyDown, не забудьте изменить значение свойства Name (по умолчанию система назначит имя Text1).

```
Private Sub Input1_KeyDown(KeyCode As Integer, Shift As Integer)
 If KeyCode = vbKeyReturn Then     ' Нажата ли клавиша Enter?
  Label1.Caption= Input1.Text      ' Перенос принятого сообщения
  Input1.Text=""                   ' Очистка окна ввода
  Input1.SetFocus()                ' Передача фокуса окну ввода
 End If
End Sub
```

В экспериментах с описанным приложением каждое нажатие клавиши <Enter> (и не только <Enter> — попробуйте нажать <Esc>) сопровождается звуковым сигналом, который может раздражать пользователя. Выдачей этого сигнала управляет операционная система, однако его можно подавить. Для этой цели надо сделать так, чтобы код клавиши <Enter> не был передан обработчику события Input1_KeyDown.

Приложение 3.3 (VB). Предварительная цензура сообщений клавиатуры

Перехват сообщения может осуществить форма, если в ее свойство KeyPreview заслать значение True. Тогда обработчик события клавиатуры, приписанный к форме, получит сообщение о нажатии клавиши раньше, чем оно будет передано в процедуру Input1_KeyDown. Таким образом, у формы появляется возможность предварительного анализа сообщений, адресованных другим компонентам. Если обработчик формы по каким-то причинам не пожелает передать поступивший код символа настоящему адресату, то он зашлет в параметр KeyAscii нулевое значение. Для того чтобы избежать отображения символа q, у которого KeyAscii совпадает с кодом клавиши <F2>, в обработчике события KeyPress выполняется присвоение KeyAscii=0 при нажатии любой клавиши:

```
Private Sub Form_KeyPress(KeyAscii As Integer)
 KeyAscii = 0
End Sub

Private Sub Text1_KeyUp(KeyCode As Integer, Shift As Integer)
 If KeyCode = vbKeyF2 Then
   Msg$ = "F2."
   If Shift And 1 Then Msg$ = "SHIFT+" & Msg$
   If Shift And 2 Then Msg$ = "CTRL+" & Msg$
   If Shift And 4 Then Msg$ = "ALT+" & Msg$
   Text1.Text = "Вы нажали " & Msg$
 End If
End Sub
```

В приведенном ранее обработчике события KeyDown операция, передающая фокус окну ввода после нажатия клавиши <Enter>, является лишней. Дело в том, что на форме находится всего два компонента. Один из них — Label1 — является пассивным и фокус получить не может. Поэтому окно ввода Input1 в данном приложении всегда находится в фокусе. Однако на форме могут находиться и другие объекты, способные стать активными, а такая передача может состояться и по инициативе пользователя. Он мог что-то набрать в одном окне, а затем, не нажимая клавиши <Enter>, мышкой активизировать другое окно. В этом случае наблюдение за событием KeyDown не позволит определить момент завершения операции ввода. Чтобы отследить такой финт со стороны пользователя, в программу нужно включить обработчик события LostFocus, возникающий в момент потери фокуса. Событие GotFocus можно использовать для подготовки окна ввода к приему нового сообщения. Продемонстрируем предлагаемые идеи на примере следующего приложения.

Приложение 3.4 (VB). Прием сообщений в разных окнах

Разместим на форме два окна ввода с именами `Input1` и `Input2`, а также два выводных окна с именами `Output1` и `Output2` (рис. 3.2).

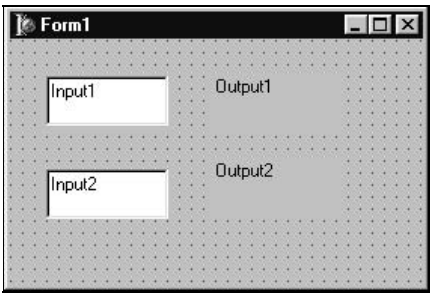


Рис. 3.2. Прием сообщений в разных окнах

Пусть схема работы приложения представляет собой бесконечный цикл, в котором:

1. Пользователь набирает очередное сообщение в активном окне ввода.
2. По нажатию клавиши <Enter> или в случае потери фокуса принятое сообщение отображается в соответствующем окне вывода.
3. Активизируется альтернативное окно ввода и все повторяется с начала.

Для выхода из бесконечного цикла можно использовать стандартные средства — кнопку с крестиком на форме или соответствующую команду системного меню. Первая половина программы приведенного ниже приложения снабжена подробным комментарием.

```
Private Sub Input1_GotFocus()
' Реакция на получение фокуса окном Input1
  Input1.Text = ""    'Сброс содержимого поля ввода
End Sub

Private Sub Input1_KeyPress(KeyAscii As Integer)
' Дополнительная реакция на код нажатой клавиши в окне Inpit1
 If KeyAscii = vbKeyReturn Then    ' Если нажата клавиша Enter
' Перенос текста в заголовок метки Output1
    Output1.Caption = Input1.Text
    Input2.SetFocus             ' Передача фокуса окну Input2
  End If
 End Sub

Private Sub Input1_LostFocus()
```

```
 ' Реакция на потерю фокуса окном Input1
 ' Перенос текста в заголовок метки Output1
  Output1.Caption = Input1.Text
  Input2.SetFocus    ' Передача фокуса окну Input2
End Sub

Private Sub Input2_GotFocus()
  Input2.Text = ""
End Sub

Private Sub Input2_KeyDown(KeyCode As Integer, Shift As Integer)
 If KeyCode = vbKeyReturn Then
    Output2.Caption = Input2.Text
    Input1.SetFocus
    KeyCode = 0
  End If
End Sub

Private Sub Input2_LostFocus()
  Output2.Caption = Input2.Text
  Input1.SetFocus
End Sub
```

К числу достаточно востребованных свойств окна ввода относится работа с выделенным фрагментом текста. Возможность такого выделения по инициативе пользователя вам, наверное, знакома по работе с текстовыми редакторами. Пользователь может прибегнуть либо к услугам клавиатуры (управление курсором при нажатой клавише <Shift>), либо воспользоваться мышкой. Информация о выделенном фрагменте поступает в следующие свойства компонента:

- `SelStart` — номер начальной позиции в строке `Text` (позиции отсчитываются от 0);
- `SelLength` — длина выделенного фрагмента;
- `SelText` — выделенный текст.

Используя перечисленные свойства объекта `TextEdit`, программа тоже имеет возможность произвести выделение любого фрагмента в окне. К таким выделениям обычно прибегают для того, чтобы удалить часть сообщения или заменить какой-либо фрагмент на что-то другое. Если окно теряет фокус, то обычно выделение сбрасывается. Но если вы зададите `HideSelection=False`, то фрагмент останется выделенный и после потери фокуса.

Возможно, что объект `TextEdit` предназначается только для вывода сообщений из программы подобно метке `Label`. Чтобы запретить пользователю доступ к содержимому окна, необходимо изменить значение свойства `Locked`:

```
Text1.Locked=True
```

Естественно, что размеры окна ввода ограничены шириной компонента, установленной вами при создании приложения (свойство `Width`). Однако длина вводимого текста может оказаться существенно больше. При этом в поле будет видна только часть текста, и для просмотра начала или конца сообщения придется двигать курсор влево или вправо. Вообще говоря, длину вводимого текста можно ограничить, изменив его свойство `MaxLength`:

```
Text1.MaxLength = 20
```

В этом случае система будет пресекать любую попытку ввести больше, чем 20 символов. На каждое нажатие сверх установленной нормы последует мелодичный звук. По умолчанию свойству `MaxLength` присвоено нулевое значение, которое не запрещает ввод совсем, а лишь ограничивает его длину 32 767 символами. Может быть, вам потребуется вводить или выводить более длинные строки. Измените свойство `MaxLength`:

```
Text1.MaxLength = 50000
```

Максимальная длина текста не должна превышать 64 Кбайт. Очевидно, что рассматривать строки такой длины — удовольствие не из приятных. Поэтому компонент `TextEdit` из строки может превратиться в страницу, для чего достаточно воспользоваться свойством `MultiLine`:

```
Text1.Multiline = True
```

В этом случае вы можете при наборе текста нажимать клавишу <Enter>, что приведет к продолжению набора с начала следующей строки. Автоматический перенос текста в начало следующей строки происходит при достижении правой границы поля ввода, если вы запретили использовать линейки прокрутки (`ScrollBars=None`). Конечно, это — не Word, не допускающий разрыва слов или делающий переносы по всем правилам.

В процессе работы приложения программным путем можно изменить шрифт (свойство `NameFont`) и установить такие его характеристики, как высота заглавных символов в пунктах (свойство `FontSize`), ширина символов (свойство `Font.Width`), утолщение контуров букв (свойство `FontBold`) и их наклон (свойство `FontItalic`), режим подчеркивания (свойство `FontStrikeThrough`) или перечеркивания символов (свойство `FontUnderline`). Например:

```
Text1.FontName = "Times New Roman"
Text1.FontSize = 12
Text1.FontItalic = True
```

При создании приложения щелчок по свойству `Font` приводит к появлению знакомого окна выбора параметров шрифта.

3.3.2. Использование объекта *MaskEdBox* в Visual Basic

Объект `MaskEdBox` предназначен для ввода символьных данных по установленному шаблону и отличается от компонента `TextEdit` тем, что на отдельные позиции поля ввода накладывается одно из двух ограничений:

- в данной позиции пользователь может набрать один из символов, разрешенных шаблоном;
- в данной позиции пользователь ничего набрать не может, т. к. ее значение повторяет соответствующую позицию шаблона. При вводе курсор такие позиции пропускает, поэтому изменить их содержимое нельзя.

Таким образом, шаблон, значение которого устанавливается в свойстве `Mask`, играет роль фильтра вводимых данных.

Среди стандартных компонент панели **General** (Общие) объект `MaskEdBox` отсутствует и для его подключения к приложению необходимо проделать следующее.

1. Войти в позицию главного меню **Project** (Проект) и выполнить команду **Components** (Компоненты).
2. В появившемся диалоговом окне на вкладке **Controls** (Элементы управления) выбрать строку **Microsoft Masked Edit Control** (Элемент управления редактор маски) (рис. 2.5), и щелчком мыши установить галочку в соответствующем квадратике слева.
3. После нажатия кнопки **OK** новый компонент дополнит стандартный набор. Этим набором сможет пользоваться только данное приложение, в противном случае вновь добавляемые к разным приложениям компоненты будут засорять экран.

Шаблон ввода представляет собой символьную строку, которую мы заносим в свойство **Mask** (Маска) либо на стадии проектирования приложения, либо во время его работы. Среди символов шаблона различают управляющие символы, контролирующие вводимые в данной позиции коды, и литералы — символы, переносимые в поле ввода программой. Перечень управляющих символов приведен в табл. 3.3.

Таблица 3.3. *Перечень символов маски, принятых в среде Visual Basic*

Управляющий символ маски	Символ, который должен или может быть набран в данной позиции
#	Цифра (ввод обязателен)
9	Цифра (ввод не обязателен, позицию можно пропустить)

Таблица 3.3 *(окончание)*

Управляющий символ маски	Символ, который должен или может быть набран в данной позиции
A	Цифра или буква (ввод обязателен)
A	Цифра или буква (ввод не обязателен)
&	Любой символ с кодом 32-126 или 128-255 (ввод обязателен)
C	Любой символ с кодом 32-126 или 128-255 (ввод не обязателен)
>	Буква будет преобразована в заглавную
<	Буква будет преобразована в строчную
.	Десятичная точка, отделяющая целую часть числа от дробной
,	Запятая — разделитель тысяч в больших числах
:	Двоеточие — разделитель компонент времени (часы: минуты: секунды)
/	Разделитель компонент даты (год / месяц / день)

Символы маски, не относящиеся к разряду управляющих, считаются литералами, которые переносятся в поле ввода без изменения и не могут быть исправлены пользователем. Естественно, возникает вопрос — а как включить в состав литералов те нормальные символы, которые отнесены к управляющим. В этом случае в маске перед нужным символом ставится обратная косая черта, которой не соответствует отдельная позиция в поле ввода. Например, `Mask = "(0\95)-###-##-##"`. Здесь символ `"9"` рассматривается не как управляющий, а как литерал, позволяющий задать междугородний телефонный код Москвы. Для включения в состав литералов символа `backslash` нужно задать две обратные косые черты — `"\\"`.

Во время выполнения программы контролируемые позиции поля ввода выделены знаком подчеркивания. Однако это не всегда удобно. Используя два свойства компонента `MaskEdBox`, вы можете заменить символ подчеркивания на любой другой, более подходящий по смыслу:

```
AllowPrompt = True      'Разрешить замену подчерка
PromtChar = "*"         'Назначить новый символ
```

Продемонстрируем использование окна ввода по шаблону на примере ввода телефонных номеров.

Приложение 3.5 (VB). Ввод номера телефона

Поместим на форму объект `MaskEdBox` и установим следующие значения его свойств:

```
Name = Input1
Mask = (831-2)-##-##-##
```

Чтобы лучше разглядеть вид шаблона на стадии проектирования, увеличим размер букв маски до 14 пунктов (рис. 3.3).

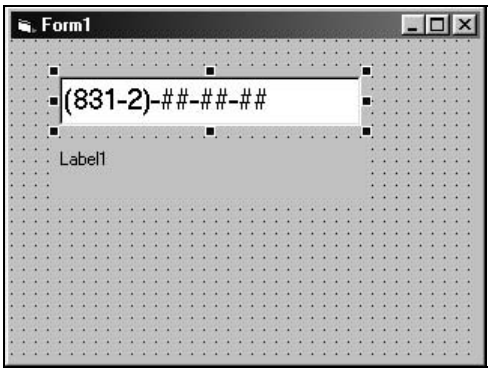


Рис. 3.3. Маска для ввода телефонных номеров

Пусть наше приложение предлагает пользователю ввести телефонный номер, и после нажатия клавиши <Enter> переносит поступившее значение в заголовок метки `Label1`. После этого на поле ввода восстанавливается маска для приема следующего номера. Ниже приводится текст модуля формы:

```
Private Sub MaskInput_KeyUp(KeyCode As Integer, Shift As Integer)
 If KeyCode = 13 Then
   Label1.Caption = MaskInput.Text
   MaskInput.Mask = ""
   MaskInput.Text = ""
   MaskInput.Mask = "(831-2)-##-##-##"
 End If
End Sub
```

Обратите внимание на то, что для очистки окна перед вводом следующего номера, приходится сначала сбросить маску, а потом восстановить ее. Очистка поля ввода без сброса маски приведет к генерации ошибочной ситуации. Вид приложения после его старта приведен на рис. 3.5.

При наборе номера можно столкнуться с некоторой недоработкой компонента. Если вы начнете набор с одной из цифр, включенных в литерал, то курсор переходит в соответствующую позицию литеральных символов вместо того, чтобы переместиться в начальную позицию номера. Правда, искажения литеральных символов при этом не происходит. Но если первая же цифра телефона не содержится в начальном литерале, то ввод без осложнений начинается с нужной позиции. В позиции поля ввода, которой соответствует управляющий символ маски `"#"`, можно набрать только цифру.

Нажатие таких клавиш как <F1> игнорируется молча, при нажатии какой-либо буквы раздается звуковой сигнал. Такой же сигнал вы услышите, если попытаетесь ввести текст, превышающий длину маски.

На рис. 3.4 видно, что подчерки в позициях ввода сливаются. Можно выделить каждую из вводимых позиций, если заменить свойство `PromptChar`, например, на звездочку (см. рис. 3.5).

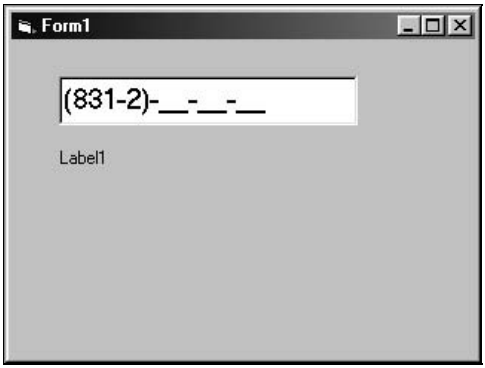


Рис. 3.4. Начальный вид приложения перед вводом номера

Рис. 3.5. Окно с нестандартной маской в процессе ввода

Литеральные символы маски попадают в свойство `Text`, однако от них можно избавиться, если выделить всю строку и скопировать выделенный текст в Clipboard. Если значение свойства `ClipMode` при этом равно `1%` (или константе `mskExcludeLiterals`), то в буфер обмена литеральные символы не передаются. По умолчанию `ClipMode=0`, что соответствует режиму сохранения литералов при копировании или вырезании.

3.3.3. Специфика компонента *RichTextBox* в Visual Basic

Компонент `RichTextBox` представляет поле, в котором может отображаться и редактироваться произвольный текст, закодированный либо в формате ASCII, либо в формате RTF (Rich Text Format, расширенный текстовый формат). В последнем случае текст может содержать различные шрифтовые выделения, примененные к отдельным фрагментам документа, выполненного, например, в MS Word (рис. 3.6). По сравнению с многострочным вариантом окна `TextBox` здесь имеется гораздо больше возможностей по редактированию текстовых документов. Во-первых, у объекта `RichTextBox` нет ограничения на длину текста. Во-вторых, с ним может быть связан txt- или rtf-файл, содержимое которого сразу же загружается, как только имя файла будет присвоено свойству `FileName`. В-третьих, с компонентом `RichTextBox` может быть автоматически связано всплывающее меню, которое появляется

в момент нажатия правой клавиши мыши (рис. 3.7). Наконец, здесь представлены некоторые возможности по редактированию параграфов и страниц RTF-документа (поля, отступы и т. п.).

Приложение 3.6 (VB)

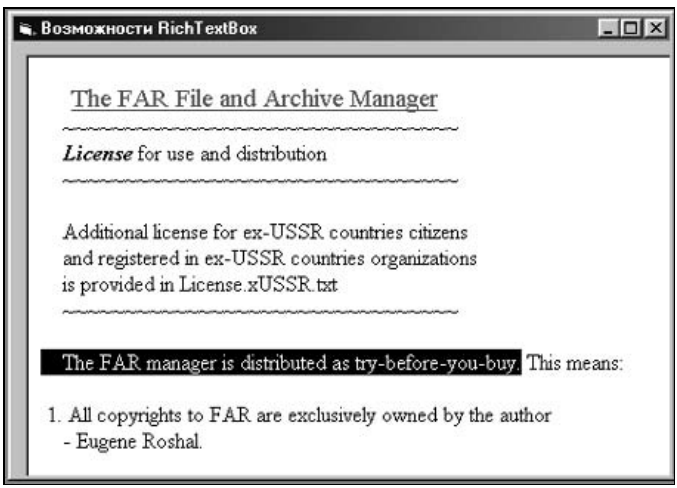


Рис. 3.6. Документ формата RTF в окне компонента `RichTextBox`

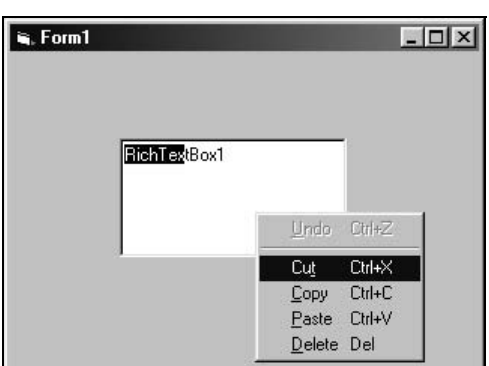


Рис. 3.7. Всплывающее меню у компонента `RichTextBox`

Методы, обслуживающие объект `RichTextBox`, позволяют производить поиск заданного текста (метод `Find`), вывод на принтер выделенного фрагмента (метод `SelPrint`), запоминать текущий документ в дисковом файле (метод `SaveFile`) и считывать содержимое другого документа (метод `LoadFile`). На базе компонента `RichTextBox` довольно легко построить несложный текстовый редактор, и в некоторых книгах вы можете найти более

или менее изящные примеры такой реализации (см. пример \Examples\Apps\RichEdit). Более подробное обсуждение возможностей объекта `RichTextBox` не входит в задачи нашей книги.

3.3.4. Использование компонент типа *TEdit* в средах Borland C++ Builder и Delphi

Компоненты типа `TEdit` используются для ввода и/или вывода однострочных символьных сообщений. Их возможности и набор свойств в значительной мере повторяют соответствующие возможности и свойства объекта `TextBox`, работающего в одностроковом режиме. Главным свойством этих компонент также является свойство `Text`, которое в BCB имеет тип `AnsiString`, а в Delphi — тип `String`.

Первое, не очень существенное, отличие объектов `Edit` от `TextBox` заключается в возможности автоматически подстраивать размеры окна ввода/вывода под длину сообщения `Text`. Это происходит в том случае, если свойству `AutoSize` присвоено значение `True`. За размеры формы при этом никто выходить не собирается — более длинные строки приходится прокручивать влево или вправо с помощью клавиш управления курсором.

Вторая особенность окон типа `TEdit` состоит в возможности выделить сообщение, находящееся в окне в момент получения им фокуса. Для этой цели появилось новое свойство `AutoSelect`. По умолчанию его значением является `False`, и для выделения текста или его фрагмента необходимо прибегнуть к установке значений свойств `SelStart` (первая позиция выделенного текста) и `SelLength` (длина выделенного текста). При `AutoSelect=True` выделение текста, находящегося в окне, происходит автоматически.

Ограничение на длину вводимого или выводимого сообщения по-прежнему устанавливается значением свойства `MaxLength`. Однако строки такой длины как 32 или 64 Кбайт здесь не употребляются. Блуждание по строкам такой длины лишено смысла.

Ограничение доступа пользователя к содержимому окна вывода здесь устанавливается свойством с более привычным названием — `ReadOnly`.

Формат и названия обработчиков событий, наиболее активно используемых при вводе/выводе текстовых сообщений, немного отличаются от аналогичных процедур VB:

- `Enter(Sender)` — событие, возникающее в момент активизации окна;
- `Exit(Sender)` — событие, возникающее в момент потери фокуса;
- `KeyDown(Sender, Key,Shift)` — событие, возникающее в момент движения нажатой клавиши вниз;

- `KeyPress(Sender, Key)` — событие, возникающее при удержании нажатой клавиши;
- `KeyUp(Sendr, Key,Shift)` — событие, возникающее при отпускании нажатой клавиши;
- `Change(Sender)` — событие, возникающее при изменении, добавлении или удалении очередного символа в поле ввода.

Параметр `Sender`, появившийся в этих процедурах, является указателем на объект, пославший соответствующее сообщение. Параметр `Key` в обработчиках событий `KeyDown` и `KeyUp` имеет тип `Word`, тогда как событие `KeyPress` передает однобайтовый код `Key` типа `char`. Параметр `Shift`, передаваемый событиями клавиатуры, представляет собой множество, которое может быть пустым или содержать от одного до трех значений, — `ssShift`, `ssAlt` и `ssCtrl` — в зависимости от сочетания управляющих клавиш, нажатых одновременно с клавишей <Key>.

Передача фокуса окну ввода производится путем обращения к методу `SetFocus`.

Напомним, что задача первого приложения заключалась в приеме кода клавиши <F2> в любой комбинации с управляющими клавишами <Alt>, <Ctrl> и <Shift>. Нажатия любых других клавиш должны игнорироваться. Для реализации последнего ограничения воспользуемся свойством формы `KeyPreview`, которое позволяет перехватить сообщение клавиатуры, проанализировать код нажатой клавиши и запретить дальнейшую передачу информации фактическому адресату.

Небольшое отличие в обработчике события `FormKeyPress` между BCB и Delphi связано с типом параметра `Key`. В C с данными типа `char` можно работать как с числами — поэтому значение `Key` можно сравнивать с числовым клавиатурным кодом клавиши <F2>. В Pascal такая операция привела бы к ошибке из-за несовпадения типов, поэтому числовой код 113 пришлось преобразовать в символ с помощью функции `Chr`. Аналогичный прием пришлось применить в Pascal и для засылки нулевого кода в переменную `Key`.

Более существенная разница между программами обработчиков события `EditKeyDown` заключается в анализе значения параметра <Shift>. Тип его значения `TShiftState` определен как множество, содержащее несколько значений, соответствующих битовым значениям в шкале состояния управляющих клавиш клавиатуры (<Alt>, <Ctrl>, <Shift>) и мыши. Если для Pascal тип данных `Set` был родным, и для анализа вхождения значения в текущий набор множества существовала операция `in`, то в BCB такой тип данных появился недавно. И вместо операции `in` приходится прибегать к методу `Contains`. Он выдает логическое значение `True`, если аргумент содержится в проверяемом объекте.

Приложение 3.7 (BCB). Анализ клавиатурного кода

```
void __fastcall TForm1::FormKeyPress(TObject *Sender, char &Key)
{ // Запрет приема кодов клавиш, отличных от F2
  if(Key != VK_F2) Key=0;
  Caption="";        // Очистка заголовка формы
}
//-----------------------------------------------------------------
void __fastcall TForm1::Edit1KeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{ AnsiString msg="Вы нажали F2 ";
  if(Key==VK_F2)        // Если нажата клавиша F2
    { msg=(Shift.Contains(ssAlt))? msg+"+Alt":msg;
      msg=(Shift.Contains(ssCtrl))? msg+"+Ctrl":msg;
      msg=(Shift.Contains(ssShift))? msg+"+Shift":msg;
      Caption=msg;
    }
  else Caption="";    // Очистка заголовка формы
}
```

Приложение 3.8 (Delphi). Анализ клавиатурного кода

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
     Key:=Chr(0);
end;

procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  msg:String;
begin
  msg:='';
  if Key=VK_F2 then
    begin
     msg:='Вы нажали F2 ';
     if ssAlt in Shift then msg:=msg+'+ Alt ';
     if ssCtrl in Shift then msg:=msg+'+ Ctrl ';
     if ssShift in Shift then msg:=msg+'+ Shift';
```

```
    end;
  Caption:=msg;
end;
```

Приложение 3.9 (BCB). Прием и подтверждение приема сообщения

```
void __fastcall TForm1::Input1KeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
  if(Key==13)
    { Label1->Caption=Input1->Text;
      Input1->Text="";
      Input1->SetFocus();
    }
}
```

Приложение 3.10 (Delphi). Прием и подтверждение приема сообщения

```
procedure TForm1.Input1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key=13 then begin
     Label1.Caption:=Input1.Text;
     Input1.Text:='';
     Input1.SetFocus;
   end;
end;
```

Приложение 3.11 (BCB). Предварительная цензура сообщений клавиатуры

```
void __fastcall TForm1::FormKeyPress(TObject *Sender, char &Key)
{
  Key=0;
}
//---------------------------------------------------------------------
----

void __fastcall TForm1::Edit1KeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
```

```
  AnsiString msg="Вы нажали F2 ";
  if(Key==VK_F2)
    { if(Shift.Contains(ssAlt))  msg += " + Alt ";
      if(Shift.Contains(ssCtrl)) msg += " + Ctrl ";
      if(Shift.Contains(ssShift))msg += " + Shift ";
      Edit1->Text=msg;
    }
}
```

В приложении 3.12 не забудьте установить свойство формы `KeyPreview=True`.

Приложение 3.12 (Delphi). Предварительная цензура сообщений клавиатуры

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  Key:=Chr(0);
end;

procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  msg:string;
begin
  if Key=VK_F2 then begin
     msg:='Вы нажали F2 ';
     if ssAlt in Shift then msg:=msg+'+ Alt ';
     if ssCtrl in Shift then msg:=msg+'+ Ctrl ';
     if ssShift in Shift then msg:=msg+'+ Shift';
     Edit1.Text:=msg;
  end;
end;
```

Приложение 3.13 (BCB). Прием сообщений в разных окнах

```
void __fastcall TForm1::FormActivate(TObject *Sender)
{
  Edit1->SetFocus();    //Передача фокуса окну Edit1
}

void __fastcall TForm1::Edit1KeyUp(TObject *Sender, WORD &Key,
```

```
      TShiftState Shift)
{
   if(Key==13)        // Если нажата клавиша Enter
     { Label1->Caption=Edit1->Text;    // Перенос принятого сообщения
       Edit2->Text="";    // Очистка окна Edit2
       Edit2->SetFocus();    // Активизация окна Edit2
     }
}
//---------------------------------------------------------------
void __fastcall TForm1::Edit2KeyUp(TObject *Sender, WORD &Key,
      TShiftState Shift)
{  if(Key==13)        // Если нажата клавиша Enter
     { Label2->Caption=Edit2->Text;    // Перенос принятого сообщения
       Edit1->Text="";    // Очистка окна Edit2
       Edit1->SetFocus();    // Активизация окна Edit1
     }
}
//-----------------------------------------------------------------------
void __fastcall TForm1::Edit1Exit(TObject *Sender)
{// Обработка ситуации "Потеря фокуса"
  Label1->Caption=Edit1->Text;
  Edit2->Text="";
}
//-----------------------------------------------------------------------
void __fastcall TForm1::Edit2Exit(TObject *Sender)
{// Обработка ситуации "Потеря фокуса"
  Label2->Caption=Edit2->Text;
  Edit1->Text="";
}
//-----------------------------------------------------------------------
```

Приложение 3.14 (Delphi). Прием сообщений в разных окнах

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Edit2.SetFocus    { Передача фокуса окну Edit2 }
end;

procedure TForm1.Edit1KeyUp(Sender: TObject; var Key: Word;
```

```
  Shift: TShiftState);
begin
  if Key=13 then begin        { Если нажата клавиша Enter }
     Label1.Caption:=Edit1.Text;    { Перенос принятого сообщения }
     Edit2.Text:='';        { Очистка окна Edit2 }
     Edit2.SetFocus;        { Активизация окна Edit2 }
  end;
end;

procedure TForm1.Edit2KeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key=13 then begin        { Если нажата клавиша Enter }
     Label2.Caption:=Edit2.Text;    { Перенос принятого сообщения }
     Edit1.Text:='';        { Очистка окна Edit2 }
     Edit1.SetFocus;        { Активизация окна Edit1 }
  end;
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  Label1.Caption:=Edit1.Text;
  Edit2.Text:='';
end;
procedure TForm1.Edit2Exit(Sender: TObject);
begin
  Label2.Caption:=Edit2.Text;
  Edit1.Text:='';
end;
```

Приложение 3.15 (VB). Индикатор клавиатуры

Описываемое ниже приложение отображает большую часть клавиатуры (рис. 3.8).

Во время его работы вы можете нажать любую клавишу в сочетании с управляющими клавишами <Shift>, <Ctrl> и <Alt>. При этом нажатые клавиши подсвечиваются (точнее — надписи на них выделяются жирным шрифтом), а в двух тестовых окнах отображаются код ASCII и клавиатурный код, соответствующие нажатой комбинации. Для правильной работы приложения переключите клавиатуру в режим набора английских букв. На рис. 3.9 вид главной формы в момент нажатия клавиши <A>.

Рис. 3.8. Клавиатура с усеченным набором клавиш

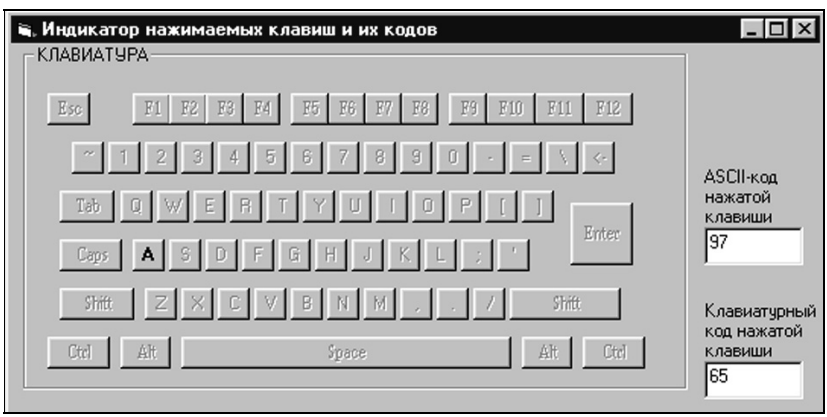


Рис. 3.9. Фрагмент работы приложения

Это приложение демонстрирует возможность объединения однородных объектов (в нашем случае таковыми являются командные кнопки) в массив. Преследуются при этом две цели. Во-первых, для однотипных событий, инициаторами которых являются элементы массива, используются общие обработчики. Во-вторых, индексы, присваиваемые объектам, могут оказаться полезными при решении конкретных задач.

Что надо сделать для того чтобы командные кнопки, имитирующие клавиши клавиатуры, стали группой? Во-первых, надо поместить на форму контейнер. В нашем примере выбран простейший элемент — `Frame`. Заменим значение свойства `Caption`, вписываемое в разрыв рамки, на слово "Клавиатура". Во-вторых, у первой же кнопки, помещаемой внутрь рамки, сформируем значение свойства `Index`. По умолчанию в соответствующей строке окна свойств ничего нет, но как только мы впишем туда какое-то числовое значение, VB понимает, что такая кнопка причисляется к группе.

Обычно индексы начинают отсчитывать от 0, автоматически увеличивая их значение на 1. Но в нашем случае удобнее в качестве индекса выбрать клавиатурный код клавиши. Это имеет и свои минусы, т. к. клавиатурные коды не образуют сплошного числового интервала. Поэтому при появлении индекса, не относящегося к выбранным клавишам, возможно возникновение аварийных ситуаций (индекс есть, а соответствующего элемента массива нет). Например, нажатие клавиши, не включенной в выбранное подмножество, вызовет сообщение об ошибке.

Несмотря на то, что при нажатии любой клавиши генерируется всего 3 события и для их обработки вполне достаточно приводимых ниже программ, объем приложения превышает 50 Кбайт. Объясняется это довольно просто — в нашем приложении присутствует более 70 командных кнопок. А для каждой из них нужно определить габариты и место в контейнере (`Height`, `Width`, `Left`, `Top`), заголовок (`Caption`), индекс (`Index`), шрифт и его параметры (`Font`), признак доступности (`Enabled`). Вот из этого и складывается объем проекта, хотя большинство его составных частей носит единообразный характер. Единственное исключение из этой схемы составляют особые клавиши, выступающие парами, — <Shift>, <Ctrl> и <Alt>. Клавиатурный код правой и левой клавиши каждой из этих пар одинаков, но у двух разных элементов массива индексы совпадать не могут. Поэтому мы искусственно назначим правым клавишам код левых клавиш, увеличенный на 1 000. Ниже приведен фрагмент проекта, соответствующий описанию командной кнопки с надписью **A**:

```
Begin VB.CommandButton Command
   Caption         =   "A"
   Enabled         =   0   'False
   BeginProperty Font
      Name            =   "MS Sans Serif"
      Size            =   8.25
      Charset         =   204
      Weight          =   400
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   Height          =   315
   Index           =   65
   Left            =   1200
   TabIndex        =   49
   TabStop         =   0   'False
   Top             =   1920
```

```
      Width            =   315
   End
```

В предыдущем приложении достаточно подробно объяснено, как формируются биты параметра `Shift` (не путать с клавишей <Shift>), передаваемые процедурам `KeyDown` и `KeyUp`. В обработчиках соответствующих событий большую часть программы занимают строки, которые устанавливают или сбрасывают значения свойств `Enabled` и `FontBold` у системных клавиш.

```
.........................
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
 Command(KeyCode).Enabled = True
 Command(KeyCode).FontBold = True
 Text2.Text = KeyCode
 If Shift And 1 Then
  Command(16).Enabled = True:
  Command(1016).Enabled = True:
  Command(16).FontBold = True:
  Command(1016).FontBold = True
 End If
 If Shift And 2 Then
  Command(17).Enabled = True:
  Command(1017).Enabled = True:
  Command(17).FontBold = True:
  Command(1017).FontBold = True
 End If
 If Shift And 4 Then
  Command(18).Enabled = True:
  Command(1018).Enabled = True:
  Command(18).FontBold = True:
  Command(1018).FontBold = True
 End If
End Sub

Private Sub Text1_KeyPress(KeyAscii As Integer)
 Text1.Text = KeyAscii
End Sub

Private Sub Text1_KeyUp(KeyCode As Integer, Shift As Integer)
 Command(KeyCode).FontBold = False
 Command(KeyCode).Enabled = False
```

```
 Command(9).FontBold = False
 Command(9).Enabled = False
 Text1.Text = ""
 Text2.Text = ""
 If Shift = 0 Then
  Command(16).Enabled = False:
  Command(1016).Enabled = False:
  Command(16).FontBold = False:
  Command(1016).FontBold = False
  Command(17).Enabled = False:
  Command(1017).Enabled = False:
  Command(17).FontBold = False:
  Command(1017).FontBold = False
  Command(18).Enabled = False:
  Command(1018).Enabled = False:
  Command(18).FontBold = False:
  Command(1018).FontBold = False
 End If
End Sub
```

3.3.5. Специфика работы с объектом *MaskEdit* в средах Borland C++ Builder и Delphi

Объект `MaskEdit` предназначен для ввода символьных строк по шаблону, заданному значением свойства `EditMask`. В отличие от среды VB набор шаблона и установка символа замещения (`Character for Blanks`) осуществляются здесь в специальном диалоговом окне (рис. 3.10). Здесь же вам продемонстрируют вид окна ввода, каким он будет в работающем приложении. Дополнительные удобства заключаются в возможности просмотреть и выбрать типовые маски для ввода наиболее распространенных данных — телефонов, кодов адреса, дат, значений времени и т. п. Галочка слева от метки **Save Literal Characters** (Сохранить литеральные символы) позволяет считывать введенный код из свойства `Text`, включая литеральные символы маски. Если галочка в этой строке отсутствует, то считанное значение содержит только набиравшиеся символы. Если содержимое окна ввода извлекается из свойства `EditMask`, то вместе с набранными символами извлекаются и литералы независимо от метки в окне **Input Mask Editor** (Редактор маски ввода). Маску ввода можно извлечь и из файла с расширением dem. Для этого в редакторе маски имеется кнопка **Masks** (Маски).

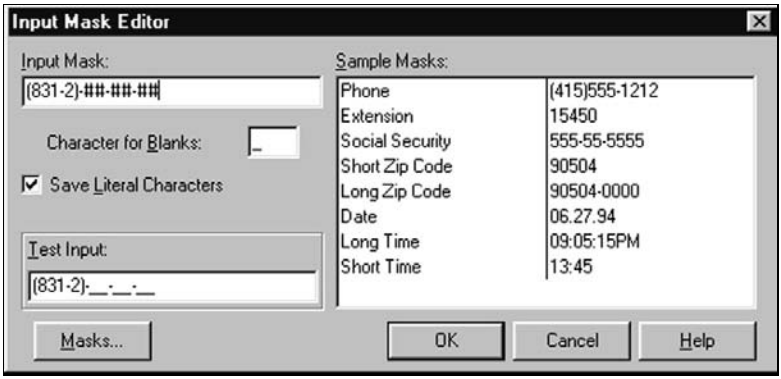


Рис 3.10. Окно редактора маски ввода

Управляющие символы маски здесь несколько иные, чем в среде VB (табл. 3.4).

Таблица 3.4. *Управляющие символы маски ввода сред BCB и Delphi*

Управляющий символ маски	Символ, который должен или может быть набран в данной позиции
!	Если это символ представлен в шаблоне, то недовведенные необязательные символы превращаются в лидирующие пробелы. В противном случае недостача восполняется пробелами в конце введенной строки
>	Все буквы в разрешенных позициях, начиная с данной, будут заменяться большими до тех пор, пока не встретится символ '<'
<	Все буквы в разрешенных позициях, начиная с данной, будут заменяться малыми до тех пор, пока не встретится символ '>'
<>	Блокировка замены вводимых букв кодами другого регистра
\	Следующий символ считать литералом, даже если он совпадает с одним из управляющих
L	Буква A-Z, a-z (ввод обязателен)
l	Буква A-Z, a-z (ввод не обязателен)
A	Буква или цифра (ввод обязателен)
a	Буква или цифра (ввод не обязателен)
C	Любой символ (ввод обязателен)
c	Любой символ (ввод не обязателен)
0	Цифра (ввод обязателен)
9	Цифра (ввод не обязателен)

Таблица 3.4 *(окончание)*

Управляющий символ маски	Символ, который должен или может быть набран в данной позиции
#	Цифра, знак + или – (ввод не обязателен)
:	Разделитель компонент времени — часов, минут, секунд (возможна замена в соответствии с настройкой операционной системы)
/	Разделитель компонент даты — день, месяц, год (возможна замена в соответствии с настройкой операционной системы)
;	Разделитель трех компонент маски
_	Символ подчерк вставляет пробелы в текст

Маска ввода, вообще говоря, представлена тремя компонентами, которые разделяются точкой с запятой. Первая часть — собственно маска. Вторая часть может быть равна `0` или `1` в зависимости от того, сохраняются или не сохраняются литералы в значении свойства `Text`. Третья часть состоит из символа замен, которым отображаются разрешенные позиции в поле ввода.

Алгоритм работы объекта `MaskEdit` отработан гораздо лучше, чем в среде VB. Курсор всегда переходит в нужную позицию, даже если вводимые символы содержатся среди литералов шаблона.

Ниже приводится текст обработчика события `OnChange`, которое происходит при вводе или изменении каждого символа в разрешенных позициях окна `MaskEdit`. Каждый раз при возникновении такого события содержимое окна ввода переносится в заголовок метки `Label1`, и для этого понадобилась всего одна строка программы. Вид окна приложения в процессе работы показан на рис. 3.11. Для сокращения времени набора код города включен в качестве литералов маски.

Приложение 3.16 (BCB). Ввод номера телефона

```
void __fastcall TForm1::MaskEdit1Change(TObject *Sender)
{
  Label1->Caption=MaskEdit1->Text;
}
```

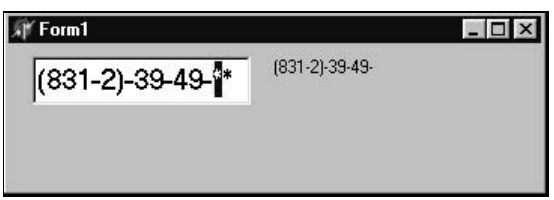


Рис. 3.11. Вид окна в процессе набора номера телефона

Приложение 3.17 (Delphi). Ввод номера телефона

В приводимом ниже приложении добавлен обработчик события `OnKeyPress`, который, принимая код клавиши <Enter>, производит очистку поля ввода. Однако засылка пустой строки в свойство `Text` восстанавливает в окне ввода маску, обеспечивая тем самым подготовку к вводу следующего номера телефона. Точно такая же "очистка" произойдет и в случае, когда вы, выделив все содержимое окна, нажмете клавишу <Del>.

```
procedure TForm1.MaskEdit1Change(Sender: TObject);
begin
  Label1.Caption:=MaskEdit1.Text;
end;

procedure TForm1.MaskEdit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key=Chr(13) then MaskEdit1.Text:='';
end;
```

3.4. Визуализация многострочных документов

3.4.1. Объект *TextBox* в многострочном режиме

Если свойству `MultiLine` присвоено значение `True`, то окно `TextBox` превращается в хранилище текстовой информации, напоминающей рулонный документ с достаточно большим количеством строк. Максимальный объем информации, которую можно поместить в окно, — 64 Кбайт. По такому тексту можно перемещаться, используя стандартные возможности клавиатуры (стрелки управления курсором, клавиши <Home>, <End>, <PageUp>, <PageDown>). С помощью свойства `ScrollBars` (возможные значения — `None`, `Both`, `Vertical`, `Horizontal`) с окном можно связать одну или две линейки прокрутки, делающие перемещения по тексту более комфортными.

Текст в окне может быть набран вручную с клавиатуры, но, наверное, это — не самое лучшее использование многострочного элемента `TextBox`. Чаще в таком окне приходится отображать содержимое какого-либо текстового файла. Загрузка файла может быть осуществлена, например, следующим образом:

```
Open "qq.txt" For Input As 1
Text1.Text=Input$(LOF(1),1)
```

Если длина строки практически не ограничена (`MaxLength=0`, что позволяет иметь строки размером до 32K), то перенос в следующую строку

вызывают управляющие символы. Ограничение длины строки небольшим числом (например, `MaxLength=40`) вызывает перенос по достижению правой границы.

Свойство `Alignment`, принимающее одно из значений `VbCenter`, `VbLeftJustify` или `VbRightJustify`, позволяет центрировать строки или прижимать их к соответствующей границе.

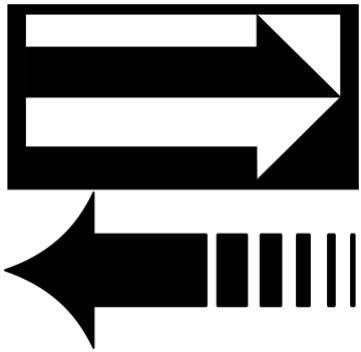
Перечень наиболее важных методов, позволяющих оперировать с содержимым многострочного окна, приведен в табл. 3.5.

Таблица 3.5. *Методы элемента* `Memo` *среды BCB*

Формат вызова метода	Выполняемая функция
`Clear()`	Очистка окна
`SelectAll()`	Выделение содержимого окна
`ClearSelestion()`	Удаление выделенного текста
`CopyToClipboard()`	Копирование выделенного текста в буфер обмена
`CutToClipboard`	Перенос выделенного текста в буфер обмена
`PasteFromClipboard()`	Вставка в окно содержимого буфера обмена
`Undo()`	Отказ от исправлений
`ClearUndo()`	Очистка буфера исправлений
`SetSelTextBuf`	Копирование выделенного текста в буфер
`GetSelTextBuf`	Выборка текста из буфера

Глава 4

Работа со списками



Объект типа *список* представляет собой нечто похожее на массив строк. В отличие от обычных массивов список может изменять свои размеры во время работы программы. Достаточно близким эквивалентом списка является страница текста, содержащая автономные строки (без переноса слов), которые можно условно перенумеровать и производить над ними различные манипуляции:

- отображать на экране весь список, прижимая содержимое его строк к левой границе, к правой границе или располагая их по центру;
- сортировать строки в алфавитном порядке;
- извлекать, пополнять, заменять или удалять указанные строки;
- перемещать строки внутри списка;
- определять позицию строки в списке;
- выделять отдельную строку, совокупность подряд идущих строк или разрозненные строки;
- копировать выделенные строки в буфер обмена и извлекать их оттуда;
- сохранять список в текстовом файле и извлекать его оттуда.

К объектам типа *список* можно отнести компоненты `Memo` (BCB/Delphi), `ListBox`, `ComboBox`, `FileListBox`, `DriveListBox`, `DirListBox`, `Outline` (BCB/Delphi), `Notepad` (BCB/Delphi). Одни из них обладают более универсальными свойствами (например — `ListBox` и `ComboBox`), другие имеют более специализированное назначение. Сравнивая системы программирования, можно заметить, что набор функциональных средств по управлению универсальными списками существенно богаче представлен в BCB и Delphi.

4.1. Работа с объектом типа *Memo*

Объект типа `Memo` (дословный перевод — записная книжка) представлен только в системах BCB и Delphi. Его основное предназначение — отображение

и редактирование многострочных текстовых документов. В этом смысле он эквивалентен компоненту `TextBox` из Visual Basic, работающему в многострочном режиме.

Содержимое поля `Memo` можно лишь отчасти рассматривать как классический список, состоящий из независимых строк. С одной стороны, свойство `Text` представляет содержимое этого объекта как одну длинную символьную строку, содержащую управляющие символы ("Перевод строки", "Возврат каретки", "Горизонтальная табуляция"). Ручное или автоматическое включение таких символов в текст позволяет отображать длинную строку в виде многострочного документа. С другой стороны, объект `Memo` располагает свойством `Lines`, обеспечивающим доступ к каждой строке по ее индексу:

```
Memo1.Lines[0]; { первая строка (Delphi) в окне с именем Memo1}
Memo1.Lines.Strings[0]; { первая строка (Delphi)}
```

Вы заметили, что к одной и той же строке можно обращаться двумя способами, и выбор той или иной формы диктуется тем методом, которым должна обрабатываться строка. Иногда обе формы эквивалентны и, естественно, не надо утруждать себя набором более длинного идентификатора:

```
Memo1.Lines[1]:='Это вторая строка';
Memo1.Lines.Strings[1]:='Это тоже вторая строка';
```

В системе программирования BCB можно встретить даже три варианта обращения к одной и той же строке списка в зависимости от требуемого формата данных:

- `Memo1->Lines[1]` — указатель на значение второй строки списка, как на данное типа `TString`;
- `Memo1->Lines->Strings[1]` — указатель на значение второй строки списка, как на данное типа `AnsiString`;
- `Memo1->Lines->Strings[1].c_str()` — указатель на значение второй строки списка, как на данное типа (`char *`).

Содержимое окна `Memo` может быть сформировано с помощью редактора строк на стадии проектирования программы или загружено из файла во время работы приложения:

```
Memo1.Lines.LoadFromFile('qq.txt');
```

С помощью свойства `Alignment` можно зафиксировать один из трех вариантов прижима текста — к левой границе (`taLeftJustify`), к правой границе (`taRightJustify`) или по центру (`taCenter`).

После старта приложения окно `Memo` может быть активизировано либо программным путем, либо по инициативе пользователя, щелкнувшим мышкой по площади окна. С этого момента вы получаете возможность производить стандартные действия по набору и редактированию текста, используя дос-

тупные клавишные операции, — вставка, удаление, выделение, копирование выделенного текста в буфер обмена и извлечение его оттуда.

Приложение может узнать, внесено ли какое-либо изменение в текст по значению логического свойства `Modified`. Если пользователь выделил фрагмент текста в окне `Memo` и по этому поводу программа должна предпринять определенные действия, то приложение может проанализировать выделенный текст (значение свойства `SelText`), узнать его длину (свойство `SelLength`) и стартовую позицию (свойство `SelStart`). Последнее значение выдает номер позиции первого выделенного символа, отсчитываемый от начала длинной строки.

Свойство `MaxLength` ограничивает длину строки `Text`. Если `MaxLength=20`, то вы не сможете набрать больше 20 символов, даже если ширина окна позволяет это сделать. По умолчанию `MaxLength=0`, что позволяет вводить строки, размером до 64K. Также по умолчанию значение свойства `WordWrap` равно `True`, поэтому по достижении правой границы окна происходит автоматический перенос набираемого текста на следующую строку. Свойство `WantReturns` определяет реакцию системы на нажатие клавиши <Enter> при наборе информации во время работы приложения. По умолчанию значение этого свойства равно `True` и нажатие клавиши <Enter> вызывает переход курсора в окне `Memo` на следующую строку. При `WantReturns=False` для перевода строки придется нажимать комбинацию клавиш <Ctrl>+<Enter>, т. к. действие клавиши <Enter> может быть связано со свойством `Default` некоторой кнопки. Такой же двусмысленностью обладает клавиша <Tab> — ее нажатие по умолчанию означает переход фокуса к следующему объекту на форме. Однако при наборе текста табуляторный пропуск (<Tab> — символ с кодом 9) может оказаться полезным, и тогда необходимо переустановить значение свойства `WantTabs`.

Манипуляции программы над строками поля `Memo`, т. е. над объектами вида `Memo1.Lines`, напоминают о том, что данный объект может считаться списком. Перечень наиболее употребительных методов, работающих с элементами `Memo`, приведен в табл. 4.1. Форматы вызова процедур или функций, указанные в первой колонке, справедливы для языка Object Pascal. В системе BCB точку следует заменить на `"->"` и вместо отсутствующих аргументов поставить пустые скобки `()`. Кроме перечисленных функций и процедур в программе можно использовать самый обычный оператор присваивания, пересылающий строки одного объекта в другой:

```
Memo1->Lines = ComboBox1->Items;
```

Таблица 4.1. *Методы объекта* `Memo` *среды Delphi*

Формат вызова метода	Описание действия
`K:=Memo1.Lines.Add(str)`	Добавление строки `str` в конец `Memo1`, возвращает индекс добавленной строки

Таблица 4.1 *(окончание)*

Формат вызова метода	Описание действия
`Memo1.Lines.AddStrings(список);`	Добавление всех строк из указанного аргумента в конец `Memo1`
`Memo1.Append(str);`	Добавление строки `str` в конец `Memo1`
`Memo1.Lines.Assign(ListBox1.Items);`	Копирование строк из списка `ListBox1` в окно `Memo1`
`Memo1.Clear;`	Очистка поля `Memo1`
`Memo1.ClearSelection;`	Очистка выделенного фрагмента
`Memo1.CopyToClipboard;`	Копирование выделенного фрагмента в буфер обмена
`Memo1.CutToClipboard;`	Вырезание выделенного фрагмента с сохранением в буфере обмена
`Memo1.Lines.Delete(index);`	Удаление строки с указанным индексом
`Memo1.Lines.Equals(Combo1.Items)`	Сравнение строк в указанных списках
`Memo1.Lines.Exchange(ind1,ind2);`	Перестановка указанных строк
`Memo1.GetTextBuf(buf,len);`	Копирование свойства `Text` в буфер заданной длины
`K:=Memo1.GetTextLen;`	Определение длины текста в поле `Memo1`
`K:=Memo1.Lines.IndexOf(str)`	Определение индекса строки списка, совпадающей с аргументом функции. Если такой строки нет, возвращается `-1`
`Memo1.Insert(str[,flag]);`	Вставка строки с текущей позиции курсора
`Memo1.Insert(str,x,y[,flag]);`	Вставка строки с указанной позиции
`Memo1.Lines.Insert(ind,str)`	Вставка строки по указанному индексу
`Memo1.Lines.LoadFromFile('name');`	Загрузка `Memo1` из файла
`Memo1.PasteFromClipboard;`	Вставка содержимого буфера обмена
`Memo1.Lines.SaveToFile('name');`	Запоминание поля `Memo1` в файле
`Memo1.Lines.Move(ind1,ind2);`	Перемещение строки с индексом `ind1` в позицию `ind2`
`Memo1.SelectAll;`	Выделение содержимого поля `Memo1`
`Memo1.UnSelect;`	Отмена выделения

По наименованию и содержанию ряда процедур видно, что объект Memo, прежде всего, был ориентирован на создание простейших текстовых редакторов. В отличие от более сложного компонента RichEdit здесь не используются шрифтовые выделения. Но конструкторам объекта этого показалось мало, и они наделили Memo свойствами и методами классического списка.

4.2. Классический список *ListBox* в системе Visual Basic

Список ListBox используется для отображения строк, предлагаемых на выбор пользователю. Каждая строка списка ListBox имеет индекс, определяющий ее положение в массиве строк. Отсчет индексов ведется от нуля:

```
List1.List(0)' — первая строка в списке с именем List1
```

Количество строк, находящихся в данный момент в списке, является значением свойства ListCount. Именно по этому счетчику организуются циклы, связанные с перебором всех строк списка:

```
For j = 0 To List1.ListCount — 1
  If List1.Selected(j) Then    ' Если j-тая строка выделена
    Label1.Caption = j
  End If
Next i
................
```

Появление конструкции With-End With в языке VB несколько упрощает написание программы, позволяя не задавать полностью составные имена:

```
With List1
For j=0 To ListCount-1
  If Selected(j) Then          ' Если j-тая строка выделена
................
  End If
End With
```

Приложение 4.1 (VB)

Основным приемом работы со списком является выделение одной или нескольких строк, заинтересовавших пользователя, и последующая обработка выделенных строк. Упорядочение строк по алфавиту существенно упрощает просмотр списка. Поэтому каждый список обладает свойством Sorted. По умолчанию его значение равно False, что соответствует не упорядоченному списку. Расположение его строк определяется последовательностью их ввода. Стоит изменить значение этого свойства (Sorted=True), как все строки

выстраиваются в алфавитном порядке. Для строк, содержащих большие и малые буквы, при сортировке разницы между строчными и прописными символами не делается (рис. 4.1).

Рис. 4.1. В левом окне `Sorted=False`, в правом — `Sorted=True`

Приложение 4.2 (VB)

Режим выделения строк зависит от значения свойства `MultiSelect`, принимающего в среде Visual Basic одно из трех значений — `0` (`None`), `1` (`Simple`) и `2` (`Extended`). В первом случае пользователь может выделить только одну строку в списке, щелкнув по ней курсором мыши или нажав клавишу <Пробел> в тот момент, когда на строку наведен пунктирный прямоугольник. Во втором случае можно выделить больше одной строки и выделенные строки не обязаны располагаться подряд. Для такого выделения нужно щелкать по требуемым строкам или использовать стрелки управления курсором в сочетании с клавишей <Пробел>. В третьем случае можно выделить мышкой участок строк, расположенных подряд, используя клавишу <Shift>, или разрозненные строки, используя клавишу <Ctrl>. Указанные ситуации отображены на рис. 4.2.

Количество выделенных строк запоминается в свойстве `SelCount`. Содержимое последней выделенной строки запоминается в свойстве `Text`, а для проверки, является ли строка `List(j)` выделенной, необходимо прибегнуть к проверке значения свойства `Selected(j)`. Оно равно `True`, если `j`-тая строка выделена.

К наиболее распространенным методам обработки списков относятся процедуры очистки (`Clear`), добавления новых строк (`AddItem`) и удаления существующих строк (`RemoveItem`):

```
List1.Clear                  'Очистка списка с именем List1
List1.AddItem str[,index]    'Добавление строки str
List1.RemoveItem index       'Удаление строки с заданным индексом
```

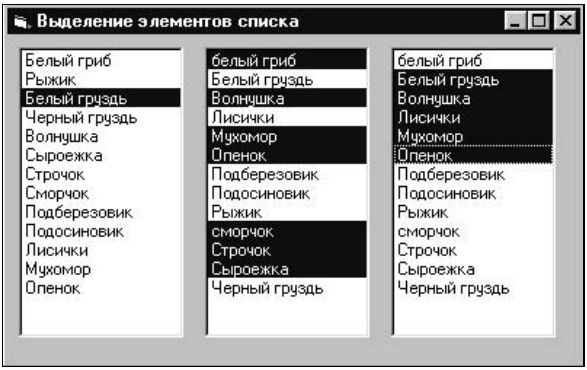


Рис. 4.2. Слева `MultiSelect=0`, в центре `MultiSelect=1`, справа `MultiSelect=2`

Процедура добавления новой строки в среде Visual Basic требует заключения содержимого строки `str` в двойные кавычки, если добавляемое значение не является числом или переменной типа `String`. Аргумент `index` не задается в двух случаях:

- при добавлении строк к упорядоченному массиву;
- при добавлении новой строки в конец списка.

Если аргумент `index` задан, то добавляемая строка вставляется на указанное место с одновременным спуском вниз на одну позицию всех остальных строк.

Приложение 4.3 (VB)

Если высота окна недостаточна для отображения всех строк списка или ширина окна меньше, чем размер самой длинной строки, то список автоматически снабжается одной или двумя линейками прокрутки, обеспечивающими доступ к невидимой информации. Списки с достаточно короткими элементами могут отображаться в две или больше колонок в зависимости от выбранного вами значения свойства `Columns` (рис. 4.3).

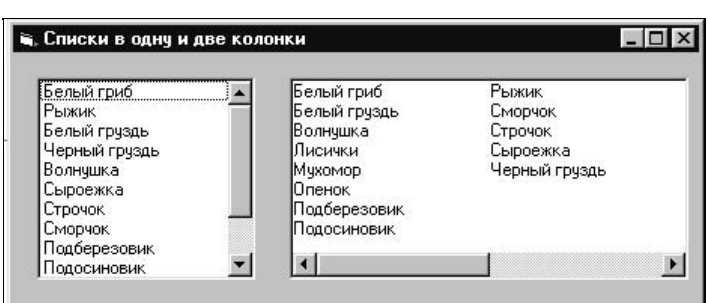


Рис. 4.3. Расположение списков в зависимости от свойства `Columns`

В свойстве `NewIndex` хранится индекс последней добавленной строки. Специфической особенностью стандартного списка VB является возможность связать с любой его строкой целое число типа `Long`:

```
List1.Add "Трюфель"
List1.ItemData(List1.NewIndex)=1500
```

Как это число может быть использовано в конкретной задаче, зависит от программиста. Например, в "грибной" задаче оно может символизировать стоимость 1 кг грибов данного вида.

В отличие от объекта `Memo`, описанного в *разд. 4.1 данной главы*, пользователь по своей инициативе не может изменить содержимое классического списка. Любые изменения во время выполнения приложения могут быть сделаны только программным путем.

4.3. Классический список *ListBox* в системах Borland C++ Builder и Delphi

По внешнему виду объект `ListBox` в системах фирмы Borland мало отличается от аналогичного компонента, описанного в *разд. 4.2 данной главы*. Однако внешнее впечатление обманчиво.

Первое принципиальное отличие заключается в том, что здесь с каждой строкой можно ассоциировать любой объект. Чаще всего в качестве такого объекта выступает небольшой растровый рисунок (пиктограмма), который может украсить отображаемый список. Однако возможна и другая интерпретация пары строка-объект. Строка может рассматриваться как информация об объекте, с которой пользователю будет удобнее работать.

Объект `ListBox` снабжен массивом указателей `Items` на значения соответствующих строк списка:

- `ListBox1.Items[0]` — указатель на первую строку в списке Delphi;
- `ListBox1->Items[0]` — указатель на первую строку в списке BCB.

К значениям этих строк можно обратиться, используя более длинный идентификатор:

- `ListBox1.Items.Strings[0]` — первая строка в списке Delphi;
- `ListBox1->Items->Strings[0]` — первая строка в списке BCB.

В зависимости от используемого метода и типа требующихся аргументов приходится выбирать тот или иной способ адресации операндов. Наличие дополнительного массива указателей позволяет организовать более эффективную работу с такими неудобными данными, как строки переменной длины. Например, при сортировке строк списка достаточно провести перестроение четырехбайтовых указателей вместо того, чтобы перемещать сами строки,

каждый раз освобождая и запрашивая память. В специальных процедурах копирования одного списка в другой нет нужды в дублировании значений строк — достаточно переустановить значения указателей в списке-получателе.

Все методы табл. 4.1, где присутствовал объект `Memo1.Lines`, с точностью до замены имени объекта на `ListBox1.Items`, остаются справедливыми. Кроме того, непосредственно с именем списка может использоваться метод `Clear` — очистка списка.

Приложение 4.4 (BCB)

Интересной особенностью списков в системах BCB и Delphi является возможность задания строк в формате `имя=значение`. Для этого при формировании содержимого списка в редакторе строк вы начинаете набор строки с имени, вплотную к которому размещается знак =, и без пробела продолжаете набор строки, относящейся к значению. К текстовым файлам подобного типа относятся ini-файлы в среде Windows. Для организации поиска строки, содержащей заданное имя, можно выполнить перебор всех строк в цикле с использованием функции `IndexOfName`:

```
for(int j=0; j<ListBox1->Count; j++)
 if(ListBox1->Items[j]->IndexOfName("имя2")>-1)
 {
    // Обработка найденной строки
 }
```

Функция `IndexOfName` возвращает индекс строки, если ее имя совпадает с аргументом функции. Еще один пример, демонстрирующий использование свойств `Names` и `Values`, приведен на рис. 4.4.

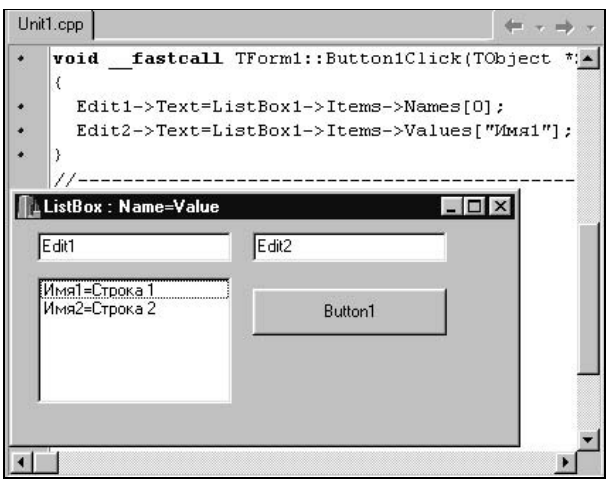


Рис. 4.4. Извлечение имени и значения из строки списка

Приложения 4.5, 4.6 (Delphi)

В системах BCB и Delphi классические списки также обладают логическим свойством `MultiSelect`, которое принимает значение `False` для выделения единственной строки (этот режим включен по умолчанию) и `True` — для множественного выделения любого количества групп строк (рис. 4.5). Дополнительное логическое свойство `ExtendedSelect` в сочетании с `MultiSelect=True` разрешает выделение единственной группы строк (`MutiSelect=ExtendedSelect=True`).

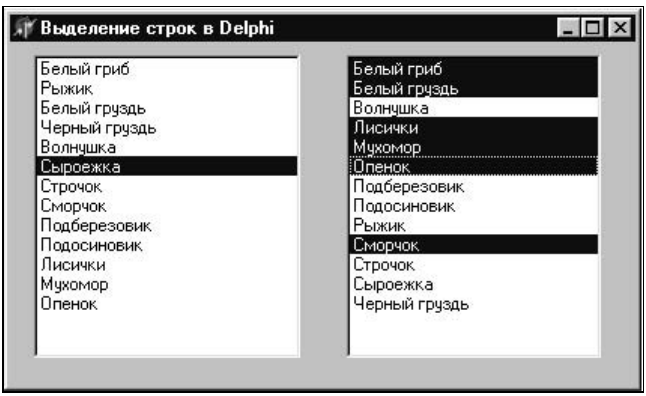


Рис. 4.5. Слева `MultiSelect=False`, справа `MultiSelect=True`

Можно упомянуть еще об одном не очень часто используемом свойстве логического типа `IntegralHeight` (Интегральная высота). Если установить для него значение `True`, то система корректирует размер окна на несколько пикселов таким образом, чтобы нижняя строка была видна целиком.

На рис. 4.6 показана возможность группировки списка в две колонки.

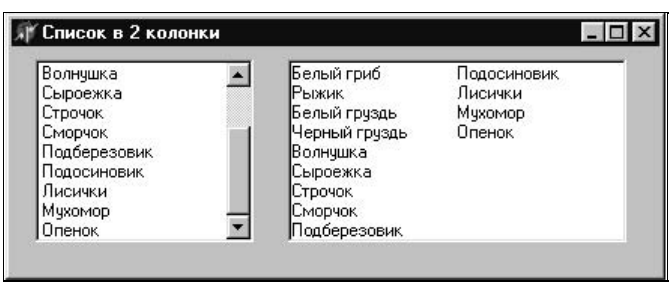


Рис. 4.6. Расположение списков в зависимости от размера окна

4.4. Компактный список *ComboBox* в системе Visual Basic

Компактный список `ComboBox` представляет собой комбинацию классического списка `ListBox` и редактируемого поля ввода `Edit`. Обычно список хранится в свернутом виде и раскрывается только в момент доступа к элементам списка. Окно ввода может предоставить пользователю возможность ввести и отредактировать строку, с которой программа распорядится предусмотренным способом. Преимущества здесь очевидны. Во-первых, один объект совмещает в себе свойства двух объектов (хотя на какой-то выигрыш по ресурсам рассчитывать не надо). Во-вторых, свернутый список занимает на форме меньше места. В-третьих, к объектам подобного рода нас приучили различные программные продукты, работающие под управлением Windows.

Приложение 4.7 (VB)

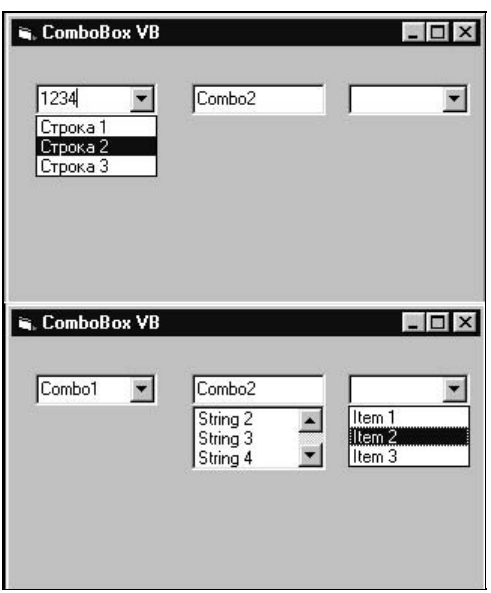


Рис. 4.7. Различные форматы комбинированного списка в VB

На рис. 4.7 представлены три разновидности объекта `ComboBox`, внешний вид которых определяется значением свойства `Style`. По умолчанию, значением этого свойства является `0` (`DropDown Combo`), что соответствует разворачивающемуся списку `Combo1` с доступным для ввода и редактирования окном. Список раскрывается по нажатию на кнопочку со стрелкой в поле

ввода (на рисунке — это самое левое окно). Значение `Style=1` (`Simple Combo`) определяет список с доступным полем ввода. В отличие от предыдущего формата в окне отсутствует кнопка со стрелкой. Поэтому при старте приложения список не виден, но если у объекта `Combo2` увеличить значение свойства `Height` (на правом рисунке `Height=1000`), список появится и при необходимости он будет снабжен линейкой прокрутки. Наконец, последний вариант стиля (`Style=2, Drop_DownList`) предоставляет разворачивающийся список с недоступным окном ввода (на рисунке — самый правый объект).

По сравнению с классическим списком `ListBox` комбинированный список обладает дополнительным свойством `Text`, представляющим содержимое редактируемого окна.

4.5. Компактный список *ComboBox* в системах Borland C++ Builder и Delphi

Приложение 4.8 (BCB)

Комбинированный список `ComboBox` в визуальных средах фирмы Borland обладает всеми свойствами и методами классических списков типа `ListBox` и имеет такие же три формата представления на экране (рис. 4.8). Видом окна управляет свойство `Style`, которое наряду со значениями `csDropDown`, `csSimple` и `csDropDownList` может принимать еще два значения — `csOwnerDrawFixed` и `csOwnerDrawVariable`. Два последних формата используются для отображения списков с нестандартным заполнением (более подробно их применение показано в примере просмотра шрифтов). Содержимое поля редактирования доступно программе через свойство `Text`.

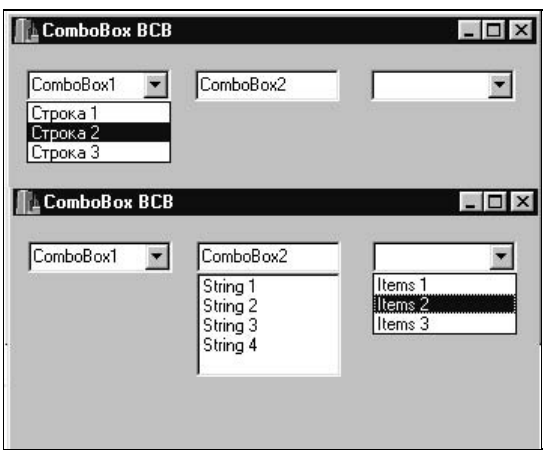


Рис. 4.8. Форматы объекта `ComboBox` в BCB

4.6. Списки *DriveListBox*, *DirListBox* и *FileListBox* в среде Visual Basic

Специализированные списки `DriveListBox`, `DirListBox` и `FileListBox` предназначены для отображения и переключения текущих наименований логического диска, каталога и файла, соответственно.

- Объект `DriveListBox` оформлен в виде стандартного комбинированного списка, строками которого являются пиктограммы и имена всех доступных логических дисков, а в поле ввода отображается имя выбранного диска. Уже во время появления этого объекта на форме в его поле ввода появляется имя текущего диска. Скорее всего, им будет диск c:\..., т. к. используемая система программирования обычно установлена в каталоге c:\Program Files (рис. 4.9).

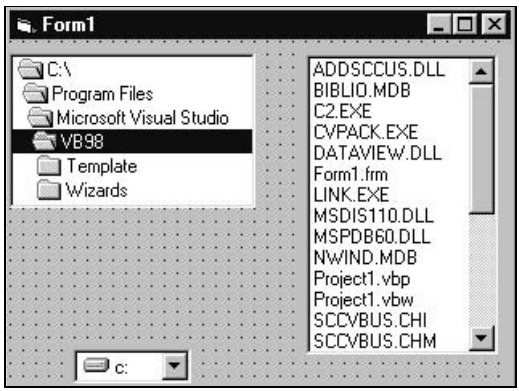


Рис. 4.9. Форма VB на стадии проектирования

- Объект `DirectoryListBox` представлен в виде классического списка типа `ListBox`, отображаемого программным путем с фиксированной высотой строки. Слева от имени каталога воспроизводится пиктограмма раскрытого или закрытого оглавления с учетом смещения, определяемого уровнем вложенности директорий. Текущий каталог представлен выделенной строкой, и полный путь к нему хранится в свойстве `Path`. Если увеличить размер окна до нескольких строк, то уже на стадии проектирования можно увидеть имя текущего каталога (VB98 на рис 4.9).
- Объект `FileListBox` оформлен в виде обычного списка с линейкой прокрутки. Имя выделенной строки запоминается в свойстве `FileName`.

Вообще говоря, каждый из этих объектов живет самостоятельной жизнью, отображая тот список, который указан в соответствующем свойстве. Любое изменение, возникающее в момент выбора нового дисковода во время работы

приложения, приводит к изменению значения свойства `Drive` и возникновению события `DriveChange`.

Приложение 4.9 (VB)

Естественно связать смену имени текущего дисковода с установкой корневого каталога выделенного диска в списке `DirectoryLisBox`. В свою очередь, изменение текущего каталога вызовет событие `DirectoryChange`, обработчик которого может изменить свойство `Path` в списке `FileListBox`. Так можно осуществить взаимоувязанные перемещения по дискам, каталогам и файлам. На рис. 4.10 представлены простейшие тексты обработчиков событий, позволяющие синхронизовать изменения текущего диска, выделенного каталога и выбираемого файла в системе VB.

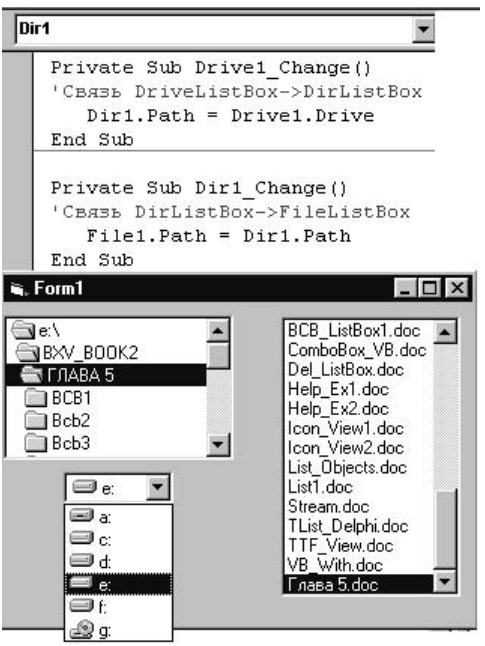


Рис. 4.10. Взаимная увязка окон выбора диска, каталога и файла

Для составления более детальной программы обработки выделенных компонент придется учитывать такие дополнительные свойства, как количество элементов в соответствующем списке (свойство `ListCount`), индекс выделенной строки (`ListIndex`), обозначение элементов в массиве строк (`List(j)`).

4.7. Списки *DriveComboBox*, *DirectoryListBox* и *FileListBox* в системах Borland C++ Builder и Delphi

Специализированные списки, используемые в визуальных системах фирмы Borland для установки текущих значений диска, каталога и файла, очень немногим отличаются от аналогичных объектов VB. Если на форме расположены три списка `DriveComboList1`, `DirectoryListBox1` и `FileListBox1`, то для их объединения в синхронно действующий механизм достаточно подключить следующий обработчик события `OnChange` для объекта `DriveComboList1`:

```
void __fastcall TForm1::DriveComboBox1Change(TObject *Sender)
{
  DirectoryListBox1->Drive=FileListBox1->Drive=DriveComboBox1->Drive;
  FileListBox1->Directory=DirectoryListBox1->Directory;
}
```

Более того, взаимосвязь между указанными объектами можно установить на стадии проектирования приложения, связав их свойства следующим образом:

```
DriveComboBox1->DirList=DirectoryListBox1;
DirectoryListBox1->FileList=FileListBox1;
```

Файлы, отображаемые в списке `FileListBox`, могут быть подвергнуты предварительной фильтрации, чтобы не загромождать окно ненужной информацией. Для этой цели используется свойство `Mask` (например, для просмотра файлов с пиктограммами нужно задать `Mask=*.ico`). Дополнительная фильтрация состоит в отборе файлов, атрибуты которых удовлетворяют шкале свойства `FileType`. Эта шкала представлена логическими битами `ftReadOnly` (только для чтения), `ftHidden` (скрытый файл), `ftSystem` (системный файл), `ftVolumeID` (идентификатор метки тома), `ftDirectory` (каталог), `ftArchive` (файл, подлежащий архивации, — не путать со сжатием), `ftNormal` (файл с любыми атрибутами). Значения этих битов устанавливаются в Инспекторе объектов путем выбора одного из возможных значений `True` или `False`.

Приложение 4.10 (BCB)

На приведенных выше примерах объектов типа `DriveListBox` и `DirectoryListBox` фигурировали списки, в которых элементы с текстовым пояснением были снабжены пиктограммами фиксированного или переменного размера. Продемонстрируем технику прорисовки такого рода списков на примере программы просмотра икон — файлов с расширением ico. Разместим на форме четыре объекта — `ListBox1` (для отображения пиктограмм

и имен ico-файлов), DriveComboBox1 (для выбора текущего дисковода), DirectoryListBox1 (для выбора текущего каталога), FileListBox (для отбора файлов с расширением ico) и Button1 (для инициализации процедуры отображения рисунков и строк).

Так как изображения пиктограмм обычно имеют размеры 32×32, то установим следующие значения свойств в объекте ListBox1:

```
ItemHeight = 40;
Style = lbOwnerDrawFixed;
```

Это означает, что мы собираемся прорисовывать список своей программой, реагирующей на событие OnDrawItem, и что установлена фиксированная ширина строки, равная 40 пикселам. Свяжем окна выбора диска, каталога и файлов описанным выше способом на стадии проектирования и установим Mask=*.ico в объекте FileListBox1. Разместим на поле редактора кода два следующих обработчика событий:

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control,
int Index, TRect &Rect, TOwnerDrawState State)
{ Graphics::TIcon *ico=new Graphics::TIcon();
  ico->LoadFromFile(FileListBox1->Items->Strings[Index]);
  ListBox1->Canvas->Draw(Rect.Left+1,Rect.Top+4,ico);
  ListBox1->Canvas->TextOut(Rect.Left+40,Rect.Top+8,
                            FileListBox1->Items->Strings[Index]);
  delete ico;
}
//-------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int j;
  ListBox1->Items->Clear();
  ListBox1->Items->AddStrings(FileListBox1->Items);
}
```

Работа с этим приложением строится по следующей схеме. Сначала мы путешествуем по дискам и каталогам до тех пор, пока не встретим папку, содержащую файлы с расширением ico. В этот момент в поле объекта FileListBox1 появится список имен таких файлов. Нажимаем кнопку **Копировать** и попадаем на обработчик события Button1Click. Здесь производится очистка списка ListBox1 и его содержимое пополняется списком файлов из FileListBox1.

Указатель ico на объект типа TIcon предназначается для того, чтобы динамически формировать объект нужного формата, в который разрешено загружать файлы пиктограмм. С помощью процедуры new резервируется память для образа иконы и в нее загружается содержимое очередного файла.

Для воспроизведения иконы, загруженной в объект `ico`, используется метод `Draw`, который переносит изображение в прямоугольник `Rect`, представляющий очередную строку списка. Небольшие смещения на 1 пиксел влево и на 4 пиксела вниз имеют целью не дать иконам прилипнуть к левой границе списка и расположиться симметрично относительно середины строки (`4 = (40 - 32)/2`). После переноса изображения в список с помощью метода `TextOut` наименование файла размещается правее иконы. Затем возвращается память, занятая объектом `ico`. Результат работы приложения представлен на рис. 4.11.

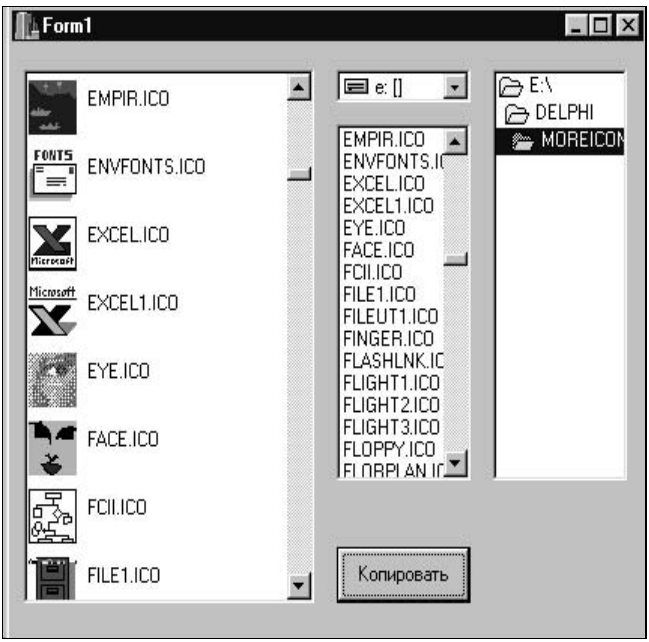


Рис. 4.11. Просмотр икон из папки Moreicon

Приложение 4.11 (Delphi)

Аналогичное Delphi-приложение приведено ниже:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
  ListBox1.Items.AddStrings(FileListBox1.Items);
end;

procedure TForm1.ListBox1DrawItem(Control: TWinControl;
```

```
Index: Integer; Rect: TRect; State: TOwnerDrawState);
var
  ico:TIcon;
begin
  ico:=TIcon.Create;
  ico.LoadFromFile(FileListBox1.Items.Strings[Index]);
  ListBox1.Canvas.Draw(Rect.Left+1,Rect.Top+4,ico);
  ListBox1.Canvas.TextOut(Rect.Left+40,Rect.Top+8,
  FileListBox1.Items.Strings[Index]);
  ico.Free;
end;
```

Объекты, описанные в *разд. 4.6 и 4.7 данной главы*, активно использовались в средах Windows 3.*x*. Однако в современных версиях операционных систем появились более удобные окна, предназначенные для выбора или сохранения файла. В системе Visual Basic такое окно представлено объектом `CommonDialog` (Общий диалог), в системах BCB/Delphi — `OpenDialog` и `SaveDialog`. По сути дела, каждый из них сформирован как комбинация трех описанных выше объектов, увязанных между собой и размещенных на общем диалоговом окне.

4.8. Присоединение объектов к строкам списка *ListBox*

Строки списков типа `Tsrings`, к которым относятся и классические списки `ListBox`, предусматривают возможность присоединения к текстовому содержимому строки (`…Items.Strings[j]`, `…->Items->Strings[j]`) какого-либо объекта, запоминаемого в массиве `Objects`.

Для этой цели кроме методов, упомянутых в табл. 4.1, применяются следующие процедуры, одним из аргументов которых является указатель на некоторый объект:

- Список→AddObject (строка, объект);
- Список→IndexOfObject (объект);
- Список→InsertObject (индекс, строка, объект).

Функция `AddObject` добавляет строку и ассоциированный с ней объект в список. Возвращаемое значение совпадает с индексом добавленного элемента. Функция `IndexOfObject` осуществляет поиск среди элементов списка такой строки, с которой ассоциирован объект, совпадающий с аргументом функции. Если таковой не находится, то функция возвращает значение `-1`. Процедура `InsertObject` вставляет строку и ассоциированный с ней объект в позицию списка, определяемую индексом.

Объекты, ассоциированные со строками списка, системой не отображаются. Такие действия должны выполнять обработчики событий `OnDrawItem` и `OnMeasureItem` подобно тому, как это демонстрировалось в программах просмотра шрифтов и икон.

4.9. Сортировка грибов

Приложение 4.12 (Delphi)

Разместим на форме два универсальных списка и заполним первый из них названиями грибов. Для этого достаточно дважды щелкнуть по строке **Items** (Элементы) в Инспекторе объектов (рис. 4.12). После набора строк в поле редактора и щелчка по кнопке **OK** набранный текст окажется в окне объекта `ListBox1`.

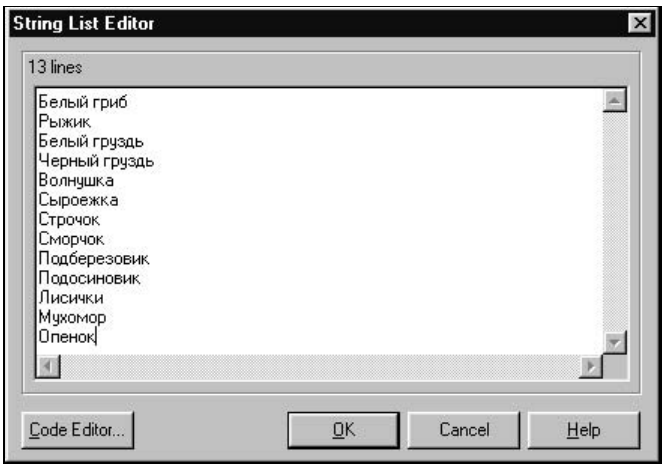


Рис. 4.12. Набор строк списка в редакторе Delphi-6.0

Второй список `ListBox2` предназначается для копирования в него строк из первого списка по нажатию кнопки **Копировать** и их упорядочения в алфавитном порядке по нажатию кнопки **Сортировать**.

Еще две кнопки с заголовками **Левый на диск** и **Правый на диск** предназначены для сохранения содержимого каждого списка в дисковых файлах с именами "Грибы1.txt" и "Грибы2.txt". Ниже приведены тексты Delphi-процедур обработки щелчков по каждой из перечисленных кнопок. На рис. 4.13 отображен экран приложения после его загрузки в память, на рис. 4.14 — после выполнения процедуры переписи, на рис. 4.15 — после выполнения процедуры сортировки. Завершив работу приложения, вы обнаружите в текущем каталоге тестовые файлы с содержимым каждого окна.

```
procedure TForm1.Button1Click(Sender: TObject);
begin {Щелчок по кнопке "Переписать"}
```

```
 ListBox2.Items:=ListBox1.Items;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin {Щелчок по кнопке "Сортировать"}
  ListBox2.Sorted:=True;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin {Щелчок по кнопке "Левый на диск"}
  ListBox1.Items.SaveToFile('Грибы1.txt');
end;

procedure TForm1.Button4Click(Sender: TObject);
begin {Щелчок по кнопке "Правый на диск"}
  ListBox2.Items.SaveToFile('Грибы2.txt')
end;
```

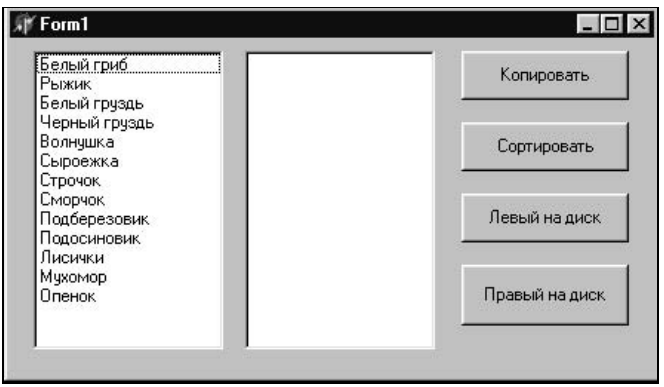


Рис. 4.13. Главная форма в момент старта приложения

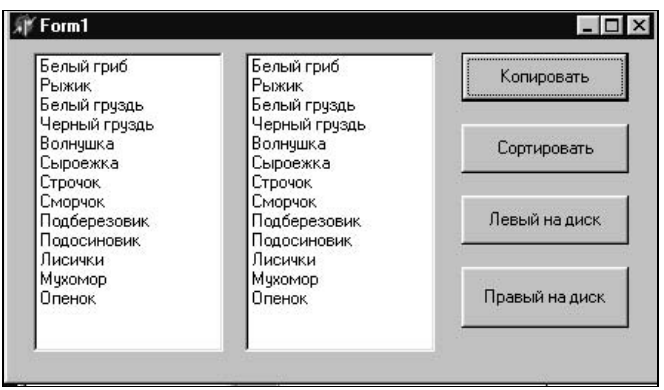


Рис. 4.14. Главная форма после копирования списка

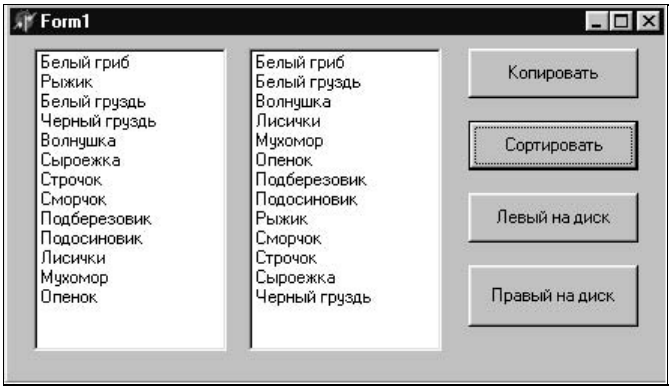


Рис. 4.15. Главная форма после сортировки списка

Отметим деталь, которая может оказаться полезной. Если после упорядочения списка отменить режим сортировки (`LisBox1.Sorted:=False;`), то первоначальное содержимое списка не восстановится. Список останется упорядоченным, но вся последующая работа с ним будет продолжаться как над несортированным списком.

Приложение 4.13 (VB)

Приложение "Сортировка грибов" в среде Visual Basic имеет некоторые отличия от только что приведенного варианта. Набор содержимого списка производится в окне **Properties** (Свойства) после щелчка по значению поля `Items`. Завершая набор очередной строки, надо нажимать комбинацию клавиш <Ctrl>+<Enter>. Нажатие клавиши <Enter> завершает процедуру набора списка.

Во-вторых, VB не позволяет скопировать список целиком, используя единственный оператор присваивания. Поэтому приходится копировать по одной строке, извлекая их из исходного списка и добавляя в `List2` с помощью метода `AddItem`. Для организации цикла по всем строкам первого списка используется свойство `ListIndex`, хранящее количество строк в списке.

В-третьих, свойство `Sorted` доступно во время выполнения программы только на чтение, и изменить его нельзя. Поэтому перенос строк пришлось совместить с сортировкой, и в методе `AddItem` не указывать индекс строки-получателя в списке `List2`.

Наконец, в среде Visual Basic отсутствует метод, позволяющий за одно обращение сохранить список в файле. Поэтому файл приходится открывать и производить в него запись строк в цикле. Дополнительное неудобство вызывает необходимость задания пути к файлу, иначе этот файл окажется в каталоге VB98.

```
Private Sub Command1_Click()
 For i = 0 To List1.ListCount - 1
  List2.AddItem List1.List(i)
 Next i
End Sub

Private Sub Command2_Click()
 Open "e:\\Грибы1.txt" For Output As #1
 For i = 0 To List1.ListCount - 1
   Print #1, List1.List(i)
 Next i
 Close #1
End Sub

Private Sub Command3_Click()
 Open "e:\\Грибы2.txt" For Output As #1
 For i = 0 To List2.ListCount - 1
   Print #1, List2.List(i)
 Next i
 Close #1
End Sub
```

4.10. Просмотр системных шрифтов

Набор системных шрифтов состоит из совокупности ttf-файлов, зарегистрированных в операционной системе. На разных компьютерах он, естественно, разный, т. к. к шрифтам, подключаемым при установке операционной системы, добавляются шрифты других пакетов программ. Например, установка CorelDraw обычно сопровождается регистрацией большого количества шрифтов, используемых дизайнерами.

Существует большое количество разных приложений, демонстрирующих полный набор символов выбранного шрифта или какой-либо его фрагмент. На примерах некоторых из них, частично заимствованных из других источников или слегка переработанных, частично — оригинальных, мы продемонстрируем особенности работы с некоторыми компонентами визуальных сред и средства управления шрифтами.

Приложение 4.14 (BCB). Прорисовка списков, выполняемая программой пользователя

Идея этого примера заимствована из приложения Examples\Apps\OwnerLst, поставляемого в составе Borland C++ Builder. Для списков типа `ListBox`, используемых в стандартном режиме (`Style=Standard`), перерисовка списка

происходит автоматически при внесении в список каких-либо изменений (добавка, удаление, перестановка или прокрутка строк). При этом для текстов всех строк действует общая установка параметров шрифта, сделанная либо во время проектирования, либо модифицированная во время работы приложения. Параметры шрифта, раскрывающиеся при щелчке по кнопочке со знаком плюс слева от свойства `Font`, включают следующие характеристики:

- `Charset` — набор символов, характерных для используемого стандарта, производителя компьютерной техники или страны (`ANSI_CHARSET`, `OEM_CHARSET`, `RUSSIAN_CHARSET`, `ARABIC_CHARSET` и др.). По умолчанию действует набор `DEFAULT_CHARSET`, совпадающий с кодовой страницей операционной системы.
- `Color` — числовой код, обычно задаваемый в шестнадцатеричной системе и определяющий интенсивности RGB-компонент цвета контуров символов.
- `Height` — высота шрифта в пикселах, представленная положительным или отрицательным числом.
- `Name` — строка с наименованием шрифта, зарегистрированным в операционной системе. Имя шрифта может включать пробелы.
- `Pitch` — свойство, управляющее шириной букв и принимающее одно из трех значений `fpDefault`, `fpVariable` или `fpFixed`. В файле каждого ttf-шрифта присутствует таблица, задающая ширину любого символа. Большинство ttf-шрифтов относятся к пропорциональным шрифтам, для которых каждый символ имеет индивидуальную ширину (например — узкая буква I, широкая W). При `pitch=fpDefault` действуют указания о ширине букв, заложенные в ttf-файл. Значения `fpVariable` заставляют системную программу отображения текстов использовать переменную ширину букв даже в том случае, когда используемый шрифт относится к моноширинным. Значение `fpFixed`, наоборот, располагает все символы в прямоугольниках равной ширины. Без особой нужды не стоит противопоставлять значение этого свойства параметрам, зашитым в ttf-файле.
- `Size` — высота шрифта в пунктах (1 пункт = 1/72 дюйма = 0.3528 мм).

Свойства `Height` и `Size` связаны между собой отношением:

```
Height = – Size * PixelPerInch / 72;
```

Большинству пользователей по аналогии с работой в редакторах типа MS Word привычнее задавать высоту надписи в пунктах. Знак высоты влияет на размер букв таким образом, что в одном случае междустрочный интервал добавляется к высоте буквы, а в другом случае — не добавляется. На форме, приведенной на рис. 4.16, расположены 2 списка типа `ListBox`. В каждом из них набрано по одной строке из нескольких букв "I".

Для того чтобы сформировать в списке перечень всех шрифтов, зарегистрированных в системе, надо воспользоваться свойством `Fonts` системного объекта `Screen`:

```
ListBox1->Items = Screen->Fonts;
```

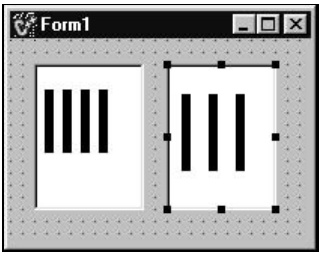


Рис. 4.16. Слева `Size=-54`, справа `Size=54`

Если такое присвоение выполнить во время работы приложения, то в списке `ListBox1` немедленно появятся имена всех системных шрифтов в алфавитном порядке, поскольку системный список упорядочен. Однако индивидуальность каждого шрифта при этом пропадет, т. к. все имена отображаются одним и тем же шрифтом, приписанным к свойству `Font->Name` объекта `ListBox1` (рис. 4.17).

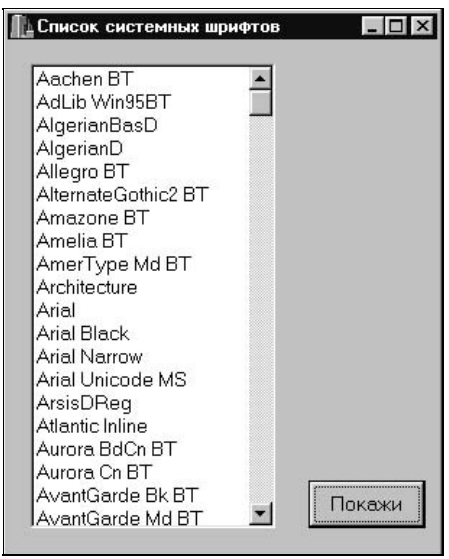


Рис. 4.17. Список шрифтов в режиме `lbStandard`

Приложение 4.15 (BCB)

Для того чтобы каждая строка списка `ListBox1` отображалась своим шрифтом, придется написать процедуру воспроизведения элемента списка — `DrawItem` и предупредить об этом систему. Последнее заключается в изменении свойства `Style`, которому можно назначить один из двух дополни-

тельных режимов — `lbOwnerDrawFixed` или `lbOwnerDrawVariable`. Первый из этих режимов означает, что высота каждой строки будет фиксированной, а второй — разрешает каждой строке иметь индивидуальную высоту.

Начнем с простейшего — `Style=lbOwnerDrawFixed`. Высоту строк в списке регулирует свойство `ItemHeight`, и мы установим его значение, равным 20 пикселам. Для перерисовки очередной наблюдаемой на экране строки списка система обращается к процедуре обработки события `OnDrawItem`:

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control,
int Index,Rect &Rect,TOwnerDrawState State)
{
  AnsiString s=ListBox1->Items->Strings[Index];
  ListBox1->Canvas->FillRect(Rect);
  ListBox1->Canvas->Font->Name = s;
  ListBox1->Canvas->TextOut(Rect.Left,Rect.Top,s.c_str());
}
```

Процедура `ListBox1DrawItem` получает указатель на объект-инициатор события `OnDrawItem` (`Control`), ссылку на прямоугольник (`Rect`), в котором должно быть построено содержимое очередной строки с номером `Index` и информацию о состоянии строки (`State`). Последняя может быть выделена, находиться в фокусе, иметь серый вид (`Grayed`) и т. п.

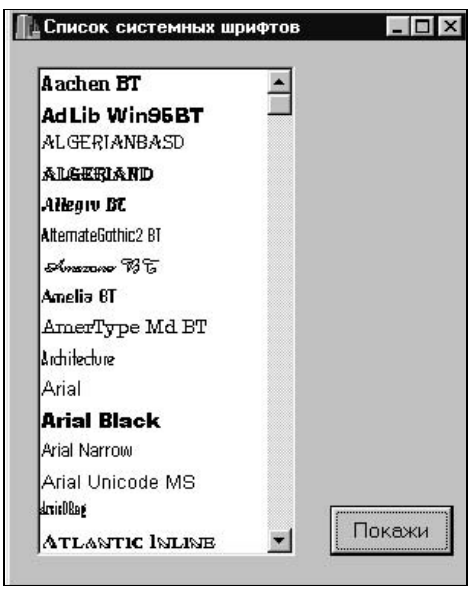


Рис. 4.18. Список шрифтов в режиме `lbOwnerDrawFixed`

Первый оператор обработчика запоминает значение строки, совпадающее с именем шрифта, в переменной типа `AnsiString`. Это сделано для сокращения текста программы, т. к. текст воспроизводимой строки в дальнейшем потребуется дважды. Второй оператор обращается к методу `FillRect` для очистки прямоугольника, расположенного на канве нашего списка. Подробнее о свойстве `Canvas` речь пойдет в *главе 8*. В третьем операторе имя шрифта присваивается свойству `Font` канвы списка (его не надо путать со свойством `Font` самого списка). Последний оператор вызывает метод `TextOut` для отображения текста `s`, начиная с точки (`x=Rect.Left`, `y=Rect.Top`). Этот метод очень напоминает функцию `outtextxy`, знакомую нам по библиотеке графических процедур BGI в системах программирования Turbo Pascal и Borland C. Обратите внимание на третий аргумент метода. Так как переменная `s` имеет тип `AnsiString`, то ее надо преобразовать к типу (`char *`), что выполняется с помощью метода `c_str()`.

Результат работы приложения, содержащего два обработчика событий, — `Button1Click` для загрузки списка системных шрифтов в `ListBox1` и только что описанную процедуру отрисовки элемента списка показан на рис. 4.18.

Приложение 4.16 (BCB)

А теперь немного усложним задачу тем, что заставим список воспроизводить каждую строку тем размером букв, который заложен в ttf-файл. Очевидно, что каждый разработчик шрифта по-своему определяет высоту символов по умолчанию. Поэтому нам придется иметь дело с разновысокими строками. Установим значение свойства `Style`, равным `lbOwnerDrawVariable` и напишем следующий обработчик события `OnMeasureItem`:

```
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control,
int Index, int &Height)
{
  ListBox1->Canvas->Font->Name = ListBox1->Items->Strings[Index];
  ListBox1->Canvas->Font->Size = 0;
  Height = ListBox1->Canvas->TextHeight("H")+2;
}
```

- Первый оператор копирует имя шрифта из строки списка в свойство `Font` канвы.
- Второй оператор делает, казалось бы, нелепую вещь, устанавливая высоту шрифта канвы равной `0`. Однако это — небольшая хитрость, почти нигде не описанная. В случае такого задания высоты фактический размер шрифта извлекается из ttf-файла.
- Последний оператор с помощью предшествовавшего таинства определяет высоту текста, обращаясь к методу `TextHeight` со строкой, содержащей

единственную букву. А добавка (+2) позволяет разумно сформировать интервалы между содержимым смежных строк.

Немного изменяется и текст обработчика события `OnDraweItem`. С учетом приведенного выше секрета строки с нулевой высотой остальные действия обработчика в дополнительных комментариях не нуждаются:

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control,
int Index, TRect &Rect, TOwnerDrawState State)
{ AnsiString s=ListBox1->Items->Strings[Index];
  ListBox1->Canvas->FillRect(Rect);
  ListBox1->Canvas->Font->Name = s;
  ListBox1->Canvas->Font->Size=0;
  ListBox1->Canvas->TextOut(Rect.Left+1,Rect.Top+1,s.c_str());
}
```

Результат работы этого приложения приведен на рис 4.19.

Рис. 4.19. Список шрифтов в режиме `lbOwnerDrawVariable`

Приложение 4.17 (BCB)

Небольшая модификация приложения, заключающаяся в добавке двух объектов `Label1` с заголовком "Выделенный шрифт" и `Label2` для содержимого выделенной строки списка, позволит получить более полную информацию о наименованиях просматриваемых шрифтов. Дело в том, что некоторые

из них не содержат букв и вместо названий шрифтов в списке фигурируют не очень понятные цепочки геометрических фигур (рис. 4.20). Полный текст соответствующего приложения приведен ниже.

Рис. 4.20. Список шрифтов с расшифровкой их имен

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  ListBox1->Items=Screen->Fonts;
}
//---------------------------------------------------------------------
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control,
int Index, TRect &Rect, TOwnerDrawState State)
{ AnsiString s=ListBox1->Items->Strings[Index];
  ListBox1->Canvas->FillRect(Rect);
  ListBox1->Canvas->Font->Name = s;
  ListBox1->Canvas->Font->Size=0;
  ListBox1->Canvas->TextOut(Rect.Left+1,Rect.Top+1,s.c_str());
}
//---------------------------------------------------------------------
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control,
int Index, int &Height)
{ ListBox1->Canvas->Font->Name = ListBox1->Items->Strings[Index];
```

```
  ListBox1->Canvas->Font->Size = 0;
  Height = ListBox1->Canvas->TextHeight("H")+2;
}
//---------------------------------------------------------------------
void __fastcall TForm1::ListBox1Click(TObject *Sender)
{ Label2->Caption=ListBox1->Items->Strings[ListBox1->ItemIndex];
}
```

Приложение 4.18 (BCB). Просмотр всех символов выделенного шрифта

Иногда у программиста возникает желание поподробнее рассмотреть все символы того или иного шрифта с заданной высотой букв. Ниже приводится пример одного из возможных вариантов такого приложения.

Для его работы понадобится текстовый файл ascii.txt, содержащий несколько строк с числовыми кодами символов от 32 до 255. Этот файл загружается в объект `Memo1` для того, чтобы каждый числовой код был заменен соответствующим символом выбранного шрифта. Такой файл может быть изготовлен программным путем, например, с помощью приведенной ниже программы ascii.cpp:

```
#include <stdio.h>
  main()
{
 FILE *f1;
 int j;
 f1=fopen("ascii.txt","wb");
 for(j=32;j<256;j++)
  {
    if((j+1)%32 != 0)
      fprintf(f1,"%c",j);
    else
      fprintf(f1,"%c",0x0A);
  }
 fclose(f1);
}
```

В конструкторе приложения (чтобы не заводить лишний обработчик события, например, `OnCreate`) производятся начальные установки — перечень зарегистрированных в системе шрифтов загружается в список `ListBox1`, индекс выделяемой строки устанавливается в начало списка, производится загрузка текстового файла ascii.txt в объект `Memo1` и назначается начальная высота шрифта.

Для удобства просмотра в списке `ListBox1` использован многоколонный режим отображения (`Columns=4`) и этот список снабжен горизонтальной

линейкой прокрутки. Из таких же соображений записная книжка помещена на объект `ScrollBox`, снабженный линейками прокрутки.

Кнопка `UpDown`, ассоциированная с окном `Edit1`, предназначена для задания высоты отображаемых символов в диапазоне от 8 до 72 пунктов с шагом 1.

В момент щелчка по выделяемому шрифту производится замена свойства `Font->Name` в `Memo1` и установка начальной высоты символов, после чего содержимое `Memo1` автоматически перерисовывается.

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
  ListBox1->Items=Screen->Fonts;
  ListBox1->ItemIndex=0;
  Memo1->Lines->LoadFromFile("ascii.txt");
}
//---------------------------------------------------------------
void __fastcall TForm1::ListBox1Click(TObject *Sender)
{
  Memo1->Font->Name=ListBox1->Items->Strings[ListBox1->ItemIndex];
}
//-----------------------------------------------------------------
void __fastcall TForm1::UpDown1Click(TObject *Sender,
TUDBtnType Button)
{
  Memo1->Font->Size=UpDown1->Position;
}
```

Пример работы приложения показан на рис. 4.21.

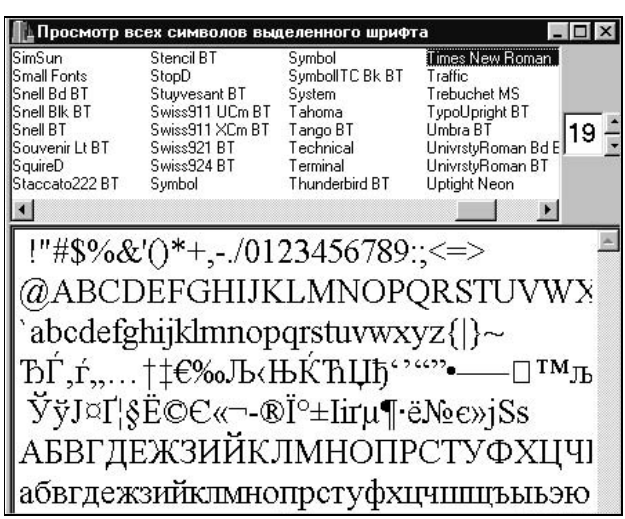


Рис. 4.21. Просмотр символов шрифта Times New Roman

Более увлекательное зрелище открывается при просмотре специальных шрифтов (рис. 4.22)

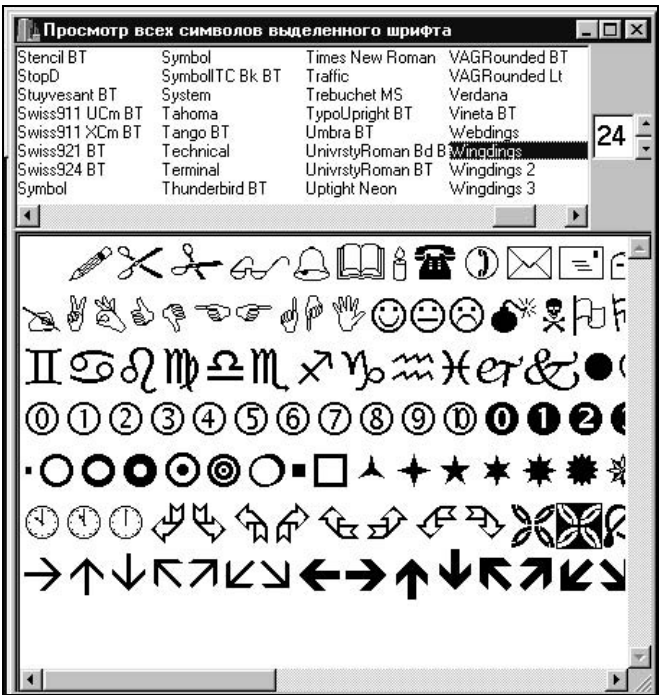
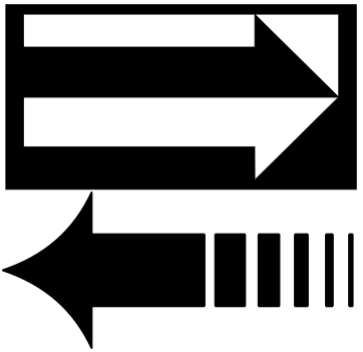


Рис. 4.22. Просмотр символов шрифта Wingdings

Глава 5

Кнопки и меню



Назначение различных кнопок и команд меню примерно одинаково — инициализировать одну из процедур приложения, включить и пометить тот или иной режим (`CheckBox`, галочка слева от названия команды). Зачастую, наиболее применяемые команды меню дублируются так называемыми быстрыми кнопками. Сходство между кнопками и командами меню в среде Visual Basic подчеркивает даже название компонента — `Command Button` (Командная кнопка).

Конечно, у кнопок немного больше различных свойств, чем у команд меню. На кнопках нужно тем или иным способом разместить надпись, нанести рисунок, который при нажатии кнопки может изменять свой вид, связать нажатие некоторых кнопок с эквивалентным действием клавиш <Enter> или <Esc>. Но их основное событие — щелчок мышью по кнопке или команде меню — является сигналом к выполнению обычно единственного предназначения — сделать то, что тебе предписано.

Перечень кнопок и специфика их внешнего вида описаны в табл. 5.1.

Таблица 5.1. *Перечень и специфика кнопок*

Обозначение кнопки		Специфика кнопки
Visual Basic	**BCB, Delphi**	
`CommandButton` (режим `Standard`)	`Button`	Обычная кнопка с надписью
`CommandButton` (режим `Graphical`)	`BitmapButton`	Кнопка с надписью и рисунком
	`SpeedButton`	Кнопка с рисунками, которая может оставаться утопленной
`CheckBox`	`CheckBox`	Квадратная кнопка с внешней надписью, иногда с галочкой

Таблица 5.1 *(окончание)*

Обозначение кнопки		Специфика кнопки
Visual Basic	**BCB, Delphi**	
`OptionButton`	`RadioButton`	Круглая кнопка с внешней надписью, иногда с жирной точкой в центре
	`RadioGroup`	Контейнер с массивом радиокнопок
`UpDown`	`UpDown`	Кнопка со стрелками увеличения и уменьшения числового значения

5.1. Командные и инструментальные кнопки

Командная кнопка `CommandButton` (системные имена `Command1`, `Command2` и т. д.) в среде VB сочетает в себе два типа кнопок, используемых в средах BCB и Delphi.

Приложение 5.1 (VB)

Кроме заголовка (свойство `Caption`) на этой кнопке можно разместить от одной до трех растровых картинок небольшого размера в формате bmp, появляющихся на кнопке в разных ее состояниях:

- `Picture` — картинка, наклеенная на не нажатую кнопку;
- `DownPicture` — картинка, появляющаяся на нажатой кнопке;
- `DisabledPicture` — картинка, соответствующая недоступному состоянию кнопки.

Для того чтобы поместить соответствующую картинку на кнопку, необходимо изменить ее стиль со стандартного (`Style=Standard`) на графический (`Style=Graphical`). Затем в окне **Properties** (Свойства) необходимо перейти на одно из указанных выше свойств типа **Picture** (Рисунок), щелкнуть по не присвоенному пока значению **None** (Нет) и сделать еще один щелчок на появившейся кнопке с тремя точками. В диалоговом окне (рис. 5.1) выберите необходимую пиктограмму и загрузите ее на кнопку. Картинку `Picture` вы увидите сразу (рис. 5.2), остальные картинки появятся на кнопке в соответствующей ситуации (рис. 5.3).

Цветом фона кнопки и цветом надписи можно управлять с помощью свойств `BackColor` и `ForeColor`. Если вы хотите, чтобы событие `Click` возникало не только в момент щелчка мышью по кнопке, но и в момент нажатия клавиши <Enter>, свойству `Default` надо присвоить значение `True`. Событие `Click` может возникнуть и момент нажатия клавиши <Esc> в том случае, если свойству `Cancel` присвоено значение `True`. Конечно, оба эти свойства не могут использоваться одновременно, т. к. клавиша <Enter>

обычно применяется для подтверждения некоторой акции, тогда как клавиша <Esc> ассоциируется с отменой этого действия.

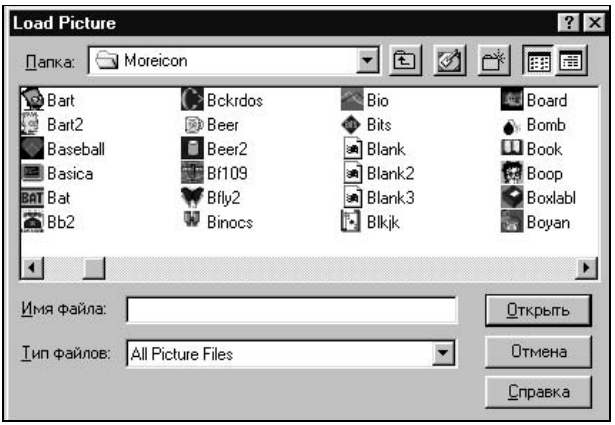


Рис. 5.1. Диалоговое окно VB для загрузки рисунка на кнопку

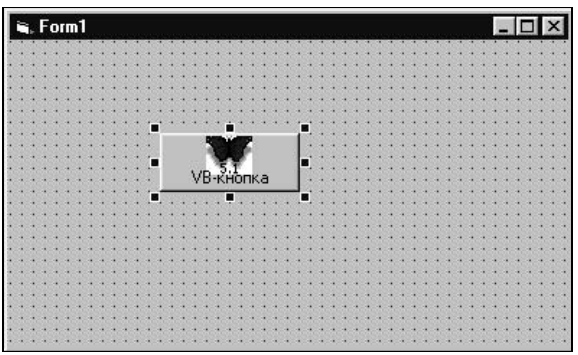


Рис. 5.2. Рисунок, загруженный в свойство `Picture`

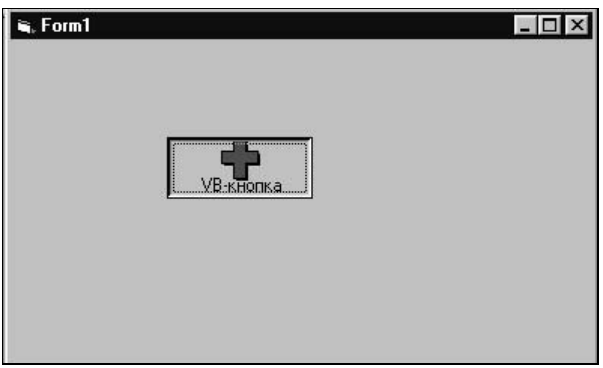


Рис. 5.3. Рисунок, загруженный в свойство `DownPicture`

Обратите внимание (на рис. 5.3 это отчетливо видно), что в момент щелчка по кнопке последняя активизируется и вокруг надписи появляется пунктирная рамка. Точно такое же зрелище наблюдается и в тот момент, когда кнопке передается фокус программным путем:

```
Command1.SetFocus
```

Приложение 5.2 (BCB)

В средах Delphi и BCB роль кнопки `Command Button` выполняют две разные кнопки — типа `Button` (системные имена `Button1`, `Button2`,) и типа `Bitmap Button` (системные имена — `BitBtn1`, `BitBtn2`). Одна из них (`Button`) явно не имеет свойства `Picture`, что разрешает разместить на поверхности кнопки только надпись, которая всегда располагается в центре кнопки. В отличие от нее кнопка `BitBtn` ориентирована на размещение на своей поверхности рисунков специального вида, представляющих свойство `Glyph`. Сюда можно загрузить группу пиктограмм одинакового размера (обычно в диапазоне от 16×16 до 32×32 пиксела), содержащую от одного до четырех изображений. Первое изображение соответствует спокойному состоянию кнопки, второе — появляется на ней, когда кнопка недоступна, третье изображение сопровождает кнопку в момент ее нажима, наконец, последнее изображение возникает на кнопке, когда она находится в утопленном положении. Готовятся такие изображения с помощью специального редактора изображений, но проще воспользоваться готовым набором, который содержится в каталоге C:\Program Files\Common Files\Borland Shared\Images\Buttons

Почти все наклейки кнопок из этого каталога представлены двумя изображениями — ярким (нормальным) и тусклым негативом, соответствующим недоступной кнопке.

Формирование значения свойства `Glyph` производится с помощью редактора рисунков — **Picture Editor** (Редактор рисунков) (рис. 5.4), который появляется после щелчков по полю значения свойства.

Обратите внимание на то, что после загрузки рисунка свойство `NumGlyph` (Количество изображений) автоматически стало равно 2. Система по размеру рисунка сама определила количество изображений. На рис. 5.5 показаны формы приложения с единственной кнопкой `BitBtn1` до ее активизации (слева) и после щелчка по кнопке, когда ее свойство `Enabled` было сброшено в `False`.

Рассматриваемые кнопки обладают рядом свойств, помогающих нужным образом разместить надпись и рисунок. Свойство `Layout`, принимающее одно из четырех возможных значений (`blGluphLeft`, `blGluphBottom`, `blGluphRight`, `blGluphTop`) позволяет разместить рисунок по отношению к подписи слева, снизу, справа или сверху. Количество пикселов между границей рисунка и соответствующей границей кнопки задается свойством

`Margin`. Если `Margin=-1`, то текст и рисунок центрируются по вертикали и горизонтали. Наконец, свойство `Spacing` регулирует величину зазора между рисунком и подписью.

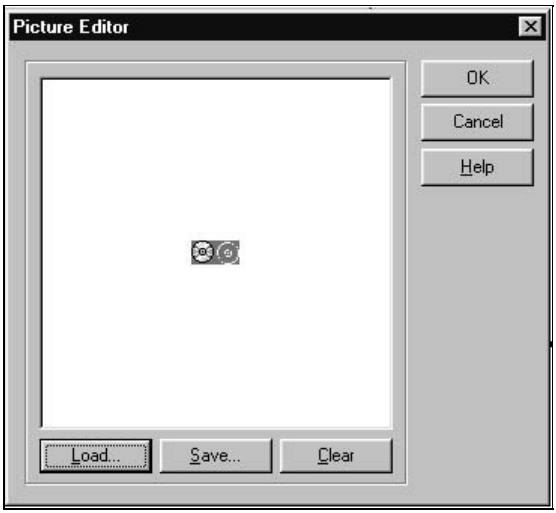


Рис. 5.4. Поле редактора после загрузки файла cddrive.bmp

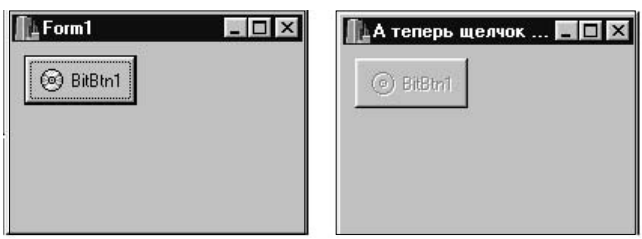


Рис. 5.5. Поверхность кнопки `BitBtn1` в моменты доступности и недоступности

Приложение 5.3 (BCB)

Очень часто в приложениях используются кнопки стандартного назначения. С их помощью пользователь выбирает одну из альтернатив — **Да** или **Нет**, **Повторить** или **Проигнорировать**, **Помочь** и т. п. Так как в большинстве Windows-приложений вид таких кнопок, рисунки и надписи устоялись, то система берет на себя заботы о внешнем виде такого рода кнопок. Для этой цели надо воспользоваться свойством `Kind`, которое может быть задано одной из следующих мнемонических констант — `bkCustom` (пользователь берет все на себя), `bkAll`, `bkCancel`, `bkClose`, `bkHelp`, `bkIgnore`, `bkNo`, `bkOK`, `bkRetry`, `bkYes` (рис. 5.6).

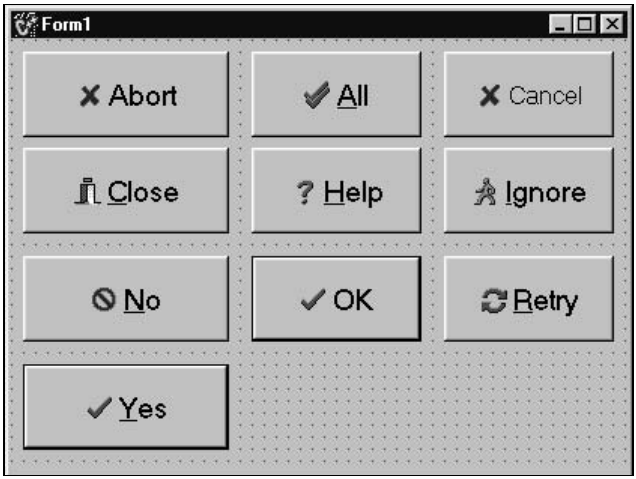


Рис. 5.6. Внешний вид стандартных кнопок типа `BitBtn`

Кнопки типа `Button` и `BitBtn` не фиксируются в утопленном положении. Для этой цели среды фирмы Borland предлагают еще одну кнопку типа `SpeedButton` (Быстрая кнопка). В дополнение к рассмотренным выше свойствам кнопки `BitBtn` быстрые кнопки обладают рядом дополнительных возможностей. Обычно их используют в качестве инструментальных кнопок, расположенных на общей панели и объединенных в группу. Каждой кнопке группы следует приписать групповой индекс (свойство `GroupIndex`), отличный от нуля. После такой процедуры групповые кнопки по первому щелчку утапливаются и остаются утопленными до следующего щелчка или до изменения свойства `Down`. После запуска приложения по умолчанию утопленной оказывается первая кнопка. Если мы хотим начать работу приложения с ситуации, когда все кнопки группы отжаты, то у каждой из них надо установить в истину значение свойства `AllowAllUp=True`. Свойство `Flat=True` позволяет работать с кнопками, которые являются плоскими и возвышаются только тогда, когда по их поверхности перемещается курсор мыши.

С нажатием кнопки **Close** (Закрыть) происходит завершение работы приложения, поэтому сообщение "Нажата кнопка Close" никогда не появится.

Кнопка, у которой значение группового индекса оставлено нулевым, в утопленном положении не фиксируется.

Аналога кнопки `SpeedButton` в среде VB 6.0 нам обнаружить не удалось, хотя в версии VB 3.0 существовала кнопка такого же типа с несколько странным названием — `SSRibbon` (ribbon — красящая лента для матричных принтеров). Следует отметить, что в версии VB 6.0 поиск нужного объекта можно сопоставить разве что с преодолением полосы препятствий.

5.2. Специальные кнопки

Специальные кнопки обеспечивают возможность индикации одного или нескольких не альтернативных вариантов (кнопка `Check Box`), выбора одного из группы альтернативных вариантов (кнопка `Option Box` в VB и аналогичная кнопка `RadioButton` в BCB и Delphi) и набора целочисленного значения путем прибавления или вычитания заданной величины из текущего значения (кнопка `UpDown`).

Приложение 5.4 (BCB)

Вид каждой из специальных кнопок среды BCB показан на рис. 5.7.

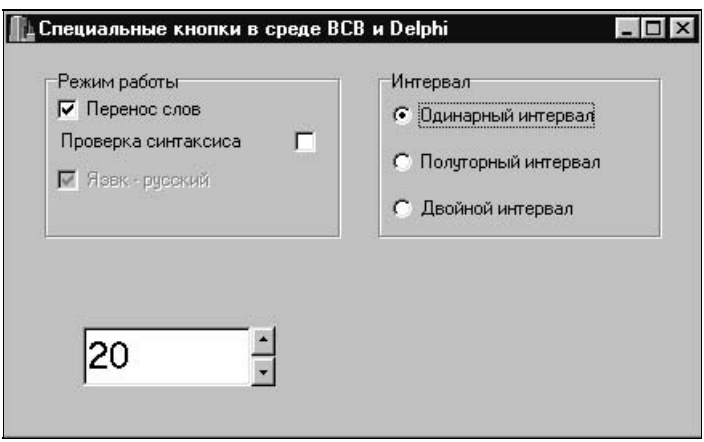


Рис. 5.7. Специальные кнопки в среде BCB

В левой рамке — в контейнере `GroupBox1` — расположены компоненты `CheckBox1`, `CheckBox2` и `CheckBox3`. Ширина каждого объекта подобрана таким образом, чтобы пояснительная надпись вместе с квадратной кнопкой помещалась в отведенном прямоугольнике. Свойство `Alignment` (Выравнивание), которое может принимать одно из двух значений — `taRightJustify` (Прижим справа) и `taLeftJustify` (Прижим слева), определяет положение пояснительной подписи (значения свойства `Caption`) относительно кнопки. Значение свойства `Checked=True` свидетельствует о наличии галочки в поле кнопки, т. е. о том, что кнопка помечена. Такую пометку может сделать пользователь, щелкнув по непомеченной кнопке во время работы приложения, или программа, присвоив значение `True` свойству `Checked`. Повторный щелчок по помеченной кнопке стирает галочку, при этом свойство `Checked` принимает значение `False`. Свойство `State` (Состояние) принимает одно из трех значений — `cbChecked` (Кнопка помечена жирной черной галочкой), `cbUnchecked` (Кнопка не помечена) и `cbGrayed` (Галочка в кнопке сероватого

цвета). В последнем случае кнопка не реагирует на щелчки пользователя, но программе она доступна. Если кнопка недоступна (`Enabled=False`), то изменяется цвет пояснительной надписи.

В правом контейнере `GroupBox2` расположены три радиокнопки `RadioButton1`, `RadioButton2` и `RadioButton3`. Своим названием они обязаны кнопкам переключения диапазонов на старых радиоприемниках. При включении любой радиокнопки одновременно выскакивает ранее нажатая и фиксируется новая кнопка. Двух одновременно нажатых радиокнопок в одной группе быть не должно, т. к. с каждой из них предполагается выполнение несовместимых действий. Для радиокнопок наиболее важными являются свойства `Caption` (Текст пояснительной надписи) и `Alignment` (Способ размещения пояснительной надписи).

В связи с тем, что радиокнопки, как правило, выступают в виде организованной группы, то выделять под них контейнер и размещать на нем каждую кнопку — дело хлопотное. Поэтому в средах BCB и Delphi предусмотрели специальный компонент `RadioGroup`, совмещающий специфику контейнера и размещенного в нем массива (свойство `Items`) радиокнопок. Индивидуального имени каждая кнопка массива не имеет, поэтому доступ к таким кнопкам возможен по индексу его элемента:

- `RadioGroup1->Items[0]` имя первой радиокнопки;
- `RadioGroup1->Items[1]` имя второй радиокнопки;
- `RadioGroup1->Items[2]` имя третьей радиокнопки.

Для того чтобы программным путем добраться до надписи на радиокнопке, приходится использовать еще более длинный идентификатор:

- `RadioGroup1->Items->Strings[0]` надпись на первой кнопке;
- `RadioGroup1->Items->Strings[1]` надпись на второй кнопке;
- `RadioGroup1->Items->Strings[2]` надпись на третьей кнопке.

Приложение 5.5 (BCB)

Кнопки в радиоконтейнере можно расположить в несколько колонок, если задать значение свойства `Columns`, отличным от `1` (рис. 5.8).

Когда пользователь выделил ту или иную радиокнопку в массиве `RadioGroup1`, то ее индекс хранится в свойстве `ItemIndex`. Поэтому в программе обработки соответствующего события мы должны делать проверку следующего типа:

```
if(RadioGroup1->Items->Strings[RadioGroup1->ItemIndex]==”Одинарный
интервал”)
```

В среде Delphi добавку `RadioGroup1` можно спрятать в операторе `with`:

```
with RadioGroup1 do
```

```
if Items.Strings[ItemIndex] = 'Одинарный интервал' then …
.................................
end;
```

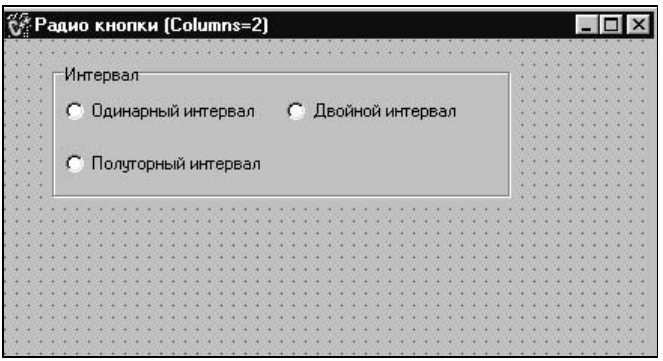


Рис. 5.8. Расположение радиокнопок в 2 колонки

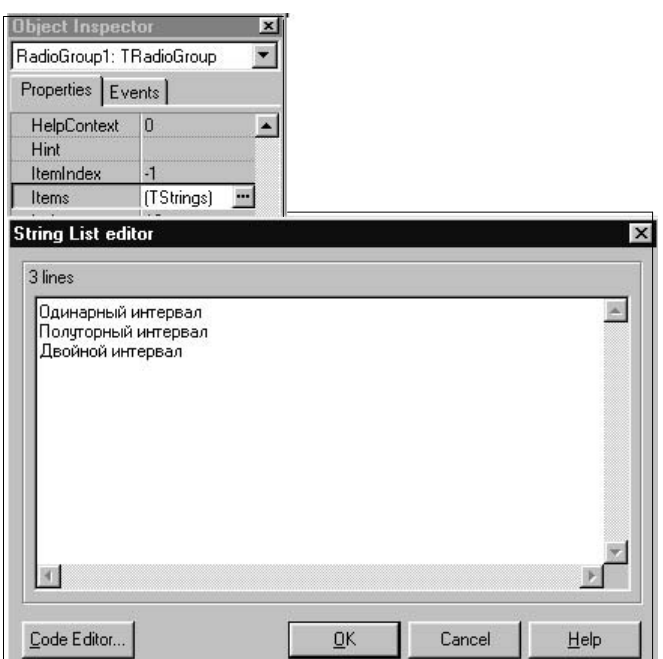


Рис. 5.9. Формирование содержимого контейнера `RadioButton1`

Кстати, такая же конструкция `with`, обеспечивающая экономию в наборе составных имен, появилась в составе входного языка последних версий VB. Теперь только язык C обделен таким удобным средством.

Когда в радиоконтейнере не выделена ни одна кнопка, то свойство `ItemIndex` принимает значение `-1`.

Если в контейнере, отличном от `RadioGroup`, размещены радиокнопки, то пояснительная надпись для каждой из них задается индивидуально путем формирования значения соответствующего свойства `Caption`. В контейнере `RadioGroup` для формирования количества радиокнопок и создания присоединенных к ним пояснительных подписей необходимо щелкнуть по значению свойства `Items` и по появляющейся справа кнопке с тремя точками. В результате будет вызван редактор строк (рис. 5.9), в котором следует набрать только текст подписей (каждая подпись в отдельной строке). После выхода их редактора строк содержимое контейнера `RadioGroup` формируется автоматически.

Приложение 5.6 (VB)

В среде Visual Basic эквивалентными объектами являются элементы `CheckBox` и `OptionButton`. В стандартном режиме (`Style = Standard`) их внешний вид почти такой же (рис. 5.10). Однако они допускают использование в графическом режиме (`Style=Graphical`), когда с поверхностью кнопки может быть связано до трех рисунков — `Picture` (появляется на не выделенной кнопке), `DownPicture` (появляется в момент нажатия кнопки) и `DisabledPicture` (появляется на заблокированной кнопке). Состояние кнопок VB определяется значением свойства `Value`. Для кнопок типа `CheckBox` это свойство может принимать одно из трех значений — `0` (`Unchecked`), `1` (`Checked`) и `2` (`Grayed`). Для кнопок типа `OptionButton` свойство `Value` может принимать только два логические значения — `True` (кнопка выделена) или `False` (кнопка не выделена). Свойства `Caption` и `Alignment` у них совпадают с ранее рассмотренными аналогами. Для того чтобы сгруппировать кнопки того или иного вида, на форму помещают контейнер `Frame` (Рамка) или `PictureBox` (Рамка для рисунка). Кнопки, помещенные в контейнер, автоматически объединяются в группу.

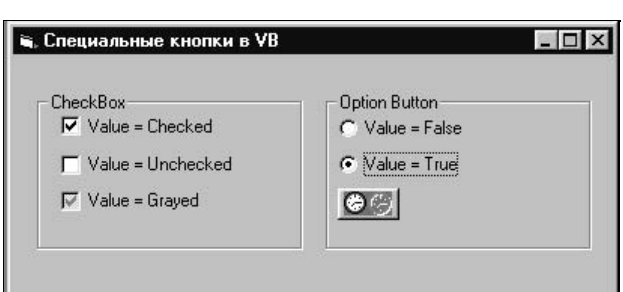


Рис. 5.10. Вид специальных кнопок в среде VB

Кнопки типа `UpDown` представлены в штатном наборе компонент BCB/Delphi явно. В среде Visual Basic сначала придется обновить вкладку **General**, выполнив команду **Components** (Компоненты) из меню **Project** (Проект) и выбрав **Microsoft Windows Common Controls-2 5.0 (SP3)**. На рис. 5.11 видно, что описание объекта `UpDown` содержится в файле comct232.ocx, расположенном в каталоге Windows\System. Поэтому при переносе такого VB-приложения на другой компьютер придется дополнительно перенести указанный файл.

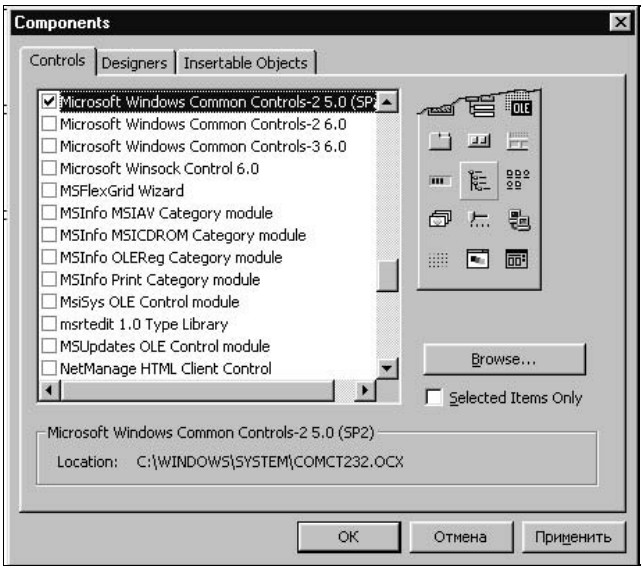


Рис. 5.11. Выбор раздела, содержащего объект `UpDown` в VB

Приложение 5.7 (Delphi)

Кнопки типа `UpDown` (Вверх/Вниз) позволяют изменять текущее числовое значение путем его коррекции на заданное приращение в сторону увеличения или уменьшения. Поверхность кнопки `UpDown` разделена пополам горизонтальной чертой, над которой нарисован треугольник вершиной вверх (`Up`). Аналогичный треугольник под чертой "смотрит" вершиной вниз (`Down`). На рис. 5.12 приведены два варианта расположения кнопки `UpDown`. Очевидно, что стрелка влево связана с уменьшением текущего значения, а стрелка вправо — с увеличением.

Выбором варианта управляет свойство `Orientation`, принимающее одно из двух следующих значений:

- `Orientation = udVertical` — BCB/Delphi (значение по умолчанию);
- `Orientation = 0 (cc2orientation Vertical)` — VB (значение по умолчанию);

- `Orientation = udHorizontal` — BCB/Delphi;
- `Orientation = 1 (cc2orientation Horizontal)` — VB.

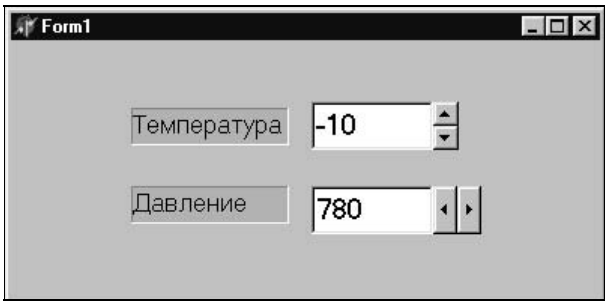


Рис. 5.12. Вертикальное и горизонтальное расположение кнопки `UpDown`

Главные свойства, определяющие поведение объекта `UpDown`, приведены в табл. 5.2.

Таблица 5.2. *Свойства объекта* `UpDown`

Свойство	Пояснение
`Min`	Нижняя граница числового значения
`Max`	Верхняя граница числового значения
`Position` или `Value` (VB)	Текущее числовое значение (показание счетчика)
`Increment`	Приращение, на которое изменяется текущее числовое значение
`Wrap`	Признак, управляющий поведением объекта при достижении текущим числовым значением верхней или нижней границы

Визуализация текущего числового значения объекта `UpDown` в средах BCB и Delphi, на наш взгляд, сделана более естественно и менее замысловато. Здесь введено свойство `Associate`, которое задает ассоциированный (связанный) со счетчиком объект, в котором отображается значение `Position`. Обычно в качестве связанного объекта выступает метка (`Label`) или окно ввода (`Edit`). Поэтому в поле значения свойства `Associate` надо выбрать имя нужного объекта, например, `Edit1`. После привязки окна кнопка автоматически приклеивается к окну справа или слева в зависимости от выбираемого вами значения свойства `AlignButton=udRight` или `AlignButton=udLeft`. При этом размер кнопки по высоте автоматически подгоняется к высоте окна.

После старта BCB- или Delphi-приложения в окне автоматически устанавливается минимальное значение, определяемое свойством `Min`. Значение в окне может изменяться в сторону увеличения (при щелчках по **Up** —

Вверх) или уменьшения (при щелчках по **Down** — Вниз) с шагом `Increment`. Попытка отредактировать значение в окне ввода выглядит удавшейся, но первый же щелчок по одному из треугольников кнопки подтверждает, что система помнит текущее значение `Position` и никак не реагирует на сделанные вами изменения в окне ввода. Действия, эквивалентные щелчкам по кнопке `UpDown`, могут быть инициированы стрелками управления курсора с клавиатуры (срабатывают только стрелки <вверх> и <вниз> независимо от ориентации кнопки), если значение свойства `ArrowKeys` равно `True`. Значение свойства `Wrap=True` превращает интервал допустимых значений счетчика в кольцо, когда максимальное и минимальное значения становятся соседними. Переход к значению `Max+Increment` переводит счетчик в положение `Min`, а переход к значению `Min-Increment` переводит счетчик в положение `Max`.

Приложение 5.8 (VB)

В среде Visual Basic привязка ассоциированного объекта, в котором может отображаться текущее значение свойства `Value`, выполняется с помощью свойства `BuddyControl`, аналогичного свойству `Associate`. Однако в VB необходимо дополнительно задать, в каком из свойств связанного объекта будет отображаться текущее значение счетчика, и требуется включить или отключить механизм синхронизации:

```
BuddyControl=Text1      'Назначение связанного объекта
BuddyProperties=Text    'Назначение свойства для Value
SyncBuddy=True          'Включение синхронизации
```

После назначения ассоциированного объекта и указания его свойства, связанного с текущим значением счетчика `UpDown`, необходимо присвоить ассоциированному свойству допустимое числовое значение. Именно оно и окажется начальным значением свойства `Value` в момент запуска VB-приложения. Если, например, значением свойства `Text` останется нечисловое `Text1`, то в качестве стартового значения `Value` будет выбрано значение `Min`. Наконец, если ассоциированный объект или его свойство не назначены, то начальное значение `Value` может быть задано на стадии проектирования.

Если механизм синхронизации включен, то одновременно с изменением текущего значения `Value` будут происходить изменения и в окне ввода. В отличие от BCB и Delphi в режиме синхронизации пользователь может изменить содержимое поля ввода в пределах интервала `[Min, Max]`, и оно после этого становится новым текущим значением счетчика `UpDown`.

Заметим, что установка ассоциированного объекта и свойства `BuddyProperties` в окне **Properties** (Свойства) на автомате устанавливает значение `SyncBuddy=True`. Обратная операция `SyncBuddy=False` также автоматически

сбрасывает ассоциированное свойство (BuddyProperties=(None)). Но это происходит только на стадии проектирования. Во время работы программы свойства SyncBuddy и BuddyProperties могут изменяться независимо друг от друга. Поэкспериментируйте с приложением, настраиваемые параметры которого приведены в табл. 5.3. Возможный вид формы приложения с расположенными на ней объектами приведен на рис. 5.13. Окно Text2 используется для отображения фактического состояния текущего значения Value, тогда как содержимое ассоциированного окна Text1 зависит от режима синхронизации.

Таблица 5.3. *Настраиваемые параметры приложения 5.8*

Объект	Свойство	Значение
Form1	—	—
Text1	Text	Text1 или 5
Text2	Text	Text2
	BuddyControl	Text1
UpDown1	BuddyProperty	Text
	Value	3

Текст обработчика события, возникающего при изменении текущего показания счетчика в результате щелчков по кнопке UpDown, приведен ниже:

```
Private Sub UpDown1_Change()
  Text2.Text=UpDown1.Value
  UpDown1.SyncBuddy=False   ' Попробуйте закомментировать эту строку
End Sub
```

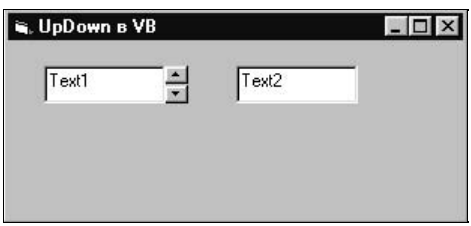


Рис. 5.13. Вид приложения со счетчиком UpDown

Разработчикам объекта UpDown в среде Visual Basic показалось мало того, что было накручено по поводу назначения ассоциированного объекта, и они ввели еще одно логическое свойство AutoBuddy. Если установить его значение True (а по умолчанию оно равно False), то выбор связанного компонента происходит автоматически. Если объект, размещенный перед кнопкой UpDown (его порядковый номер TabIndex на 1 меньше), годится для отобра-

жения значения `Value`, то он и становится ассоциированным компонентом. При этом также автоматически выбирается и значение `BuddyProperties`. Оно устанавливается равным `(None)`, если `SyncBuddy=False` и `(Default)` при `SyncBuddy=True`. Хорошо, что для окна ввода связанным свойством по умолчанию `(Default)` является свойство `Text`, хотя в явном виде об этом нигде не сказано. Если объект, предшествующий кнопке, нужным требованиям не соответствует, то его поиск продолжается среди элементов, установленных на форму после `UpDown`, и первый подходящий среди них становится ассоциированным компонентом. На наш взгляд, это — типичное архитектурное излишество в погоне за ненужным универсализмом. Совершенно не объясним и тот факт, что для выбора ассоциированного свойства VB предлагает список всех свойств связанного объекта, хотя многие из них никак не могут представлять и, уж тем более, отображать числовые значения.

5.3. Специфика обработки событий кнопок

Выше упоминалось, что главным событием основных кнопок, на которое должен реагировать соответствующий обработчик, является щелчок мышью по кнопке (`Click` в VB, `OnClick` в BCB и Delphi). По времени событию `Click` предшествует переход фокуса к кнопке. В среде Visual Basic такая ситуация вызывает событие `GotFocus`, в средах BCB и Delphi аналогичное событие называется `OnEnter`. Когда кнопка теряет фокус, возникает событие `LostFocus` (в VB) или `OnExit` (в BCB и Delphi). Каждое из перечисленных выше событий передает сообщение выбранной кнопке (точнее — обработчику соответствующего события этой кнопки), не содержащее никаких параметров.

Однако одновременно с выбором кнопки и возникновением одного из трех упомянутых событий пользователь может нажать какую-нибудь клавишу или сочетание клавиш на клавиатуре. В этой ситуации последовательно возникают события `KeyDown`, `KeyPress` и `KeyUp`, которые адресуются кнопке, находящейся в фокусе. В этих сообщениях обработчики событий получают код нажатой клавиши и битовую шкалу с комбинацией одновременно нажатых управляющих клавиш <Alt>, <Ctrl> и <Shift>.

Приложение 5.9 (VB)

Довольно часто кнопки объединяют в группы с целью упрощения обработки однотипных событий. Рассмотрим в качестве примера VB-приложения наборную панель микрокалькулятора, где будут реализованы только две функции — сброс окна операнда и набор целочисленного значения с помощью цифровых кнопок (рис. 5.14).

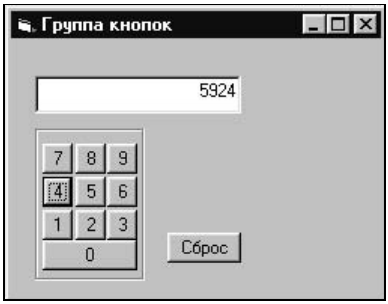


Рис. 5.14. Окно VB-приложения

Для формирования окна поместим рамку (`Frame1`), разместим на ней 10 командных кнопок (`Command1`, `Command2`). Выровняем их размеры и заменим стандартные заголовки на цифры от 0 до 9. Кнопки уже сгруппированы, но это выражается лишь в том, что перемещение рамки автоматически сдвигает и ее содержимое. Чтобы достичь большего, попробуем заменить значения свойств `Name` у всех кнопок на одно и то же имя, например, на `Cif`. Для первой кнопки такая замена проходит без замечаний. На второй — система предупредит нас о существовании объекта с таким же именем (рис. 5.15).

Рис. 5.15. Запрос подтверждения на создание массива компонент

Так как мы действительно хотим создать массив кнопок, что облегчит процедуру написания обработчика событий, то отвечаем утвердительно, и дальнейшее переименование проходит без замечаний. Переименование разумно вести в порядке возрастания цифр, соответствующих надписям на кнопках. Дело в том, что появления массива с кнопками порождает у каждой кнопки новое свойство — `Index`. Совпадение индекса кнопки с ее надписью также поможет нам упростить структуру обработчика события `Click`, который в случае массива будет обслуживать одновременно все кнопки:

```
Private Sub Cif_Click(Index As Integer)
 If Len(Text1.Text)=0 And Index=0 Then Exit Sub
 Text1.Text=Text1.Text & Chr(Asc(0) + Index)
End Sub
```

Обратите внимание на то, что теперь сообщение о щелчке передает индекс кнопки и этого достаточно, чтобы присоединить выделенную цифру к со-

держимому окна ввода. Первая строка обработчика не допускает набора нуля в качестве старшей цифры числа.

Еще одна кнопка `Command11` с надписью **Сброс** предназначена для очистки окна ввода:

```
Private Sub Command11_Click()
 Text1.Text = ""
End Sub
```

Приложение 5.10 (BCB)

Создадим аналогичный проект в среде BCB. Для объединения кнопок разместим на форме объект `GroupBox` и положим на него одну кнопку типа `TButton`. Подберем разумные размеры выделенной кнопки и скопируем ее в Clipboard командой **Copy** (Копировать) из меню **Edit** (Правка) (копировать можно не только выделенные фрагменты текста, но и выделенные объекты). Затем вставим скопированный объект и, зажав клавишу <Ctrl>, стрелками перемещения курсора отбуксируем кнопку на свое место. Повторим эти операции для каждой цифровой кнопки и заменим их заголовки (`Caption`) соответствующими цифрами. В качестве строки, предназначенной для индикации набираемого числа, придется выбрать объект `Label`. Более подходящее окно ввода `Edit`, к сожалению, не умеет прижимать текст к правой границе. Чтобы поле метки не потерялось на сером фоне формы, поменяем цвет метки (`Color=clBtnHeighlight`). Запретим метке подгонять свою ширину под размер текста (`AutoSize=false`) и установим правый прижим (`Alignment=taRightJustify`).

Для написания общего обработчика события `OnClick` на Си потребуется некоторая хитрость. Во-первых, надо выделить все цифровые кнопки и на закладке **Events** (События) в Инспекторе объектов щелкнуть по событию `OnClick`. Это позволит приписать всем кнопкам общую процедуру. Во-вторых, обработчик события получит указатель на объект `Sender`, пославший сообщение. Но этот указатель имеет обобщенный тип `TObject`. Поэтому нельзя напрямую обратиться за свойством `Sender->Caption`. Придется завести указатель типа `TButton` и выполнить приведение типа параметра `Sender` с помощью функции `static_cast`:

```
void __fastcall TForm1::Button10Click(TObject *Sender)
{//Прием цифры и отображение числа
 AnsiString k;
 TButton *pb=static_cast<TButton *>(Sender);
 K = pb->Caption;
 if(!(Label1->Caption == "" && k == '0'))
      Label1->Caption = Label1->Caption + k;
```

```
}
//-----------------------------------------------------------
void __fastcall TForm1::Button11Click(TObject *Sender)
{//Сброс числа
  Label1->Caption="";
}
//-----------------------------------------------------------
```

В приведенном примере для анализа нажатой кнопки было использовано свойство `Caption`, имеющее тип `AnsiString`. Такие данные удобно сравнивать, присоединять к другим текстовым величинам. В другой задаче более полезным могло оказаться целочисленное свойство `Tag`. По умолчанию его значение равно `0`, но мы можем присвоить для каждой кнопки соответствующее значение.

Приложение 5.11 (Delphi)

Разработка аналогичного проекта в среде Delphi тоже связана с преобразованием типа данных для параметра `Sender`. Однако в Паскале это преобразование сделано существенно проще и не требует применения дополнительного указателя:

```
procedure TForm1.Button10Click(Sender: TObject);
begin
  with Sender As TButton do
  begin
    if not((Label1.Caption='')and(Caption ='0')) then
      Label1.Caption:=Label1.Caption + Caption;
  end
end;

procedure TForm1.Button11Click(Sender: TObject);
begin
  Label1.Caption:='';
end;
```

5.4. Создание главного меню приложения

Строка главного меню, в которой расположены заголовки, и связанные с ней ниспадающие меню являются принадлежностью большинства Windows-приложений. За исключением чисто внешнего оформления заголовки меню и команды ниспадающих меню функционально не отличаются

от обычных кнопок. Основная разница заключается в том, что кнопки уже находятся в составе стандартных объектов визуальных сред, откуда их можно извлечь двойным щелчком мыши, а элементы меню надо описать.

Подобно любому объекту управления каждому элементу меню приписываются такие свойства, как `Caption` (обозначение элемента в строке меню, появляющееся на экране), `Name` (идентификатор элемента в обработчике события), `Enabled` (признак доступности), `Visible` (признак видимости) и некоторые другие.

К числу особых свойств элементов меню относятся следующие:

- наличие или отсутствие признака выделения исполненной команды-галочки, размещаемой слева от наименования (характеристика `Checked`);
- наличие или отсутствие клавишного эквивалента команды функциональной клавиши или комбинации управляющей клавиши (<Ctrl>, <Shift>) с какой-либо обычной клавишей (характеристика `Shortcut`);
- выделение в заголовке меню или в обозначении команды *"горячей" клавиши*, набор которой в сочетании с клавишей <Alt> эквивалентен щелчку по элементу меню.

Наконец, немаловажным обстоятельством в организации меню является его древовидная структура. В ее корне располагается заголовок, под которым "висит" меню первого уровня. Выполнение некоторых команд меню первого уровня может порождать меню второго уровня и т. д.

Помимо главного меню (объект `MainMenu`), разделы которого размещаются под заголовком формы, с формой или с любым из находящихся на ней объектов может быть связано индивидуальное всплывающее меню (`PopupMenu`), эквивалентное одному разделу главного меню. Его единственная особенность заключается в том, что всплывающее меню активизируется (показывается на экране) в момент нажатия правой клавиши мыши, когда ее курсор находится над ассоциированным объектом.

Создание главного или всплывающего меню осуществляется с помощью редактора меню, вызов которого в разных визуальных средах и техника набора элементов меню слегка различаются.

5.4.1. Работа с *Menu Editor* в среде Visual Basic

Перед тем, как обратиться к **Menu Editor** (Редактору меню), необходимо активизировать форму, для которой мы собираемся создавать меню. Для этого достаточно щелкнуть по любой свободной части рабочего поля формы. Затем нужно перейти в раздел **Tools** (Инструменты) главного меню и выполнить команду **Menu Editor** (Редактор меню). Немного быстрее все это можно проделать с помощью клавишной команды <Ctrl>+<E>.

В любом случае перед нами возникает окно редактора меню, представленное на рис. 5.16. В этом окне происходит формирование заголовков разделов главного меню и команд меню разных уровней с присущими им атрибутами — выделенными (подчеркнутыми) буквами, клавишными эквивалентами команд (**Shortcut**), наличием отметки (**Checked**), признаками доступности (**Enabled**) и видимости (**Visible**).

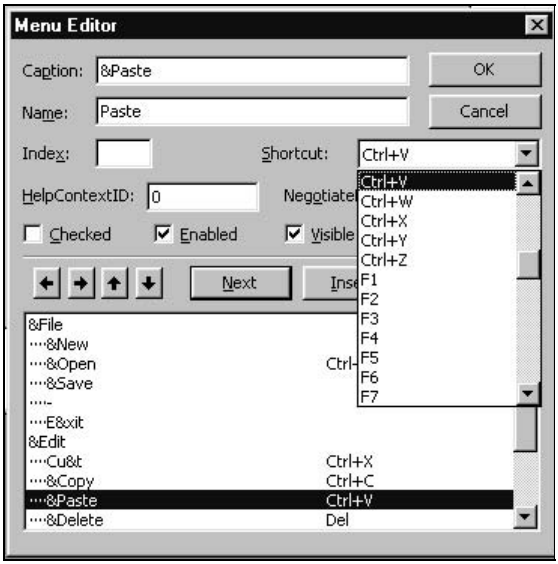


Рис. 5.16. Окно редактора меню в среде VB 6.0

Окно **Menu Editor** состоит из двух половинок. В верхней его части находится несколько полей ввода, логических и командных кнопок, а нижняя часть предназначена для отображения формируемых элементов меню и уровней их вложенности. Условимся называть эту часть окна *полем меню.*

Каждая строка поля меню предназначена для хранения одного элемента меню — либо заголовка (корня меню), либо команды того или иного уровня. Имя заголовка всегда располагается с 1-й позиции. Если в какой-то его позиции находится символ "&", то следующая за ним буква символизирует *"горячую" клавишу.* В созданном меню она будет подчеркнута, и комбинация этой буквы с управляющей клавишей <Alt> эквивалентна щелчку по соответствующему элементу меню.

Командам из меню первого уровня в строке поля меню предшествуют четыре точки. Такой сдвиг создается, если в процессе формирования очередного элемента меню мы нажали на кнопку с жирной правой стрелкой. Для следующего элемента того же уровня автоматически сохраняется сдвиг от предыдущей команды. Повторное нажатие на кнопку с правой стрелкой вызовет дополнительный сдвиг еще на четыре позиции, что будет соответст-

вовать командам меню второго уровня и т. д. Любое перемещение по уровням вложенности организуется с помощью кнопок с горизонтальными стрелками. Если имя команды состоит из единственного знака "минус", то в меню ей соответствует разделительная линия. Таким способом стараются отделить группу команд, имеющих общее функциональное назначение.

Кнопки с вертикальными стрелками обеспечивают вертикальные перемещения текущей (выделенной синим цветом) строки по полю меню. Такие же перемещения можно организовать с помощью мыши — щелкните по какой-либо строке, и она станет текущей. Это может понадобиться для модернизации ранее набранной информации, отображенной в текущей строке. Вы можете отредактировать текущий элемент меню или все связанные с ним характеристики, удалить элемент или вставить перед ним новый. Две последние операции выполняются с помощью кнопок **Delete** (Удалить) и **Insert** (Вставить), не имеющих ничего общего с аналогичными клавишами на клавиатуре.

Кнопку **Next** (Далее) или клавишу <Enter> нажимают после того, как завершен набор всех характеристик очередного элемента меню. После ввода описаний всех элементов меню нажимают кнопку **OK**, что приведет к завершению работы Редактора меню и присоединению построенной нами схемы к активной форме. Не забывайте, что таким образом создается только оболочка меню. Все внешние атрибуты меню на форме появятся (рис. 5.17), каждое ниспадающее меню после щелчка по заголовку раздела будет открываться, но никаких действий в ответ на щелчки по командам меню на стадии выполнения не последует, т. к. мы еще не предусмотрели набор соответствующих событийных процедур.

5.4.2. Пример создания оболочки меню в среде Visual Basic

Предположим, что мы хотим разместить на форме меню, разделы которого со всеми нюансами представлены на рис. 5.17.

Приложение 5.12 (VB)

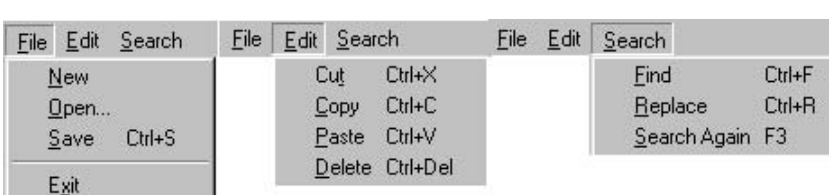


Рис. 5.17. Главное меню с тремя разделами

Активизируем форму и вызываем **Menu Editor**. Из приведенного состава меню следует, что содержимое строк поля меню, которое нам предстоит сформировать, должно выглядеть следующим образом:

&File

.&New

.&Open

.&Save Ctrl+S

.-

.E&xit

&Edit

.Cu&t Ctrl+X

.&Copy Ctrl+C

.&Paste Ctrl+V

.&Delete Ctrl+Del

&Search

.&Find Ctrl+F

.&Replace Ctrl+R

.&Search Again F3

Для формирования строки 1 в поле меню набираем в окне с обозначением **Caption** (Заголовок) текст **&File**. Повторяем его же, но уже без амперсанда, в окне с обозначением **Name** (Имя). В принципе, программное имя первого заголовка меню может быть и другим. Однако чаще всего имена характеристик **Caption** (Заголовок) и **Name** (Имя) дублируют, чтобы не запутаться в двойных обозначениях. Для перехода к описанию следующего элемента меню нажимаем кнопку **Next** (Следующая).

Перед набором следующих пяти команд меню, открываемого заголовком **File** (Файл), стоит один раз нажать кнопку с жирной правой стрелкой для установки первого уровня вложенности. После этого набор 2-й и 3-й строк ничем не отличается от описания первого элемента меню.

Формирование 4-й строки на поле меню сопряжено с некоторым нюансом — мы должны пополнить наименование элемента меню обозначением клавишной команды (<Ctrl>+<S>). Для этого открываем список **Shortkeys** (Горячие клавиши) и выбираем в нем строку с клавишной командой <Ctrl>+<S>. Она автоматически будет перенесена в текущую строку. Этим же приемом придется воспользоваться и при наборе 8-й и последующих строк.

Еще один нюанс нас ожидает при наборе 5-й строки, представляющей описание разделительной линии в меню. В поле **Caption** (Заголовок) достаточно

набрать единственный символ минус. Несмотря на то, что линия в меню весьма условно может быть названа объектом управления, ей, тем не менее, придется присвоить имя в поле **Name** (Имя). В противном случае в момент выхода из Редактора меню вы получите сообщение о нехватке данных. Естественно, что имя разделительной линии не может начинаться со знака минус. Каждый программист должен как-то единообразно именовать разделительные линии, например — `qq1`, `qq2`, `qq3`, .

Перед набором 7-й строки необходимо вернуться по уровню вложенности назад, воспользовавшись кнопкой с левой стрелкой. Набор остальных элементов меню выполняется без каких-либо проблем. Перед тем, как нажать клавишу **OK** и выйти из Редактора меню, вы можете еще раз совершить путешествие по всем строкам поля меню и проследить за всеми характеристиками описанных элементов. Они автоматически отображаются в окнах ввода и на отметках логических кнопок.

Небольшая неточность в реализации данного меню связана с выбором клавишной комбинации для команды **Delete** (Удалить). В среде Visual Basic для нее принята одноклавишная команда <Del>, а не сочетание клавиш <Ctrl>+<Del>, как это сделано в системах фирмы Borland. Поэтому из списка **Shortkeys** (Горячие клавиши) пришлось извлекать то, что там есть.

Обращаем внимание на то, что при описании элементов меню вы не должны отключать признаки `Enabled` и `Visible`. В противном случае такие строки меню окажутся недоступными на стадии проектирования программы, и вы не сможете написать для них событийные процедуры.

После выхода из окна **Menu Editor** (Редактор меню) действующая оболочка меню появится на форме (рис. 5.18), и для каждой команды в соответствующем разделе меню придется набрать обработчик события `Click` точно таким же образом, как и для любой кнопки.

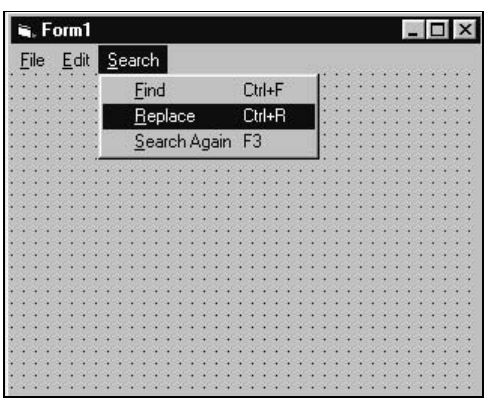


Рис. 5.18. Внешний вид оболочки главного меню

5.4.3. Пример создания оболочки меню в среде Borland C++ Builder

Для создания главного меню в системах BCB/Delphi на форму следует поместить объект `MainMenu` (рис. 5.19).

Приложение 5.13 (BCB)

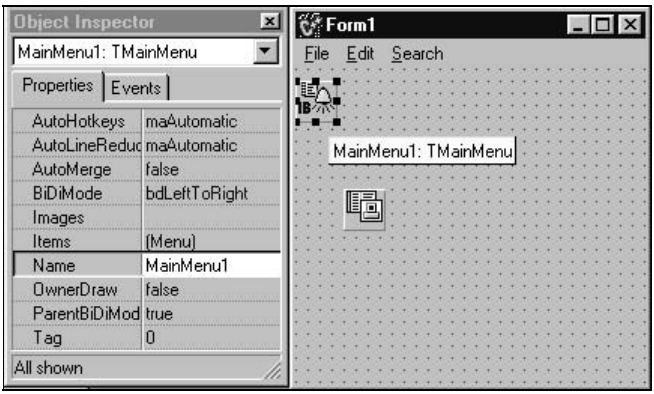


Рис. 5.19. Свойства компонента `MainMenu`

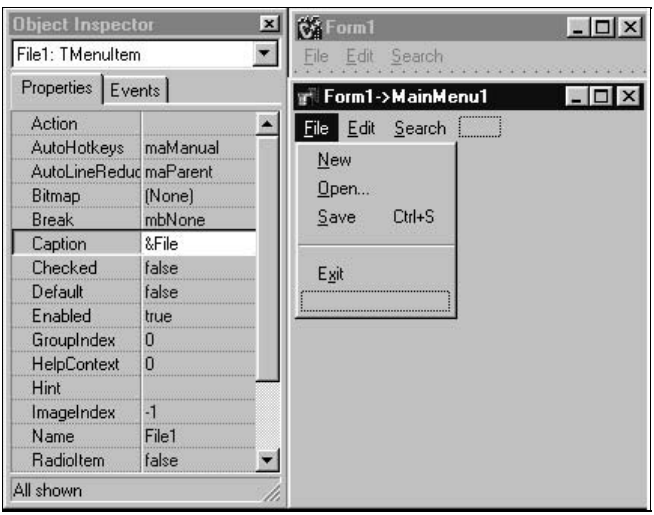


Рис. 5.20. Окно редактора меню в среде BCB

Для вызова редактора меню достаточно два раза щелкнуть по значению свойства `Items` или по пиктограмме объекта `MainMenu`, после чего на экране появится окно редактора меню (рис. 5.20) В BCB и Delphi для этого редак-

тора нет специального названия, но заголовок окна отражает существо дела. В отличие от автономного редактора VB, где набор всех элементов команды меню ведется на одной форме, в среде BCB/Delphi постоянно приходится переключаться между окном редактора меню и страницей значений свойств команды — объекта типа `TMenuItem`.

В окне редактора в начальный момент находится выделенное окно, предназначенное для фиксации наименования первого раздела меню, а в окне Инспектора объектов на вкладке свойств нам придется набирать необходимые значения. Для названия раздела **File** (Файл) достаточно в поле значения **Caption** (Заголовок) набрать `&File`. После нажатия клавиши <Enter> в окне редактора появится первый заголовок — **File**, а прямоугольник сместится вправо для набора заголовка следующего раздела. Так как меню удобнее набирать по разделам, то в окне редактора можно принудительно щелкнуть под названием первого раздела или нажать клавишу, перемещающую курсор клавиатуры влево, — к предыдущему разделу. Тогда появится пунктирный прямоугольник для набора первой команды меню **File** (Файл) (рис. 5.21)

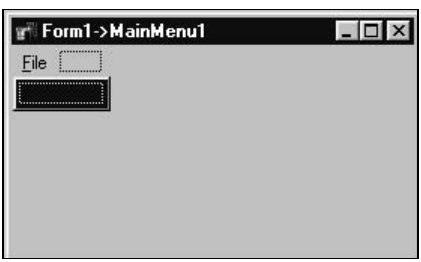


Рис. 5.21. Подготовка к набору команд раздела **File**

Одновременно в Инспекторе объектов открывается страничка для очередного объекта типа `TMenuItem`. Текст команды набирается в окне `Caption` с использованием символа `&` перед подчеркиваемой буквой. Клавишная комбинация, эквивалентная команде, выбирается из списка `Shortcut`. Значение параметра `Name`, которое в VB приходилось набирать пользователю, здесь назначается автоматически. Например, если `Caption=New`, то в свойство `Name` засылается значение `New1`. Однако среди наименований команд могут встретиться неподходящие комбинации — например, линия раздела имеет заголовок, состоящий из единственного знака минус. Заголовок команды может быть написан на русском языке, но идентификатор `Name` не может содержать русские буквы. В таких случаях система присваивает подобным командам имена типа `N1, N2, N3,…` . В остальном набор команд меню мало чем отличается от технологии, описанной в *разд. 5.4.2 данной главы*. Окно редактора меню, фактически, служит экраном, на котором динамически отображается процесс построения оболочки меню. И лишь изредка помогает в конструировании разделов меню и переходов в подменю других уровней.

Приложение 5.14 (Delphi)

В среде Delphi 6.0 появилось новое окно просмотра объектов в виде дерева (`Object TreeView`), что особенно удобно при работе с меню (рис. 5.22).

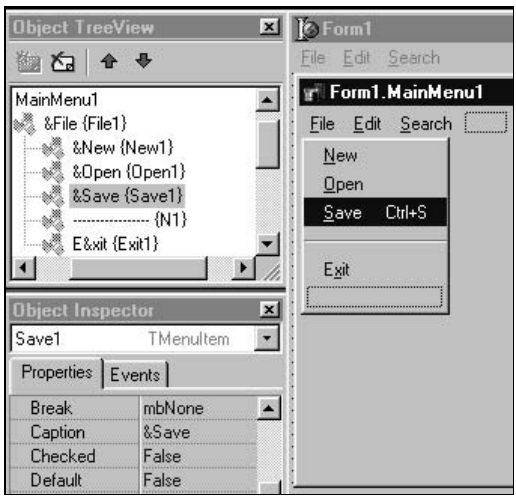


Рис. 5.22. Процесс создания главного меню в среде Delphi 6.0

5.4.4. Создание всплывающих меню в Visual Basic

Создание всплывающего меню выполняется точно так же, как и построение главного меню, содержащего единственный раздел. Этот раздел может быть снабжен подменю второго уровня и подменю более высоких уровней. Единственное отличие всплывающего меню от главного заключается в том, что при формировании строки начала раздела нужно отключить признак видимости, т. е. удалить галочку из пометки `Visible` (рис. 5.23). Все остальные команды раздела должны набираться как видимые.

Завершив построение меню, мы не обнаружим его на форме, т. к. заголовок раздела был объявлен невидимым. Но в списке объектов, входящих в состав нашего приложения, мы обнаружим имена всех команд только что созданного меню. Имя `Pop1`, которое было указано в качестве свойства `Name` для невидимого заголовка раздела, система рассматривает как идентификатор всплывающего меню.

К сожалению, в среде Visual Basic недостаточно нажать правую клавишу мыши, чтобы всплывающее меню появилось на экране. Делать это приходится принудительно, подключив к приложению обработчик события `MouseDown`, в котором производится вызов метода `PopupMenu`. Текст этого обработчика для всплывающего меню, связанного с формой и содержащего

всего три строки, приведен на рис. 5.24. Если всплывающее меню должно быть привязано к другому объекту, то аналогичное обращение к методу `PopupMenu` следует поместить в процедуру обработки события `MouseDown` для ассоциированного объекта.

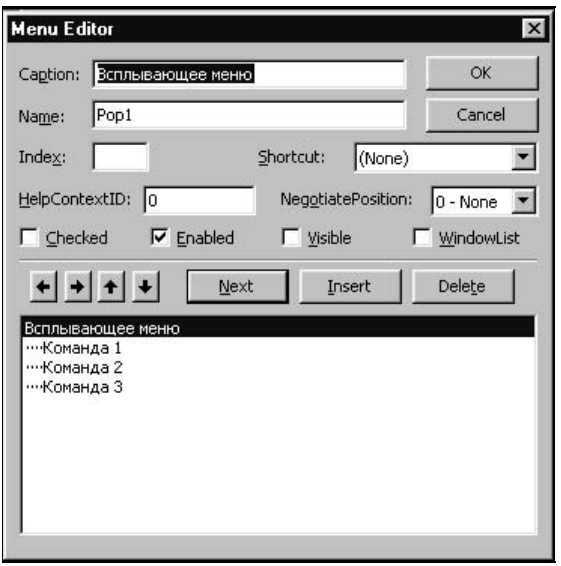


Рис. 5.23. Набор команд всплывающего меню

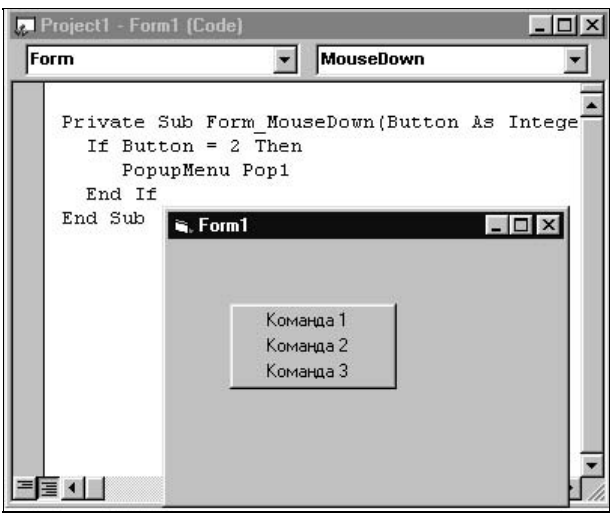


Рис. 5.24. Всплывающее меню на фоне обработчика события

5.4.5. Создание всплывающих меню в Borland C++ Builer и Delphi

В визуальных средах BCB и Delphi всплывающие меню создаются и проявляются в работе немного проще. С любым объектом, так же как и с формой можно связать всплывающее меню. Создается такое меню тем же способом, что и главное. Только вместо объекта `MainMenu` на форму вызывается компонент `PoupMenu` и его имя заносится в одноименное свойство хозяина всплывающего меню (рис. 5.25). Двойной щелчок по пиктограмме объекта переводит нас в окно редактора (рис. 5.26).

Программа может в определенные моменты времени запрещать или разрешать автоматическое появление всплывающего меню, манипулируя значением свойства `AutoPop`. В последних версиях BCB и Delphi появилась возможность создания более комфортабельных условий для левши. Свойство `TrackButton`, по умолчанию имеющее значение `tbRightButton`, можно изменить на `tbLeftButton`.

На прилагаемом к книге носителе приведены три идентичных приложения, содержащие главное и всплывающее меню. Каждое из них фиксирует выбранную строку меню и сообщает об этом в диалоговом окне **ShowMessage**. Никакой другой содержательной работы по исполнению команд меню в этих приложениях не предусмотрено.

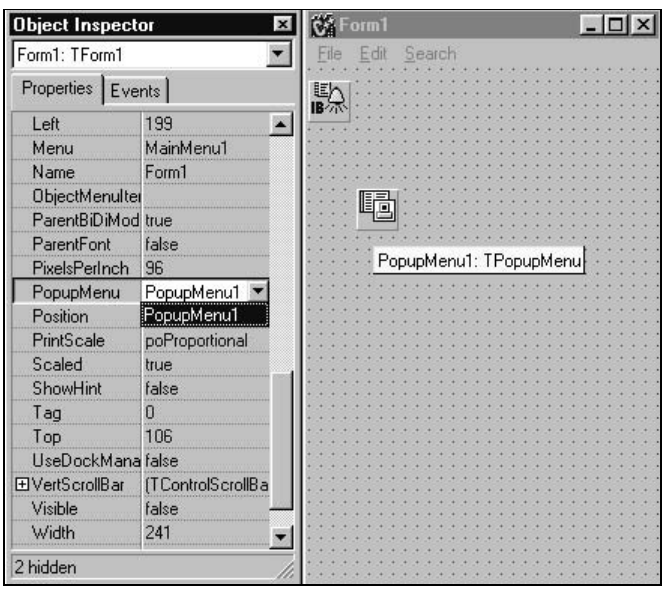


Рис. 5.25. Подготовка к созданию всплывающего меню в BCB

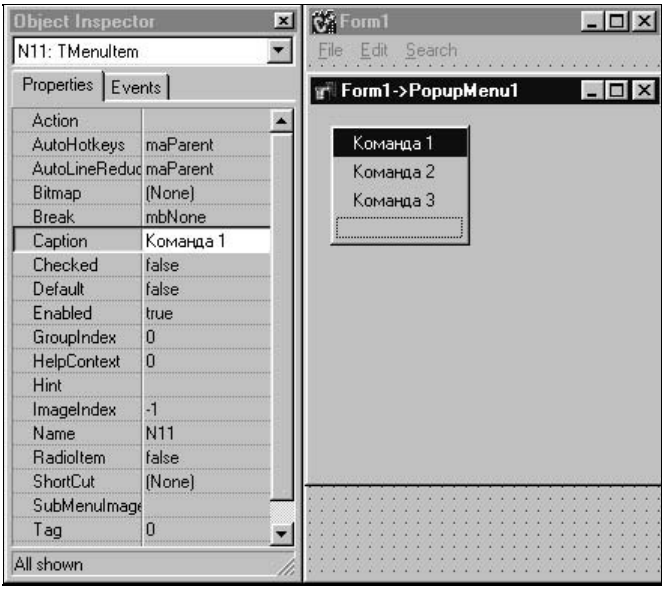


Рис. 5.26. Набор команд всплывающего меню

5.5. Нестандартное меню

В приложении можно создать несколько главных и всплывающих меню и, в зависимости от сложившейся ситуации, присваивать свойствам `MainMenu` и `PopupMenu` нужные значения. Кроме того, главное меню головной формы может сливаться с пунктами меню дочерних форм. Отдельные пункты меню можно исключать, делать недоступными или заменять новыми. Однако, во-первых, это занятие достаточно хлопотное. Во-вторых, инициатива входа в меню и право выбора той или иной команды целиком принадлежат пользователю.

Приложение 5.15 (BCB)

В некоторых приложениях полезно высвечивать различные меню по запланированной логике программы и требовать от пользователя обязательного выбора одного из предложенных блюд. Именно для таких ситуаций предлагается нестандартное решение, описываемое в данном разделе. Идея нестандартного меню заключается в размещении очередного раздела команд в окне списка типа `ListBox`, размещенного на вспомогательной форме `FormMenu`, которая динамически создается, формируется по номеру раздела и отображается в режиме `ShowModal`. Отмахнуться от такого окна пользователь просто так не может, а если еще заголовок окна лишить управляющих

кнопок, то единственный выход продолжить работу приложения состоит в выборе одной из строк предложенного раздела меню.

Любая функция приложения может обратиться к процедуре `ShowMenu`, передав ей в качестве единственного параметра целочисленный номер раздела меню. После подготовительной работы по созданию окна **FormMenu** управление будет передано вспомогательной форме, в момент активизации которой возникнет событие `OnActivate`. В обработчике этого события и сосредоточены все операции по формированию заданного раздела и подгонке размеров формы под габариты списка. В заголовке формы `FormMenu` предлагается расположить название раздела отображаемого меню. Для связи по данным между главной и вспомогательной формами номер текущего раздела меню можно передать в глобальную переменную `Num_Menu`.

В системе BCB функция `ShowMenu` должна быть включена в состав модуля `Unit1`, обслуживающего головную форму, и может быть реализована следующим образом:

```
int ShowMenu(int k)
{
  Num_Menu=k;
  if(FormMenu != NULL) delete FormMenu;
  FormMenu = new TFormMenu(Application);
  return FormMenu->ShowModal();
}
```

Проверка указателя `FormMenu` предпринята для того, чтобы в случае необходимости удалить форму с предыдущим разделом из памяти. Затем создается новая форма с помощью оператора `new` и ей передается управление в модальном режиме. Возврат управления в функцию `ShowMenu` теперь произойдет только тогда, когда один из обработчиков событий формы `FormMenu` зашлет в системную переменную `ModalResult` ненулевое значение. И в качестве такого значения естественно выбрать номер выделенной строки списка, увеличенный на 1 (строки в списке нумеруются от 0).

Для демонстрации изложенной идеи мы ограничились формированием двух разделов меню, содержащих соответственно 3 и 6 команд. А вместо выполнения выбранной команды на экран выдается сообщение о выборе пользователя. Сформированное меню подгоняется по высоте под число команд в разделе, а форма `FormMenu` смещается в верхний правый угол, чтобы не занимать экран. Дополнительные смещения в командах подгонки высоты и положения объектов пришлось подбирать экспериментальным путем. Возможно, что при другом размере или разрешении экрана вам придется проделать то же самое.

Обработчик события `OnKeyDown` пришлось подключить потому, что без него список реагирует на нажатие некоторых клавиш — стрелки управления кур-

сором, пробел, <End>, <Home>, <Page Up>, <Page Down>. При этом помеченной всегда оказывалась первая строка списка. Один из фрагментов работы с нестандартным меню показан на рис. 5.27.

```
// Модуль Unit1
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "Unit2.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
int Num_Menu;  // Глобальная переменная для номера меню
int ShowMenu(int k);
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent *Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject* /*Sender*/)
{// Имитация работы с меню
  int resp;
  resp=ShowMenu(1); // Вывод меню с командами раздела 1
  switch (resp)
  { case 1: ShowMessage("Выбрана Команда 1-1");break;
    case 2: ShowMessage("Выбрана Команда 1-2");break;
    case 3: ShowMessage("Выбрана Команда 1-3");
  }
  resp=ShowMenu(2); // Вывод меню с командами раздела 2
  switch (resp)
  { case 1: ShowMessage("Выбрана Команда 2-1");break;
    case 2: ShowMessage("Выбрана Команда 2-2");break;
    case 3: ShowMessage("Выбрана Команда 2-3");break;
    case 4: ShowMessage("Выбрана Команда 2-4");break;
    case 5: ShowMessage("Выбрана Команда 2-5");break;
    case 6: ShowMessage("Выбрана Команда 2-6");
  }
}
```

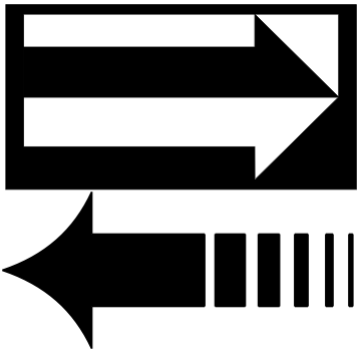
```
//---------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{// Освобождение занятых ресурсов
  if(FormMenu != NULL) delete FormMenu;
}
//---------------------------------------------------------------------
int ShowMenu(int k)
{// Подготовка и создание формы для меню
  Num_Menu=k;
  if(FormMenu != NULL) delete FormMenu;
  FormMenu = new TFormMenu(Application);
  return FormMenu->ShowModal();
}

// Модуль Unit2
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TFormMenu *FormMenu;
extern int Num_Menu;
//---------------------------------------------------------------------
__fastcall TFormMenu::TFormMenu(TComponent *Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TFormMenu::FormActivate(TObject *Sender)
{// Формирование команд меню в списке
  ListMenu->Clear();
  switch(Num_Menu)
  {
    case 1: Caption="Раздел 1";
            ListMenu->Items->Add("Команда 1-1");
            ListMenu->Items->Add("Команда 1-2");
            ListMenu->Items->Add("Команда 1-3");
            break;
```

```
    case 2: Caption="Раздел 2";
            ListMenu->Items->Add("Команда 2-1");
            ListMenu->Items->Add("Команда 2-2");
            ListMenu->Items->Add("Команда 2-3");
            ListMenu->Items->Add("Команда 2-4");
            ListMenu->Items->Add("Команда 2-5");
            ListMenu->Items->Add("Команда 2-6");
            break;
   }
  Top=0;                        // Прижим формы к верхней части экрана
  Left=Screen->Width-Width-2;   // Прижим формы к правой границе
// Установка высоты списка с небольшим запасом
  ListMenu->Height=ListMenu->Items->Count*ListMenu->ItemHeight+5;
// Установка высоты формы с небольшим запасом
  Height=ListMenu->Height+27;
}
//---------------------------------------------------------------------
void __fastcall TFormMenu::ListMenuClick(TObject *Sender)
{
  ModalResult=ListMenu->ItemIndex+1;
}
//---------------------------------------------------------------------
void __fastcall TFormMenu::ListMenuKeyDown(TObject *Sender,
                              WORD &Key, TShiftState Shift)
{
  Key=0;
}
```

Рис. 5.27. Нестандартное меню рядом с главной формой

Глава 6

Стандартные интерфейсные объекты



В этой главе вы познакомитесь с компонентами, представленными типовыми диалоговыми окнами. Большинство из них используется в таких Windows-приложениях, как изучаемые системы визуального программирования, MS Word, CorelDRAW и во многих других. Совсем немного усилий вам придется приложить для того, чтобы использовать эти объекты в своих приложениях.

6.1. Диалоговое окно для ввода текста

В системе Visual Basic для ввода текстовых сообщений используется диалоговая функция `InputBox`:

```
A$ = InputBox(Prompt,Caption,Default,x,y,HelpFile,Context)
```

Приложение 6.1 (VB)

Единственным обязательным параметром этой функции является аргумент `Prompt` — значение типа `String`, представляющее подсказку, появляющуюся в диалоговом окне. Например, в результате обращения `Name$=InputBox("Введите свою фамилию")` из приложения "Анкета" на экране появится диалоговое окно, изображенное на рис. 6.1.

Рис. 6.1. Диалоговое окно для ввода строки

После набора затребованной информации пользователь может нажать клавишу <Enter> или щелкнуть по кнопке **OK**. И в том, и в другом случае функция `InputBox` возвратит содержимое поля ввода. В поле ввода до сигнала завершения набора можно производить исправления, допустимые средствами клавиатуры. Если пользователь щелкает по кнопке **Cancel** (Отмена) или нажимает клавишу <Esc>, то функция возвращает строку нулевой длины независимо от того, что находится в поле ввода.

Длина подсказки `Prompt` может быть достаточно большой (до 1 000 символов) и даже занимать несколько строк. Но разделители между строками в виде управляющих символов с кодами `&H0D` и `&H0A` вы должны включить в текст сами.

Параметр `Caption`, если он задан, появляется в качестве заголовка диалогового окна. На рис. 6.1 заголовок диалогового окна совпадает с наименованием приложения (не путать с заголовком формы). Третий параметр `Default` — значение типа `String` — задает предполагаемый ответ по умолчанию. Он появляется в поле ввода и может сэкономить время набора, если ответ пользователя совпадает с предполагаемым текстом.

Приложение 6.2 (VB)

Например, в результате следующего обращения появится окно, представленное на рис. 6.2.:

```
A$=InputBox("Я хотел бы жить и умереть в...",_
            "Надо продолжить фразу", "в Париже")
```

```
Private Sub Form_Load()
  A$ = InputBox("Я хотел бы жить и умереть …", _
                "Надо продолжить фразу", "в Париже")
  Caption = Len(A$)
End Sub
```

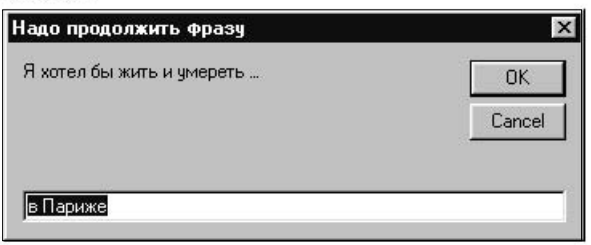


Рис. 6.2. Диалоговое окно с тремя параметрами

Два следующие параметра (`x`, `y`) определяют смещения диалогового окна по горизонтали и вертикали относительно левого верхнего угла экрана. Значения этих координат задаются в твипах. При их отсутствии диалоговое окно появляется в центре экрана.

Наконец, два последние необязательные параметра задают имя файла помощи и номер кадра в справочной информации, которые должны появиться, если пользователь нажмет клавишу <F1>.

Приложение 6.3 (BCB)

Окно **InputBox** (рис. 6.3) доступно и для пользователей систем BCB/Delphi:

```
S := InputBox(Caption, Prompt, Default);
```

Отличается оно от аналогичного объекта в VB только перестановкой параметров и отсутствием возможности задать позицию окна на экране. В системе BCB все аргументы функции являются данными типа `AnsiString`, в системе Delphi — данными типа `String`. Несколько иначе реагирует это окно на щелчок по кнопке **Cancel** — возвращается не пустая строка, а значение `Default`.

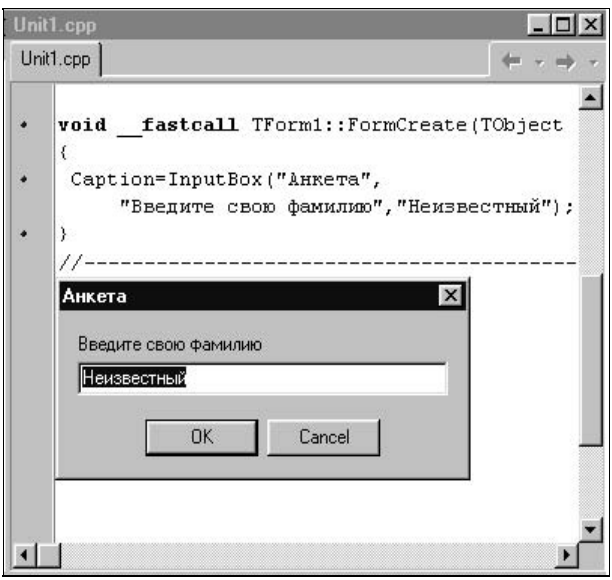


Рис. 6.3. Диалоговое окно **InputBox** на фоне фрагмента программы ввода

Чуть более широкие возможности по вводу символьных данных в BCB/Delphi предоставляет диалоговое окно **InputQuery**, которое наряду с формированием значения результирующей строки возвращает логическое значение, определяющее выбор пользователя:

```
bool InputQuery(Caption, Prompt, &Value);
```

Третий аргумент в этой функции предназначен для размещения результирующей строки (в BCB он задан ссылкой на строку типа `AnsiString`). Если

пользователь щелкнул по кнопке **OK** или нажал клавишу <Enter>, то функция `InputQuery` возвращает значение `True`.

6.2. Диалоговые окна для вывода сообщений

6.2.1. Функция *MsgBox* в системе Visual Basic

Функция `MsgBox` предназначена для вывода сообщения, адресованного пользователю, и ожидания реакции пользователя в виде числового кода нажатой кнопки:

```
K = MsgBox(Msg,Buttons,Caption,HelpFile,Context])
```

Приложение 6.4 (VB)

Единственным обязательным параметром функции `MsgBox` является текст сообщения `Msg`. Если задан только он, то в диалоговом окне наряду с сообщением `Msg` появляется только кнопка **OK** (рис. 6.4). Ничего другого, кроме как нажать единственную кнопку, пользователю не остается — крестик и тот не доступен. Числовой код, соответствующий нажимаемой кнопке, в данном случае равен `1` (системная константа `vbOK`).

Если вы хотите ограничиться только выводом сообщения, то к функции `MsgBox` можно обратиться как к оператору:

```
MsgBox Msg
```

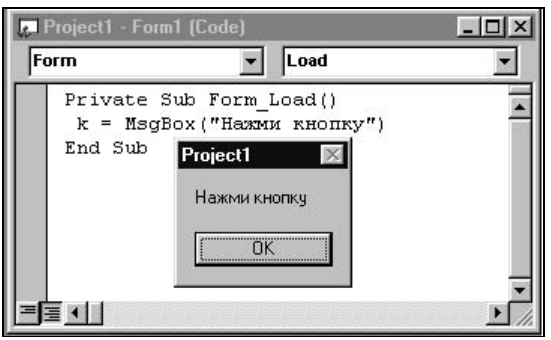


Рис. 6.4. Самое простое диалоговое окно типа `MsgBox`

Параметр `buttons`, представленный целочисленным значением, определяет, какие кнопки и пиктограммы должны быть размещены в диалоговом окне. Формируется оно как сумма четырех слагаемых, первое из которых определяет совокупность кнопок, второе — тип иконки, третье определяет порядковый номер кнопки по умолчанию, четвертое — режим модальности.

К последнему слагаемому прибегают крайне редко. А список наиболее употребительных констант и соответствующих числовых значений приведен в табл. 6.1.

Таблица 6.1. *Символьные константы и их числовые значения, доступные для параметра* `buttons`

Число	Константа	Пояснение
		Комбинации кнопок
0	`VbOKOnly`	Только кнопка **OK**
1	`VbOKCancel`	Кнопки **OK** и **Cancel** (Отмена)
2	`VbAbortRetryIgnore`	Кнопки **Abort** (Стоп), **Retry** (Повтор), **Ignore** (Пропустить)
3	`VbYesNoCancel`	Кнопки **Yes** (Да), **No** (Нет), **Cancel** (Отмена)
4	`VbYesNo`	Кнопки **Yes** (Да), **No** (Нет)
5	`VbRetryCancel`	Кнопки **Retry** (Повтор), **Cancel** (Отмена)
		Тип пиктограммы
16	`VbCritical`	Запрещающий знак в кружочке
32	`VbQuestion`	Вопросительный знак в кружочке
48	`VbExclamation`	Восклицательный знак в кружочке
64	`VbInformation`	Буква `i` в кружочке
		Номер активизируемой кнопки, срабатывающей по умолчанию
0	`VbDefaultButton1`	Кнопка номер 1
256	`VbDefaultButton2`	Кнопка номер 2
512	`VbDefaultButton3`	Кнопка номер 3
768	`VbDefaultButton4`	Кнопка номер 4, в ее роли выступает **Help** (Справка)

Числовое значение, возвращаемое функцией `MsgBox`, определяет кнопку, нажатую пользователем:

- 1 — **OK**;
- 2 — **Cancel** (Отмена);
- 3 — **Abort** (Стоп);
- 4 — **Retry** (Повтор);
- 5 — **Ignore** (Пропустить);
- 6 — **Yes** (Да);
- 7 — **No** (Нет).

Приложения 6.5, 6.6 (VB)

Наличие пиктограммы в окне может придать более эмоциональную окраску тексту сообщения (рис. 6.5 и 6.6).

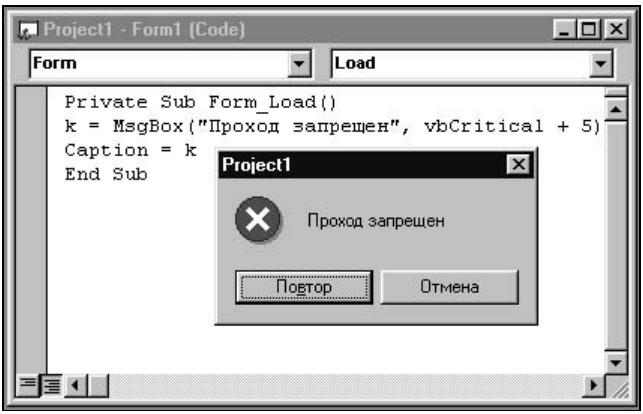


Рис. 6.5. Сообщение с запрещающим знаком

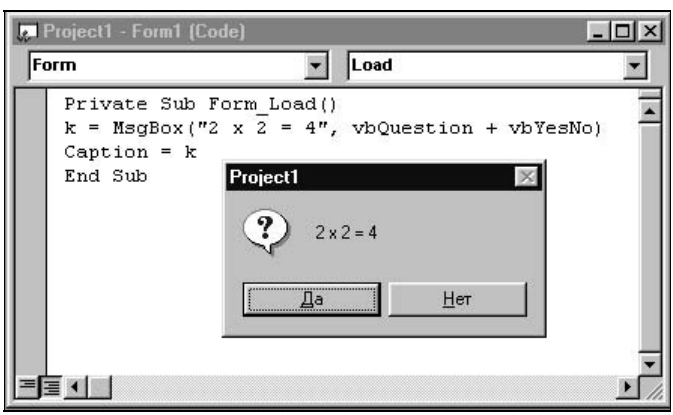


Рис. 6.6. Сообщение в форме вопроса

6.2.2. Вывод сообщений в системах BCB и Delphi

В системах BCB и Delphi существует довольно большое количество диалоговых окон, предназначенных для вывода сообщений и анализа реакции пользователя, — `ShowMessage`, `MessageBox`, `MessageDlg`, `MessageDlgPos`. Однако их функциональные возможности мало чем отличаются от окна типа `MsgBox` в системе Visual Basic.

Процедура `ShowMessage` отображает заданное сообщение в диалоговом окне с единственной кнопкой **OK**. Заголовок диалогового окна совпадает с именем приложения.

Функция `MessageBox` обладает несколько более широкими возможностями, т. к. располагает тремя аргументами:

```
int k = MessageBox(Message, Caption, Buttons);
```

Первый ее аргумент представляет указатель на текст отображаемого сообщения. Оно может быть достаточно длинным и автоматически разбивается на строки, помещающиеся в окне. Второй параметр задает строку, которая выступает в качестве заголовка диалогового окна. Комбинация кнопок в диалоговом окне задается числовым аргументом `Buttons`. Для их задания предусмотрен набор мнемонических констант, приведенных в табл. 6.2.

Таблица 6.2. *Мнемонические константы, доступные для аргумента* `Buttons`

Константы Delphi	Константы BCB	Набор кнопок
`mbAbortRetryIgnore`	`MB_ABORTRETRYIGNORE`	Кнопки **Abort** (Стоп), **Retry** (Повтор) и **Ignore** (Пропустить)
`mbOK`	`MB_OK`	Одна кнопка **OK**
`mbOKCancel`	`MB_OKCANCEL`	Кнопки **OK** и **Cancel** (Отмена)
`mbRertryCancel`	`MB_RETRYCANCEL`	Кнопки **Retry** (Повторить) и **Cancel** (Отмена)
`mbYesNo`	`MB_YESNO`	Кнопки **Yes** (Да), **No** (Нет)
`mbYesNoCancel`	`MB_YESNOCANCEL`	Кнопки **Yes** (Да), **No** (Нет) и **Cancel** (Отмена)

Функция `MessageBox` возвращает целочисленное значение из интервала [0,7] в зависимости от нажатой кнопки (табл. 6.3).

Таблица 6.3. *Целочисленное значение, возвращаемое функцией* `MessageBox` *в зависимости от нажатой кнопки*

Число	Константа BCB	Константа Delphi	Нажатая кнопка
1	`IDOK`	`mrOK`	**OK**
2	`IDCANCEL`	`mrCancel`	**Cancel** (Отмена)
3	`IDABORT`	`mrAbort`	**Abort** (Стоп)
4	`IDRETRY`	`mrRetry`	**Retry** (Повторить)
5	`IDIGNORE`	`mrIgnore`	**Ignore** (Пропустить)
6	`IDYES`	`mrYes`	**Yes** (Да)
7	`IDNO`	`mrNo`	**No** (Нет)

Нулевой результат свидетельствует о неудачной попытке создать диалоговое окно, например, по причине исчерпания ресурсов памяти.

Функция `MessageDlg` обладает большей гибкостью и несколько более широкими возможностями:

```
MessageDlg(Msg, DlgType, Buttons, HelpCtx);
```

Первый аргумент — значение типа `AnsiString` в BCB или типа `String` в Delphi — представляет текст отображаемого сообщения. Второй аргумент определяет тип пиктограммы, включаемой в диалоговое окно, и обычно задается с помощью одной из следующих мнемонических констант:

- `mtWarning` — в диалоговом окне появляется восклицательный знак на фоне желтого треугольника, в заголовке окна слово **Warning** (Предупреждение);
- `mtError` — в диалоговом окне появляется стоп-знак (косой крест) в красном кружочке, в заголовке окна слово **Error** (Ошибка);
- `mtInformation` — в диалоговом окне появляется буква `i` на синем фоне, а в заголовке окна слово **Information** (Информация);
- `mtConfirmation` — в диалоговом окне появляется вопросительный знак на зеленом фоне, а в заголовке окна слово **Confirmation** (Подтверждение);
- `mtCustom` — диалоговое окно не содержит пиктограммы, заголовок окна совпадает с именем приложения.

Приложение 6.7 (Delphi)

Аргумент `Buttons` представлен множеством, элементы которого определяют набор кнопок, присутствующих в диалоговом окне. Для языка Object Pascal множество (`set`) является каноническим типом данных, и для задания его значения достаточно перечень допустимых элементов просто заключить в квадратные скобки (рис. 6.7).

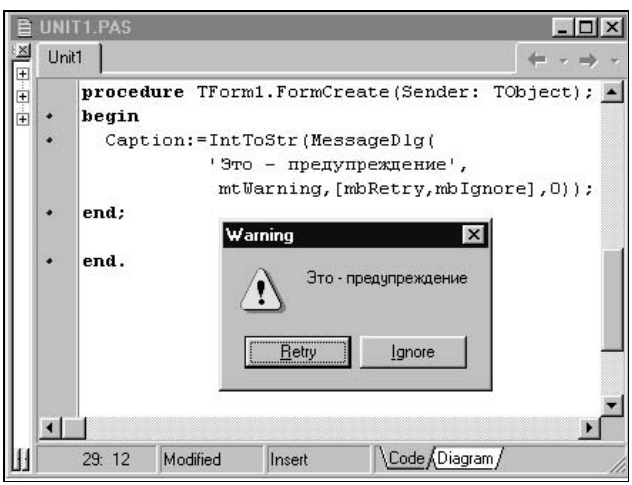


Рис. 6.7. Формат обращения к функции `MessageDlg` в Delphi

Приложение 6.8 (BCB)

Однако для программистов, использующих язык C, данные типа *множество* являются новинкой, и способ задания соответствующих значений выглядит не совсем обычно (рис. 6.8).

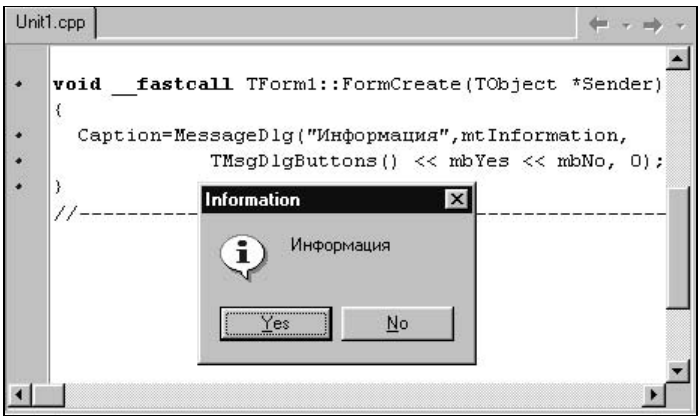


Рис. 6.8. Пример формирования множества в BCB

Последний аргумент `HelpCtx` определяет номер кадра помощи в Help-файле, который должен появиться при нажатии клавиши <F1>.

Вообще говоря, в системах BCB и Delphi существует несколько вариантов функции `MessageDlg`, отличающихся набором параметров. Такая возможность в объектно-ориентированных системах составляет существо термина *полиморфизм*, и ей пользуются довольно часто. Однако чтобы не вносить сумятицу в головы начинающих программистов, мы рекомендуем использовать еще одну диалоговую функцию `MessageDlgPos`, которая в дополнение к четырем аргументам предыдущей функции требует задания координат (x, y), задающих в пикселах положение окна на экране.

Обе функции возвращают целочисленное значение — числовой код, приписанный нажатой кнопке (см. табл. 6.3).

6.3. Универсальные и специализированные диалоги

Довольно большое количество Windows-приложений использует устоявшийся набор диалоговых окон для задания различных параметров системы (например, полей страниц, выводимых на принтер) и выполнения типовых процедур (выбор и запоминание файлов, поиск и замена слов и т. п.). Все они в той или иной мере присутствуют в качестве диалоговых объектов в наших визуальных средах.

В среде Visual Basic таким объектом является многоликий `CommonDialog`, управляя методами которого можно сформировать вызов нужного окна. В системах BCB и Delphi функции стандартных диалогов разбросаны по отдельным объектам — `OpenDialog` и `SaveDialog`, `OpenPictureDioalog` и `SavePictureDialog`, `PrintDialog` и `PrinterSetupDialog`, `FindDialog` и `ReplaceDialog`, `ColorDialog`, `FontDialog`. В связи с тем, что все перечисленные объекты базируются на соответствующих средствах операционной системы, внешний вид диалоговых окон у всех визуальных сред одинаков. Разница наблюдается лишь в названиях методов и свойств.

На стандартной панели компонент в системе Visual Basic объект `CommonDialog`, к сожалению не представлен. Поэтому его придется извлекать из хранилища объектов с помощью команды **Components** (Компоненты) в меню **Project** (Проект). В открывающемся окне на закладке **Controls** (Элементы управления) отыщите строку **Microsoft Common Dialog Control 6.0** и пометьте ее. Если такой строки у вас не окажется, необходимо нажать на кнопку **Browse** (Просмотр) и в каталоге System щелкнуть по файлу Comdlg32.ocx. После этого объект `CommonDialog` появится на панели компонент и будет доступен данному приложению.

6.3.1. Выбор файла в режиме диалога

Несмотря на то, что за диалоговым окном по выбору файла закреплено название `OpenDialog`, оно предназначено только для того, чтобы облегчить пользователю выбор нужного файла. А открывать его в том смысле, как это принято делать в программах, должны операторы соответствующего алгоритмического языка — `OPEN`, `_rtl_open`, `fopen`, `assign` и т. п.

Приложения 6.9, 6.10 (VB)

В системе Visual Basic окно типа `OpenDialog` появляется в результате вызова метода `ShowOpen` для объекта `CommonDialog`. Окно это, по сути дела, объединяет три несколько устаревших системных списка — `DriveListBox`, `DirListBox` и `FileListBox`. По умолчанию на экране возникает системный каталог (рис. 6.9), с которым интересы приложения могут и не совпадать. Поэтому в программе перед вызовом метода `ShowOpen` можно изменить свойство `InitDir`, задав в нем нужное оглавление:

```
CommonDialog1.InitDir = "E:\BXV_BOOK2\"
```

Тогда в диалоговом окне появится заказанный каталог, что может ускорить процесс путешествия по дискам и папкам (рис. 6.10).

Чтобы не загромождать окно списком всех файлов, перед обращением к методу `ShowOpen` целесообразно установить набор нужных шаблонов в свойстве `Filter`:

```
CommonDialog1.Filter = "Текстовые файлы (*.txt)|*.txt|"&_
                       "Модули VB (*.bas)|*.bas"
```

Рис. 6.9. Стандартное окно типа `OpenDialog`

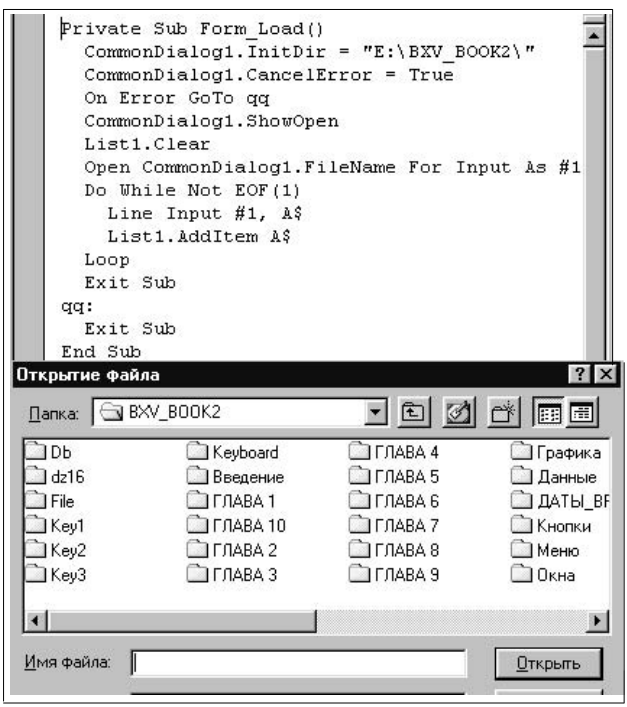


Рис. 6.10. Выбор начального каталога в окне типа `OpenDialog`

Такая установка может быть выполнена и на стадии проектирования в окне **Properties** (Свойства). Если вы задаете несколько шаблонов для фильтрации имен, то среди них можно выделить одно, которое появится первым в поле Тип файлов:

```
CommonDialog1.FilterIndex = 2
```

В данном примере в момент появления диалогового окна будут отображены модули VB с расширением bas (цепочка шаблонов нумеруется от 1). Дополнительным условием отбора может послужить свойство `Flags`, в котором можно задать такие атрибуты файлов, как `ReadOnly`, `Archive` и т. п.:

```
CommonDialog1.Flags = CdlOFNHideReadOnly
```

На самом деле, свойство `Flags` имеет более широкое назначение. Со всеми его возможностями и наборами соответствующих констант вы можете познакомиться, заглянув в файле помощи в раздел **Flags Property** — Свойства флагов (**Open**, **Save As Dialogs**).

Заголовок "Открытие файла" можно заменить более содержательной фразой, задав ее в свойстве `DialogTitle`:

```
CommonDialog1.Title = "Задайте цифровую базу"
```

После нажатия кнопки **Открыть** (Open) и удаления диалогового окна имя файла, выделенного пользователем, можно извлечь из свойства `CommonDialog1.Filename`:

```
OPEN CommonDialog1.Filename FOR INPUT As #1
```

Окно диалога позволяет пользователю отказаться от предложенного выбора — например, он может нажать на кнопку **Отмена** (Cancel). В этом случае в программе должна быть предусмотрена соответствующая ветка:

```
CommonDialog1.CancelError = True
  On Error GoTo qq


qq:
 'Какие-то действия в случае ошибки


  Exit Sub
End Sub
```

Если такая возможность в программе не предусмотрена, то отказ пользователя от выбора файла нажатие клавиши <Esc> или кнопки **Отмена** (Cancel) приведет к возникновению аварийной ситуации с выдачей сообщения "Path/File access error" и прекращению работы приложения.

На рис. 6.11 приводится программа заполнения списка `ListBox1` строками из файла, выбранного пользователем, и результат ее работы — в окне `OpenDialog` выбран текст модуля приложения.

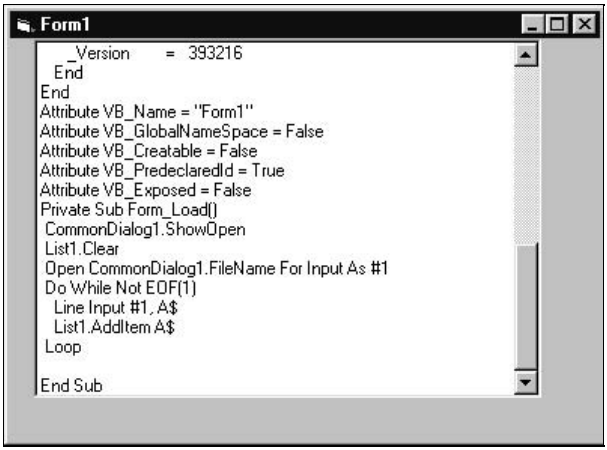


Рис. 6.11. Формирование списка `List1` по содержимому файла

Приложение 6.11 (BCB)

В системах BCB и Delphi выбор файла осуществляется с помощью объекта `OpenDialog`, расположенного в палитре компонентов на вкладке **Dialogs** (Диалоги). Во время работы приложения диалоговое окно по выбору файла появляется в результате вызова метода `Execute`:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  if(OpenDialog1->Execute())
    Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
}
```

Приложение 6.12 (Delphi)

Аналогичный фрагмент в системе Delphi выглядел бы следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if OpenDialog1.Execute then
     Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
```

Для загрузки содержимого выбранного файла в объект, обслуживаемый методом `LoadFromFile`, открывать файл не нужно. Но если мы собираемся обмениваться данными с файлом с помощью обычных операторов чтения (`_rtl_read` или `read`, `_rtl_write` или `write`), то выбранный файл должен открываться стандартным образом. Ряд из упоминавшихся выше свойств

окна `OpenDialog` повторяются и здесь, разве что с небольшим изменением обозначений — `FileName`, `Filter`, `FilterIndex`, `Title`, `InitialDir`. Никаких проблем в связи с отказом от выбора файла в BCB и Delphi не возникает. Роль параметра `Flags` здесь выполняют два десятка подсвойств, скрывающихся за составным свойством `Options`.

6.3.2.Выбор файла для сохранения данных

Диалоговое окно типа `SaveDialog` (рис. 6.12), с помощью которого подбирается имя существующего или вновь создаваемого файла, предназначенного для сохранения результатов приложения, мало чем отличается от окна типа `OpenDialog`. Одна из дополнительных возможностей здесь состоит в том, что свойство `DefaultExt` позволяет при наборе имени файла не задать его расширение, которое добавится на автомате. Имя файла, выбранное или набранное пользователем, по-прежнему располагается в свойстве `FileName`.

Для появления диалогового окна типа `SaveDialog` в системе VB необходимо вызвать метод `ShowSaveAs`:

```
CommonDialog1.ShowSave
```

Приложение 6.13 (BCB)

В системах BCB и Delphi для этой же цели к объекту `SaveDialog` необходимо применить метод `Execute`:

```
if(SaveDialog1->Execute())
  { Memo1->SaveToFile(SaveDialog1->FileName);
```

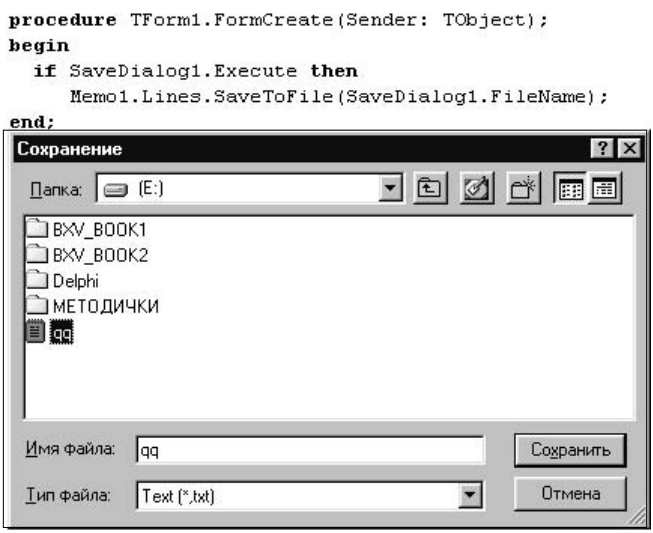


Рис. 6.12. Сохранение содержимого `Memo1` в файле qq.txt

6.3.3. Диалог по поводу выбора шрифта

Стандартное диалоговое окно для выбора шрифта и установки его параметров (см. рис. 2.11) появляется в результате обращения к методу `ShowFont` (в VB) или `Execute` (в BCB или Delphi):

```
CommonDialog1.ShowFont       'Visual Basic
FontDialog1->Execute();       // BCB
FontDialog1.Execute;          {Delphi}
```

Полный перечень свойств, присущих выбранному шрифту, подробно обсуждался в *разд. 4.10*.

6.3.4. Диалог по выбору цвета

Стандартное диалоговое окно для выбора цвета (рис. 6.13) появляется в результате обращения к методу `ShowFont` (в VB) или `Execute` (в BCB или Delphi):

```
CommonDialog1.ShowColor      'Visual Basic
ColorDialog1->Execute();      // BCB
ColorDialog1.Execute;        {Delphi}
```

Приложение 6.14 (Delphi)

Щелкнув мышкой по выбранному квадратику, мы неявно переносим соответствующую числовую константу в свойство `Color`. Если компьютер переключен в режим *truecolor* с 24-разрядным кодом цветности, то 3 младшие байта константы определяют интенсивность каждой из трех цветовых составляющих — R, G, B. Если основная палитра вас не удовлетворяет, то диалоговое окно можно раскрыть в расширенном варианте (рис 6.14). Это можно сделать одним из двух способов — задать компоненту `cdFullOpen=True` в раскрытом свойстве `Option` или нажать кнопку **Определить цвет** в окне основной палитры. В расширенном варианте можно на глаз выбрать цветовой оттенок в большом квадрате или прибегнуть к заданию числовых компонент в одной из двух цветовых палитр — HSB (Hue — Интенсивность, Saturation — Контраст, Brightness — Яркость) или RGB (Red — Красный, Green — Зеленый, Blue — Синий). Числа эти набираются в соответствующих окнах и должны принадлежать диапазону [0, 255].

Цвет, выбранный в режиме диалога, можно использовать для задания цвета пера (`Pen`) или кисти (`Brush`). В первом случае этим цветом окрашиваются различные линии (то, что принято называть цветом переднего плана), во втором — производится заливка замкнутых областей (цвет фона).

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 ColorDialog1.Execute;
end;

end.
```

Рис. 6.13. Основная палитра для выбора цвета

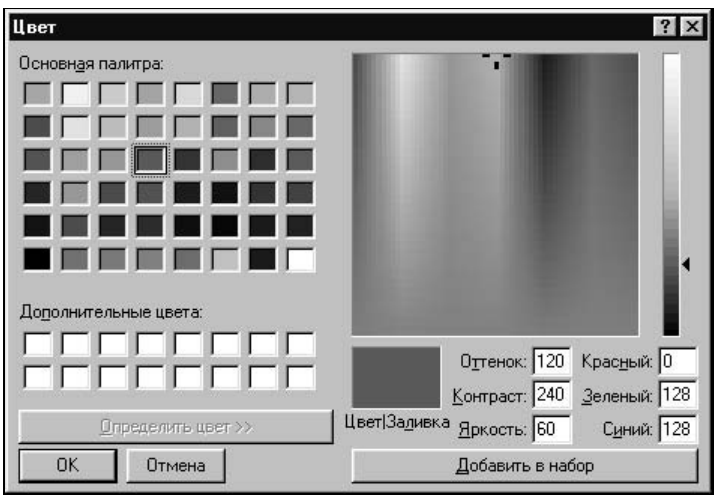


Рис. 6.14. Дополнительная палитра для выбора цвета

6.3.5. Диалоги по установкам и настройкам принтера

Система Visual Basic предлагает единственное диалоговое окно, появляющееся в результате обращения к методу `ShowPrint`. В системах BCB и Delphi возможности по управлению принтером гораздо шире. Здесь предлагается два диалоговых окна — **PrintDialog** (Диалоговое окно печать) и **PrinterSetupDialog** (Диалоговое окно установки принтера), изображенного на рис. 6.16. Каждое

из них активизируется при обращении к методу `Execute` и включает несколько страниц. Вид этих страниц может немного изменяться в зависимости от используемого принтера. На приводимых ниже иллюстрациях отражена специфика струйного принтера Hewlett-Packard 420. Обращаем ваше внимание на то, что упомянутые диалоговые окна не обеспечивают непосредственный вывод текста, подготовленного вашим приложением. Они только настраивают параметры, управляющие драйвером печати. Для непосредственной печати вам придется обратиться к специализированным методам.

Окно начального диалога (рис. 6.15) позволяет установить диапазон выводимых страниц, число копий документа, порядок вывода страниц при печати нескольких копий. В последнем случае имеется возможность напечатать несколько первых, затем несколько вторых, третьих и т. д. страниц. Хотя при этом экземпляры не разобраны по копиям, такой режим обеспечивает более высокую скорость печати. Наличие недоступных окон на рисунке обусловлено заданием соответствующих свойств. Например, можно выбрать режим печати либо всех страниц (`PrintRange = prAllPages`), либо страниц из заданного диапазона (от страницы с номером `FromPage` до страницы с номером `ToPage` в режиме `PrintRange=prPageNums`), либо выделенного фрагмента текста (`PrintRange=prSelection`).

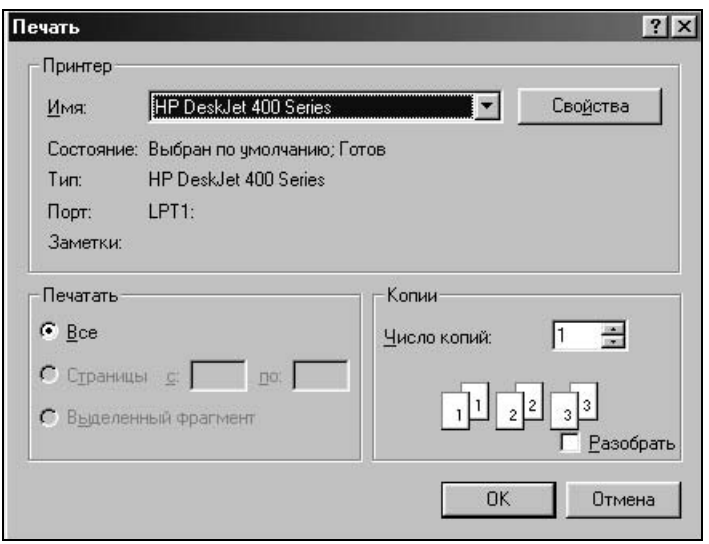


Рис. 6.15. Формат основного окна **PrintDialog**

Свойство `PrintToFile`, принимающее одно из двух логических значений, позволяет направить вывод в файл, который можно перенести на другой компьютер, укомплектованный принтером нужного типа. Щелчок по кнопке **Свойства** (Properties) в основном окне приводит к появлению следующего диалогового окна, где можно выбрать формат страниц и способ их загрузки

в принтер (вручную, на автомате из лотка), расположение текста в пределах страницы (книжная или альбомная ориентация).

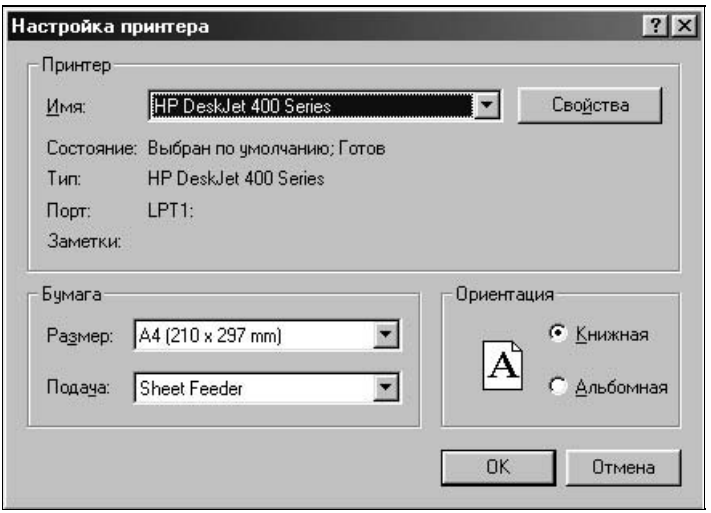


Рис. 6.16. Диалоговое окно первичной настройки принтера

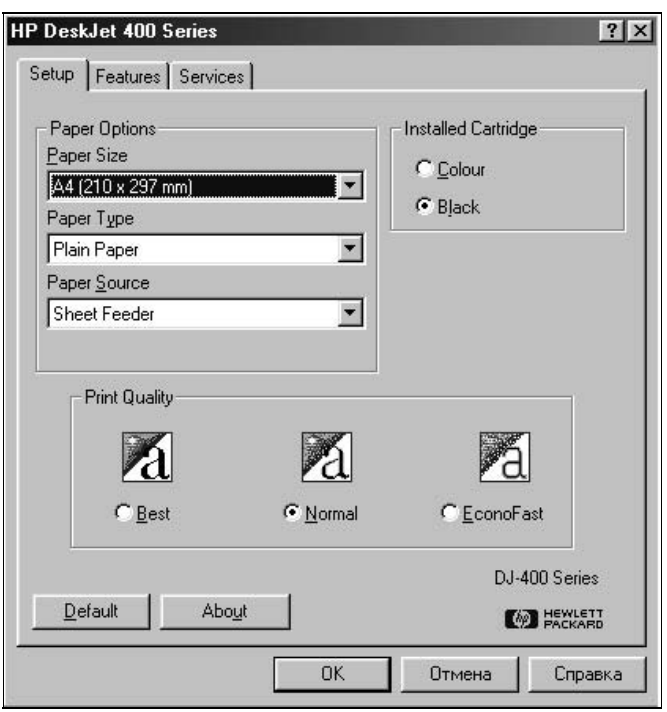


Рис. 6.17. Панель установок для принтера HP DeskJet-420

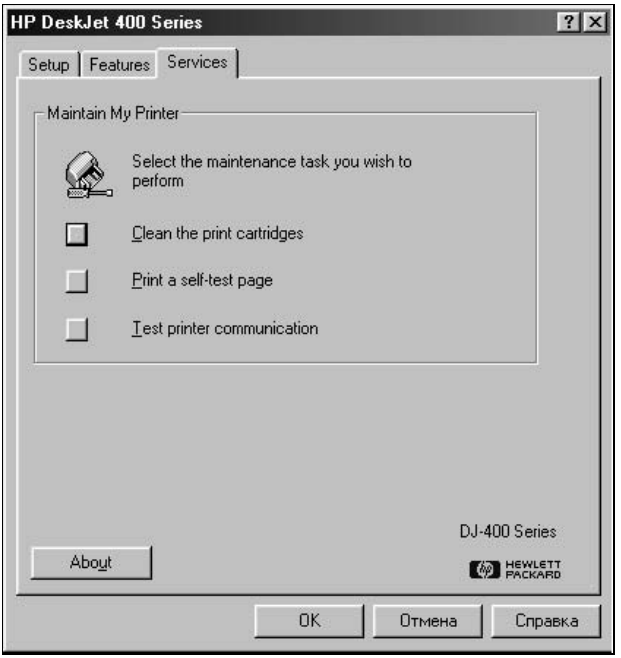


Рис. 6.18. Панель технического обслуживания принтера HP DeskJet-420

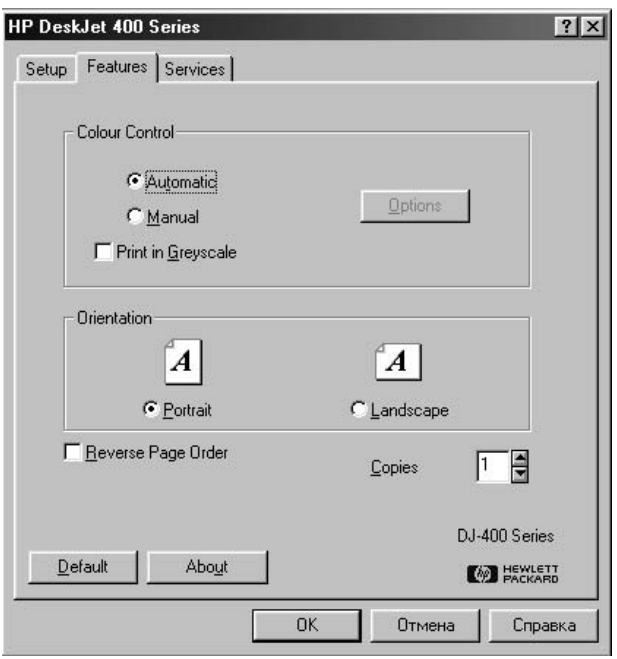


Рис. 6.19. Панель дополнительных возможностей HP DeskJet-420

Щелчок по кнопке **Свойства** (Properties) в окне первичной настройки вызывает появление следующего диалогового окна с тремя вкладками — **Setup** (Настройки), **Features** (Возможности) и **Services** (Обслуживание). Вид каждого из этих окон приведен на рис. 6.17—6.19. Здесь вы можете установить более специфические свойства — наличие цветного или черного картриджа, выбор качества печати (**Best** — Наилучшее, **Normal** — Нормальное, **EconomFast** — Быстрое и Экономичное), определяющего расход чернил, тип бумаги (листы, рулон) и т. п.

На панели технического обслуживания вы можете произвести чистку слегка засохшей печатающей головки, выдать тестовую страницу для проверки качества печати, проверить правильность подключения принтера к компьютеру.

6.3.6. Диалоговые окна для поиска и замены текста

Объектами для поиска (`FindDialog`), поиска и замены (`ReplaceDialog`) располагают только системы BCB и Delphi. Функции их узко специализированы, и работают эти объекты только с текстами, находящимися на поле компонент `RichEdit`. Приведенный ниже текст приложения представляет собой слегка переработанный пример из help-файла системы BCB. Для его функционирования на форме должны быть расположены объекты `RichEdit1`, `Button1` и `FindDialog1`. В составе приложения содержится всего два обработчика событий `OnClick` для кнопки и `OnFind` для диалога поиска.

Приложение 6.15 (BCB)

После запуска приложения на поле редактора `RichEdit1` надо набрать какой-нибудь текст и переместить курсор в начало текста. Затем нужно щелкнуть по кнопке с надписью **Начать поиск** для инициализации диалогового окна `FindDialog1`. На поле ввода открывшегося окна набирается поисковый фрагмент, который может содержать более одного слова. Запуск процедуры поиска начинается после щелчка по кнопке **Найти далее**, в результате которого возникает событие `OnFind`, и управление передается второму обработчику.

Содержимое поля редактора используется как одна длинная строка (`RichEdit1->Text`), в которой поиск начинается либо с самого первого символа (`from=0`), либо с позиции, следующей за ранее найденным выделенным фрагментом, если таковой существует. Переменная `len` хранит количество символов, оставшихся до конца длинной строки, и вместе с начальной позицией `from` определяет область поиска.

При нажатии кнопки **Найти далее** поисковый образ из окна ввода переносится в свойство `FindDialog1->FindText`. Собственно поиск осуществляет метод `FindText`, принадлежащий классу `TRichEdit`. Первый аргумент этого

метода представляет поисковый образ, второй и третий аргументы задают область поиска. Четвертый аргумент представляет собой множество, элементы которого уточняют специфику поиска. Единственный член этого множества, использованный в нашем примере (stMatchCase), предписывает при сравнении текстов игнорировать разницу между большими и малыми буквами. К сожалению, эта возможность не распространяется на русскоязычные тексты. Если поисковый образ обнаружен в заданной зоне, то метод возвращает номер первой позиции найденного фрагмента. В противном случае возвращается -1. Обратите внимание на передачу фокуса RichEdit1->SetFocus(). Пока мы набирали искомое слово и щелкали по кнопке, в центре внимания операционной системы находилось диалоговое окно типа FindDialog. Но для того чтобы подсветить найденный текст, надо не только установить новые значения свойств SelStart и SelLength, но и активизировать объект RichEdit1. На рис. 6.20 приведен момент работы приложения.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  FindDialog1->Execute();
}
//---------------------------------------------------------------
void __fastcall TForm1::FindDialog1Find(TObject *Sender)
{
  int from=0, len, pos;
  if(RichEdit1->SelLength != 0)
    from = RichEdit1->SelStart + RichEdit1->SelLength;
  len = RichEdit1->Text.Length() - from;
  pos = RichEdit1->FindText(FindDialog1->FindText,from,len,
                            TSearchTypes()<< stMatchCase);
  if(pos!=-1)
    { RichEdit1->SetFocus();
      RichEdit1->SelStart=pos;
      RichEdit1->SelLength=FindDialog1->FindText.Length();
    }
}
```

Однако с его помощью можно вести поиск и замену не только в полнофункциональных редакторах текста RichEdit, но и в более простых компонентах типа Memo. Ниже приводится и комментируется пример такого диалога, заимствованный из help-файла. На форме расположены компоненты Memo1, кнопка Button1 (инициатор появления окна диалога) и ReplaceDialog1. Диалоговое окно возникает на экране в результате обращения к методу Execute. Вообще говоря, обработчик события OnClick можно

бы несколько расширить за счет указания позиции диалогового окна на экране. Например, разместив его под окном `Memo1`:

```
ReplaceDialog1->Pos=Point(Left+5,Top+40+Memo1->Height);
```

Но не так уж и трудно вручную перетащить появившееся окно в более удобное место.

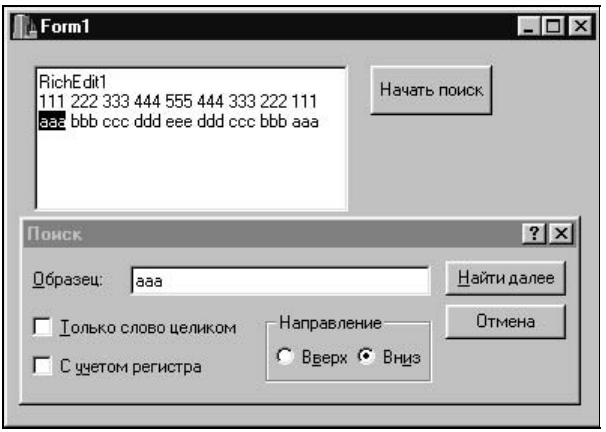


Рис. 6.20. Объект `FindDialog` в работе

Диалоговое окно типа `ReplaceDialog` построено как расширенный вариант окна.

Обработчик события `OnReplace` получит управление после щелчка по одной из кнопок **Заменить** (Replace) или **Заменить все** (Replace All). Правда, до этого нужно разместить в поле **Memo1** какой-либо текст и набрать в полях диалогового окна искомый и заменяющий его фрагмент (рис. 6.22). Обратите внимание на способ приведения типа объекта `Sender`, пославшего сообщение `OnReplace` и имеющего тип `TObject*`, к типу указателя `dlg`, выполненное по классическому стандарту языка C.

Приложение 6.16 (BCB)

Для поиска фрагмента, подлежащего замене, используется метод `Pos`, примененный к содержимому `Memo1` (`Memo1->Lines->Text`). Если искомый фрагмент обнаружен, то производится выделение найденного текста (`Memo1->SelText`) путем формирования начального индекса в строке (`SelStart`) и длины выделенного текста (`SelLength`), совпадающей с длиной поискового образа. После этого выделенный фрагмент заменяется на заданное значение `ReplaceText`. Если поисковый образ не обнаружен (функция `Pos` возвратила нулевое значение), выдается короткий звуковой сигнал.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
  ReplaceDialog1->Execute();
}
//---------------------------------------------------------------------
void __fastcall TForm1::ReplaceDialog1Replace(TObject *Sender)
{
   TReplaceDialog *dlg = (TReplaceDialog *)Sender;
  int pos = Memo1->Lines->Text.Pos(dlg->FindText);
  if (pos > 0)
  {
    Memo1->SelStart = pos - 1;
    Memo1->SelLength = dlg->FindText.Length();
    Memo1->SelText = dlg->ReplaceText;
  }
  else
    MessageBeep(0);
}
```

Для управления деталями поиска и замены можно использовать как кнопки и пометки в диалоговом окне, так и значения подсвойств комплексного свойства `Option`. Среди них параметры, задающие направление поиска (`frDown`), поиск отдельного слова (`frWholeWord`) или произвольного вхождения (`frDisableWholeWord`), указание об игнорировании или учете разницы между большими и малыми буквами (`frMatchCase`, `frDisable MatchCase`), необходимость одноразовой (`frReplace`) или многократной (`frReplaceAll`) замены и др. Каждое из таких свойств принимает одно из двух логических значений, устанавливая одновременно правильное значение противоположного по смыслу свойства.

На рис. 6.21 и 6.22 представлены кадры работы приложения до и после замены фрагмента 'bbb ccc' на тройку символов '666'.

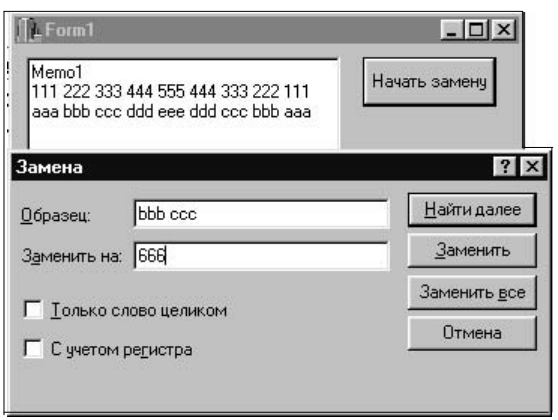


Рис. 6.21. Диалоговое окно перед началом замены

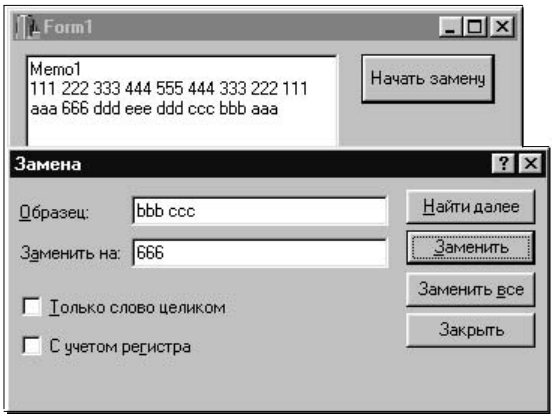


Рис. 6.22. Редактируемый текст после однократной замены

Приложение 6.17 (BCB)

Функция `Pos` может быть с успехом использована не только в обработчике события `OnReplace`, но и в обработчике события `OnFind`. И это дает возможность производить поиск не только в продвинутых редакторах типа `RichEdit`, но и в более простых объектах типа `Memo`. Однако здесь потребуется небольшая хитрость. Дело в том, что при поиске и замене найденный фрагмент заменяется указанным текстом и при продолжении замен поиск в строке `Memo->Lines->Text` можно начинать с начала строки. При просто поиске потребуется пропускать ранее найденные вхождения. Поэтому надо контролировать позицию предыдущего вхождения, и продолжать поиск вслед за очередным выделенным фрагментом. В этом нам поможет глобальная переменная `len`, предназначенная для хранения длины уже обработанной части строки, и локальная переменная `q`, в которую каждый раз переносится необработанный хвост. Ниже приводится текст приложения, которое может искать вхождения как в объектах `RichEdit`, так и в записных книжках типа `Memo`. Для его работы существенно то, что при инициализации поиска в глобальную переменную `len` заносится `0`.

```
int len;       // глобальная переменная
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  len=0;
  FindDialog1->Execute();
}
//---------------------------------------------------------------------
void __fastcall TForm1::FindDialog1Find(TObject *Sender)
{
```

```
  AnsiString q=RichEdit1->Lines->Text.Delete(1,len);
  TFindDialog *dlg = (TFindDialog *)Sender;
  int pos = q.Pos(dlg->FindText);
  if (pos > 0)
  {
    RichEdit1->SelStart = len+pos-1;
    len=RichEdit1->SelStart+dlg->FindText.Length();
    RichEdit1->SelLength = dlg->FindText.Length();
    RichEdit1->SetFocus();
  }
  else MessageBeep(0);
}
```

Приложение 6.18 (Delphi)

Аналогичный вариант для Delphi выглядит следующим образом:

```
var
  Form1: TForm1;
  len : integer; {глобальная переменная}

implementation
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  len:=0;
  FindDialog1.Execute;
end;

procedure TForm1.FindDialog1Find(Sender: TObject);
var
  j,p:integer;
  q1,q:String;
begin
  p:=length(Memo1.Lines.Text);
  q1:=FindDialog1.FindText;
{перенос хвоста строки в q}
  for j:=len+1 to p do
    q:=q+Memo1.Lines.Text[j];
```

```
{поиск вхождения q1 в q}
  p:=Pos(q1,q);
  if p>0 then begin
    Memo1.SelStart:=len+p-1;
    len:=Memo1.SelStart+Length(FindDialog1.FindText);
    Memo1.SelLength:=Length(FindDialog1.FindText);
    Memo1.SetFocus;   end
  else MessageBeep(0);
end;
end.
```

От переменной с именем `pos` здесь пришлось отказаться, т. к., во-первых, Object Pascal в отличие от C++ не делает разницы между большими и малыми буквами, и имя переменной начинало конфликтовать с функцией `Pos`. Вместо нее использована переменная `p`. Во-вторых, вместо удаления обработанного начала длинной строки (функция `Delete` в предыдущем примере) пришлось переносить необработанный хвост посимвольно. В системе Delphi функция `Delete` предназначена для другой цели — она удаляет строку с указанным номером из `Memo`.

6.3.7. Выбор графических файлов

Диалоговые окна типа `OpenPictureDialog` и `SavePictureDialog` выполнены как модификация стандартных окон по выбору файлов типа `OpenDialog` и `SaveDialog`. Две наиболее важные добавки заключаются в том, что пользователю не приходится набирать наиболее распространенные расширения графических файлов, и ему предоставляется возможность предварительного просмотра выбранного изображения (рис. 6.23). В остальном поведение этих окон и их свойства те же, что и описанные в *разд. 6.3.1* и *6.3.2* данной главы.

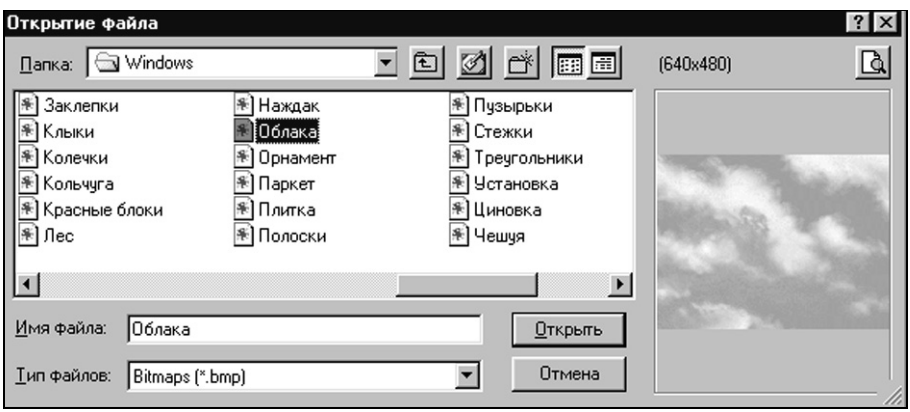
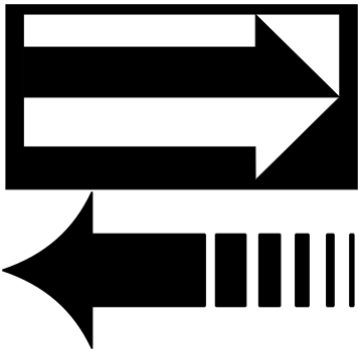


Рис. 6.23. Вид окна `OpenPictureDialog`

Глава 7

Работа с файлами



Модуль управления файлами — один из наиболее устоявшихся компонентов операционных систем, отвечающий за обмен данными между программой и различными внешними устройствами. Он обеспечивает ввод и вывод данных, представленных во внешнем или внутреннем формате, временное хранение результатов расчета на внешних носителях и их последующее использование в повторных сеансах или в других программах, прием и передачу данных по каналам связи, просмотр и редактирование некоторых наборов данных, управление различными периферийными устройствами на логическом и физическом уровнях и т. п. Вообще говоря, это — достаточно сложная, разветвленная и постоянно развивающаяся технология программирования, требующая знаний не только библиотечных функций по управлению файлами в той или иной операционной системе и соответствующей среде программирования, но и умения программировать на языке ассемблера, понимания временной последовательности реально происходящих процессов и многое другое.

Однако в большинстве случаев программистам приходится иметь дело с весьма ограниченным набором операций по запоминанию и извлечению данных, расположенных на жестких или гибких дисках. А эта работа, по существу, сводится к четырем-пяти следующим операциям:

- создание новых или уничтожение существующих файлов;
- выделение ресурсов и приведение файла в состояние готовности к обмену (именно это скрывается за термином *открыть файл*);
- чтение *(ввод из файла)* или запись *(вывод в файл)* очередной порции данных;
- перемещения указателя файла в начало нужной порции данных;
- возврат выделенных ресурсов и завершение неоконченных операций (этому соответствует термин *закрыть файл*).

Несмотря на кажущуюся простоту перечисленных операций, при работе с ними приходится учитывать форматы представления данных и структуры их хранения в оперативной памяти и на диске.

Дисковые файлы организованы в иерархическую структуру данных, вершиной которых является корневой каталог логического диска. Обычно такую структуру называют *деревом* по аналогии с обычным деревом с его корнем, основным стволом, ветвями и листьями. В программировании принято переворачивать обычные деревья с ног на голову. В корневом каталоге хранится информация обо всех элементах первого уровня — подкаталогах и файлах. Каждый из этих элементов имеет имя, которое в среде Windows может быть длинным, содержать русские буквы и даже пробелы. В каталоге запоминаются дисковый адрес, определяющий начало расположения соответствующего элемента, длина набора данных, дата его создания или последнего обновления и атрибуты, регулирующие способ доступа к данным и их видимость. *Корневой каталог* (каталог 0-го уровня) и подкаталоги следующих уровней занимают на диске непрерывное адресное пространство, тогда как файлы данных могут быть разбросаны порциями фиксированной длины — *кластерами.* Кластер образуется несколькими идущими подряд секторами, физическая емкость каждого равна 512 байтам. В зависимости от способа организации данных на винчестере (FAT12, FAT16, FAT32 и др.) и емкости винчестера размер кластеров может быть разным, но для хранения данных файлу всегда предоставляют целое число кластеров. Последовательность кластеров, выделенных файлу и необязательно следующих подряд, регистрируется в таблице распределения дискового пространства — FAT (File Allocation Table, таблица распределения файлов). Каталог ссылается на начальный кластер файла, а в соответствующей позиции FAT находится ссылка на продолжение цепочки данных. Последний кластер с данными, принадлежащими файлу, помечается специальным кодом — признаком конца файла (EOF, End-Of-File).

С точки зрения внутренней организации данных в дисковом файле и, соответственно, методов доступа к ним различают *текстовые* (строковые) и *двоичные* наборы данных.

В текстовых файлах числовая и строковая информация представлена цепочками символов переменной длины, в которых используются те или иные разделители данных (запятые, пробелы, кавычки) в пределах одной строки и управляющие коды, разграничивающие строки. Система управления файлами поддерживает последовательный доступ к строкам (порциям) таких данных. При этом после `k`-той строки читается или записывается `(k+1)`-я строка. Обмен с текстовыми файлами сопровождается прямыми или обратными преобразованиями числовой информации, что требует дополнительного времени и, как правило, сопряжено с увеличением объема занимаемой внешней памяти. Так, например, длинные целые числа в машинном формате занимают 4 байта, тогда как для символьной записи достаточно большого

числа может потребоваться до 11 байт в строке. Плюс разделительные байты, которые должны быть вставлены между числами. Кроме того, системные программы должны анализировать каждый байт, участвующий в последовательном обмене, на предмет обнаружения признаков конца строки или конца файла. Зато символьное представление информации не зависит от типа компьютера и используемой операционной системы. Поэтому символьные данные могут быть переданы другим вычислительным средствам или отправлены по каналам связи абоненту, работающему на такой же или другой *платформе* (под этим термином имеют в виду процессор и/или операционную систему другого типа).

Двоичные файлы хранят информацию в том же виде, в котором она находилась и в оперативной памяти. Поэтому здесь экономится время обмена (никаких преобразований данных не происходит) и не требуется дополнительной памяти на диске. Более того, в двоичных файлах становится возможной операция по перемещению указателя файла на любую порцию данных, что позволяет обмениваться данными не только последовательно, но и в произвольном порядке. Иногда такой доступ называют *прямым* (Direct Access), иногда — *случайным* (Random Access). Для работы с двоичными файлами системы программирования обычно предлагают два варианта обмена. В первом случае содержимое файла рассматривается как набор записей одинакового размера, и такая запись ассоциируется с соответствующим типом данных (`Type-End Type` в VB, `struct` в C++, `record` в Pascal). Во втором случае пользователю разрешен обмен порциями произвольной длины. Но со структурой данных в очередной порции программа должна разбираться сама.

7.1. Файлы в системе Visual Basic

Для создания новых файлов или открывания существующих файлов Visual Basic предлагает единственный оператор `Open`, в котором может присутствовать довольно много обязательных и необязательных параметров:

Open "имя_файла" **For** режим [**Access** доступ] [запрет] **As** [**#**]n [**Len**=k]

Ключевые слова языка здесь выделены жирным шрифтом, необязательные параметры заключены в квадратные скобки. Имя файла может быть указано с полным путем. В качестве режима работы может быть использовано одно из следующих служебных слов:

- `Append` — в существующий текстовый файл предполагается дописывать новые строки;
- `Binary` — открывается существующий или новый файл в двоичном режиме;
- `Input` — открывается существующий текстовый файл для ввода;

- `Output` — открывается текстовый файл для вывода;
- `Random` — открывается новый или существующий файл для прямого обмена записями.

Для файлов, ориентированных на режимы `Binary` и `Random`, дополнительно может быть указан один из трех возможных методов доступа:

- `Access Read` — только для чтения;
- `Access Write` — только для записи;
- `Access Read Write` — для чтения и записи.

Смысл *запрета* заключается в том, чтобы ограничить доступ к содержимому файла другим программам, и включается он с помощью одного из следующих служебных словосочетаний:

- `Shared` — разрешить свободный доступ всем;
- `Lock Read` — запретить чтение;
- `Lock Write` — запретить запись;
- `Lock Read Write` — запретить и чтение, и запись.

Служебное слово `As` предшествует программному номеру файла, который будет использоваться во всех последующих операциях вместо имени файла. Номер файла может задаваться как с предшествующим символом `#`, так и без него, и должен принадлежать интервалу [1, 511]. Закрыв файл с номером `#5`, вы в последующей части программы можете использовать этот же номер для связи с другим файлом.

Указание `Len=k` используется только для файлов, открываемых в режиме `Random`. Оно задает длину `k` записи в байтах, которая не должна превосходить 32 767.

Открытые файлы закрываются оператором `Close` или `Reset`:

```
Close  #4,#2
Close
Reset
```

Оператор `Close` без параметров и `Reset` действуют одинаково — они закрывают все открытые к этому моменту файлы.

Для записи в последовательные файлы, открытые в режимах `For Output` или `For Append`, используются операторы `Print #n` или `Write #n`.

Способ форматирования данных, записываемых в файл по оператору `Print`, ничем не отличается от того, что вы привыкли видеть на экране дисплея:

```
Print #n,[список_вывода]
```

Список вывода может включать любые допустимые выражения, разделяемые пробелами, запятыми, точками с запятыми и управляющими функциями

`Spc(m)`, `Tab[(m)]`. Пробел или точка с запятой размещают очередное значение в строке сразу вслед за предыдущим значением. Запятая включает в выводимую строку несколько пробелов для того чтобы очередное значение попало на границу зоны (выводимая строка условно разбита на зоны по 14 позиций). Функция `Spc(m)` разделяет два соседних значения указанным количеством пробелов. Функция `Tab(m)` с необязательным аргументом продолжает формирование выводимой строки либо с позиции с номером `m`, либо с начала следующей зоны, если аргумент был опущен. Логические значения выводятся словами `True` или `False`. Значения календарных дат попадают в выводимую строку в коротком формате без лидирующих нулей.

Данные, записанные в файл по оператору `Print#`, обычно извлекаются в строковую переменную с помощью оператора `Line Input#`:

```
Line Input #5, A$
```

Основная особенность оператора `Write#` заключается в том, что при формировании выводимой строки он вставляет запятые между числами и окаймляет кавычками строковые величины. Логические значения, выводимые словами, тоже окаймляются спецсимволами — `#TRUE#` или `#FALSE#`. Данные, записанные в последовательный файл по оператору `Write#`, обычно считываются оператором `Input#`.

В файлы прямого доступа, открытые в режиме `For Binary` или `For Random`, данные записываются с помощью оператора `Put`:

```
Put [#]n, [номер_записи], vr
```

В указанное место файла или в текущую запись переносится содержимое переменной `vr`, в качестве которой обычно выступает либо структура, объявленная блоком `Type — End Type`, либо массив типа `Byte`. Для файлов типа `Random` рекомендуется, чтобы длина записи совпадала со значением, объявленным при открытии файла. Но если в качестве источника данных выступает строковая переменная, длина которой может изменяться во время работы программы, то значение `k` (`Len=k`) должно, по крайней мере, на два байта превосходить длину записываемой строки (в эти два байта система запишет длину выводимых данных). Такие же два дополнительные байта пишутся на диск, если очередную запись представляет значение типа `Variant`.

Для файлов, открытых в режиме `Binary`, параметр `Len=k` в операторе `Open` не принимается во внимание и на диск записывается все, что находится в переменной `vr`. Номера записей или байтов в файлах прямого доступа отсчитываются от `1`. Если в операторе вывода номер записи не указан (`Put #5,,v1`), то прямой доступ вырождается в последовательный, и каждая новая запись выводится вслед за предыдущей.

Считывание данных из файла прямого доступа осуществляет оператор `Get`:

```
Get [#]n, [номер_записи], vr
```

Еще один оператор `Seek` позволяет переместить указатель файла на запись или байт с указанным номером:

```
Seek [#]n, номер_записи
```

Им можно воспользоваться в тех случаях, когда время обработки очередной записи достаточно велико, и его полезно совместить с подводом головок диска к нужной дорожке, что выполняется аппаратурой управления винчестером параллельно с работой центрального процессора.

Среди оставшихся функций, используемых чаще других, отметим следующие:

- `EOF(n)` — возвращает значение `True`, если в последней операции считывания данных был достигнут признак конца файла с номером `n`;
- `LOF(n)` — возвращает длину открытого файла в байтах;
- `Input(k,[#]n)` — возвращает строку, содержащую не более `k` символов, считанных из последовательного файла с номером `n`.

7.2. Файлы в среде Borland C++ Builder

Двоичные файлы, уже существующие на диске, в BCB принято открывать с помощью функции `_rtl_open`, которая заменяет устаревшую функцию `_open`:

```
nf = _rtl_open(*filename, flags);
```

Ее первый параметр является указателем типа `char *` на строку с именем открываемого файла или явно содержит эту строку. При явном задании имени файла с полным путем не забывайте вместо символа '\' набирать два символа '\\'. В качестве второго аргумента используют одну из мнемонических констант `O_RDONLY`, `O_WRONLY` или `O_RDWR`, определяющих режим использования файла (для чтения, для записи, для чтения и записи).

Функция `_rtl_open` возвращает целое число `nf`, заменяющее имя файла в последующих операциях обмена. Иногда его называют *дескриптором файла* (в англоязычной документации — file handle), но мы предпочитаем более простое обозначение — *номер файла*. Ошибка при открывании файла может быть вызвана указанием несуществующего пути, попыткой открыть файл, используемый в данное время другой программой, или неверным заданием режима. В этом случае возвращаемое значение равно `-1`.

Для создания нового двоичного файла или использования существующего с уничтожением содержащихся в нем данных используется функция `_rtl_creat`:

```
nf = _rtl_creat(*filename, attrib);
```

Ее второй аргумент определяет такие атрибуты файла, как *только для чтения* (константа `_A_RDONLY`), *скрытый* (константа `_A_HIDDEN`), *системный* (кон-

станта _A_SYSTEM), *нормальный* (константа _A_NORMAL), *подлежащий архивации* (константа _A_ARCH). При необходимости присвоить файлу несколько атрибутов одновременно, следует написать логическую сумму таких констант, например — _A_RDONLY OR _A_HIDDEN. Указанные мнемонические константы находятся в том же заголовочном файле io.h, где находятся и прототипы функций _rtl_open, _rtl_creat, _rtl_close, _rtl_write и _rtl_read. Создаваемый таким образом файл доступен и для записи, и для чтения.

Запись в двоичный файл осуществляется с помощью функции _rtl_write, которая заменила устаревшую функцию _write:

```
int _rtl_write(int nf, void *buf, unsigned len);
```

Первым ее аргументом является номер файла. Информация, которая должна быть записана на диск, готовится в массиве оперативной памяти, на который "смотрит" указатель buf, а длина этого массива в байтах задается третьим параметром len. Значение, возвращаемое функцией, определяет количество успешно записанных байтов. Как правило, оно должно совпадать с заказанной длиной len. Меньшее значение свидетельствует об ошибке, скорее всего — о переполнении диска.

Функция извлечения данных из двоичного файла имеет очень похожий формат и аргументы у нее те же самые:

```
int _rtl_read(int nf, void *buf, unsigned len);
```

Она заменила устаревшую функцию _read. Если возвращаемое значение меньше, чем запрашиваемое количество байт (len), то программа прочитала последнюю порцию данных. И после этого функция возвратит только нулевое значение.

Наряду с новыми функциями, описанными выше, в заголовочном файле io.h находятся заголовки ранее эксплуатировавшихся функций — open, creat, creatnew, creattmp, с помощью которых можно открыть или создать файлы в текстовом или двоичном режиме:

```
nf = open(*filename, access[,mode]);
```

Первый аргумент по-прежнему определяет имя открываемого файла. Второй параметр определяет способ доступа к данным и задается как непротиворечивая логическая комбинация из следующих констант:

- O_RDONLY — файл открывается только для чтения;
- O_WRONLY — файл открывается только для записи;
- O_RDWR — файл открывается для записи и чтения;
- O_APPEND — открывается существующий файл для пополнения имеющегося в нем набора данных (при открытии такого файла его указатель устанавливается вслед за последней существующей записью);
- O_CREAT — используется для создания нового или открытия существующего файла;

- O_TRUNC — позволяет удалить данные в существующем файле;
- O_BINARY — открываемый файл будет работать в двоичном режиме;
- O_TEXT — открываемый файл будет работать в текстовом режиме.

Необязательный третий аргумент может принимать одно из трех следующих значений — S_IREAD, S_IWRITE или S_IREAD|S_IWRITE. Он определяет направление обмена. Вообще говоря, текстовый или двоичный файл может быть открыт и без указания режима во втором параметре. Дело в том, что в файле io.h объявлена глобальная переменная _fmode, значением которой по умолчанию является константа O_TEXT. Она и оказывает влияние на тип открываемого файла, если признак O_TEXT или O_BINARY опущен при открытии или создании файла.

Функция creat создает новый или открывает существующий файл. В отличие от этого функция creatnew сообщает об ошибке при попытке создать уже существующий файл. Функция creattmp позволяет создать временный файл в указанном каталоге. Для временного файла система сама формирует уникальное имя и удаляет файл после его закрытия. Все три функции создают текстовый или двоичный файл в зависимости от значения глобальной переменной _fmode. Функции open и creat возвращают целочисленный номер открытого файла или выдают значение -1, если попытка открыть файл не удалась.

Форматы вызова функций write и read полностью совпадают с обращениями к функциям _rtl_write и _rtl_read.

Для перемещения указателя двоичного файла с номером nf в нужную позицию предназначена функция lseek:

```
k=long lseek(nf, offset, fromwhere);
```

Позиция (номер байта), в которую следует переместить указатель файла, определяется как заданное целочисленное смещение offset относительно одной из трех точек:

- относительно текущей позиции указателя, если третий аргумент равен SEEK_CUR. В этом случае смещение может быть как положительным, так и отрицательным;
- относительно начала файла, если третий аргумент равен SEEK_SET. В этом случае смещение должно быть положительным;
- относительно конца файла, если третий аргумент равен SEEK_END. В этом случае смещение должно быть отрицательным.

Система BCB предлагает довольно большой набор функций по управлению файлами и каталогами, реализуемых в конечном итоге через процедуры Windows. Их перечень приведен в табл. 7.1 без указания формата их вызова. Как правило, имена файлов и каталогов среди параметров этих процедур имеют тип AnsiString. При желании вы можете сами заглянуть в файл

помощи и получить нужную справку. Табл. 7.1 лишь дает представление о функциональных возможностях системных процедур.

Таблица 7.1. *Назначение системных процедур, предназначенных для работы с файлами и каталогами*

Функция	Назначение
`CreateDir`	Создание нового каталога
`DeleteFile`	Удаление файла с диска
`DirectoryExists`	Определение, существует ли указанный каталог
`DiskFree`	Определение размера свободного места на указанном диске
`DiskSize`	Определение полного объема указанного диска
`FileAge`	Определение даты и времени создания или последнего обновления указанного файла
`FileClose`	Закрытие указанного файла
`FileDateToDateTime`	Преобразование даты и времени создания файла в значение типа `TDateTime`
`FileExists`	Определение, существует ли указанный файл
`FileGetAttr`	Опрос атрибутов указанного файла
`FileGetDate`	Опрос даты и времени создания файла в формате MS-DOS
`FileOpen`	Открывание файла с указанным режимом доступа
`FileRead`	Чтение указанного числа байтов из файла
`FileSearch`	Поиск полного пути к указанному файлу
`FileSeek`	Перемещение указателя в открытом файле
`FileSetAttr`	Установка атрибутов файла
`FileSetDate`	Изменение даты и времени создания файла в формате MS-DOS
`FileWrite`	Запись содержимого буфера в файл, начиная с текущей позиции
`FindClose`	Освобождение памяти, занятой процедурой `FindFirst`
`FindFirst`	Поиск первого файла с заданными атрибутами в указанном каталоге
`FindNext`	Поиск следующего файла с заданными атрибутами в указанном каталоге
`ForceDirectories`	Создание всех каталогов, соответствующих заданному пути

Таблица 7.1 *(окончание)*

Функция	Назначение
GetCurrentDir	Опрос имени текущего каталога
RemoveDir	Удаление существующего пустого каталога
RenameFile	Изменение имени файла
SetCurrentDir	Изменение имени текущего каталога

7.3. Файлы в Delphi

В Pascal традиционно используются файлы трех типов — текстовые, типизированные и двоичные. В отличие от Basic и C, где с именами открываемых файлов связываются программные числовые номера, в Pscal их заменяют переменные файлового типа. Эти переменные объявляются в разделе описания переменных:

```
type
 PhoneRec = record
    Name:string[20];
    Phone:string [15];
    Address:string [40];
    BirthDay:TDateTime;
  end;

var
  ft:TextFile;          {файловая переменная для текстового файла}
  fb:File;              {файловая переменная для двоичного файла}
  fr:File of PhoneRec; {файловая переменная для типизированного файла}
```

После этого в разделе исполняемых операторов имена файловых переменных с помощью процедуры `AssignFile` связываются с именами дисковых файлов соответствующего типа:

```
AssignFile(ft,'abc1.txt');
AssignFile(fb,'abc2.bin');
AssignFile(fr,'abc3.rec');
```

И только теперь можно приступить к процедуре открывания файлов. Не в обиду будет сказано, но авторы Паскаля здесь умудрились выбиться из общепринятой колеи и вместо четкого `Open` взяли на вооружение гораздо менее привычные служебные слова `Reset` (для файлов ввода), `Rewrite` (для файлов вывода) и `Append` (для пополнения текстовых файлов):

```
Reset(ft);        Rewrite(ft);       Append(ft);
```

```
Reset(fb[,len]);  Rewrite(fb[,len]);
Reset(fr);        Rewrite(fr);
```

Второй необязательный аргумент `len`, используемый при открытии двоичных файлов, устанавливает в байтах размер порции данных, которые принимают участие в процедурах обмена. По умолчанию используются блоки размером по 128 байт, но эта цифра, скорее, дань традиции Pascal. С точки зрения оптимизации скорости обмена длина блока должна быть равной длине кластера. Однако и это не совсем точно, т. к. на время передачи данных существенно большее влияние оказывает аппаратура управления винчестером, располагающая своим собственным кэшем.

Открытые файлы любого типа закрываются с помощью процедуры `CloseFile`:

```
CloseFile(ft);     CloseFile(fb);      CloseFile(fr);
```

Для обмена с текстовыми файлами используются обычные операторы `read/readln` и `write/writeln`, первым аргументом которых являются имена переменных файлового типа:

```
write(ft, x1:10,x2:6);
writeln(ft,' ',x3);
.......................
readln(ft,y1,y2,y3);
```

Чтобы очередная порция данных, записываемая в текстовый файл, была завершена признаком конца строки (символ "Возврат каретки" или пара символов "Возврат каретки" и "Перевод строки"), вывод последней (или единственной) части строки должен быть завершен оператором `writeln`. Кроме этого, пользователь должен позаботиться о том, чтобы числовые данные, направляемые на диск, разделялись, как минимум, одним пробелом. В цикле считывания строк из текстового файла программа может определить момент исчерпания данных по значению логической функции `Eof(ft)`. Она принимает значение `True`, если между текущей позицией указателя и концом файла ничего кроме пробелов нет.

Приложение 7.1 (Delphi)

Приводимое ниже приложение осуществляет побайтовое копирование содержимого одного текстового файла в другой. Первый текстовый файл должен существовать, т. к. его имя выбирается пользователем в стандартном диалоговом окне `Opendialog1`. Второй файл может быть создан — такая возможность предусмотрена в диалоговом окне `SaveDialog1`. Процедура переписи включена в состав обработчика события `OnActivate`.

```
unit Unit1;
interface
```

```
uses
  Windows,Messages,SysUtils,Variants,Classes,Graphics,Controls,Forms,
  Dialogs;
type
  TForm1 = class(TForm)
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.FormActivate(Sender: TObject);
var
  ft1, ft2: TextFile;
  ch: Char;
begin
  if OpenDialog1.Execute then begin
    AssignFile(ft1,OpenDialog1.Filename);
    Reset(ft1);
    if SaveDialog1.Execute then
    begin
      AssignFile(ft2,SaveDialog1.Filename);
      Rewrite(ft2);
      while not Eof(ft1) do begin
        Read(ft1,ch);
        Write(ft2,ch);
      end;
      CloseFile(ft2);
    end;
    CloseFile(ft1);
  end;
end;
end.
```

В обмене с типизированными файлами используются только операторы `read` и `write`. При этом источником или получателем данных должна быть единственная переменная типа `record`, имеющая такую же структуру, которая была использована при объявлении файла.

Приложение 7.2 (Delphi)

В приведенном ниже приложении на форме находятся две кнопки — `Button1` с надписью **Записать в файл** и `Button2` с надписью **Прочитать из файла**. Щелчок по первой кнопке приводит к созданию типизированного файла с именем Poly.dat, записи которого представляют координаты вещественных точек. В обработчике этого же события координаты точки `P1` записываются в файл. Щелчок по второй кнопке извлекает координаты первой точки из файла в запись `P2`.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
type
  DPoint=record
    x:double;
    y:double
  end;
var
  Form1: TForm1;
  P1,P2:DPoint;
  fp:file of DPoint;
```

```
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
  AssignFile(fp,'Poly.dat');
  Rewrite(fp);
  P1.x:=1;
  P1.y:=2;
  Write(fp,P1);
  CloseFile(fp);
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  AssignFile(fp,'Poly.dat');
  Reset(fp);
  Read(fp,P2);
  CloseFile(fp);
end;
end.
```

При обмене с типизированным файлом допустим переход к любой существующей записи с заданным номером (записи в файле нумеруются от `0`). Установка указателя файла на запись с номером `nr` осуществляется процедурой `Seek`:

```
Seek(fr,nr);
```

С двоичными файлами обмен производится блоками, размер которых установлен либо по умолчанию (128 байт), либо был объявлен в момент открывания файла. Для этой цели используются специальные операторы `BlockWrite` и `BlockRead`:

```
BlockWrite(fb, buf, n_block,v1);
BlockRead(fb, buf, n_block,v1);
```

Здесь `buf` — массив (обычно типа `byte`), в котором либо находятся данные, подготовленные для записи на диск, либо предназначенный для хранения считываемых с диска данных. Параметр `n_block` задает количество блоков, участвующих в обмене. В переменную `v1` заносится фактическое количество записанных или считанных блоков. Несовпадение между `n_block` и значением `v1` при чтении обычно связано с тем, что реальная длина файла не всегда кратна размеру блока. При выводе такое событие, как правило, свидетельствует об исчерпании дисковой памяти. Подобно записям в типизированных файлах блоки данных в двоичном файле тоже нумеруются от `0`.

И для перехода к блоку с нужным номером необходимо воспользоваться процедурой `Seek`.

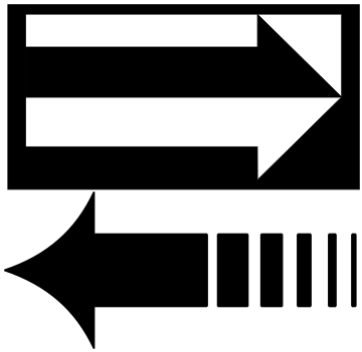
Перечень дополнительных процедур и функций по управлению файлами в системе Delphi приведен в табл. 7.2.

Таблица 7.2. *Дополнительные процедуры и функции для работы с файлами, принятые в среде Delphi*

Процедура/функция	Назначение
`ChDir`	Назначает новый текущий каталог на текущем диске
`Eoln`	Возвращает значение `True`, если в строке текстового файла обнаружен признак конца строки
`Erase`	Уничтожает указанный файл
`FilePos`	Возвращает номер текущей записи или текущего блока в типизированном или двоичном файле
`FileSize`	Возвращает количество записей или блоков в типизированном или двоичном файле
`Flush`	Выталкивает содержимое буфера обмена в выводной текстовый файл
`GetDir`	Возвращает имя текущего каталога на указанном логическом диске
`IOResult`	Возвращает числовой код, свидетельствующий о результате завершения предыдущей операции с дисковым файлом. При нормальном завершении `IOResult=0`
`MkDir`	Создает новый подкаталог
`Rename`	Переименовывает указанный файл
`RmDir`	Удаляет пустой подкаталог
`SeekEoln`	Возвращает значение `True`, если между текущей позицией указателя и концом строки в текстовом файле нет значащих символов кроме пробела
`SetTextBuf`	Изменяет длину встроенного буфера у текстового файла
`Truncate`	Удаляет данные от текущей позиции указателя до конца типизированного или двоичного файла

Глава 8

Машинная графика



Результаты работы программы, дополненные соответствующими графическими изображениями, производят гораздо большее впечатление, чем сухие колонки цифр. Размышляя о машинной графике, мы собираемся обсудить три ее аспекта. Во-первых, речь может идти об изображениях, хранящихся на дисках и полученных из различных источников, — слайды, сформированные с помощью средств цифровой фото- и видеотехники, результаты работы многочисленных графических пакетов, картинки из Интернета и т. п. Такие рисунки можно размещать на поверхности некоторых объектов нашей программы, производить масштабирование изображений и выделять интересующие нас фрагменты. Второй аспект машинной графики связан с непосредственным творчеством пользователя, которому предоставляется набор инструментов для построения и раскраски различных геометрических фигур. Его характеризует диалоговый режим общения с пользователем и обработка различных событий, связанных, в основном, с мышью. Наконец, к третьей категории средств машинной графики относятся автоматические построения, выполняемые программным путем с использованием процедур соответствующей библиотеки.

Графические возможности и методы базируются на функциях графического ядра операционной системы Windows — *Graphic Device Interface (GDI)*. Для обеспечения независимости от устройства отображения графической информации (дисплей, принтер, плоттер) библиотеки процедур GDI используют так называемый контекст устройства — *Windows device context (DC)*. По сути дела, контекст конкретного устройства представляет собой массив в оперативной памяти, в котором хранятся параметры, управляющие процессом вывода, — размеры рабочего поля, разрешающая возможность, характеристики пишущего узла (толщина и цвет пера, тип линии), режим взаимодействия выводимых пикселов с ранее построенной частью изображения и многое другое. Обилие этих деталей и учет специфики Windows затрудняют прямое использование функций GDI.

Основным преимуществом графических объектов визуальных сред является то, что они скрывают от пользователя многие нюансы взаимодействия с процедурами GDI — запрос и освобождение контекста устройства, создание и уничтожение данных типа `TPen` и `TBrush`, специфику обработки графических файлов разного формата (bmp, ico, wmf, emf) и т. п.

8.1. Рисуем? Где? Чем?

Условимся называть *графическими* те объекты визуальных сред, на поверхности которых можно рисовать, используя различные графические методы, или размещать изображения, хранящиеся в наиболее распространенных графических форматах. В системе Visual Basic к таким объектам относятся компоненты `Form`, `PictureBox`, `Image` и `Shape`. Графические объекты в системах BCB и Delphi снабжены специальным свойством — *канвой* (`Canvas`). Таких объектов здесь десятка полтора — `Animate`, `Bitmap`, `ComboBox`, `ControlBar`, `DrawGrid`, `Form`, `Image`, `ListBox`, `ListView`, `PaintBox`, `Printer`, `StringGrid`, `ToolBar` и др.

Несмотря на отсутствие единства в терминологии, используемой в разных визуальных средах, мы будем оперировать такими понятиями, как *канва* (поверхность, на которой рисуют), *перо* (инструмент для воспроизведения контуров геометрических фигур и символов шрифтов) и *кисть* (инструмент для создания фона канвы). Эти термины близки к объектам GDI, интуитивно понятны и являются стандартными свойствами для систем BCB/Delphi.

У большинства графических объектов физическую основу канвы составляет часть экрана монитора, на которой нарисован объект, т. е. соответствующий участок видеопамяти. Если этот участок по той или иной причине будет перекрыт, то изображение, находившееся на канве, как правило, может пропасть. В большинстве случаев системная программа перерисовки объекта не восстанавливает содержимое канвы, это должна делать программа пользователя в ответ на событие `OnPaint`.

Однако среди графических объектов есть и такие, у которых содержимое канвы дублируется в специально организованном участке оперативной памяти под названием *bitmap* — битовая плоскость. Изображение канвы там представлено последовательностью двоичных разрядов, хранящих код цветности каждой точки. Кроме информации о цвете здесь же хранятся сведения о размерах изображения — число пикселов в строке и количество строк. Одним из таких графических объектов является компонент `Image`. При его перерисовке, выполняемой служебными программами, содержимое канвы восстанавливается автоматически. Программа пользователя может рисовать на объекте типа `TBitmap`, которому выделяется аналогичный участок в оперативной памяти, а потом копировать содержимое этого невидимого объекта на канву того или иного элемента формы. К такому копированию прибе-

гают в тех случаях, когда время рисования сложного графического изображения достаточно велико, и его повторная перерисовка не целесообразна.

Фон канвы создается кистью (`Brush`), для которой должны быть заданы цвет и способ заполнения (вспомните процедуры `setfillstyle` и `setfillpattern` в библиотеке BGI). Существует два способа заполнения замкнутой области, каковой является и поверхность канвы. В первом случае вдоль канвы дискретно перемещается регулярный шаблон фиксированного размера (обычно — 8×8), представляющий собой двоичную матрицу, единичные пикселы которой окрашивают точки канвы в заданный цвет. После заполнения одной полосы шаблон повторяет движение по следующей полосе и т. д. Каждая из визуальных сред предоставляет пользователю набор системных шаблонов, с помощью которых организуются наиболее распространенные варианты заливки или штриховки. Обозначения констант, с помощью которых устанавливается текущий стиль кисти, приведены в табл. 8.1.

Таблица 8.1. *Описание констант, устанавливающих текущий стиль кисти*

Константа VB	Константа BCB/Delphi	Способ заполнения канвы
`VbFSSolid`	`bsSolid`	Сплошная заливка (шаблон из единиц)
`VbFSTransparent`	`bsClear`	Отсутствие заливки (шаблон из нулей)
`VbFSHorizontalLine`	`bsHorizontal`	Горизонтальная штриховка
`VbVerticalLine`	`bsVertical`	Вертикальная штриховка
`VbUpwardDiagonal`	`bsFDiagonal`	Штриховка под углом 135°
`VbDownwardDiagonal`	`bsDiagonal`	Штриховка под углом 45°
`VbCross`	`bsCross`	Двойная штриховка под углами 0° и 90°
`VbDiagonalCross`	`bsDiagCross`	Двойная штриховка под углами 45° и 135°

Для изменения стиля кисти необходимо изменить свойство `FillStyle` у объекта в VB или свойство `Brush->Style/Brush.Style` у канвы соответствующего объекта в BCB или Delphi:

```
Picture1.FillStyle = VbVerticalLine              ' VB
PaintBox1->Canvas->Brush->Style= bsVertical;     // BCB
PaintBox1.Brush.Style:= bsVertical;              { Delphi }
```

Приложение 8.1 (VB)

На рис. 8.1 приведены образцы заполнения квадратной области перечисленными выше способами.

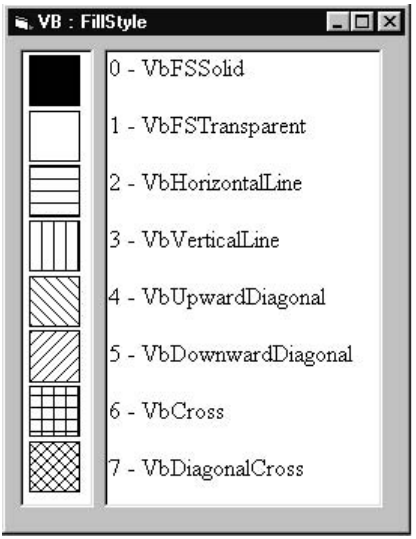


Рис. 8.1. Способы заливки и штриховки

Под цветом кисти (цветом фона) понимают цвет пикселов рабочей поверхности канвы в момент ее создания или стирания. По умолчанию цвет кисти совпадает с цветом стандартного фона формы. Цвет единичных пикселов шаблона в системе Visual Basic задается с помощью свойства `FillColor`:

```
Picture1.FillColor = код_цвета
```

Для формирования числового значения кода цвета (величины типа `long`) можно воспользоваться одной из двух функций — `RGB` или `QBColor`:

```
Picture1.FillColor = RGB(red,green,blue)
Picture1.FillColor = QBColor(n)
```

Аргументами функции `RGB` являются интенсивности красного, зеленого и синего цветов, значения которых должны принадлежать диапазону `[0, 255]`. Функция `QBColor` сохраняет преемственность со способами задания цвета в системе QBasic, где использовалось всего 16 цветовых оттенков. Способ их нумерации приведен в табл. 8.2.

Если в системе VB цвет кисти явно не задан, то он принимается равным значению свойства `BackColor`, установленному для объекта, на поверхности которого происходит рисование. По умолчанию цвет заливки — черный.

Таблица 8.2. *Оттенок цвета в зависимости от числового значения параметра `n` (среда Visual Basic)*

n	Цвет	n	Цвет
0	Черный	8	Серый
1	Синий	9	Голубой

Таблица 8.2 *(окончание)*

n	Цвет	n	Цвет
2	Зеленый	10	Светло-зеленый
3	Бирюзовый	11	Светло-бирюзовый
4	Красный	12	Светло-красный
5	Малиновый	13	Светло-малиновый
6	Коричневый	14	Желтый
7	Светло-серый	15	Ярко-белый

В системах BCB и Delphi для задания цвета кисти необходимо установить значение свойства `Brush->Color` у канвы соответствующего объекта. В качестве такого значения можно либо выбрать одну из системных констант (табл.8.3), либо сформировать целочисленное значение из предполагаемых интенсивностей цветовых компонент. В системе BCB такое значение должно быть преобразовано к типу `TColor`:

```
int col[16]=
  { 0x000000, // черный
    0x800000, // синий
    0x00A800, // зеленый
    0x0000FF, // красный
    0x800080, // фиолетовый
    0x4080C0, // коричневый
    0x008080, // темно-оливковый
    0x00FF00, // светло-зеленый
    0xFFFF00, // бирюзовый
    0x4477E4, // оранжевый
    0x00FFFF, // желтый
    0xFFFFFF  // белый
  };
.......................................................
  Form1->PaintBox1->Canvas->Brush->Color=(TColor)col[j];
```

Таблица 8.3. *Оттенок цвета в зависимости от значения символьной константы (среда BCB)*

Имя константы	Цвет	Имя константы	Цвет
`clAqua`	Светло-бирюзовый	`clNavy`	Темно-синий
`clBlack`	Черный	`clOlive`	Оливковый

Таблица 8.3 *(окончание)*

Имя константы	Цвет	Имя константы	Цвет
clBlue	Синий	clPurple	Темно-фиолетовый
clFuchsia	Светло-фиолетовый	clRed	Красный
clGray	Светло-серый	clSilver	Темно-серый
clLime	Лимонный	clTeal	Темно-бирюзовый
clMaaron	Темно-красный	clWhite	Белый

Приложение 8.2 (Delphi)

В системе Delphi цвет кисти тоже можно сформировать в виде целочисленной константы, но преобразование к другому типу данных здесь не требуется:

```
var
  R,G,B:integer;
begin
  R:=Random(255);
  G:=Random(255);
  B:=Random(255);
  Canvas.Brush.Color:=(B*256+G)*256+R;
```

Кроме перечисленных выше стилей штриховки пользователь может приписать кисти узор, представленный объектом типа Bitmap. Им может быть графический рисунок, хранящийся в одном из допустимых форматов в дисковом файле, или узор, сформированный программным путем. Узор заменяет одновременную установку обеих характеристик (цвет и стиль) и "наносится на кисть" с помощью следующего оператора:

```
имя_объекта->Canvas->Brush->Bitmap=объект_типа_Bitmap;
```

Если размер наносимого узора превышает 8×8 пикселов, то в качестве шаблона кисти используется соответствующий фрагмент из левого верхнего угла узора. Для возврата к системным шаблонам необходимо сбросить указатель Bitmap:

```
имя_объекта->Canvas->Brush->Bitmap=NULL;
```

Для рисования контуров различных геометрических фигур и отдельных точек на канве используется перо (в BCB и Delphi свойство типа TPen). Перо обладает четырьмя основными характеристиками — шириной, задающей толщину рисуемых линий, цветом, стилем воспроизведения линий и режимом взаимодействия цвета пера с цветом пикселов на канве.

В системе Visual Basic цвет воспроизводимого контура либо задается явно среди параметров вызванного метода, либо устанавливается по значению

свойства `ForeColor`, приписанному объекту, на поверхности которого происходит рисование.

В системах BCB и Delphi цвет пера устанавливается присвоением нужного значения свойству `Pen->Color/Pen.Color` у канвы нужного объекта. Не забывайте, что в системе BCB целочисленное значение цвета должно быть преобразовано к типу `TColor`. По умолчанию цвет пера — черный.

Ширина пера — свойство `Pen->Width/Pen.Width` (BCB/Delphi) или `DrawWidth` (VB), — т. е. толщина воспроизводимой линии по умолчанию равна 1 пикселу. Новое значение, которое вы собираетесь присвоить этой характеристике, тоже должно быть задано в пикселах (мы специально обращаем ваше внимание на это потому, что стандартной единицей измерения линейных величин в VB являются твипы). Следует сразу же заметить, что на перья с шириной больше 1 стиль пера никакого влияния не оказывает. Все толстые контуры рисуются сплошной линией, независимо от установленного стиля.

Стиль пера — свойство `Pen->Style/Pen.Style` (BCB/Delphi) или `DrawStyle` (VB) — задает битовый шаблон, в соответствии с которым одни пикселы линии рисуются, а другие не рисуются. Если битовый шаблон состоит из сплошных единиц, то рисуется каждый пиксел контура, т. е. он воспроизводится сплошной линией. Если в шаблоне встречаются нулевые биты, то соответствующий участок контура не воспроизводится. Обычно длина шаблона ограничена (в MS-DOS использовались 16-разрядные шаблоны, задаваемые коротким целочисленным значением, в Windows длина шаблона равна 32 битам). Поэтому после воспроизведения очередных 32 точек линии шаблон начинает периодически повторяться. Константы, управляющие стилем пера и приведенные в табл. 8.4, имеют смысл только при ширине пера в 1 пиксел.

Таблица 8.4. *Описание символьных констант, определяющих стиль пера*

Стиль VB	Стиль BCB/Delphi	Тип линии
`VbSolid`	`psSolid`	Сплошная линия
`VbDash`	`psDash`	Штриховая линия
`VbDot`	`psDot`	Пунктирная линия
`VbDashDot`	`psDashDot`	Штрих-пунктирная линия
`VbDashDotDot`	`psDashDotDot`	Штрих-пунктир-пунктирная линия
`VbInvisible`	`psClear`	Невидимая линия (цветом фона)
`VbInsideSolid`	`psInsideFrame`	Сплошная линия, не выходящая за пределы рамки

Приложения 8.3 (Delphi), 8.4 (BCB), 8.5 (Delphi)

Образцы воспроизведения линий разного стиля приведены на рис. 8.2. Попытка применить такие же стили для пера толщиной в 3 пиксела показана на рис. 8.3.

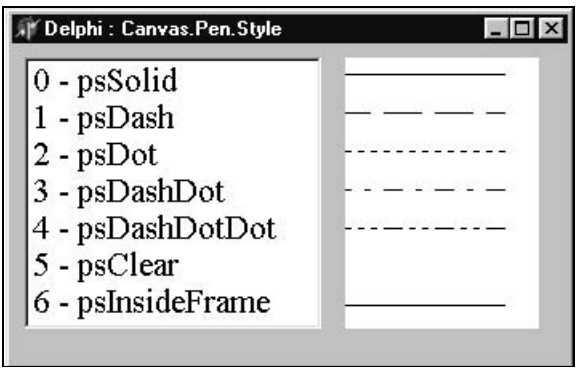


Рис. 8.2. Линии разных типов при ширине пера 1

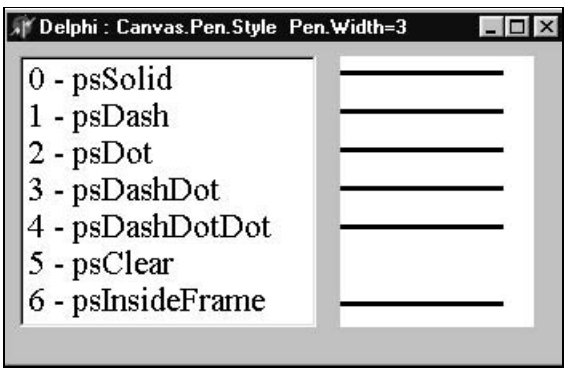


Рис. 8.3. Для толстых линий тип всегда один

Режим пера — свойство `Pen->Mode/Pen.Mode` (BCB/Delphi) или `DrawMode` (VB) — определяет способ взаимодействия цвета воспроизводимых пикселов линии с цветом соответствующих пикселов канвы. Это соответствует возможности аппаратуры формировать результирующий цвет пиксела в видеопамяти как результат логического взаимодействия прежнего кода цветности пиксела с вновь поступившим кодом. Для характеристики `Mode` предусмотрены 16 различных режимов формирования результирующего цвета отображаемых пикселов (табл. 8.5). По умолчанию действует режим `pmCopy`, при котором пиксел принимает цвет пера.

Таблица 8.5. *Символьные константы, определяющие режим формирования цвета (доступны для параметра* `Mode`*)*

Режим пера, VB	Режим пера, BCB	Способ формирования цвета воспроизводимого пиксела
`VbBlackness`	`pmBlack`	Отображаемый пиксел закрашивается черным цветом независимо от цвета пера и своего прежнего цвета
`VbWhiteness`	`pmWhite`	Пиксел закрашивается белым цветом
`VbNop`	`pmNop`	Сохраняется прежний цвет пиксела
`VbInvert`	`pmNot`	Прежний код цветности пиксела меняется на противоположный (инвертирование битов)
`VbCopyPen`	`pmCopy`	Пиксел закрашивается цветом пера
`VbNotCopyPen`	`pmNotCopy`	Пиксел закрашивается инвертированным цветом пера
`VbMergePen`	`pmMerge`	Прежний код цветности логически складывается с цветом пера
`VbNotMergePen`	`pmNotMerge`	Инвертированный цвет `pmMerge`
`VbMergePenNot`	`pmMergePenNot`	Цвет пера логически складывается с инвертированным прежним цветом
`VbMergeNotPen`	`pmMergeNotPen`	Инвертированный цвет пера логически складывается с прежним цветом
`VbMaskPen`	`pmMask`	Прежний код цветности логически умножается на цвет пера
`VbNotMaskPen`	`pmNotMask`	Инвертированный код прежней цветности пиксела логически умножается на цвет пера
`VbMaskNotPen`	`pmMaskNotPen`	Инвертированный цвет пера логически умножается на прежний цвет
`VbMaskPenNot`	`pmMaskPenNot`	Инвертированный прежний цвет логически умножается на цвет пера
`VbXorPen`	`pmXor`	Прежний цвет пиксела "складывается" с цветом пера по операции `XOR`
`VbNotXorPen`	`pmNotXor`	Прежний цвет пиксела "складывается" с цветом пера по операции `NOT XOR`

Для работы с изображениями особый интерес представляет режим сложения старого и нового цвета пиксела по операции `XOR`. Дело в том, что нулевой результат, воспринимаемый системой как цвет фона, может получиться только при взаимодействии двух одинаковых цветов. Этим часто пользуются

для того, чтобы стереть изображение и затем воспроизвести его в новой позиции для достижения эффекта анимации.

8.2. Что позволяет рисовать система VB?

Прежде всего, познакомимся с выбором системы координат и единиц измерения линейных величин, т. к. Visual Basic допускает гораздо более широкие возможности по сравнению со своими именитыми коллегами. По умолчанию, в каждом графическом объекте действует своя локальная система координат, начало которой расположено в левом верхнем углу. *Ось x* направлена в общепринятом направлении — вправо, а *ось y* — вниз. В качестве единиц измерения по умолчанию действуют твипы. В одном логическом дюйме содержится 1 440 твипов, в одном логическом сантиметре — 567 твипов. Термины *логический дюйм*, *логический сантиметр* используются здесь потому, что на экране нельзя точно измерять расстояния из-за возможности сжимать или растягивать изображение с помощью соответствующих средств управления монитором. Но при выдаче изображений на принтер логические единицы практически совпадают с общепринятыми.

Возможность ввести свою систему координат, в которой и оси направлены в привычном направлении, и физическая область определения рисунка вписывается в отведенное ей место на экране, предоставляется методом `Scale`. Идеология этого метода заимствована из техники масштабирования изображений, выводимых на плоттеры. Перед тем как построить какое-либо изображение на листе установленного формата, плоттеру сообщают, каким программным координатам должны соответствовать противоположные углы листа бумаги. Примерно так же поступает и метод `Scale`:

```
object.Scale (x1,y1)-(x2,y2)
```

Точка с координатами `(x1,y1)` определяет положение левого верхнего угла объекта (точнее — допустимой области рисования на объекте), а точка с координатами `(x2,y2)` — положение правого нижнего угла объекта. По заданным координатам система будет преобразовывать координаты точек в систему локальных координат объекта (а в конечном итоге — в пикселы экрана). Если метод `Scale` использован без указания графического объекта, то система координат устанавливается для формы приложения.

Метод `Scale` может быть вызван без указания координат обеих точек. В этом случае восстанавливается система координат объекта, действовавшая по умолчанию.

Система координат, установленная для объекта по умолчанию или выбранная пользователем, позволяет задавать точки канвы как в абсолютных, так и в относительных координатах. Координаты точек всегда заключаются в круглые скобки и имеют тип `Single`, т. е. могут быть заданы вещественными значениями. Перед относительными координатами точек записывается

служебное слово `STEP`. Почти все графические системы используют понятие графического курсора или так называемой текущей точки (Current Point, CP). В Visual Basic для обозначения позиции текущей точки на канве графического объекта используются свойства `CurrentX` и `CurrentY`. Перед началом работы текущая точка системой устанавливается в начало координат, а потом, в зависимости от выполняемых графических операций, перемещается в ту или иную позицию. Например, при построении отрезка невидимый графический курсор перемещается в конечную точку отрезка. Относительные координаты фактически являются приращениями к позиции текущей точки.

Единицы измерения линейных величин на канве графического объекта определяются целочисленным значением свойства `ScaleMode`, которое может быть задано с помощью одной из мнемонических констант, приведенных в табл. 8.6.

Таблица 8.6. *Описание мнемонических констант, доступных для* `ScaleMode`

Значение	Константа	Единицы измерения
0	`vbUser`	Определены пользователем с помощью метода `Scale`
1	`vbTwips`	Твипы (1 твип = 0,01764 мм, в 1 дюйме 1 440 твипов)
2	`vbPoints`	Пункты (1 пункт = 0,3528 мм, в 1 дюйме 72 пункта)
3	`vbPixels`	Пикселы (размер зависит от монитора и разрешения)
4	`vbCharacters`	Символы (по горизонтали одна единица = 120 твипам, по вертикали — 240 твипам, размер знакоместа = 2,117×4,233 мм)
5	`vbInches`	Дюймы (1 дюйм = 2,54 см)
6	`vbMillimeters`	Миллиметры
7	`vbCentimeters`	Сантиметры

При выборе любой единицы измерения длины, кроме `ScaleMode=vbUser`, начало координат располагается в левом верхнем углу графического объекта, и ось y направлена вниз. В пользовательской системе координат ось x может быть направлена как вправо, так и влево, а ось y — вверх или вниз. В любой системе координат положение левого верхнего угла канвы графического объекта определяется значениями свойств `ScaleLeft` и `ScaleTop`. Ширина и высота канвы объекта в выбранной системе координат совпадают со значениями свойств `ScaleWidth` и `ScaleHeight`.

Приложение 8.6 (VB)

Следующий пример демонстрирует использование каждой из единиц измерения длины:

```
Private Sub Command1_Click()
  Cls
  ScaleMode = 1
  Caption = "vbTwips, сторона=1440"
  Line (15, 15)-(1455, 1455), , B
End Sub
Private Sub Command2_Click()
 Cls
  ScaleMode = 2
  Caption = "vbPoints, сторона=72"
  Line (1, 1)-(73, 73), , B
End Sub
Private Sub Command3_Click()
  Cls
  ScaleMode = 3
  Caption = "vbPixels, сторона=96"
  Line (1, 1)-(97, 97), , B
End Sub
Private Sub Command4_Click()
  Cls
  ScaleMode = 4
  Caption = "vbCharacters, стороны = 12 x 6"
  Line (0, 0)-(12, 6), , B
End Sub
Private Sub Command5_Click()
  Cls
  ScaleMode = 5
  Caption = "vbInches, сторона = 1"
  Line (0, 0)-(1, 1), , B
End Sub
Private Sub Command6_Click()
  Cls
  ScaleMode = 6
  Caption = "vbMillimeters, сторона =25.4"
  Line (0, 0)-(25.4, 25.4), , B
```

```
End Sub
Private Sub Command7_Click()
  Cls
  ScaleMode = 7
  Caption = "vbCentimeters, сторона =2.54"
  Line (0, 0)-(2.54, 2.54), , B
End Sub

Private Sub Command8_Click()
  Cls
  Scale (-100, 100)-(100, -100)
  Circle (0, 0), 20
  Caption = "vbUser"
End Sub
```

На форме этого приложения расположены 8 командных кнопок с надписями, соответствующими той или иной единице измерения длины (рис. 8.4). Щелчок по любой из кнопок приводит к отображению квадрата со стороной, равной 1 дюйму. Исключение составляет режим `vbUser`, для которого в центре формы (т. е. в начале координат, заданном пользователем) воспроизводится окружность.

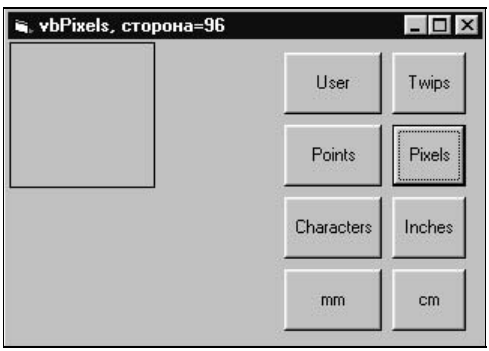


Рис. 8.4. Выбор единиц измерения длины

8.2.1. Работа с отдельными точками

Для построения точки в заданной позиции канвы используется метод `PSet`:

```
[object.]PSet [Step](x,y),[color]
```

Абсолютная или относительная позиция воспроизводимой точки задается парой вещественных значений `(x,y)`.

Необязательный параметр color определяет цвет отображаемой точки. Он может быть задан значением одной из функций RGB или QBColor. Если при вызове метода PSet цвет не указан, то в его роли выступает значение свойства ForeColor, установленное для объекта, на поверхности которого рисуется точка. Вообще говоря, на окончательный цвет точки влияет режим DrawMode (см. выше). Если мы хотим стереть точку, ее можно воспроизвести цветом BackColor.

Отсутствие объекта означает, что рисование происходит на поверхности формы.

Размер точки определяется шириной пера, т. е. значением свойства DrawWidth. Если ширина пера превышает 1, то центр точки приблизительно совпадает с указанной позицией (точное совпадение может быть только при нечетной ширине пера).

Для опроса цвета точки с указанными координатами используется функция Point, которая возвращает длинное целое число (тип Long), соответствующее представлению цвет в формате функции RGB:

```
color=[object.]Point(x, y)
```

Если точка с указанными координатами находится за пределами объекта, то функция Point возвращает значение -1.

8.2.2. Отрезки прямых и прямоугольники

Для построения линий или прямоугольников вызывается метод Line:

```
[object.]Line [Step][(x1,y1)]-[Step](x2,y2),[color],[B][F]
```

Точки с абсолютными или относительными координатами (x1,y1) и (x2,y2) задают концы воспроизводимого отрезка или определяют диагональ отображаемого прямоугольника. Координаты первой точки не являются обязательными аргументами, в случае их отсутствия роль начальной точки выполняет текущая точка. Единицы, в которых заданы координаты, определяются значением свойства ScaleMode.

Цвет линии или контура прямоугольника определяется необязательным параметром color. Как правило, код цветности задается одной из функций RGB или QBColor. При отсутствии параметра color цветом пера считается значение свойства ForeColor. Дополнительное влияние на результирующий цвет может оказать значение свойства DrawMode.

Для построения контура прямоугольника должен быть задан необязательный параметр — буква B. Комбинация BF означает, что прямоугольник должен быть залит с использованием текущих значений свойств FillColor и FillStyle.

На способ воспроизведения линии или границы прямоугольника оказывают влияние ширина пера (значение свойства DrawWidth) и стиль линии (значение свойства DrawStyle) при DrawWidth=1.

После воспроизведения линии или прямоугольника текущая точка перемещается в конец отрезка или диагонали.

8.2.3. Окружности, эллипсы, дуги и сектора

Для начала напомним, что канонические уравнения окружности и эллипса с центром в начале координат выглядят следующим образом:

$$\frac{x^2}{r^2}+\frac{y^2}{r^2}=1,$$

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1.$$

Здесь r — радиус окружности, a и b — полуоси эллипса вдоль оси x и y, соответственно. При $a = b$ эллипс превращается в окружность, а величина, пропорциональная отношению полуосей, определяет эксцентриситет эллипса. Более точно, эксцентриситетом эллипса называют величину e ($1 > e \geq 0$), которая вычисляется по следующей формуле:

$$e=\sqrt{1-k^2}\ ,$$

где $k = \min(a/b,\ b/a)$.

При $e = 0$ оба фокуса эллипса, расположенные на большей полуоси, сливаются в центр окружности. Увеличение эксцентриситета пропорционально увеличению расстояния между фокусами, что эквивалентно растяжению окружности по той или иной оси.

Для построения окружности, эллипса или их дуги вызывается один и тот же метод `Circle`:

```
[object.]Circle [Step](x,y),r,[color,a1,a2,e]
```

Точка с абсолютными или относительными координатами `(x,y)` задает центр окружности или эллипса.

Параметр `r` задает радиус окружности или участвует в определении полуосей эллипса, если задан последний параметр — коэффициент сжатия `e`, который может быть как больше, так и меньше 1. Значения координат, радиуса и коэффициента сжатия могут быть вещественными (тип `Single`).

Цвет контура определяется необязательным параметром `color`, значение которого имеет тип `Long`. Код цветности можно задать с помощью одной из функций `RGB` или `QBColor`. При отсутствии параметра `color` цветом пера считается значение свойства `ForeColor`. Дополнительное влияние на результирующий цвет может оказать значение свойства `DrawMode`.

Параметры `a1` и `a2` задают в радианах угол начального и конечного радиус-векторов дуги. Углы отсчитываются в направлении против часовой стрелки от оси x и могут меняться в диапазоне от -2π до 2π. По умолчанию `a1 = 0`,

a2 = 2π. Построение дуги всегда выполняется от угла a1 к углу a2. Вообще говоря, в определении начальной и конечной точки дуги всегда участвуют абсолютные значения углов a1,a2. Знак минус у любого из них воспринимается как указание о соединении соответствующей граничной точки с центром. Если оба угла заданы со знаками минус, то вместо дуги будет построен круговой или эллиптический сектор.

На толщину и способ отображения контура влияют текущие значения свойств DrawWidth и DrawStyle.

Замкнутая фигура — полная окружность или полный эллипс — могут быть залиты или заштрихованы, если предварительно были заданы соответствующие значения свойств FillColor и FillStyle.

8.2.4. Очистка канвы графического объекта

Для очистки поверхности графического объекта от следов рисования или вывода текстовой информации необходимо вызвать метод Cls:

```
[object.]Cls
```

Если объект не указан, то очищается поверхность формы. Естественно, что при этом не удаляются компоненты, размещенные на форме.

8.3. Что позволяют рисовать системы Borland C++ Builder и Delphi?

Канва — объект типа TCanvas — в качестве самостоятельного объекта обычно не выступает. Как правило, она является свойством тех компонент, на которых можно рисовать или воспроизводить графические изображения. Кроме упоминавшихся выше пера (объекта типа TPen) и кисти (объекта типа TBrush) в состав канвы встроены следующие объекты, выступающие в качестве ее свойств:

- *позиция пера* (объект типа PenPos), фиксирующая текущие координаты пера на канве (PenPos.x и PenPos.y);
- *шрифт* (объект типа TFont), определяющий состав и конфигурацию символов, их высоту, цвет и способ отображения надписей (подчерк, наклон, толщина контуров);
- *двумерный массив пикселов* (объект типа TPixels), заполняющих прямоугольную рабочую область канвы и фиксирующих цвет каждой точки.

Свойство Canvas и его характеристики доступны только во время работы приложения. Поэтому методы и характеристики канвы можно использовать только программным путем. Стандартная система координат канвы похожа на экранную — начало располагается в левом верхнем углу объекта,

ось x направлена вправо, ось y — вниз, координаты измеряются в пикселах. Поэтому все операции, связанные с масштабированием геометрических фигур, возлагаются на программу пользователя.

Положение текущей точки или позиции пера, определяемой свойством `PenPos`, — переменной типа `TPoint`, можно изменить обычными операторами присваивания:

```
TPoint A=Point(20,30);                  // BCB
имя_объекта->Canvas->PenPos = A;        // BCB
имя_объекта->Canvas->PenPos.x=20;       // BCB
имя_объекта->Canvas->PenPos.y=30;       // BCB
имя_объекта.Canvas.PenPos.x:=20;       {Delphi}
имя_объекта.Canvas.PenPos.y:=30;       {Delphi}
```

Точно такого же эффекта можно добиться обращением к методу `MoveTo`:

```
имя_объекта->Canvas->MoveTo(20,30);     // BCB
имя_объекта.Canvas.MoveTo(20,30);       {Delphi}
```

Наконец, положение текущей точки можно изменить, отобразив точку с указанными координатами путем присвоения нужного цвета элементу массива `Pixels`:

```
имя_объекта->Canvas->Pixels[0][30]=clRed;    // BCB
имя_объекта.Canvas.Pixels[20,30]=clRed;      {Delphi}
```

8.3.1. Отрезки прямых и ломаные

Как правило, построению отрезка предшествует вызов метода `MoveTo`, переводящий текущую точку в начало отрезка. Собственно построение отрезка прямой выполняется методом `LineTo`, аргументом которого являются координаты конечной точки:

```
имя_объекта->Canvas->LineTo(x,y);
```

Конец отрезка после работы метода `LineTo` становится новой текущей позицией пера. На вид отрезка оказывают влияние цвет пера (`Color`), его ширина (`Width`), способ взаимодействия пикселов канвы с цветом пера (`Mode`). При ширине пера в 1 пиксел дополнительное влияние оказывает установленный стиль линии (`Style`).

Ломаная линия, проходящая через вершины, заданные массивом координат `xy` (тип данных — `TPoint`), воспроизводится с помощью метода `Polyline`:

```
имя_объекта->Canvas->Polyline(xy,last_ind);      // BCB
имя_объекта.Canvas.Polyline(xy);                {Delphi}
```

В системе BCB второй аргумент определяет индекс последней точки ломаной, и он должен быть, по крайней мере, на 1 меньше, чем общее число точек в массиве `xy`. Построение ломаной всегда начинается с начальной точки `xy[0]`.

Приложение 8.7 (BCB)

В качестве примера приведем приложение, которое, реагируя на событие `OnPaint`, перерисовывает стрелку на объекте `PaintBox1`. В обработчике этого события приведены три варианта инициализации массива вершин ломаной, два из которых записаны в виде комментария:

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
  Windows::TPoint points[5]=
  {// Наиболее экономный вариант инициализации
    Point(0,50),
    Point(150,50),
    Point(130,40),
    Point(130,60),
    Point(150,50)
  };
/* // Второй способ задания точек
  points[0] = Point(0,50);
  points[1] = Point(150,50);
  points[2] = Point(130,40);
  points[3] = Point(130,60);
  points[4] = Point(150,50);
*/
/* // Третий способ задания точек
  points[0].x=0;
  points[0].y=50;
  points[1].x=150;
  points[1].y=50;
  points[2].x=130;
  points[2].y=40;
  points[3].x=130;
  points[3].y=60;
  points[4].x=150;
  points[4].y=50;
*/
  ((TPaintBox *)Sender)->Canvas->Polyline(points,4);
}
```

Конечно, последний оператор в этом обработчике можно было записать и попроще:

```
PaintBox1->Canvas->Polyline(points,4);
```

Но пример лишний раз демонстрирует приведение указателя на инициатор события `Sender` к типу нужного объекта. Это может пригодиться, когда один и тот же обработчик обслуживает несколько объектов одного класса. Результат работы приложения показан на рис. 8.5.

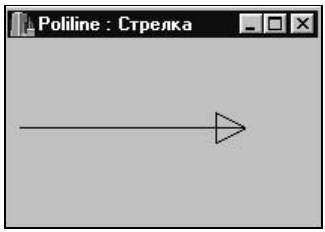


Рис. 8.5. Построение стрелки

В системе Delphi метод `Polyline` использует в качестве единственного аргумента ссылку на открытый массив точек. Это означает, что индексы в массиве отсчитываются от 0, а значение максимального индекса процедура `Polyline` узнает сама с помощью функции `High`. Поэтому не стоит особо удивляться примеру воспроизведения пятиконечной звезды, который приведен в справочном файле:

```
Canvas.Polyline([Point(40,10),Point(20,60),Point(70,30),
                Point(10,30),Point(60,60),Point(40,10)]);
```

8.3.2. Стандартные прямоугольники

Основной информацией для построения прямоугольников являются целочисленные координаты левого верхнего угла `(x1,y1)` и правого нижнего угла `(x2,y2)`. В некоторых методах эти значения должны быть заданы как данные типа `TRect`, в других допускается явное перечисление всех четырех чисел. Любое нарушение в последовательности задания координат методами построения прямоугольников игнорируется, изображение они не строят, но и никаких сообщений не выдают. Если одна из угловых точек находится за пределами графического объекта, то изображение прямоугольника будет обрезано, и пропадут две его стороны.

С помощью метода `FrameRect` отображается граница прямоугольника, которая воспроизводится в соответствии с текущими параметрами пера:

```
TRect r=Rect(10,10,100,200);    // BCB
.....................................
  Form1->Canvas->FrameRect(r);  // BCB
  Form1->Canvas->FrameRect(Rect(5,15,50,50)); // BCB


var r:Trect;                    {Delphi}
```

```
.....................................
r:=Rect(10,10,100,200);         {Delphi}
  Form1.Canvas.FrameRect(r);    {Delphi}
  Form1.Canvas.FrameRect(Rect(5,15,50,50)); {Delphi}
```

Метод `FillRect` позволяет заполнить внутренность прямоугольника `r` в соответствии с текущими параметрами кисти:

```
имя_объекта->Canvas->FillRect(r);     // BCB
имя_объекта.Canvas.FillRect(r);       {Delphi}
```

Метод `Rectangle` совмещает в себе обе функции — и строит границу, и заливает внутренность прямоугольника. К этому методу можно обратиться либо с одним аргументом типа `TRect`, либо с четырьмя целочисленными значениями, координатами левого верхнего и правого нижнего углов:

```
имя_объекта->Canvas->Rectangle(x1,y1,x2,y2);     // BCB
имя_объекта->Canvas->Rectangle(r);               // BCB
имя_объекта.Canvas.Rectangle(x1,y1,x2,y2);       {Delphi}
имя_объекта.Canvas.Rectangle(r);                 {Delphi}
```

Прямоугольник со скругленными углами строится с помощью метода `RoundRect`

```
имя_объекта->Canvas->RoundRect(x1,y1,x2,y2,w,h);// BCB
имя_объекта.Canvas.RoundRect(x1,y1,x2,y2,w,h);  {Delphi}
```

Дополнительные аргументы определяют ширину (`w`) и высоту (`h`) эллипса, четвертушки дуг которого используются для сопряжения смежных сторон. При `w=h` в сопряжении участвуют соответствующие четверти окружности.

Специальный прямоугольник с пунктирной границей воспроизводится методом `DrawFocusRect`:

```
имя_объекта->Canvas->DrawFocusRect(r);     // BCB
имя_объекта.Canvas.DrawFocusRect(r);       {Delphi}
```

Название этого метода связано с отображением кнопок, которым передается фокус. Их заголовок обычно окаймляется пунктирным прямоугольником. Пунктирный прямоугольник, динамически отслеживающий перемещение мыши, очень часто используют для выделения прямоугольной области какого-либо изображения. Его обычно называют *резиновым* прямоугольником. Техника его построения базируется на использовании режима `XOR` при определении результирующего цвета пиксела. В момент нажатия правой клавиши мыши (событие `OnMouseDown`) фиксируются координаты одного из углов резинового прямоугольника, а затем отслеживаются координаты курсора во время перемещения мыши с зажатой клавишей (периодически возникающее событие `OnMouseMove`). Каждый раз в режиме `XOR` стирается предыдущее изображение пунктирного прямоугольника и строится новое. Процесс прекращается в момент отпускания клавиши мыши (событие `OnMouseUp`).

Приложение 8.8 (BCB)

Описанная технология реализована в следующем приложении:

```
int x0,y0,x1,y1;
bool Down=false;
// ---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  x0=X;  StatusBar1->Panels->Items[1]->Text=X;
  y0=Y;  StatusBar1->Panels->Items[3]->Text=Y;
  x1=X; y1=Y;
  Down=true;
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseUp(TObject *Sender,
TMouseButton Button, TShiftState Shift, int X, int Y)
{
  Down=false;
  StatusBar1->Panels->Items[9]->Text = X;
  StatusBar1->Panels->Items[11]->Text= Y;
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseMove(TObject *Sender,
TShiftState Shift, int X, int Y)
{
  TRect r;
  if(Down)
   {
    r.Left  =(x0<x1)? x0 : x1;
    r.Right =(x0<x1)? x1 : x0;
    r.Top   =(y0<y1)? y0 : y1;
    r.Bottom=(y0<y1)? y1 : y0;
    Image1->Canvas->DrawFocusRect(r);
```

```
    r.Left  =(x0<X)? x0 : X;
    r.Right =(x0<X)? X : x0;
    r.Top=   (y0<Y)? y0 : Y;
    r.Bottom=(y0<Y)? Y : y0;
    Image1->Canvas->DrawFocusRect(r);
    x1=X;
    y1=Y;
    StatusBar1->Panels->Items[5]->Text=X;
    StatusBar1->Panels->Items[7]->Text=Y;
   }
}
```

Глобальная переменная `Down` используется для фиксации состояния — мышь перемешается по канве `Image1` с нажатой или не нажатой клавишей. В глобальных переменных (`x0,y0`) и (`x1,y1`) хранятся текущие значения левой верхней и правой нижней вершин резинового прямоугольника. При этом для правильного его воспроизведения в `x0` всегда заносится минимум из (`x0,x1`), а в `x1` — максимум из (`x0,x1`). Соответственно, в `y0` заносится максимум из (`y0,y1`), а в `y0` — минимум из (`y0,y1`). Если этого не делать, то прямоугольник будет рисоваться только при движении курсора мыши от начальной точки вниз и вправо.

Так как метод `DrawFocusRect` автоматически включает режим `pmXor`, то в программе пользователя дублировать такое переключение не обязательно. Повторное воспроизведение резинового прямоугольника по его прежним координатам уничтожает предшествующее изображение прямоугольника и восстанавливает поверхность канвы, перекрытую прежними границами.

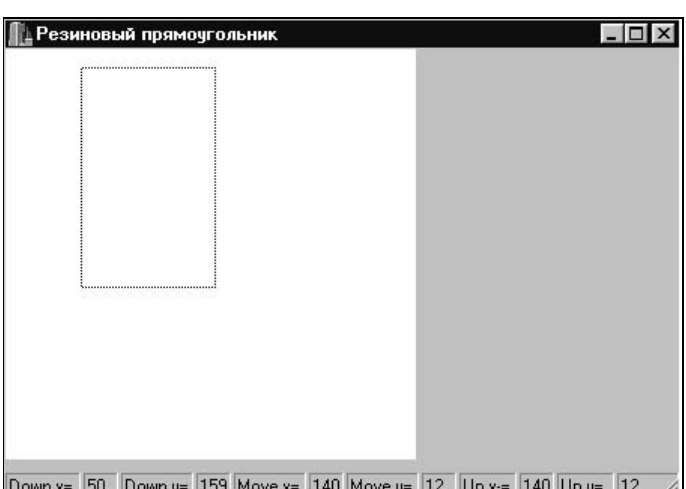


Рис. 8.6. Воспроизведение "резинового" прямоугольника

Строка состояния StatusBar1 использована для отображения координат мыши во время действия того или иного события мыши — при нажатии кнопки, при движении с нажатой кнопкой и при ее отпускании. Результат работы приложения приведен на рис. 8.6.

Приложение 8.9 (BCB)

Конечно, резиновый прямоугольник можно создать и другими средствами, меняя, например, стиль воспроизведения линий. Однако самый подходящий "точечный" стиль psDot смотрится гораздо хуже — точки больше смахивают на штрихи:

```
int x0,y0,x1,y1;
bool Down=false;
TRect r;
// ---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
// ---------------------------------------------------------------
void __fastcall TForm1::FormActivate(TObject *Sender)
{
      Image1->Canvas->Brush->Color=clWhite;
      Image2->Canvas->Brush->Color=clWhite;
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  x0=X; y0=Y;
  x1=X; y1=Y;
  Down=true;
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseMove(TObject *Sender,
TShiftState Shift, int X, int Y)
{ if(Down)
    { r.Left=(x0<x1)?x0:x1;
      r.Right=(x0<x1)?x1:x0;
      r.Top=(y0<y1)?y0:y1;
      r.Bottom=(y0<y1)?y1:y0;
```

```
      Image2->Canvas->MoveTo(r.Left,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Top);
      r.Left=(x0<X)?x0:X;
      r.Right=(x0<X)?X:x0;
      r.Top=(y0<Y)?y0:Y;
      r.Bottom=(y0<Y)?Y:y0;
      Image2->Canvas->MoveTo(r.Left,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Top);
      x1=X;
      y1=Y;
    }
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseUp(TObject *Sender,
TMouseButton Button, TShiftState Shift, int X, int Y)
{
  Down=false;
}
// ----------------------------------------------------------------
void __fastcall TForm1::Image2MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  x0=X; y0=Y;
  x1=X; y1=Y;
  Down=true;
  Image2->Canvas->Pen->Style=psDot;
  Image2->Canvas->Pen->Mode=pmXor;
}
// ----------------------------------------------------------------
void __fastcall TForm1::Image2MouseMove(TObject *Sender,
TShiftState Shift, int X, int Y)
{
  if(Down)
    { r.Left=(x0<x1)?x0:x1;
```

```
      r.Right=(x0<x1)?x1:x0;
      r.Top=(y0<y1)?y0:y1;
      r.Bottom=(y0<y1)?y1:y0;
      Image2->Canvas->MoveTo(r.Left,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Top);
      r1.Left=(x0<X)?x0:X;
      r1.Right=(x0<X)?X:x0;
      r1.Top=(y0<Y)?y0:Y;
      r1.Bottom=(y0<Y)?Y:y0;
      Image2->Canvas->MoveTo(r.Left,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Top);
      Image2->Canvas->LineTo(r.Right,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Bottom);
      Image2->Canvas->LineTo(r.Left,r.Top);
      x1=X;  y1=Y;
    }
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image2MouseUp(TObject *Sender,
TMouseButton Button, TShiftState Shift, int X, int Y)
{
  Down=false;
}
```

В приведенном выше примере на форме находятся два объекта типа `TImage`. На левом объекте "резиновый" прямоугольник воспроизводится методом `DrawFocusRect`, а на правом — путем построения отрезков, состоящих из пунктирных линий. Результат работы такого приложения приведен на рис. 8.7.

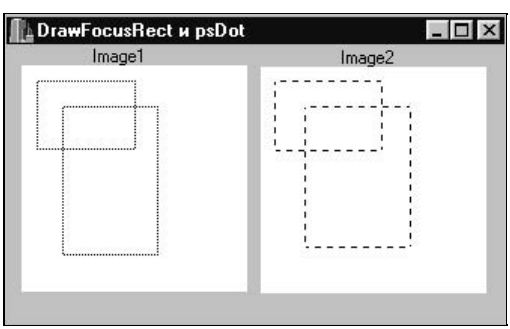


Рис. 8.7. Два стиля "резиновых" прямоугольников

8.3.3. Нестандартные многоугольники

Для построения контура произвольного многоугольника можно воспользоваться методом `Polyline`, предусмотрев дублирование начальной и конечной точек в массиве `xy`. В отличие от такой ломаной можно залить (заштриховать) внутреннюю область многоугольника `xy` с учетом текущих характеристик кисти, обратившись к методу `Polygon`. Контур многоугольника при этом не обводится, но замыкается он автоматически, т. е. координаты начальной и конечной точек в массиве `xy` могут не совпадать:

```
имя_объекта->Canvas->Polygon(xy,last_ind);      // BCB
имя_объекта.Canvas.Polygon(xy);                 {Delphi}
```

Единственным аргументом в Delphi является ссылка на открытый массив, поэтому нужда в дополнительном параметре отпала.

Для заливки замкнутой фигуры, граница которой обведена сплошной линией известного цвета, можно обратиться и к методу `FloodFill`:

```
имя_объекта->Canvas->FloodFill(x,y,col,fs);
```

Заливка фигуры начинается из точки с координатами (`x,y`). Продолжение операции зависит от значения параметров `col` и `fs`. При `fs=fsSurface` заливка продолжается до тех пор, пока в окрестности текущей точки находятся пикселы цвета `col`. При `fs=fsBorder` заливка продолжается до выхода на граничные пикселы цвета `col`.

Обращаем ваше внимание на то, что заливка фигур правильно выполняется в тех случаях, когда в контуре отсутствуют самопересечения. На примере звезды, проведенной единым росчерком пера через 5 наиболее удаленных вершин, вы можете убедиться, что просто так залить полученную фигуру нельзя ни методом `Polygon`, ни методом `FloodFill`. Если бы мы строили звезду, проводя короткие отрезки между каждыми смежными вершинами, то получилась бы нормальная невыпуклая фигура, которую бы полностью залил как метод `Polygon`, так и метод `FloodFill`.

Приложение 8.10 (Delphi)

В приведенном ниже приложении на форме помимо объекта `PaintBox1` находится компонент `StatusBar1` (Строка состояния), на котором выделены 4 панели для фиксации координат мыши при перемещении ее курсора. В панели с индексами 0 и 2, соответственно, внесен текст "`x=`" и "`y=`".

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
var
  p:array [0..5] of Tpoint;
begin
  p[0].X:=140;    p[0].Y:=10;
```

```
  p[1].X:=120;    p[1].Y:=60;
  p[2].X:=170;    p[2].Y:=30;
  p[3].X:=110;    p[3].Y:=30;
  p[4].X:=160;    p[4].Y:=60;
  p[5].X:=140;    p[5].Y:=10;

  with Sender as TPaintBox do
  begin
    Canvas.Pen.Color := clWhite;
    Canvas.Brush.Color := clBlack;
    Canvas.Polygon([Point(40, 10),Point(20, 60),
           Point(70, 30),Point(10, 30),Point(60, 60)]);
    Canvas.Polyline(p);
    Canvas.FloodFill(140,40,clWhite,fsBorder);
  end;
end;

procedure TForm1.PaintBox1MouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
begin
  StatusBar1.Panels[1].Text:=IntToStr(X);
  StatusBar1.Panels[3].Text:=IntToStr(Y);
end;

end.
```

Рис. 8.8. Заливка сложной фигуры

Пример работы этого приложения показан на рис. 8.8. Нетрудно догадаться, что для полной заливки правой звезды потребовалось бы еще 5 обращений

к методу `FloodFill` с указанием координат внутренних точек в каждом луче. Но даже и в этом случае внутренняя часть звезды отделялась бы от лучей белой границей. Так что гораздо проще добавить еще 5 точек с координатами внутренних вершин и один раз обратиться к методу `Polygon`.

8.3.4. Кривые второго порядка

К кривым второго порядка относятся только эллипсы, их дуги, сегменты и сектора, вписывающиеся в прямоугольник, заданный координатами противоположных вершин, — (`x1,y1`) и (`x2,y2`). Если окаймляющий прямоугольник оказывается квадратом, то воспроизводятся окружности или их фрагменты (дуги, сегменты или сектора). Лучи, проведенные из центра прямоугольника в точки (`x3,y3`) и (`x4,y4`), определяют границы фрагментов эллипса. Дуги, сегменты и сектора строятся от начального радиус-вектора к конечному радиус-вектору против часовой стрелки в Windows 95/98 и по часовой стрелке в Windows NT.

Для построения границы дуги с использованием текущих характеристик пера применяется метод `Arc`:

```
  имя_объекта->Canvas->Arc(x1,y1,x2,y2,x3,y3,x4,y4); // BCB
имя_объекта.Canvas.Arc(x1,y1,x2,y2,x3,y3,x4,y4);     {Delphi}
```

Построение границы эллипса и заливка его сегмента (т. е. области между дугой и хордой) осуществляется с помощью метода `Chord`:

```
имя_объекта->Canvas->Chord(x1,y1,x2,y2,x3,y3,x4,y4);  // BCB
имя_объекта.Canvas.Chord(x1,y1,x2,y2,x3,y3,x4,y4);    {Delphi}
```

Для аналогичного построения залитого сектора используется метод `Pie`:

```
имя_объекта->Canvas->Pie(x1,y1,x2,y2,x3,y3,x4,y4);    // BCB
имя_объекта.Canvas.Pie(x1,y1,x2,y2,x3,y3,x4,y4);      {Delphi}
```

Построение границы полного эллипса и заливка его внутренности производятся с помощью метода `Ellipse`:

```
имя_объекта->Canvas->Ellipse(x1,y1,x2,y2);             // BCB
имя_объекта.Canvas.Ellipse(x1,y1,x2,y2);               {Delphi}
```

8.3.5. Обмен с графическими файлами

Канва графических объектов может быть покрыта растровым изображением, хранящимся в графическом файле одного из форматов BMP, EMF, ICO, JPG, WMF.

```
имя_объекта->Canvas->LoadFromFile("имя_файла");  // BCB
имя_объекта.Canvas.LoadFromFile('имя_файла');    {Delphi}
```

8.4. Вывод символьных и числовых данных на канве

Довольно часто графические изображения сопровождаются пояснительными подписями. Кроме того, в среде Windows существует возможность выбора разнообразных шрифтов и предоставляется набор средств по управлению размерами букв, их наклоном, толщиной контура, подчеркиванием и перечеркиванием. Все это диктует необходимость отображать символьные данные не только в поле специальных объектов, таких как метки (`Label`), поля ввода (`InputBox, Edit`) и др., но и непосредственно на канве графического объекта. А уж такой графический объект, как принтер, по определению, должен уметь воспроизводить алфавитно-цифровую информацию.

8.4.1. Отображение текстовой и числовой информации в среде Visual Basic

Создатели Visual Basic поступили очень мудро, сохранив все возможности оператора (теперь метода) `Print`, к которому привыкли пользователи ранних версий этого популярного языка программирования:

```
[object.]Print список_вывода
[object.]Print Using "format"; список_вывода
```

В дополнение к хорошо сохранившемуся старому добавилась возможность произвести предварительную установку используемого шрифта и всех его характеристик, принятых в среде Windows.

8.4.2. Работа с текстами в Borland C++ Builder и Delphi

Для вывода текста на поверхности канвы используется один из методов `TextOut` или `TextRect`. Их общими аргументами являются координаты `(x,y)` точки привязки (левый верхний угол текста) и сам текст — объект `text` типа `String` в Delphi и `AnsiString` в BCB:

```
TRect R=Rect(10,10,100,100);            // BCB
  имя_объекта->Canvas->TextOut(x,y,text);
  имя_объекта->Canvas->TextRect(R,x,y,text);

  имя_объекта.Canvas,TextOut(x,y,text);    {Delphi}
  имя_объекта.Canvas.TextRect(R,x,y,text); {Delphi}
```

Аргумент `R` — объект типа `TRect` — определяет прямоугольник, внутри которого должна находиться воспроизводимая надпись. Части надписи, выходящие за границы этого прямоугольника, отсекаются.

Если наряду с текстовой информацией на канве нужно разместить и числовые данные, то их можно предварительно преобразовать из числового формата, использовав одну из функций типа sprintf, itoa, ltoa, ftoa, IntToStr, FloatToStr и др. Данные, преобразованные в строку типа char, следует конвертировать в формат AnsiString. Например:

```
char msg[80];
sprintf(msg,"x=%d y=%d",x,y);
Canvas->TextOut(10,10,AnsiString(msg));

var s:string;
s:='x='+IntToStr(x)+' y='+IntToStr(y);
Canvas.TextOut(10,10,s);
```

Вспомогательные методы TextWidth и TextHeight могут быть использованы для определения ширины и высоты еще не выведенной надписи в пикселах:

```
w = имя_объекта->Canvas->TextWidth(text);
h = имя_объекта->Canvas->TextHeight(text);
```

8.5. Специфика работы с графическими объектами

8.5.1. Графические объекты в системе Visual Basic

Наиболее универсальный объект PictureBox предназначен для построения геометрических фигур и размещения растровых изображений, хранящихся в одном из графических форматов bmp, cur, dib, emf, gif, ico, jpg, rle и wmf. По умолчанию объектам PictureBox присваиваются системные имена Picture1, Picture2, ... Точки, линии, прямоугольники, окружности, эллипсы и их дуги воспроизводятся на канве объекта Picture1 с помощью методов Picture1.PSet, Picture1.Line и Picture1.Circle. На поведение объекта PictureBox наибольшее влияние оказывают свойства AutoSize, AutoRedraw, Picture и Image.

Когда объект виден на экране, его канву представляет соответствующий участок видеопамяти, который можно ассоциировать со свойством Picture. Все, что мы рисуем с помощью графических методов, временно запоминается в этой памяти, но полученное изображение хранится до тех пор, пока канва видна полностью. Стоит заменить форму пиктограммой или какому-либо окну перекрыть часть объекта, как прежний участок видеопамяти захватит другой элемент, и часть нашей картинки или все ее изображение будет потеряно. Восстановление границ канвы объекта в прежних размерах не всегда приводит к автоматическому восстановлению графического изображения. Здесь возможны два варианта.

Во-первых, содержимое свойства Picture может автоматически дублироваться после выполнения каждого графического оператора в некотором участке оперативной памяти, где размещается свойство Image. Тогда после восстановления границ объекта содержимое свойства Image автоматически копируется в свойство Picture, и изображение восстанавливается без нашего участия. Но свойство Image не создается при объекте PictureBox автоматически. Для его возникновения необходимо выполнение одного из двух условий. Если в какой-то момент времени установлено значение свойства AutoRedraw=True, то объекту PictureBox система выделяет дополнительную оперативную память для размещения свойства Image. С этого момента все, что записывается в свойство Picture, тут же дублируется и в свойстве Image. И только эта часть рисунка будет автоматически восстановлена после частичной или полной утраты изображения. Область Image, принадлежащая объекту Picture1, автоматически приписывается объекту Picture2 после выполнения следующей операции присваивания:

```
Picture2.Picture = Picture1.Image
```

При этом область Image, ранее выделенная объекту Picture1, второй раз не дублируется. Просто между памятью Image объекта Picture1 и объектом Picture2 фиксируется связь. Впечатление такое, как будто у Picture2 появилась своя собственная копия изображения, и независимо от установки значения свойства Picture2.AutoRedraw изображение объекта Picture2 будет восстанавливаться автоматически по содержимому Picture1.Image.

Приложение 8.11 (VB)

Приводимое ниже приложение демонстрирует эти возможности. На форме находится два объекта Picture1 и Picture2 одинакового размера. В момент запуска приложения устанавливается в истину свойство Picture1.AutoRedraw, что автоматически снабжает первый объект свойством для дублирования изображения. После щелчка по поверхности Picture1 на этом объекте возникает очередная залитая окружность. Все, что нарисовано в результате последовательных щелчков по Picture1, будет автоматически восстановлено. После первого щелчка по Picture2 содержимое первого объекта копируется на канву второго объекта, который с этого момента совместно с первым объектом использует область Picture1.Image. После переписи изображения сбрасывается значение свойства Picture1.AutoRedraw. Теперь щелчки по первому объекту добавляют на нем новые окружности, но в Image они уже не дублируются. Поэтому восстановлению подлежит только та часть изображения, которая была накоплена в период истинности свойства AutoRedraw. Если бы в программе переноса изображения из Picture1 в Picture2 мы включили свойство Picture2.AutoRedraw, то содержимое Picture2 было бы скопировано в общий Image. И тогда восстановление обоих изображений было бы более полным. Поэкспериментируйте с этим

приложением, меняя порядок и сочетание замен свойств `AutoRedraw`. Внешний вид его формы показан на рис. 8.9.

```
Private Sub Form_Load()
  'Picture2.AutoRedraw = True
  Picture1.AutoRedraw = True
End Sub

Private Sub Picture1_Click()
  Dim PW, PH
  Picture1.FillStyle = vbFSSolid
  Picture1.FillColor = QBColor(Int(Rnd * 15))
  PW = Picture1.ScaleWidth
  PH = Picture1.ScaleHeight
  Picture1.Circle (Int(Rnd * PW), Int(Rnd * PH)), 250
End Sub

Private Sub Picture2_Click()
 Picture1.AutoRedraw = False
 Picture2.Picture = Picture1.Image
 Caption = " Picture2.AutoRedraw = " & Picture2.AutoRedraw
' Caption = " Picture1.AutoRedraw = " & Picture1.AutoRedraw

End Sub
```

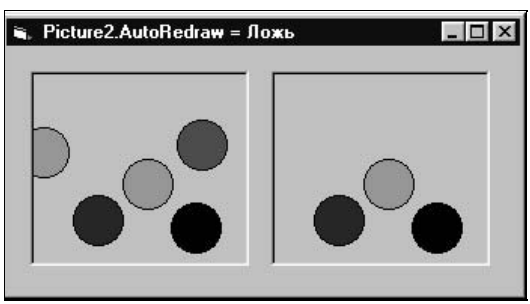


Рис. 8.9. Дублирование и восстановление изображений

Если объект `PictureBox` лишен копии в оперативной памяти, то после восстановления первоначальных размеров изображения система генерирует событие `Paint`. И в этом случае ничего не остается другого, кроме как повторить формирование изображения заново программным путем.

Логическое свойство `AutoSize` управляет размерами окна `PictureBox` в момент загрузки изображения на канву из графического файла:

```
Picture1.Picture = LoadPicture("qq.bmp")
```

При `AutoSize=True` размер объекта старается подстроиться под размер загружаемого изображения. Однако результат такой загрузки не всегда может оказаться удовлетворительным. Окно типа `PictureBox` не может выйти за границы формы, его левый верхний угол остается на прежнем месте, линейки прокрутки большого графического изображения в VB не предусмотрены. Поэтому желательно проделать загрузку предполагаемых рисунков на стадии проектирования, чтобы заранее произвести подгонку размеров и положения соответствующих окон.

Для стирания ранее загруженного изображения можно воспользоваться методом `LoadPicture` без аргумента:

```
Picture1.Picture=LoadPicture()
```

При `AutoSize=False` изображение прижимается к левому верхнему углу объекта, размер окна не изменяется, и мы рискуем увидеть либо часть изображения, обрезанную по границам окна, либо незаполненную часть окна.

Изображение, загруженное из файла, автоматически восстанавливается после любых нарушений видимости рисунка.

Форма тоже обладает графическими свойствами — `Picture`, `Image` и `Icon`. Первое свойство позволяет рисовать на форме, используя методы `PSet`, `Line` или `Circle`, а также размещать растровые изображения, загружая их из файла или копируя из других графических объектов:

```
Form1.Picture = LoadPicture("qq.bmp")
```

Свойство `Icon`, как правило, используется для загрузки пиктограммы, заменяющей изображение свернутой формы:

```
Form1.Icon= LoadPicture("qq.ico")
```

Для запоминания в дисковом файле изображения, построенного на форме, можно прибегнуть к оператору `SavePicture`:

```
SavePicture Form1.Image, "abc.bmp"
```

Обратите внимание на то, что при загрузке содержимого графического файла в качестве адресата выступало свойство `Form1.Picture`. Формат оператора запоминания изображения использует свойство `Form1.Image`.

Еще один графический объект `Image` обладает гораздо меньшими возможностями, нежели `PictureBox`. На поверхность этого объекта можно загрузить содержимое графического файла или скопировать растровую картинку из другого объекта:

```
Set Image1.Picture = LoadPicture("qq.bmp")
Image2.Picture = Image1.Picture
```

Приложение 8.12 (VB)

Установка логического свойства `Stretch=True` заставляет окно типа `Image` увеличиться или уменьшиться до размера загружаемого или копируемого рисунка. Однако выход за пределы формы по-прежнему запрещен, и линейками прокрутки большого изображения объект `Image` не располагает. На рис. 8.10 демонстрируется процесс копирования пиктограмм в окна типа `Picture1` и `Image2`, загруженных на стадии проектирования в объекты `Picture1` и `Image1`. При начальной загрузке пиктограмм размер `Picture1` сохранился, тогда как объект `Image1` принял размеры рисунка. Такое же сжатие произошло и после копирования рисунка в объект `Image2`.

```
Private Sub Form_Click()
 Picture2.Picture = Picture1.Picture
 Image2.Picture = Image1.Picture
End Sub
```

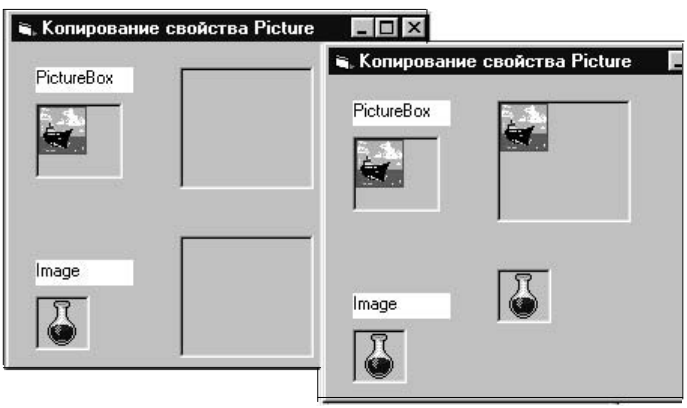


Рис. 8.10. Копирование свойств `Picture` из объектов `Picture1` и `Image1`

К графическим объектам системы Visual Basic относятся компоненты `Line` и `Shape`. Их использование имеет ограниченный характер. Дело в том, что их можно применять на стадии проектирования для создания на поверхности тех или иных объектов изображений прямых и простейших геометрических фигур — квадратов и прямоугольников с прямыми или закругленными углами, овалов и окружностей. Примерно так же как и в соответствующих графических методах, применяемых во время работы приложения, здесь можно управлять такими свойствами, как толщина, цвет и стиль линий, цвет и способ заливки геометрических фигур.

Главные свойства объекта `Line` — `X1`, `Y1`, `X2`, `Y2` — определяют координаты начала и конца отрезка в системе координат контейнера. На способ воспроизведения линии влияют установки свойств `BorderStyle` и `BorderWidth`.

Отрезки прямых, созданные на стадии проектирования на формах, рисунках и рамках, восстанавливаются автоматически независимо от значения свойства `AutoRedraw`.

Форма геометрической фигуры, создаваемой с помощью объекта `Shape`, задается значением свойства с одноименным названием `Shape`. Последнее может принимать одно из шести целочисленных значений, приведенных в табл. 8.7.

Таблица 8.7. *Тип геометрической фигуры в зависимости от целочисленного значения или символьной константы*

Число	Константа	Создаваемая фигура
0	`vbShapeRectangle`	Прямоугольник
1	`vbShapeSquare`	Квадрат
2	`vbShapeOvale`	Овал
3	`vbShapeCircle`	Окружность
4	`vbShapeRoundedRectangle`	Прямоугольник с закругленными углами
5	`vbShapeRoundedSquare`	Квадрат с закругленными углами

Фигура, создаваемая с помощью объекта `Shape`, должна вписываться в прямоугольник, определяемый шириной (`Width`) и высотой (`Height`) объекта `Shape`. Для ее заливки используются свойства `FillStyle` и `FillColor`. Фигуры, включенные в приложение на стадии проектирования, восстанавливаются автоматически после любых нарушений зон видимости.

8.5.2. Графические объекты в системах Borland C++ Builder и Delphi

Графические средства позволяют воспроизводить на экране дисплея статические или динамические рисунки, отображающие действия программы или пользователя, манипулировать с изображениями или их фрагментами, сохранять рисунки в дисковых файлах или передавать их другим приложениям.

В системах BCB и Delphi существует более десятка компонент и графических объектов, используемых для визуализации различных изображений, создаваемых программным путем или отображающих содержимое некоторых графических файлов. Их всех роднит свойство `Canvas`, которое собственно и обеспечивает компоненты поверхностью (канвой), способной хранить графические образы.

С целью экономии ресурсов под канву большинства видимых графических объектов временно выделяется соответствующий участок видеопамяти. Поэтому такие объекты иногда называют экранными, и созданное на их

поверхности изображение остается видимым до тех пор, пока содержащее их окно не будет свернуто или перекрыто другим окном. Чтобы изображение появилось вновь, программист должен предусмотреть индивидуальные средства восстановления рисунка в обработчике события `OnPaint`. Экранные объекты оперативно отражают результат каждой графической процедуры, выполненной на их канве.

В отличие от сугубо экранных объектов существуют графические компоненты, канва которых продублирована соответствующим участком оперативной памяти. Одним из таких объектов является компонент типа `TImage`. Изображения, создаваемые на их поверхности, сохраняются до тех пор, пока соответствующий объект не будет уничтожен. Единственное неудобство объекта типа `TImage` заключается в том, что содержимое его канвы продолжает оставаться невидимым до тех пор, пока процесс построения соответствующего графического образа не будет завершен. Зато графические изображения, выполненные на поверхности компонента типа `TImage`, автоматически восстанавливаются при обновлении свернутой или частично перекрытой формы.

В системах BCB и Delphi существует группа графических объектов, которые располагают канвой только в оперативной памяти, но построенное на ней изображение остается невидимым до тех пор, пока содержимое канвы внеэкранного объекта не будет скопировано на канву видимого графического компонента.

Объекты типа *TBitmap*

Графические объекты типа `TBitmap` (bitmap — битовая карта) предназначены для хранения прямоугольных растровых изображений и манипуляций над ними. Они создаются и уничтожаются в процессе работы программы. Процессы объявления и создания объекта можно совместить:

```
Graphics::TBitmap *pBMP = new Graphics::TBitmap();  // BCB
pBMP:TBitmap;                                       {Delphi}
...............
pBMP:=TBitmap.Create;
```

Здесь `pBMP` — указатель на создаваемый объект. Для объекта, на котором мы собираемся рисовать, необходимо установить ширину (`pBMP→Width/pBMP.Width`) и высоту (`pBMP→Height/pBMP.Height`) прямоугольной области. Если в объект предполагается загрузить другое изображение, то ресурсы для его хранения система выделит сама. Значение свойства `pBMP→Empty=true` свидетельствует о том, что объект пуст.

Для уничтожения объекта используется процедура `delete` (BCB) и `free` (Delphi):

```
delete pBMP;        // BCB
pBMP.Free;         {Delphi}
```

Основная информация об изображении, хранящемся в объекте типа `TBitmap`, представлена двумерным массивом `pBMP→Canvas→Pixels[x][y]`, значениями элементов которого являются коды цветности пикселов. Они формируются в процессе построения графических фигур или манипуляций над ними с помощью методов канвы, при загрузке изображения из графического файла (метод `LoadFromFile`) или из буфера обмена (методы `Assign` и `LoadFromClipboardFormat`). Для монохромного изображения (свойство `pBMP→Monochrome=true`) в массиве `Pixels` отводится 1 бит на пиксел (свойство `pBMP→PixelFormat=pf1bit`). В зависимости от установленной палитры на каждый пиксел выделяется 4, 8, 15, 16, 24 или 32 бита. Таким образом, цветное изображение большого размера в формате *truecolor* занимает в оперативной памяти достаточно много места.

Объекты типа `TBitmap` могут участвовать в операторах присвоения или быть операндами метода `Assign`:

```
Bitmap1=Bitmap2;
```

```
Bitmap1→Assign(Bitmap2);
```

Приведенные выше два действия не являются эквивалентными в полном смысле слова. Обычный оператор присваивания в отличие от процедуры `Assign` не создает копию объекта `Bitmap2`, а всего лишь присваивает указателю `Bitmap1` значение адреса `Bitmap2`.

Среди наиболее распространенных операций над объектами типа `TBitmap` — копирование их содержимого в дисковый файл с расширением bmp (метод `SaveToFile`) и считывание растрового изображения из дискового файла (метод `LoadFromFile`):

```
pBMP→LoadFromFile("имя bmp-файла");
```

```
pBMP→SaveToFile("имя bmp-файла");
```

Объекты типа *TIcon*

Графические объекты типа `TIcon` представляют собой небольшие цветные или полутоновые изображения, как правило, размером 32×32 пиксела. Обычно они используются как графические индикаторы свернутых приложений, но могут выступать и в роли пояснительных миниатюр на поверхности каких-либо компонент. Дисковые файлы хранят такие изображения в формате ico.

Объекты типа *TMetafile*

Графические файлы Microsoft Metafile поначалу были задуманы как внутренний векторный формат операционных систем Windows. В ранних версиях использовались файлы с расширением wmf (Windows 3.1 Metafile). Новые операционные системы используют более современный формат Win32 Enhanced Metafile (расширение emf). Формат Metafile оказался настолько

удачным, что в последние годы используется как стандарт для обмена графической информацией в компьютерных сетях. Помимо стандартной шапки файлы типа Metafile состоят из записей переменной длины — упакованных команд обращения к графическим процедурам GDI. В первом слове такой записи хранится ее длина. Во втором слове — порядковый номер функции GDI. В последующих словах в машинном формате заданы аргументы функции. Распаковка и интерпретация этой информации связана с дополнительными затратами машинного времени, однако масштабирование и повороты векторных изображений выполняются без искажений. С помощью методов канвы содержимое метафайлов может быть превращено в растровое изображение на видимых компонентах или формах. Обратная операция по преобразованию растрового изображения в векторное стандартными средствами C++ Builder не поддерживается. Это довольно трудная задача, относящаяся к теории распознавания образов.

Объекты типа `TMetafile` обладают рядом специфических свойств. Во-первых, помимо описания собственно графического изображения в файл можно включить текстовый комментарий (свойство `Description`) и указать его создателей (свойство `CreatedBy`). Для повышения точности воспроизведения кроме обычных параметров типа `Width` и `Height`, задающих ширину и высоту в пикселах, здесь присутствуют аналогичные характеристики в сотых долях миллиметра (свойства `MMWidth` и `MMHeight`). Еще одно свойство `Inch` определяет количество шагов в дюйме. Если в качестве единицы длины в метафайле были использованы твипсы, то `Inch=1440`. Эти параметры более уместны при воспроизведении графических объектов на прецизионных плоттерах.

Свойство `Transparent=true` обеспечивает воспроизведение рисунка с прозрачным фоном. Под таким рисунком просвечивает фон принимающего объекта с ранее созданным изображением.

Объекты типа *TPicture*

Объекты типа `TPicture` выступают в роли универсальных графических контейнеров, в которые можно загружать растровые изображения в форматах .bmp и .ico или векторные изображения — метафайлы. Таким образом, один объект типа `TPicture` может выступать в программе и как `TBitmap`, и как `Ticon`, и как `TMetafile`. Для создания объекта типа `TPicture` используют оператор `new`:

```
TPicture *pic1=new TPicture;
```

В созданный таким образом объект мы можем загрузить графический файл любого формата:

```
pic1→LoadFromFile("c:\\windows\\teach.ico");

pic1→LoadFromFile("c:\\windows\\ОБЛАКА.bmp");

pic1→LoadFromFile("c:\\windows\\system\\spbanner.wmf");
```

Для отображения загруженной картинки в левом верхнем углу формы можно воспользоваться методом Draw, указав в качестве его третьего аргумента обобщенное имя графического объекта:

```
Canvas→Draw(0,0,pic1→Graphic);
```

Независимо от типа загруженного изображения программа копирования разберется с графическим форматом рисунка и правильно воспроизведет его. Однако, загрузку и отображение растрового рисунка в формате .bmp, например, можно выполнить и по-другому, воспользовавшись свойством Bitmap:

```
pic1->Bitmap->LoadFromFile("c:\\windows\\ОБЛАКА.bmp");
Canvas->Draw(0,0,pic1->Bitmap);
```

Для других графических файлов пригодятся имена типа pic1->Icon или pic1->Metafile. Попытка загрузить файл неподобающего формата по такому конкретному имени приведет к аварийной ситуации.

Объекты типа TPicture могут использоваться в процедурах копирования Draw и StretchDraw или участвовать в обменах с Clipboard.

Объект типа *Clipboard*

Буфер обмена (Clipboard) представляет собой универсальное временное хранилище, способное принимать и выдавать текстовые документы, графические изображения в разных форматах и даже компоненты. Для работы с буфером обмена вы должны подключить к своему модулю заголовочный файл Clipbrd.hpp:

```
#include <vcl\Clipbrd.hpp>
```

Как правило, для записи в буфер обмена описаний графических объектов и извлечения их оттуда используется метод Assign:

```
Bitmap1->Assign(Clipboard()); // Запись в буфер обмена
Clipboard()->Assign(Bitmap1); // Извлечение из буфера обмена
```

Чтобы не создавать аварийную ситуацию, перед чтением из буфера обмена следует убедиться в том, что находящееся там изображение имеет нужный формат:

```
if (Clipboard()->HasFormat(CF_xxx)...
```

Аргумент CF_xxx может принимать одно из следующих значений — CF_TEXT, CF_BITMAP, CF_METAFILEPICT, CF_PICTURE или CF_OBJECT. Функция HasFormat принимает значение true, если содержимое буфера имеет формат, соответствующий аргументу.

Экранные и внеэкранные графические объекты

Различия между экранными и внеэкранными графическими объектами демонстрирует следующее приложение.

Разместим на форме три объекта размером 100×100 пикселов, располагающие свойством `Canvas` — `Image1` с началом в точке `(120,0)`, `PaintBox1` с началом в точке `(240,0)` и `StatusBar1` с началом в точке `(360,0)`. Компонент типа `TStatusBar` (строка состояния), на котором обычно никто не рисует, выбран специально для демонстрации универсальности свойства `Canvas`. В процедуре создания формы заведем объект типа `TBitmap`, указатель на который назовем `Fbmp`:

Приложение 8.13 (BCB)

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Fbmp=new Graphics::TBitmap; // Создание Bitmap'a
  Fbmp->Width=100;            // Начальная ширина
  Fbmp->Height=100;           // Начальная высота
}
```

Для того чтобы объект `Fbmp` стал доступным всем методам класса нашей формы, добавим в файл `Unit1.h` следующую строку:

```
........
  class TForm1: public TForm;
{
...........
  private
    Graphics::TBitmap *Fbmp;
...........
};
```

По событию `OnClick` (щелчок по любому свободному месту формы) организуем следующие построения. На каждом из пяти объектов, включая форму, обведем контуры квадрата размером 100×100, залитого цветом своего фона. Затем организуем цикл построения 10 000 случайных точек в каждом из квадратов. По окончании цикла скопируем содержимое `Bitmap` на канву формы, начиная с координаты `(0,120)`.

```
void __fastcall TForm1::Form1Click(TObject *Sender)
{
  int j,x,y;
// Установка цвета фона и построение залитых квадратов
  Canvas->Brush->Color=clGreen;
  Canvas->Rectangle(0,0,100,100);
  Image1->Canvas->Brush->Color=clYellow;
  Image1->Canvas->Rectangle(0,0,100,100);
  PaintBox1->Canvas->Brush->Color=clBlue;
  PaintBox1->Canvas->Rectangle(0,0,100,100);
```

```
  StatusBar1->Canvas->Brush->Color=clRed;
  StatusBar1->Canvas->Rectangle(0,0,100,100);
  Fbmp->Canvas->Brush->Color=clAqua;
  Fbmp->Canvas->Rectangle(0,0,100,100);
// Заполнение квадратов случайными черными точками
  for (j=0;j<10000;j++)
  {
    x=random(100);
    y=random(100);
    Canvas->Pixels[x][y]=clBlack;
    Image1->Canvas->Pixels[x][y]=clBlack;
    PaintBox1->Canvas->Pixels[x][y]=clBlack;
    StatusBar1->Canvas->Pixels[x][y]=clBlack;
    Fbmp->Canvas->Pixels[x][y]=clBlack;
  }
  Canvas->Draw(0,120,Fbmp); // Копирование Bitmap'a на форму
}
```

Не забудем при уничтожении формы освободить ресурсы, выделенные под `Fbmp`:

```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  if(Fbmp!=NULL)delete Fbmp; // Удаление Bitmap'a
}
```

Вместо копирования содержимого `Bitmap` мы могли бы поместить на форму объект, обладающий свойством `Picture` (например — `Image2`). Если это свойство приравнять к указателю на `Fbmp`, то собственником `Bitmap`'а становится видимый графический объект:

```
Image2->Picture->Graphic = Fbmp;
```

Это избавляет нас не только от прямого копирования содержимого `Bitmap`'а, но и от его удаления из памяти. Последнее произойдет автоматически после разрушения собственника.

Запустите приложение и понаблюдайте за процессом построения изображений. После завершения работы (сигналом к этому будет появление изображений, созданных на `Image1` и `Fbmp`) сверните форму в иконку и восстановите ее. Можно подвергнуть окно и другому испытанию, заслонив нашу форму каким-либо другим окном. На все эти манипуляции реагирует операционная система, которая пошлет нашему приложению сообщение `WM_PAINT` с информацией о координатах перекрытой части формы. Поскольку в первом варианте приложения обработчик события `FormPaint` не был предусмотрен, то после активизации нашей формы на ней пропадут перекрытые части изображений, сформированные на канве экранных компонент.

Для восстановления пропавших изображений подключим к нашему приложению обработчик события `OnPaint`, в котором следует повторить формирование содержимого канвы формы, объектов `PaintBox1` и `StatusBar1`, а также процедуру копирования `Fbmp`. При этом необходимо учесть следующие детали. Сообщение операционной системы `WM_PAINT` трансформируется в событие визуальной среды `OnPaint`, и независимо от причины, вызвавшей это событие, управление будет передано обработчику `FormPaint`. Форма перерисует себя (точнее, ту часть, которая была повреждена) и выполнит все действия, запрограммированные в обработчике. Перед возвратом из функции `FormPaint` будут сгенерированы сообщения `OnPaint`, адресованные всем размещенным на форме компонентам. Один из моментов работы описанного приложения приведен на рис. 8.11.

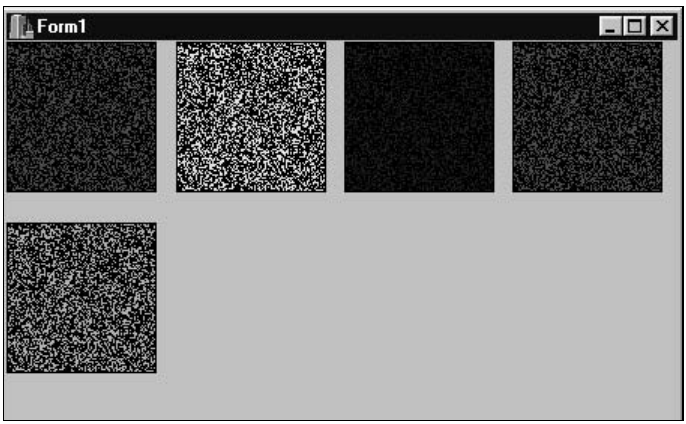


Рис. 8.11. Восстановление изображения на разных объектах

Каждый компонент реагирует на событие `OnPaint` стандартным образом. Если список его событий вообще не включает реакцию на `OnPaint` или пользователем не была предусмотрена соответствующая процедура, то компонент просто перерисовывает себя на форме. Строка состояния, например, восстановит окаймляющую ее рамку с тремя наклонными штрихами в нижнем правом углу, очистит свое рабочее поле и передаст сообщение `OnPaint` всем компонентам, расположенным на `StatusBar`.

Объект `PaintBox`, который не содержит никаких видимых элементов, перерисовку не выполняет, но сообщение `OnPaint` всем своим компонентам пошлет.

Поэтому тело нашего обработчика `FormPaint` начинается с обращения к функции `Update`, которая требует перерисовать все компоненты, расположенные на форме, в самом начале обработки события `FormPaint`. В противном случае наши действия по восстановлению канвы экранных компонент будут перечеркнуты завершающей перерисовкой поврежденных компонент. После работы функции `Update` все видимые элементы экранных компонент

будут восстановлены и повторная их перерисовка производиться не будет. Текст обработчика FormPaint приведен ниже.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  int j,x,y;
  Form1->Update(); // Перерисовка формы и всех ее компонент
// Установка цвета фона и построение залитых квадратов
// экранных компонент
  Canvas->Brush->Color=clGreen;
  Canvas->Rectangle(0,0,100,100);
  PaintBox1->Canvas->Brush->Color=clBlue;
  PaintBox1->Canvas->Rectangle(0,0,100,100);
  StatusBar1->Canvas->Brush->Color=clRed;
  StatusBar1->Canvas->Rectangle(0,0,100,100);
// Заполнение квадратов экранных компонент
// случайными черными точками
  for (j=0;j<10000;j++)
  {
    x=random(100);
    y=random(100);
    Canvas->Pixels[x][y]=clBlack;
    PaintBox1->Canvas->Pixels[x][y]=clBlack;
    StatusBar1->Canvas->Pixels[x][y]=clBlack;
  }
}
```

Приведенный пример объясняет, почему авторы многих книг демонстрируют примеры графических построений в составе обработчика события FormPaint. Им не грозят никакие трансформации формы на экране, т. к. при поступлении сообщения WM_PAINT все построения повторяются заново.

8.5.3. Копирование растровых изображений

Растровые изображения, сформированные на канве одного объекта, могут быть целиком (методы Draw и StretchDraw) или частично (методы CopyRect и BrushCopy) скопированы на канву другого графического объекта.

При копировании части на канве с *исходным изображением* (Source Image) и на *принимающей канве* (Destination Image) должны быть выделены прямоугольные области, границы которых определяются записями типа TRect. Размеры этих областей не обязаны совпадать. Копируемая часть изображения обязательно вписывается в принимающую область с пропорциональным изменением масштаба по каждой координате. Кроме стандартного копирования с сохранением расцветки части исходного изображения (метод

CopyRect) имеется возможность произвести замену одного из цветов исходного изображения на цвет фона (Brush->Color) принимающего объекта (метод BrushCopy). Если указанный цвет в исходном изображении отсутствует, то цветовая коррекция не производится.

При копировании изображения целиком в принимающем объекте либо указывается положение верхнего левого угла (метод Draw), либо ссылка на прямоугольную область типа TRect, в которую объект вписывается целиком (метод StretchDraw). В последнем случае объект обязательно подвергается пропорциональной деформации.

Приложение 8.14 (BCB)

Приведенное ниже приложение демонстрирует работу всех методов копирования. Для его работы сформируем исходное изображение на Bitmap'е размером 200×200 пикселов в виде красного прямоугольника чуть меньшего размера с закругленными краями на зеленом фоне (толщина линии — 5 пикселов).

В верхней полосе выделим три прямоугольные области — R1 (размером 100×100), R2 (размером 150×100) и R3 (размером 50×50) для стандартного копирования части объекта (область R0 размером 100×100, расположенная в верхнем левом углу исходного изображения), инициируемого нажатием кнопки с надписью CopyRect.

В средней полосе разместим три прямоугольных области R4, R5 и R6 аналогичных размеров для демонстрации замены цветов по методу BrushCopy.

Наконец, в нижней полосе выделим область R7 для копирования объекта целиком, что произойдет после щелчка по кнопке с надписью Draw. Рядом расположим в два этажа области R8 и R9 для копирования объекта целиком с деформацией по одной и обеим координатам.

В каких-либо дополнительных комментариях приведенный ниже текст программы не нуждается.

```
// ---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
Graphics::TBitmap *Bitmap; // Объявление графического объекта
TRect R0=Rect(0,0,100,100);// Объявление области в объекте
// ---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
```

```
// Создание графического объекта и его заполнение
  Bitmap = new Graphics::TBitmap;
  Bitmap->Width=200;
  Bitmap->Height=200;
  Bitmap->Canvas->Pen->Color=clRed;
  Bitmap->Canvas->Pen->Width=5;
  Bitmap->Canvas->Brush->Color=clGreen;
  Bitmap->Canvas->RoundRect(10,10,190,190,10,10);
// Установка цвета фона принимающего объекта
  Form1->Canvas->Brush->Color=clBlack;
}
// ---------------------------------------------------------------------
void __fastcall TForm1::btCopyRectClick(TObject *Sender)
{
  TRect R1=Rect(0,0,100, 100);
  TRect R2=Rect(110,0,260, 100);
  TRect R3=Rect(270,50,320,100);
// Копирование части изображения без деформации
  Form1->Canvas->CopyRect(R1, Bitmap->Canvas, R0);
// Копирование части изображения с растягиванием по оси x
  Form1->Canvas->CopyRect(R2, Bitmap->Canvas, R0);
// Копирование части изображения с деформацией по осям x и y
  Form1->Canvas->CopyRect(R3, Bitmap->Canvas, R0);
}
// ---------------------------------------------------------------------
void __fastcall TForm1::btBrushCopyClick(TObject *Sender)
{
  TRect R4=Rect(0,110,100,210);
  TRect R5=Rect(110,110,260,210);
  TRect R6=Rect(270,160,320,210);
// Копирование части изображения с заменой красного цвета на черный
  Form1->Canvas->BrushCopy(R4,Bitmap,R0,clRed);
// Копирование части изображения с заменой зеленого цвета на черный
  Form1->Canvas->BrushCopy(R5,Bitmap,R0,clGreen);
// Копирование части изображения с заменой несуществующего цвета
  Form1->Canvas->BrushCopy(R6,Bitmap,R0,clYellow);
}
// ---------------------------------------------------------------------
void __fastcall TForm1::btDrawClick(TObject *Sender)
{
  TRect R7=Rect(0,220,200,420);
// Копирование изображения целиком без деформации
```

```
  Form1->Canvas->Draw(0,220, Bitmap);
}
// ---------------------------------------------------------------
void __fastcall TForm1::btStretchDrawClick(TObject *Sender)
{
  TRect R8=Rect(210,220,360,320);
  TRect R9=Rect(270,340,320,390);
// Копирование изображения целиком с деформацией по оси x
  Form1->Canvas->StretchDraw(R8,Bitmap);
// Копирование изображения целиком с деформацией по осям x и y
  Form1->Canvas->StretchDraw(R9,Bitmap);
}
// ---------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
// Уничтожение графического объекта
  delete Bitmap;
}
// ---------------------------------------------------------------
```

Образец работы этого приложения показан на рис. 8.12

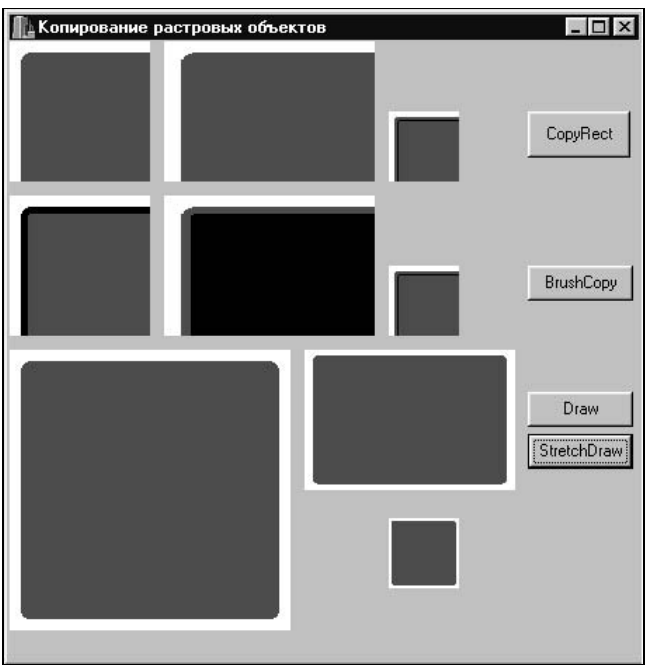


Рис. 8.12. Разные способы копирования растровых изображений

В процессе копирования растровых изображений имеется возможность управлять раскраской принимающего объекта с помощью свойства его канвы `CopyMode`. Смысл такого управления напоминает взаимодействие цвета пера (свойство `PenMode`) с предшествующим цветом пиксела в области рисования. В случае копирования речь идет о взаимодействии пикселов принимающего объекта с пикселами источника. По умолчанию изображение источника заменяет содержимое соответствующей прямоугольной области получателя (`CopyMode=cmSrcCopy`). В табл. 8.8 приведены возможные варианты режима копирования.

Таблица 8.8. *Режимы копирования в зависимости от символьной константы, доступной для* `CopyMode`

`CopyMode`	**Способ формирования области получателя**
`cmBlackness`	Область канвы получателя заливается черным цветом
`cmWhiteness`	Область канвы получателя заливается белым цветом
`cmSrcCopy`	Источник без изменения переносится в область получателя
`cmNotSrcCopy`	Инверсия источника переносится в область получателя
`cmPatCopy`	На канву получателя копируется фон (кисть) источника
`cmPatPaint`	Логическая сумма (`OR`) инвертированного изображения источника, его фона и изображения получателя
`cmPatInver`	Логическая сумма (`XOR`) фона источника и изображения получателя
`cmDstInvert`	Инверсия изображения получателя (источник игнорируется)
`cmMergeCopy`	Логическое умножение (`AND`) изображений источника и получателя
`cmSrcAnd`	Логическое умножение (`AND`) изображений источника и получателя
`cmMergePaint`	Логическая сумма (`OR`) инверсии источника и изображения получателя
`cmNotSrcErase`	Инверсия логической суммы (`OR`) источника и получателя
`cmSrcErase`	Логическое умножение (`AND`) изображения источника и инверсии изображения получателя
`cmSrcInvert`	Логическая сумма (`XOR`) изображений источника и получателя
`cmSrcPaint`	Логическая сумма (`OR`) изображений источника и получателя

8.6. Создание монотонно изменяющегося фона

Установка большинства программных продуктов фирмы Microsoft происходит на красивом фоне монотонно изменяющегося синего цвета вплоть до черного оттенка.

Приложение 8.15 (BCB)

Продемонстрируем технику построения такого фона, начальный цвет которого можно выбрать из всплывающего меню (рис. 8.13). Разместим на форме единственный компонент `PopupMenu1` и с помощью *Редактора Меню* (**Menu Editor**) сформируем в меню семь следующих строк:

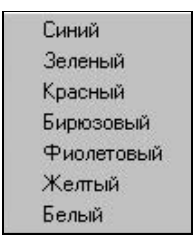


Рис. 8.13. Меню для выбора цвета

Для каждого элемента в меню присвоим свойству `Tag` числовой код из диапазона [1,7], определяющий своими двоичными разрядами присутствие компонент красного (`r`), зеленого (`g`) и синего (`b`) в формировании соответствующего цвета (табл. 8.9)

Таблица 8.9. *Цвет в зависимости от численного значения, присвоенного значению* `Tag`

Цвет	`Tag`	Комбинация RGB-компонент		
		Красный	Зеленый	Синий
Синий	1	0	0	1
Зеленый	2	0	1	0
Красный	4	1	0	0
Бирюзовый	3	0	1	1
Фиолетовый	5	1	0	1
Желтый	6	1	1	0
Белый	7	1	1	1

Для обработки щелчков по тому или иному цвету в меню создадим общий обработчик, который анализирует значение свойства `Tag` у объекта-инициатора события `OnClick` и формирует значения глобальных переменных — `r`, `g`, `b`:

```
void __fastcall TForm1::N1Click(TObject *Sender)
{
  TMenuItem *col=(TMenuItem *)Sender;
  int k=col->Tag;
  b=k % 2; k=(k-b)/2;
  g=k % 2; k=(k-g)/2;
  r=k;
  FormPaint(Sender);
}
```

Общий обработчик приводит объект-инициатор события (`Sender`) к типу `MenuItem`, извлекает его свойство `Tag` и переводит его в двоичную систему. Разряды запоминаются в глобальных переменных `r`, `g` и `b`, которые участвуют в образовании кода цвета кисти. После этого инициируется событие `OnPaint` и в действие вступает следующий обработчик:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 char c;
 int j_max=ClientHeight;
 for(int j=0; j<j_max; j++)
   {
     c=255*(1.-(double)j/j_max);
     Canvas->Brush->Color=TColor(RGB(c*r,c*g,c*b));
     Canvas->FillRect(Rect(0,j,ClientWidth,j+1));
   }
}
```

Устроен он крайне просто — в цикле пробегаются строки области клиента сверху донизу и каждый раз прямоугольник, образованный текущей и следующей строкой закрашивается цветом с переменно убывающей интенсивностью `c`. Результат работы этого приложения можно продемонстрировать только в черно-белой палитре (рис. 8.14).

Рис. 8.14. Монотонно изменяющийся фон

8.7. Формирование регулярных заливок замкнутых областей

Штатные средства Borland C++ Builder предоставляют пользователю всего 8 типовых шаблонов — стандартные значения свойства `BrushStyle` (`bsSolid`, `bsClear`, ...). Кроме них имеется возможность использовать любой шаблон размером 8×8 пикселов в формате bmp-файла, который в нужный момент загружается с помощью метода `LoadFromFile` в свойство `Brush->Bitmap`. Не так уж и сложно создать один такой файл или небольшое их количество с помощью утилиты Image Editor. Тем не менее, в формате 256 цветов размер такого шаблона составляет 1 142 байта, т. к. в нем же хранится и вся палитра, занимающая 256×4 байт. И в каждом шаблоне одна и та же палитра будет дублироваться. Формат *truecolor* (24 бит на пиксел) в данном случае более экономичен, он требует всего 246 байт на шаблон (54 байта — шапка и 3×64 — собственно цвета пикселов). Если приложение связано с использованием большого количества таких шаблонов, да еще и разного цвета, то формирование дисковых файлов превращается в утомительное занятие и сильно захламляет каталоги. Кроме того, Image Editor не рассчитан на изготовление bmp-масок в формате *truecolor*.

Приложение 8.16 (BCB)

В описываемое ниже приложение включен автономный модуль `PenBrush.cpp`, предоставляющий пользователю возможность манипулиро-

вать с сотнями и даже тысячами заливочных шаблонов, формируемых непосредственно в оперативной памяти. Перечень функций этого модуля и их назначение приведены в табл. 8.10.

Таблица 8.10. *Описание функций модуля* `PenBrush.cpp`

Формат вызова	Назначение функции
`InitPenBrush(pCGO)`	Инициализация модуля (`pCGO` — указатель на канву графического объекта)
`ClosePenBrush()`	Возврат ресурсов
`SetBrush(col,num_pat)`	Установка параметров кисти (`col` — номер цвета, `num_pat` — номер шаблона)
`GetBrush(col,num_pat)`	Опрос текущих установок кисти
`RestoreBrush()`	Восстановление предыдущих установок кисти
`SetPen(col,style,width)`	Установка параметров пера (`col` — номер цвета, `style` — номер стиля, `width` — ширина в пикселах)
`GetPen(col,style,width)`	Опрос текущих установок пера
`RestorePen()`	Восстановление предыдущих установок пера

Выделение этих функций в автономный модуль объясняется желанием построить на его базе библиотеку, которая может быть подключена к любому приложению. Некоторые функции модуля `PenBrush.cpp`, приведенного ниже, нуждаются в комментариях.

Обилие статических переменных в начале модуля объясняется желанием сократить количество передаваемых параметров между функциями (переменные, объявленные как `static`, считаются в пределах модуля глобальными, но в других модулях они недоступны). Смысл каждой глобальной переменной указан в комментарии.

Массив заливочных шаблонов заимствован из нашей первой книги. При желании вы можете сохранить только интересующие вас шаблоны, не забыв изменить при этом значение `n_brush`. Точно также вы можете поступить и с массивом допустимых цветов.

```
// Модуль PenBrush.cpp
#include <vcl.h>
#pragma hdrstop

#include "PenBrush.h"

// ---------------------------------------------------------------
```

```
#pragma package(smart_init)
// ---------------------------------------------------------------
static TCanvas *pC;  // указатель на канву графического объекта
static Graphics::TBitmap *BmpBrush; // bitmap для кисти

static int old_col_pen=0;   // предыдущий цвет пера
static int old_wid_pen=1;   // предыдущая ширина пера (в пикселах)
static int old_num_pen=0;   // предыдущий номер (стиль) пера

static int cur_col_pen=0;   // текущий цвет пера
static int cur_wid_pen=1;   // текущая ширина пера (в пикселах)
static int cur_num_pen=0;   // текущий номер (стиль) пера

static int old_col_brush=0; // предыдущий цвет кисти
static int old_num_brush=0; // предыдущий номер (стиль) кисти

static int cur_col_brush=0; // текущий цвет кисти
static int cur_num_brush=0; // текущий номер (стиль) кисти

static int n_col=25;        // максимальное количество цветов в палитре
static int n_brush=111;     // максимальное количество шаблонов

// Массив заливочных шаблонов 8x8
static unsigned char pattern[][8]={
// Сплошная заливка
  {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
// 0=очистка
  {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF},
// 1=сплошная заливка
// ==== Штриховка сплошными линиями

// по горизонтали
  {0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
// 2=горизонтали (1+7)
  {0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0x00},
// 3=горизонтали (2+6)
  {0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00},
// 4=горизонтали (3+5)
  {0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00},
```

```
// 5=горизонтали (4+4)
  {0xFF,0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00},
// 6=горизонтали (5+3)
  {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,0x00},
// 7=горизонтали (6+2)
  {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00},
// 8=горизонтали (7+1)
  {0xFF,0x00,0xFF,0x00,0xFF,0x00,0xFF,0x00},
// 9=горизонтали (1+1)
  {0xFF,0x00,0x00,0x00,0xFF,0x00,0x00,0x00},
// 10=горизонтали (1+3)
  {0xFF,0xFF,0xFF,0x00,0xFF,0xFF,0xFF,0x00},
// 11=горизонтали (3+1)
  {0xFF,0xFF,0x00,0x00,0xFF,0xFF,0x00,0x00},
// 12=горизонтали (2+2)

// по вертикали
  {0x80,0x80,0x80,0x80,0x80,0x80,0x80,0x80},
// 13=вертикали (1+7)
  {0x88,0x88,0x88,0x88,0x88,0x88,0x88,0x88},
// 14=вертикали (1+3)
  {0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0},
// 15=вертикали (2+6)
  {0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0},
// 16=вертикали (3+5)
  {0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0,0xF0},
// 17=вертикали (4+4)
  {0xF8,0xF8,0xF8,0xF8,0xF8,0xF8,0xF8,0xF8},
// 18=вертикали (5+3)
  {0xFC,0xFC,0xFC,0xFC,0xFC,0xFC,0xFC,0xFC},
// 19=вертикали (6+2)
  {0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFE},
// 20=вертикали (7+1)
  {0xAA,0xAA,0xAA,0xAA,0xAA,0xAA,0xAA,0xAA},
// 21=вертикали (1+1)
  {0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE},
// 22=вертикали (3+1)
  {0xCC,0xCC,0xCC,0xCC,0xCC,0xCC,0xCC,0xCC},
// 23=вертикали (2+2)

  // прямоугольная двойная штриховка
```

```
  {0xFF,0x80,0x80,0x80,0x80,0x80,0x80,0x80},
// 24=сетка (1+7,1+7)
  {0xFF,0xFF,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0},
// 25=сетка (2+6,2+6)
  {0xFF,0xFF,0xFF,0xE0,0xE0,0xE0,0xE0,0xE0},
// 26=сетка (3+5,3+5)
  {0xFF,0xFF,0xFF,0xFF,0xF0,0xF0,0xF0,0xF0},
// 27=сетка (4+4,4+4)
  {0xFF,0xFF,0xFF,0xFF,0xFF,0xF8,0xF8,0xF8},
// 28=сетка (5+3,5+3)
  {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFC,0xFC},
// 29=сетка (6+2,6+2)
  {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFE},
// 30=сетка (7+1,7+1)
  {0xFF,0xFF,0x99,0x99,0xFF,0xFF,0x99,0x99},
// 31=сетка (2+2,2+2)

  // по диагонали
  {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80},
// 32=справа налево (1+10)
  {0x83,0x07,0x0E,0x1C,0x38,0x70,0xE0,0xC1},
// 33=справа налево (3+7)
  {0xC7,0x8F,0x1F,0x3E,0x7C,0xF8,0xF1,0xE3},
// 34=справа налево (6+3)
  {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01},
// 35=слева направо (1+10)
  {0xC1,0xE0,0x70,0x38,0x1C,0x0E,0x07,0x83},
// 36=слева направо (3+7)
  {0xE3,0xF1,0xF8,0x7C,0x3E,0x1F,0x8F,0xC7},
// 37=слева направо (6+3)

// косоугольная двойная штриховка
  {0x81,0x42,0x24,0x18,0x18,0x24,0x42,0x81},
// 38=штриховка (1+6,1+6)

// ==== Штриховка штриховыми линиями

// по горизонтали
  {0xF0,0x00,0x00,0x00,0x0F,0x00,0x00,0x00},
```

```
// 39=штрихи (4*1+4,4) в шашечку
  {0xF8,0x00,0x00,0x00,0x1F,0x00,0x00,0x00},
// 40=штрихи (5*1+3,4) в шашечку
  {0xF0,0xF0,0x00,0x00,0x0F,0x0F,0x00,0x00},
// 41=штрихи (4*2+4,4) в шашечку
  {0xF8,0xF8,0x00,0x00,0x1F,0x1F,0x00,0x00},
// 42=штрихи (5*2+3,4) в шашечку
  {0xF0,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
// 43=редкие штрихи (4*1+4,7)
  {0xFF,0x00,0x00,0x00,0xF0,0x00,0x00,0x00},
// 44=сплошная и штрихи

// по вертикали
  {0x80,0x80,0x80,0x80,0x00,0x00,0x00,0x00},
// 45=редкие штрихи (4*1+4,7)
  {0x80,0x80,0x80,0x80,0x08,0x08,0x08,0x08},
// 46=штрихи (4*1+4,4) в шашечку

// ==== Волнообразные линии
// по горизонтали
  {0xF0,0x0F,0x00,0x00,0x00,0x00,0x00,0x00},
// 47=с небольшой амплитудой (1)
  {0xF0,0xFF,0x0F,0x00,0x00,0x00,0x00,0x00},
// 48=с небольшой амплитудой (2)
  {0x81,0x42,0x24,0x18,0x00,0x00,0x00,0x00},
// 49=с большой амплитудой (1)
  {0x81,0xC3,0x66,0x3C,0x18,0x00,0x00,0x00},
// 50=с большой амплитудой 2п

// по диагонали
  {0x10,0x10,0x10,0xF0,0x01,0x01,0x01,0x0F},
// 51=слева направо

// ==== Растровые сетки
  {0x00,0x55,0x00,0xAA,0x00,0x55,0x00,0xAA},
// 52=густые точки в шашечку (1+1,1)
  {0x00,0x44,0x00,0x11,0x00,0x44,0x00,0x11},
// 53=густые точки в шашечку (1+3,2)
  {0x22,0x00,0x00,0x00,0x88,0x00,0x00,0x00},
```

```
// 54=мелкие (1 пиксел) редкие точки в шашечку
  {0x00,0x66,0x66,0x00,0x00,0x99,0x99,0x00},
// 55=средние (2x2) точки в шашечку
  {0x00,0x00,0x30,0x30,0x00,0x00,0x03,0x03},
// 56=средние (2x2) редкие точки в шашечку
  {0x00,0x00,0x00,0x18,0x18,0x00,0x00,0x00},
// 57=средние (2x2) очень редкие точки в шашечку
  {0x00,0x00,0x3C,0x3C,0x3C,0x3C,0x00,0x00},
// 58=крупные (4x4) точки
  {0xCC,0x33,0xCC,0x33,0xCC,0x33,0xCC,0x33},
// 59=густая в шахматном порядке
  {0xAA,0x55,0xAA,0x55,0xAA,0x55,0xAA,0x55},
// 60=очень густая в шахматном порядке

// ==== Фигуры в узлах прямоугольной сетки
  {0x10,0x10,0x7C,0x10,0x10,0x00,0x00,0x00},
// 61=крестики
  {0x10,0x00,0x54,0x00,0x10,0x00,0x00,0x00},
// 62=пунктирные крестики
  {0x00,0x00,0x18,0x3C,0x3C,0x18,0x00,0x00},
// 63=жирные кресты
  {0x00,0x18,0x3C,0x7E,0x7E,0x3C,0x18,0x00},
// 64=сетка из больших ромбиков
  {0x00,0x7E,0x7E,0x7E,0x7E,0x7E,0x7E,0x00},
// 65=сетка из больших квадратов
  {0xF8,0xF8,0xF8,0xF8,0xF8,0x00,0x00,0x00},
// 66=залитые квадраты 5x5
  {0xFC,0xFC,0xFC,0xFC,0xFC,0xFC,0x00,0x00},
// 67=залитые квадраты 6x6
  {0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0x00},
// 68=залитые квадраты 7x7
  {0x00,0x7E,0x42,0x42,0x42,0x42,0x7E,0x00},
// 69=отдельные большие квадратики
  {0x00,0x00,0x3C,0x24,0x24,0x3C,0x00,0x00},
// 70=отдельные маленькие квадраты
  {0x00,0x7E,0x42,0x5A,0x5A,0x42,0x7E,0x00},
// 71=квадратики с точками в центре
  {0x00,0x42,0x24,0x18,0x18,0x24,0x42,0x00},
```

```
// 72=символ "x"
  {0x00,0x44,0x28,0x10,0x10,0x10,0x00,0x00},
// 73=символ "y"
  {0x00,0x42,0x24,0x18,0x00,0x00,0x00,0x00},
// 74=галочки
  {0x00,0x00,0x00,0x18,0x24,0x42,0x00,0x00},
// 75=перевернутые галочки
  {0x10,0x38,0x54,0x10,0x10,0x10,0x00,0x00},
// 76=тонкие стрелки вверх
  {0x10,0x10,0x10,0x54,0x38,0x10,0x00,0x00},
// 77=тонкие стрелки вниз
  {0x00,0x20,0x40,0xFC,0x40,0x20,0x00,0x00},
// 78=тонкие стрелки влево
  {0x00,0x10,0x08,0xFC,0x08,0x10,0x00,0x00},
// 79=тонкие стрелки вправо
  {0x00,0x08,0x18,0x3F,0x3F,0x18,0x08,0x00},
// 80=жирные стрелки влево
  {0x20,0x70,0xA8,0x22,0x22,0x8A,0x07,0x02},
// 81=двойные стрелки по вертикали
  {0x24,0x02,0x1F,0x02,0x24,0x40,0xF8,0x40},
// 82=двойные стрелки по горизонтали
  {0x00,0x10,0x28,0x44,0x28,0x10,0x00,0x00},
// 83=контуры ромбиков по сетке
  {0x81,0x18,0x3C,0x7E,0x7E,0x3C,0x18,0x81},
// 84=залитые ромбики и точки
  {0x18,0x3C,0x3C,0x7E,0x7E,0x3C,0x3C,0x18},
// 85=залитые ромбики по вертикали
  {0x00,0x3C,0x42,0x42,0x42,0x42,0x3C,0x00},
// 86=кружочки на сетке (d=6)
  {0x44,0x38,0x00,0x00,0x00,0x38,0x44,0x44},
// 87=кружочки на сетке  (d=5)
  {0x80,0x7F,0x41,0x41,0x41,0x41,0x41,0x7F},
// 88=незалитые квадратики
  {0x08,0x00,0x08,0x5D,0x08,0x00,0x08,0x00},
// 89=точечная сетка с точкам в узлах

// ==== Фигуры в шахматном порядке

  {0x00,0x18,0x3C,0x7E,0x7E,0x3C,0x18,0x00},
```

```
// 90=ромбики в шашечку
  {0xF0,0xF0,0xF0,0xF0,0x0F,0x0F,0x0F,0x0F},
// 91=густые мелкие квадратики (4x4)в шашечку
  {0xCC,0x33,0xCC,0x33,0xCC,0x33,0xCC,0x33},
// 92=более крупная шахматная сетка
  {0x42,0xC3,0x3C,0x24,0x24,0x3C,0xC3,0x42},
// 93=контуры квадратиков в шахматном порядке

  // ==== Узоры

  {0x00,0xFB,0xFB,0xFB,0x00,0xDF,0xDF,0xDF},
// 94=залитые кирпичики по горизонтали
  {0x00,0xFB,0x0A,0xFB,0x00,0xDF,0x50,0xDF},
// 95=незалитые кирпичики по горизонтали
  {0xEE,0xEE,0x0E,0xEE,0xEE,0xE0,0xEE,0xEE},
// 96=залитые кирпичики по вертикали
  {0x01,0x82,0x44,0x28,0x10,0x20,0x40,0x80},
// 97=незалитые кирпичи по диагонали
  {0x80,0x41,0x22,0x14,0x08,0x04,0x02,0x01},
// 98=незалитые кирпичи по диагонали
  {0x94,0x84,0x48,0x30,0x00,0xC1,0x22,0x14},
// 99=волнистый узор
  {0xFF,0x01,0x7D,0x45,0x5D,0x41,0x7F,0x00},
// 100=спиральный узор
  {0x00,0xEC,0x2A,0x2A,0x2A,0x2A,0xEC,0x00},
// 101=узор 1
  {0x00,0x00,0x7E,0x42,0x7E,0x42,0x7E,0x42},
// 102=узор 2
  {0x00,0x5D,0x3E,0x6D,0x7F,0x63,0x36,0xF0},
// 103=узор 3
  {0x92,0x24,0x49,0x92,0x24,0x49,0x92,0x24},
// 104=узор 4
  {0xB1,0x22,0x14,0x14,0x22,0x91,0x48,0x24},
// 105=узор 5
  {0x02,0x91,0x68,0x08,0x10,0x16,0x89,0x40},
// 106=узор 6
  {0xC3,0x42,0x5A,0x7E,0x7E,0x5A,0x42,0xC3},
// 107=узор 7
  {0x03,0x18,0x60,0x80,0x80,0x40,0x20,0x18},
```

```
// 108=узор 8
  {0xBF,0xA0,0xA0,0xBF,0xFB,0x0A,0x0A,0xFB},
// 109=узор 9
  {0xFF,0x81,0xBD,0xA5,0xA5,0xBD,0x81,0xFF},
// 110=узор 10
  {0xFF,0x81,0x81,0x99,0x99,0x81,0x81,0xFF}};
// 111=узор 11

// Шкала цветов типа truecolor (24 бита на пиксел)
static int col[]={
                0x0,    // 0=Черный
           0x3F3F3F,    // 1=темно-серый
           0xB2B2B2,    // 2=серый
           0xCCCCCC,    // 3=светло-серый
           0xFFFFFF,    // 4=белый
           0x0000FF,    // 5=ярко-красный
           0x0000E5,    // 6=красный
           0x0000CC,    // 7=темно-красный
           0xFF0000,    // 8=ярко-синий
           0xE50000,    // 9=синий
           0xCC0000,    // 10=темно-синий
           0x00FF00,    // 11=ярко-зеленый
           0x00E500,    // 12=зеленый
           0x00CC00,    // 13=темно-зеленый
           0x00FFFF,    // 14=ярко-желтый
           0x00E5E5,    // 15=желтый
           0x00CCCC,    // 16=темно-желтый
           0xFFFF00,    // 17=ярко-голубой
           0xE5E500,    // 18=голубой
           0xCCCC00,    // 19=темно-голубой
           0xFF00FF,    // 20=ярко-фиолетовый
           0xE500E5,    // 21=фиолетовый
           0xCC00CC,    // 22=темно-фиолетовый
           0x0066FF,    // 23=оранжевый
           0x000080,    // 24=темно-коричневый
           0x1fffffff   // 25=clNone
           };
// ------------------------------------------------------------
void InitPenBrush(TCanvas *pCGO)
```

```
{// Инициализация области рисования
// pCGO — указатель на канву области рисования
{
  pC=pCGO;
}
// ---------------------------------------------------------------
void ClosePenBrush()
{// Завершение работы
  if(BmpBrush!=NULL)
     delete BmpBrush; // Освобождение ресурсов
}
// ---------------------------------------------------------------
void fill_area(int *xy,int n)
// Заливка области, ограниченной точками xy (n-число точек)
{
  pC->Polygon((TPoint*)xy,n-1);
}
// ---------------------------------------------------------------
void GetBrush(int& color, int& num_pat)
{// Опрос текущих параметров кисти
  color=cur_col_brush;
  num_pat=cur_num_brush;
}
// ---------------------------------------------------------------
void SetBrush(int color, int num_pat)
{// Установка новых параметров кисти
// Шкала для формирования пикселов в bitmap'e
  unsigned char scale[8]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
  Byte *pBmp;
  unsigned char *plin,k,p;
// Формирование параметров bitmap для кисти
  if(num_pat<0 || num_pat>n_brush ||
                  color<0 || color>n_col)
    { ShowMessage("Кисть с такими параметрами недоступна");
      return;
    }
  if(BmpBrush!=NULL)delete BmpBrush; // Удаление предыдущего bitmap
  BmpBrush=new Graphics::TBitmap();  // Запрос памяти
  BmpBrush->Width=8;                 // Ширина = 8 пикселов
```

```
  BmpBrush->Height=8;                // Высота = 8 пикселов
  BmpBrush->PixelFormat=pf24bit;     // Формат = 24 бита на пиксел
// формирование битов цветности в bitmap
  for(int y=7; y>=0;y--)
    {
      pBmp=(unsigned char *)BmpBrush->ScanLine[y];
      plin=&pattern[num_pat][y];
      for(k=0;k<8;k++)
        {
          if(*plin & scale[k])
            { p=3*k;
              pBmp[p]  =(col[color]>>16)&0xFF;
              pBmp[p+1]=(col[color]>>8)&0xFF;
              pBmp[p+2]= col[color]&0xFF;
            }
        }
    }
  pC->Brush->Style=bsClear;
  pC->Brush->Bitmap=BmpBrush;
  old_col_brush=cur_col_brush;
  old_num_brush=cur_num_brush;
  cur_col_brush=color;
  cur_num_brush=num_pat;
}
// ----------------------------------------------------------------
void RestoreBrush()
{// Восстановление параметров кисти
  set_brush(old_col_brush,old_num_pen);
}
// ----------------------------------------------------------------
void GetPen(int& color,int& style,int& width)
{// Опрос текущих параметров пера
  color=cur_col_pen;
  style=cur_num_pen;
  width=cur_wid_pen;
}
// ----------------------------------------------------------------
void SetPen(int color,int style,int width)
{// Установка параметров пера
```

```
  if(color>n_col || style>7)
    { ShowMessage("Перо с такими параметрами недоступно");
      return; }
  pC->Pen->Color=(TColor)col[color];
  pC->Pen->Style=(TPenStyle)style;
  pC->Pen->Width=width;
  old_col_pen=cur_col_pen;
  old_num_pen=cur_num_pen;
  old_wid_pen=cur_wid_pen;
  cur_col_pen=color;
  cur_num_pen=style;
  cur_wid_pen=width;
}
// ---------------------------------------------------------------
void RestorePen()
{// Восстановление параметров пера
  set_pen(old_col_pen,old_num_pen,old_wid_pen);
}
```

Инициализация модуля сводится к переносу в статическую переменную `pC` указателя на канву графического объекта, на поверхности которого будут работать выбранные перо и кисть.

Функция установки новых параметров пера особых тонкостей не содержит, кроме преобразования целочисленных значений цвета и стиля к типам соответствующих свойств.

Наибольший интерес представляет функция установки параметров кисти. Для того чтобы создать растровый образ кисти в оперативной памяти, мы должны завести объект типа `Graphics::TBitmap` (класс `Graphics` добавлять надо, т. к. существует еще тип данных `Windows::Bitmap`). Выделив память под объект `BmpBrush`, мы задаем его высоту и ширину (8×8 пикселов), а также устанавливаем глубину цвета — 24 бита на пиксел. Стандартная шапка объекта `BmpBrush`, содержащая 54 байта, формируется системой автоматически. Однако нам предстоит заполнить каждые 24 бита, характеризующие цвет очередного пиксела. В шаблоне каждому окрашиваемому пикселу соответствует единичный двоичный разряд. С помощью свойства `ScanLine` мы устанавливаем указатель `pBmp` на очередную порцию из 24 бит в растре объекта `BmpBrush`, проверяем бит шаблона и, в случае необходимости, заносим тройку байт из значения `color` в строку `bitmap`. Единственный нюанс, который при этом приходится учитывать, заключается в том, что первой строке шаблона соответствует последняя строка `bitmap`. И поэтому заполнение ведется с конца.

Перед установкой сформированного значения bitmap в свойство `Brush->Bitmap` необходимо сбрасывать стиль кисти (`Brush->Style=bsClear`).

Для отображения залитой области используется метод `Canvas->Brush->Polygon` и единственная посредническая функция процедуры `fill_area` заключается в преобразовании типа данных из массива координат в массив точек и коррекции длины массива.

Включение модуля `PenBrush.cpp` в состав приведенного ниже приложения позволяет протестировать все варианты заливок. Один из фрагментов работы приложения показан на рис. 8.15.

```
// ---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "test_pz.h"
#include "disp_pz.h"
// ---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;
int xy[]={100,10,250,110,150,210,50,110}; // контур, в пикселах
int nxy=4; // число точек в контуре
int col_brush=0;
int num_brush=0;

// ---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  InitPenBrush(Image1->Canvas);
  SetPen(10,6,1);
}
// ---------------------------------------------------------------
void __fastcall TForm1::FillClick(TObject *Sender)
{// Заливка области
  fill_area(xy,nxy);
}
// ---------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
TCloseAction &Action)
{
  ClosePenBrush();
}
// ---------------------------------------------------------------
```

```
void __fastcall TForm1::ClearClick(TObject *Sender)
{// Очистка области
  SetBrush(0,0);
  fill_area(xy,nxy);
  SetBrush(col_brush,num_brush);
}
// ---------------------------------------------------------------
void __fastcall TForm1::ColBrushClick(TObject *Sender)
{// Изменение цвета кисти
  col_brush++;
  if(col_brush>24)col_brush=0;
  Label2->Caption=col_brush;
  SetBrush(col_brush,num_brush);
  fill_area(xy,nxy);
}
// ---------------------------------------------------------------
void __fastcall TForm1::NumBrushClick(TObject *Sender)
{// Изменение номера кисти
  num_brush++;
  if(num_brush>111)num_brush=0;
  Label1->Caption=num_brush;
  SetBrush(col_brush,num_brush);
  fill_area(xy,nxy);
}
```

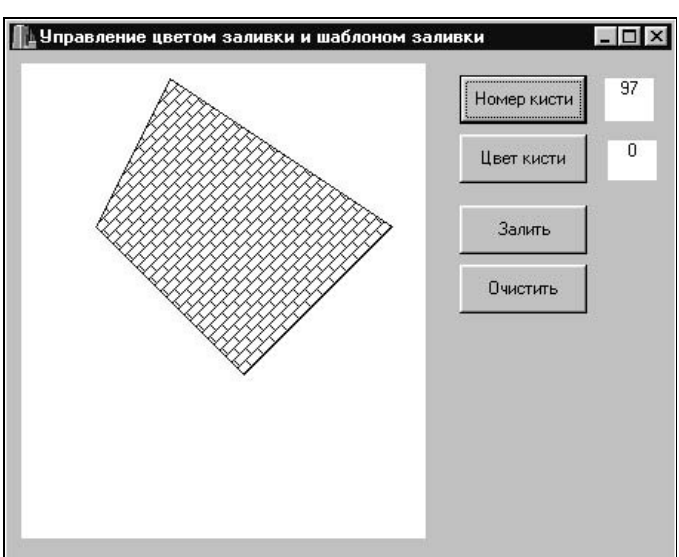


Рис. 8.15. Применение нестандартных шаблонов заливки

Приложение 8.17 (Delphi)

Несмотря на достаточно большой объем приведенного выше приложения и тесные родственные связи между BCB-приложениями и Delphi-приложениями, мы приведем реализацию этой же программы на Паскале. Переписывание операторов один в один здесь не везде проходит, да и разбиение программы на два автономные модуля вызывает определенные трудности. Кроме того, система Delphi 6.0 не поддерживает глубину цвета в 24 бита, ближайший подходящий вариант связан с 32-битной кодировкой кода цветности каждого пиксела (хотя, на самом деле, старший байт в RGB-палитре пока не задействован). Обращение к методу `Polygon` здесь требует открытого массива. В модуле Unit2.pas (аналог модуля PenBrush.cpp) пришлось добавить имена дополнительных модулей (недостача была обнаружена путем исключения неразрешенных ссылок).

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls, ExtCtrls,Unit2;

type
  TForm1 = class(TForm)
    Image1: TImage;
    NumBrush: TButton;
    ColBrush: TButton;
    Fill: TButton;
    Clear: TButton;
    Label1: TLabel;
    Label2: TLabel;
    procedure NumBrushClick(Sender: TObject);
    procedure ColBrushClick(Sender: TObject);
    procedure FillClick(Sender: TObject);
    procedure ClearClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var
  Form1: TForm1;
  col_brush:integer=0;
  num_brush:integer=0;
  xy:array [0..3] of TPoint;
  nxy:integer=4; {число точек в контуре}

implementation

{$R *.dfm}

procedure TForm1.NumBrushClick(Sender: TObject);
begin {Изменение номера кисти}
  inc(num_brush);
  if num_brush>111 then num_brush:=0;
  Label1.Caption:=IntToStr(num_brush);
  SetBrush(col_brush,num_brush);
  fill_area(xy,nxy);
end;

procedure TForm1.ColBrushClick(Sender: TObject);
begin {Изменение номера цвета}
  inc(col_brush);
  if col_brush>24 then col_brush:=0;
  Label2.Caption:=IntToStr(col_brush);
  SetBrush(col_brush,num_brush);
  fill_area(xy,nxy);
end;

procedure TForm1.FillClick(Sender: TObject);
begin {Заливка}
  fill_area(xy,nxy);
end;

procedure TForm1.ClearClick(Sender: TObject);
begin {Очистка}
  SetBrush(0,0);
```

```
  fill_area(xy,nxy);
  SetBrush(col_brush,num_brush);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  InitPenBrush(Image1.Canvas);
  SetPen(10,6,1);
  xy[0]:=Point(100,10);
  xy[1]:=Point(250,110);
  xy[2]:=Point(150,210);
  xy[3]:=Point(50,110);
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ClosePenBrush();
end;

end.


unit Unit2;

interface

uses Graphics,Types,QDialogs,SysUtils;

procedure InitPenBrush(pcObject:TCanvas);
procedure ClosePenBrush;
procedure fill_area(xy:array of Tpoint; n:integer);
procedure GetBrush(var color,num_pat:integer);
procedure SetBrush(color,num_pat:integer);
procedure RestoreBrush;
procedure GetPen(var color,style,width:integer);
procedure SetPen(color,style,width:integer);
procedure RestorePen;

implementation
```

```
var
  pC:TCanvas;  {указатель на канву графического объекта}
  BmpBrush:TBitmap; {bitmap для кисти}

  old_col_pen:integer=0;   {предыдущий цвет пера}
  old_wid_pen:integer=1;   {предыдущая ширина пера (в пикселах)}
  old_num_pen:integer=0;   {предыдущий номер (стиль) пера}
  cur_col_pen:integer=0;   {текущий цвет пера}
  cur_wid_pen:integer=1;   {текущая ширина пера (в пикселах)}
  cur_num_pen:integer=0;   {текущий номер (стиль) пера}
  old_col_brush:integer=0; {предыдущий цвет кисти}
  old_num_brush:integer=0; {предыдущий номер (стиль) кисти}
  cur_col_brush:integer=0; {текущий цвет кисти}
  cur_num_brush:integer=0; {текущий номер (стиль) кисти}
  n_col:integer=25;        {максимальное количество цветов в палитре}
  n_brush:integer=111;     {максимальное количество шаблонов}

{ Массив заливочных шаблонов 8x8}
pattern: array [0..111,0..7]of byte=(
{Сплошная заливка}
  ($00,$00,$00,$00,$00,$00,$00,$00),
{0=очистка}
  ($FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF),
{1=сплошная заливка}
{==== Штриховка сплошными линиями}
{ по горизонтали }
  ($FF,$00,$00,$00,$00,$00,$00,$00),
{2=горизонтали (1+7)}
  ($FF,$FF,$00,$00,$00,$00,$00,$00),
{3=горизонтали (2+6)}
  ($FF,$FF,$FF,$00,$00,$00,$00,$00),
{4=горизонтали (3+5)}
  ($FF,$FF,$FF,$FF,$00,$00,$00,$00),
{5=горизонтали (4+4)}
  ($FF,$FF,$FF,$FF,$FF,$00,$00,$00),
{6=горизонтали (5+3)}
  ($FF,$FF,$FF,$FF,$FF,$FF,$00,$00),
{7=горизонтали (6+2)}
  ($FF,$FF,$FF,$FF,$FF,$FF,$FF,$00),
```

```
{8=горизонтали (7+1)}
  ($FF,$00,$FF,$00,$FF,$00,$FF,$00),
{9=горизонтали (1+1)}
  ($FF,$00,$00,$00,$FF,$00,$00,$00),
{10=горизонтали (1+3)}
  ($FF,$FF,$FF,$00,$FF,$FF,$FF,$00),
{11=горизонтали (3+1)}
  ($FF,$FF,$00,$00,$FF,$FF,$00,$00),
{12=горизонтали (2+2)}
{ по вертикали}
  ($80,$80,$80,$80,$80,$80,$80,$80),
{13=вертикали (1+7)}
  ($88,$88,$88,$88,$88,$88,$88,$88),
{14=вертикали (1+3)}
  ($C0,$C0,$C0,$C0,$C0,$C0,$C0,$C0),
{15=вертикали (2+6)}
  ($E0,$E0,$E0,$E0,$E0,$E0,$E0,$E0),
{16=вертикали (3+5)}
  ($F0,$F0,$F0,$F0,$F0,$F0,$F0,$F0),
{17=вертикали (4+4)}
  ($F8,$F8,$F8,$F8,$F8,$F8,$F8,$F8),
{18=вертикали (5+3)}
  ($FC,$FC,$FC,$FC,$FC,$FC,$FC,$FC),
{19=вертикали (6+2)}
  ($FE,$FE,$FE,$FE,$FE,$FE,$FE,$FE),
{20=вертикали (7+1)}
  ($AA,$AA,$AA,$AA,$AA,$AA,$AA,$AA),
{21=вертикали (1+1)}
  ($EE,$EE,$EE,$EE,$EE,$EE,$EE,$EE),
{22=вертикали (3+1)}
  ($CC,$CC,$CC,$CC,$CC,$CC,$CC,$CC),
{23=вертикали (2+2)}

  { прямоугольная двойная штриховка}
  ($FF,$80,$80,$80,$80,$80,$80,$80),
{24=сетка (1+7,1+7)}
  ($FF,$FF,$C0,$C0,$C0,$C0,$C0,$C0),
{25=сетка (2+6,2+6)}
  ($FF,$FF,$FF,$E0,$E0,$E0,$E0,$E0),
```

```
{26=сетка (3+5,3+5)}
  ($FF,$FF,$FF,$FF,$F0,$F0,$F0,$F0),
{27=сетка (4+4,4+4)}
  ($FF,$FF,$FF,$FF,$FF,$F8,$F8,$F8),
{28=сетка (5+3,5+3)}
  ($FF,$FF,$FF,$FF,$FF,$FF,$FC,$FC),
{29=сетка (6+2,6+2)}
  ($FF,$FF,$FF,$FF,$FF,$FF,$FF,$FE),
{30=сетка (7+1,7+1)}
  ($FF,$FF,$99,$99,$FF,$FF,$99,$99),
{31=сетка (2+2,2+2)}

  { по диагонали}
  ($01,$02,$04,$08,$10,$20,$40,$80),
{32=справа налево (1+10)}
  ($83,$07,$0E,$1C,$38,$70,$E0,$C1),
{33=справа налево (3+7)}
  ($C7,$8F,$1F,$3E,$7C,$F8,$F1,$E3),
{34=справа налево (6+3)}
  ($80,$40,$20,$10,$08,$04,$02,$01),
{35=слева направо (1+10)}
  ($C1,$E0,$70,$38,$1C,$0E,$07,$83),
{36=слева направо (3+7)}
  ($E3,$F1,$F8,$7C,$3E,$1F,$8F,$C7),
{37=слева направо (6+3)}

{ косоугольная двойная штриховка}
  ($81,$42,$24,$18,$18,$24,$42,$81),
{38=штриховка (1+6,1+6)}

{==== Штриховка штриховыми линиями}
{ по горизонтали}
  ($F0,$00,$00,$00,$0F,$00,$00,$00),
{39=штрихи (4*1+4,4) в шашечку}
  ($F8,$00,$00,$00,$1F,$00,$00,$00),
{40=штрихи (5*1+3,4) в шашечку}
  ($F0,$F0,$00,$00,$0F,$0F,$00,$00),
{41=штрихи (4*2+4,4) в шашечку}
  ($F8,$F8,$00,$00,$1F,$1F,$00,$00),
```

```
{42=штрихи (5*2+3,4) в шашечку}
  ($F0,$00,$00,$00,$00,$00,$00,$00),
{43=редкие штрихи (4*1+4,7)}
  ($FF,$00,$00,$00,$F0,$00,$00,$00),
{44=сплошная и штрихи}
{ по вертикали}
  ($80,$80,$80,$80,$00,$00,$00,$00),
{45=редкие штрихи (4*1+4,7)}
  ($80,$80,$80,$80,$08,$08,$08,$08),
{46=штрихи (4*1+4,4) в шашечку}

{==== Волнообразные линии}
{ по горизонтали}
  ($F0,$0F,$00,$00,$00,$00,$00,$00),
{47=с небольшой амплитудой (1)}
  ($F0,$FF,$0F,$00,$00,$00,$00,$00),
{48=с небольшой амплитудой (2)}
  ($81,$42,$24,$18,$00,$00,$00,$00),
{49=с большой амплитудой (1)}
  ($81,$C3,$66,$3C,$18,$00,$00,$00),
{50=с большой амплитудой (2)}
{ по вертикали }

{по диагонали }
  ($10,$10,$10,$F0,$01,$01,$01,$0F),
{51=слева направо}

{==== Растровые сетки }
  ($00,$55,$00,$AA,$00,$55,$00,$AA),
{52=густые точки в шашечку (1+1,1)}
  ($00,$44,$00,$11,$00,$44,$00,$11),
{53=густые точки в шашечку (1+3,2)}
  ($22,$00,$00,$00,$88,$00,$00,$00),
{54=мелкие (1 пиксел) редкие точки в шашечку}
  ($00,$66,$66,$00,$00,$99,$99,$00),
{55=средние (2x2) точки в шашечку}
  ($00,$00,$30,$30,$00,$00,$03,$03),
{56=средние (2x2) редкие точки в шашечку}
  ($00,$00,$00,$18,$18,$00,$00,$00),
```

```
{57=средние (2x2) очень редкие точки в шашечку}
  ($00,$00,$3C,$3C,$3C,$3C,$00,$00),
{58=крупные (4x4) точки}
  ($CC,$33,$CC,$33,$CC,$33,$CC,$33),
{59=густая шахматка}
  ($AA,$55,$AA,$55,$AA,$55,$AA,$55),
{60=очень густая шахматка}

{==== Фигуры в узлах прямоугольной сетки}
  ($10,$10,$7C,$10,$10,$00,$00,$00),
{61=крестики}
  ($10,$00,$54,$00,$10,$00,$00,$00),
{62=пунктирные крестики}
  ($00,$00,$18,$3C,$3C,$18,$00,$00),
{63=жирные кресты}
  ($00,$18,$3C,$7E,$7E,$3C,$18,$00),
{64=сетка из больших ромбиков}
  ($00,$7E,$7E,$7E,$7E,$7E,$7E,$00),
{65=сетка из больших квадратов}
  ($F8,$F8,$F8,$F8,$F8,$00,$00,$00),
{66=залитые квадраты 5x5}
  ($FC,$FC,$FC,$FC,$FC,$FC,$00,$00),
{67=залитые квадраты 6x6}
  ($FE,$FE,$FE,$FE,$FE,$FE,$FE,$00),
{68=залитые квадраты 7x7}
  ($00,$7E,$42,$42,$42,$42,$7E,$00),
{69=отдельные большие квадратики}
  ($00,$00,$3C,$24,$24,$3C,$00,$00),
{70=отдельные маленькие квадраты}
  ($00,$7E,$42,$5A,$5A,$42,$7E,$00),
{71=квадратики с точками в центре}
  ($00,$42,$24,$18,$18,$24,$42,$00),
{72=символ "x"}
  ($00,$44,$28,$10,$10,$10,$00,$00),
{73=символ "y"}
  ($00,$42,$24,$18,$00,$00,$00,$00),
{74=галочки}
  ($00,$00,$00,$18,$24,$42,$00,$00),
{75=перевернутые галочки}
```

```
  ($10,$38,$54,$10,$10,$10,$00,$00),
{76=тонкие стрелки вверх}
  ($10,$10,$10,$54,$38,$10,$00,$00),
{77=тонкие стрелки вниз}
  ($00,$20,$40,$FC,$40,$20,$00,$00),
{78=тонкие стрелки влево}
  ($00,$10,$08,$FC,$08,$10,$00,$00),
{79=тонкие стрелки вправо}
  ($00,$08,$18,$3F,$3F,$18,$08,$00),
{80=жирные стрелки влево}
  ($20,$70,$A8,$22,$22,$8A,$07,$02),
{81=двойные стрелки по вертикали}
  ($24,$02,$1F,$02,$24,$40,$F8,$40),
{82=двойные стрелки по горизонтали}
  ($00,$10,$28,$44,$28,$10,$00,$00),
{83=контуры ромбиков по сетке}
  ($81,$18,$3C,$7E,$7E,$3C,$18,$81),
{84=залитые ромбики и точки}
  ($18,$3C,$3C,$7E,$7E,$3C,$3C,$18),
{85=залитые ромбики по вертикали}
  ($00,$3C,$42,$42,$42,$42,$3C,$00),
{86=кружочки на сетке (d=6)}
  ($44,$38,$00,$00,$00,$38,$44,$44),
{87=кружочки на сетке  (d=5)}
  ($80,$7F,$41,$41,$41,$41,$41,$7F),
{88=незалитые квадратики}
  ($08,$00,$08,$5D,$08,$00,$08,$00),
{89=точечная сетка с точкам в узлах}

{==== Фигуры в шахматном порядке}

  ($00,$18,$3C,$7E,$7E,$3C,$18,$00),
{90=ромбики в шашечку}
  ($F0,$F0,$F0,$F0,$0F,$0F,$0F,$0F),
{91=густые мелкие квадратики (4x4)в шашечку}
  ($CC,$33,$CC,$33,$CC,$33,$CC,$33),
{92=более крупная шахматка}
  ($42,$C3,$3C,$24,$24,$3C,$C3,$42),
```

```
{93=контуры квадратиков в шахматку}

  {==== Узоры }

  ($00,$FB,$FB,$FB,$00,$DF,$DF,$DF),
{94=залитые кирпичи по горизонтали}
  ($00,$FB,$0A,$FB,$00,$DF,$50,$DF),
{95=незалитые кирпичи по горизонтали}
  ($EE,$EE,$0E,$EE,$EE,$E0,$EE,$EE),
{96=залитые кирпичи по вертикали}
  ($01,$82,$44,$28,$10,$20,$40,$80),
{97=незалитые кирпичи по диагонали}
  ($80,$41,$22,$14,$08,$04,$02,$01),
{98=незалитые кирпичи по диагонали}
  ($94,$84,$48,$30,$00,$C1,$22,$14),
{99=волнистый узор}
  ($FF,$01,$7D,$45,$5D,$41,$7F,$00),
{100=спиральный узор}
  ($00,$EC,$2A,$2A,$2A,$2A,$EC,$00),
{101=узор 1}
  ($00,$00,$7E,$42,$7E,$42,$7E,$42),
{102=узор 2}
  ($00,$5D,$3E,$6D,$7F,$63,$36,$F0),
{103=узор 3}
  ($92,$24,$49,$92,$24,$49,$92,$24),
{104=узор 4}
  ($B1,$22,$14,$14,$22,$91,$48,$24),
{105=узор 5}
  ($02,$91,$68,$08,$10,$16,$89,$40),
{106=узор 6}
  ($C3,$42,$5A,$7E,$7E,$5A,$42,$C3),
{107=узор 7}
  ($03,$18,$60,$80,$80,$40,$20,$18),
{108=узор 8}
  ($BF,$A0,$A0,$BF,$FB,$0A,$0A,$FB),
{109=узор 9}
  ($FF,$81,$BD,$A5,$A5,$BD,$81,$FF),
{110=узор 10}
  ($FF,$81,$81,$99,$99,$81,$81,$FF));
```

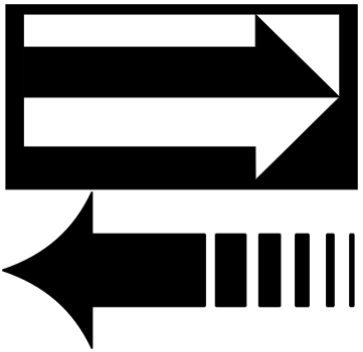
```
{111=узор 11}

{ Шкала цветов типа true color (24 бита на пиксел)}
  col:array[0..25]of integer=(
                $0,    {0=черный}
           $3F3F3F,    {1=темно-серый}
           $B2B2B2,    {2=серый}
           $CCCCCC,    {3=светло-серый}
           $FFFFFF,    {4=белый}
           $0000FF,    {5=ярко-красный}
           $0000E5,    {6=красный}
           $0000CC,    {7=темно-красный}
           $FF0000,    {8=ярко-синий}
           $E50000,    {9=синий}
           $CC0000,    {10=темно-синий}
           $00FF00,    {11=ярко-зеленый}
           $00E500,    {12=зеленый}
           $00CC00,    {13=темно-зеленый}
           $00FFFF,    {14=ярко-желтый}
           $00E5E5,    {15=желтый}
           $00CCCC,    {16=темно-желтый}
           $FFFF00,    {17=ярко-голубой}
           $E5E500,    {18=голубой}
           $CCCC00,    {19=темно-голубой}
           $FF00FF,    {20=ярко-фиолетовый}
           $E500E5,    {21=фиолетовый}
           $CC00CC,    {22=темно-фиолетовый}
           $0066FF,    {23=оранжевый}
           $000080,    {24=темно-коричневый}
           $1ffffff   {25=clNone}
           );

procedure InitPenBrush(pcObject:TCanvas);
{Инициализация области рисования}
{pcObject — указатель на канву области рисования}
begin
   pC:=pcObject;
end;

procedure ClosePenBrush;
```

```
{Завершение работы с экраном}
begin
  if BmpBrush<>nil then
     BmpBrush.Free; {Освобождение области из-под bitmap'a}
end;

procedure fill_area(xy:array of Tpoint; n:integer);
{Заливка области, ограниченной точками xy (n-число точек)}
begin
  pC.Polygon(xy);
end;

procedure GetBrush(var color,num_pat:integer);
{Опрос параметров кисти}
begin
  color:=cur_col_brush;
  num_pat:=cur_num_brush;
end;

procedure SetBrush(color,num_pat:integer);
{Формирование bitmap'a для кисти}
{ Шкала для формирования пикселов в bitmap'e }
const
  scale:array [0..7]of byte=($80,$40,$20,$10,$08,$04,$02,$01);
var
  pBmp:PByteArray;
  y,k,p:byte;
  plin:byte;
{Формирование параметров bitmap для кисти}
begin
  if(num_pat<0)or(num_pat>n_brush)or
                (color<0)or(color>n_col) then
    begin
      ShowMessage('Кисть с такими параметрами недоступна');
      exit;
    end;
  if BmpBrush<>nil then BmpBrush.Free;
{ Удаление предыдущего bitmap}
  BmpBrush:=TBitmap.Create;          { Запрос памяти}
```

```
  BmpBrush.Width:=8;                  { Ширина = 8 пикселов}
  BmpBrush.Height:=8;                 { Высота = 8 пикселов}
  BmpBrush.PixelFormat:=pf32bit;      { Формат = 32 бита на пиксел}
{ перенос байтов шаблона в bitmap }
  for y:=7 downto 0 do
    begin
      pBmp:=BmpBrush.ScanLine[y];
      plin:=pattern[num_pat,y];
      for k:=0 to 7 do
        begin
          if (plin and scale[k])<>0 then
            begin
              p:=4*k;
              pBmp[p]:=(col[color] shr 16) and $FF;
              pBmp[p+1]:=(col[color] shr 8) and $FF;
              pBmp[p+2]:=col[color]and $FF;
            end;
        end;
    end;
  pC.Brush.Style:=bsClear;
  pC.Brush.Bitmap:=BmpBrush;
  old_col_brush:=cur_col_brush;
  old_num_brush:=cur_num_brush;
  cur_col_brush:=color;
  cur_num_brush:=num_pat;
end;

procedure RestoreBrush;
{Восстановление параметров кисти}
begin
  SetBrush(old_col_brush,old_num_pen);
end;

procedure GetPen(var color,style,width:integer);
{Опрос параметров пера}
begin
  color:=cur_col_pen;
  style:=cur_num_pen;
  width:=cur_wid_pen;
```

```
end;

procedure SetPen(color,style,width:integer);
{Установка параметров пера}
begin
  if (color>n_col) or (style>7) then
    begin
      ShowMessage('Перо с такими параметрами недоступно');
      exit;
    end;
  pC.Pen.Color:=col[color];
  pC.Pen.Style:=TPenStyle(style);
  pC.Pen.Width:=width;
  old_col_pen:=cur_col_pen;
  old_num_pen:=cur_num_pen;
  old_wid_pen:=cur_wid_pen;
  cur_col_pen:=color;
  cur_num_pen:=style;
  cur_wid_pen:=width;
end;

procedure RestorePen;
{Восстановление параметров пера}
begin
  SetPen(old_col_pen,old_num_pen,old_wid_pen);
end;

end.
```

Глава 9



Обработка календарных дат и временных интервалов

9.1. Дата и время в среде Visual Basic

Данные типа дата/время представлены в среде Visual Basic 8-байтовыми вещественными значениями (так называемый формат IEEE 64-bit). Диапазон допустимых дат — от 1 января 100 г. до 31 декабря 9999. Целая часть вещественного значения соответствует количеству дней, предшествовавших нулевой отметке или прошедших после точки начала отсчета. За нулевую отметку выбрана полночь 30 декабря 1899 года. Даты, предшествующие нулевой отметке, отрицательны. Нулевое значение дробной части соответствует началу суток после полуночи, в полдень дробная часть даты достигает величины 0,5, к концу суток дробная часть приближается к 1, что соответствует показанию часов 23:59:59.

Переменные, в которых предполагается размещать значения дат и/или времени, могут быть двух типов — специализированные типа `Date` и универсальные типа `Variant`. В примере, приведенном на рис. 9.1, `d1` — специализированная переменная, а `d2` — универсальная. Разница между ними в занимаемой оперативной памяти и в скорости выполнения операций. Естественно, что формат универсальных данных требует больше места в памяти, и обработка таких значений сопряжена с дополнительными накладными расходами.

Конкретные значения дат и времени, присваиваемые тем или иным переменным, должны заключаться в специальные скобки, обозначаемые символом `#`. При этом допускаются различные форматы задания даты и времени, отличающиеся количеством цифр, отведенных под ту или иную компоненту (1 или 2 — для дня, месяца или часов, 2 или 4 — для года, сокращенное или полное наименование месяца), и разделителем компонент. Например — `#2/01/02#` или `#February 1, 2002#` или `#1 Feb 02#`. Последовательность задания дня, месяца и символьные обозначения месяцев определяются установками операционной системы.

Приложение 9.1 (VB)

Три наиболее популярные системные переменные `Date`, `Time` и `Now` используются для извлечения текущих показаний календаря и часов, поддерживаемых в компьютере даже в выключенном состоянии. Пример на рис. 9.1 демонстрирует такие операции и возможность вывода значений дат и времени.

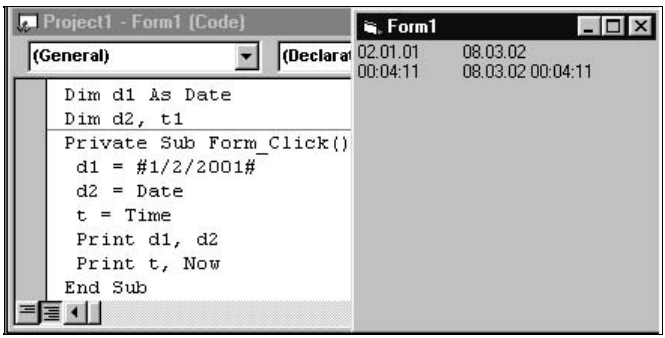


Рис. 9.1. Опрос системного календаря и компьютерных часов

Содержимое системных переменных `Date` и `Time` можно изменить, поместив их имена в левой части оператора присваивания. Однако коррекцию часов и календаря лучше осуществлять через операционную систему.

Если дата (`date`) или время (`time`) заданы в формате `Variant(Date)`, вещественного значения или строки допустимого формата, то отдельные компоненты могут быть извлечены и возвращены в формате целого числа `Variant(Integer)` с помощью следующих функций:

- `Day(date)` — день в диапазоне от 1 до 31;
- `Month(date)` — месяц в диапазоне от 1 до 12;
- `Year(date)` — год в диапазоне от 100 до 9999;
- `Weekday(date[,n0])` — день недели;
- `Hour(time)` — часы в диапазоне от 0 до 23;
- `Minute(time)` — минуты в диапазоне от 0 до 59;
- `Second(time)` — секунды в диапазоне от 0 до 59.

В определении дня недели присутствует необязательный второй аргумент `n0`, определяющий, какой день недели должен считаться первым. От его задания зависит и диапазон возвращаемых значений.

Отдельные компоненты даты и времени могут быть преобразованы в формат `Variant(Date)` с помощью следующих функций:

- `DateSerial(year,month,day)` — преобразование даты;

- `TimeSerial(hour,min,sec)` — преобразование времени.

Аргументы этих функций являются разумными значениями типа `Integer`, однако для некоторых из них имеются дополнительные соглашения. Например, год в диапазоне от 0 до 29 интерпретируется как год от 2000 до 2029, год от 30 до 99 — как год от 1930 до 1999. В значениях месяца, дня, минут и секунд допускается задание числовых значений, превышающих допустимый интервал. Система берет на себя пересчет соответствующих аргументов. Например, 90 секунд будут преобразованы в 1 минуту и 30 секунд, после чего значение минут увеличится на 1.

Если мы имеем дело с датой или временем, заданным в виде строки, то соответствующее преобразование в формат `Variant(Date)` осуществляется с помощью следующих функций:

- `DataValue("12/21/2001")` — преобразование даты;
- `TimeValue("9:46:57 PM")` — преобразование времени.

Для этих функций тоже характерна гибкость в задании аргумента. Например, дата 21 декабря 2001 года может быть представлена как `"12.21.01"`, `"December 21, 2001"`, `"Dec 21, 01"`.

Функция `DateAdd` позволяет определить дату и/или время `date2`, которая получится из даты `date1` в результате добавления или вычитания `k` временных интервалов:

```
date2=DateAdd(interval,k,date1)
```

Первый аргумент, задаваемый строкой, содержащей от 1 до 4 символов, определяет цену интервала (табл. 9.1), количество которых (`k`) добавляется (при `k>0`) или вычитается (при `k<0`).

Таблица 9.1. *Цены интервала в зависимости от символа (среда Visual Basic)*

Интервал	Цена	Интервал	Цена	Интервал	Цена
"d"	день	"yyyy"	год	"h"	час
"m"	месяц	"y"	день года	"n"	минута
"q"	квартал	"ww"	неделя	"s"	секунда

Прибавление интервала в 1 месяц не приведет к несуществующей дате. Например, через месяц после 31 января наступит 28 или 29 февраля.

Количество временных интервалов между двумя датами `date1` и `date2` можно определить с помощью функции `DateDiff`:

```
k=DateDiff(interval,date1,date2)
```

На самом деле, в этой функции кроме указанных обязательных аргументов могут присутствовать еще два, определяющие первый день недели и первый день года.

9.2. Данные типа *TDateTime* в системах Borland C++ Builder и Delphi

В рассматриваемых средах данные типа дата/время представлены объектами, порожденными классом `TDateTime`, в котором описан единственный элемент данных формата `double` (8-байтовое вещественное число). Целая часть этого числа равна числу дней, прошедших от полуночи 30 декабря 1899 года, дробная часть определяет время дня (0.0 соответствует началу суток, 0.5 — 12 часам пополудни, 0.999 — концу суток). Упаковку данных временного типа, представленных в одном из форматов, принятых в Windows, их распаковку и извлечение нужных компонент (день, месяц, год, часы, минуты, секунды, день недели и т. п.) осуществляют соответствующие функции (методы), принадлежащие классу `TDateTime`. Перечень наиболее важных функций приведен в табл. 9.2. Переменные `dt` и `as`, использованные в этой таблице, имеют тип `DateTime` и `AnsiString` (в Delphi — типа `String`), соответственно. В системе Delphi 6.0 общее количество функций раздела **time/date routines** превышает 160, но большинство из них нельзя отнести к предметам первой необходимости.

Таблица 9.2. *Некоторые методы класса `TDateTime` (среда BCB)*

Формат вызова функции/метода (BCB)	Назначение
`dt=CurrentDate();`	Опрос текущей даты
`dt=CurrentDateTime();`	Опрос текущей даты и текущего времени
`dt=CurrentTime();`	Опрос текущего времени
`as=dt.DateString()`	Преобразование даты в строку
`as=dt.DateTimeString();`	Преобразование даты и времени в строку
`int k=dt.DayOfWeek();`	Определение дня недели (1– воскресенье, 7 — суббота)
`dt.DecodeDate(&year,&month, &day);`	Распаковка даты (все аргументы типа `unsigned short`)
`dt.DecodeTime(&hour,&min,&sec, &msec);`	Распаковка времени (все аргументы типа `unsigned short`)
`int k=dt.FileDate();`	Преобразование даты в формат времени создания файла в MS-DOS
`dt=FileDateToDateTime(int k);`	Преобразование даты создания файла в формат типа `TDateTime`

Таблица 9.2 *(окончание)*

Формат вызова функции/метода (BCB)	Назначение
`as=dt.FormatString("format”);`	Преобразование даты и времени в строку по заданному формату
`TDateTime dt[([arg])];`	Объявление и инициализация данных типа `TDateTime`
`as=dt.TimeString();`	Преобразование времени в строку

Остановимся несколько подробнее на объявлении и возможной инициализации переменных типа `DateTime` с помощью метода `TDateTime`, который принято называть конструктором. Аргумент в этом методе может отсутствовать вообще (`TDateTime dt;`) и тогда в программе появляется неинициализированная переменная `dt`, занимающая в оперативной памяти 8 байт. Если в таком объявлении присутствуют пустые скобки или задан нулевой аргумент, то переменной `dt` присваивается значение 0, что соответствует 24 часам 30 декабря 1899 года (т. е. точке начала отсчета календарных дат). В качестве аргумента `arg` может быть задано вещественное число или имя переменной типа `DateTime`, которой ранее было присвоено значение. Тогда это же значение и будет начальным для переменной `dt`. Наконец, аргумент `arg` может быть представлен списком из трех или четырех целочисленных значений:

```
TDateTime d(2001,9,29);  // год, месяц, число
TDateTime t(22,30,15,0); // часы, минуты, секунды, доли секунд
```

Очень много возможностей по преобразованию данных типа `DateTime` в строку предоставляют форматные указатели в методе `FormatString`. Так, например, день месяца может быть представлен в результирующей строке как число из интервала [1, 31] (указатель `d`), двухзначное число из интервала [01, 31] (указатель `dd`), сокращенное (указатель `ddd`) или полное (указатель `ddddd`) обозначение дня недели, число и сокращенное (указатель `dddd`) или полное (указатель `dddddd`) обозначение дня недели. Такие же возможности имеются и для обозначения месяца (порядковый номер в году — `m`, сокращенное — `mm` или полное — `mmm` наименование месяца). Год может отображаться как двухзначное (`yy`) или четырехзначное (`yyyy`) число. Аналогичные способы управления распространяются и на все компоненты времени.

Над данными типа `DateTime` в BCB определены операции сложения (+, ++) и вычитания (−, −−). Их можно сравнивать, используя все операции отношения.

9.3. Объекты, связанные с датами и временем

В наших визуальных средах чаще всего используются два объекта, отображающие время или диапазон дат, — `DTPicker` (в VB) или `DateTimePicker` (в BCB и Delphi) и `MonthView` (в VB) или `MonthCalendar` (в BCB и Delphi). В системе Visual Basic вы должны зайти в раздел главного меню **Project** (Проект), выполнить команду **Components** (Компоненты), нажать кнопку **Browse** (Просмотр) и выбрать файл mscomct2.ocx. В системах BCB и Delphi аналогичные компоненты размещены на вкладке Win32. Внешний вид однотипных компонент в разных системах почти одинаков, однако набор свойств, доступных программисту, несколько отличается — в системе Visual Basic их немного больше. В системах BCB и Delphi недостающие свойства компенсируются соответствующими методами.

Приложение 9.2 (Delphi)

Объект `DateTimePicker` представляет собой гибрид календаря, отображающего текущую или заданную дату, часов и комбинированного окна ввода. Он имеет два варианта представления, определяемых значением свойства `DateMode`.

По умолчанию `DateMode=dmComboBox` и во время работы приложения щелчок по кнопке раскрывает календарь, соответствующий установленной или текущей дате (рис. 9.2). Кнопки со стрелками вправо и влево позволяют перемещаться на соседние месяцы и даже перешагивать через границы года в диапазоне дат, определенном значениями свойств `MinDate`, `MaxDate`. Щелчок по дате календаря приводит к ее переносу в окно ввода. Окно ввода устроено довольно нестандартно. Каждая компонента даты может быть выделена отдельно, и тогда ее можно изменить путем набора нужного значения на клавиатуре. Когда используется длинный формат даты (`DateFormat=dtLong`) и месяц отображается не числом, а словом, то изменение месяца все равно производится набором числа от 1 до 12. Объект `DateTimePicker` удобен еще и тем, что он автоматически следит за правильностью набора даты.

Если установить `DateMode=UpDown`, то объект лишается выпадающего календаря и обеспечивает только ввод новой даты или отображение значения свойства `Date`, установленное тем или иным путем. На способ отображения даты влияет строка, содержащая от одного до четырех символов, установленных в свойстве `Format`. Здесь используются те же сочетания, что и в свойстве `FormatString` (*см. раздел 9.2*).

Для того чтобы в окне ввода отображалось время, достаточно изменить значение свойства `Kind=dtkTime`. В этом случае содержимое окна ввода и значение свойства `Time` будут совпадать, но объект будет представлен только в формате `UpDown` (см. рис. 9.2).

Рис. 9.2. Объекты `DateTimePicker`

Свойство `CalAlignment` позволяет прижать дату или время, отображаемые в окне ввода к левой (по умолчанию `CalAlignment=dtaLeft`) или к правой (`dtaRight`) границе.

Объект `DTPicker` в системе Visual Basic имеет практически такой же внешний вид. Выбор варианта использования управляется свойствами `UpDown` (`True` или `False`) и `Format` (`dtpLongDate`, `dtpShortDate`, `dtpTime` и `dtpCustom`). Среди его свойств присутствуют такие, как `Day`, `DayOfWeek`, `Month`, `Week` и `Year`, с помощью которых можно извлечь любую компоненту из значения установленной даты (свойство `Value`). Свойство `CustomFormat` используется для формирования пользовательского шаблона отображения дат подобно тому, как это показано в *разд. 9.2 данной главы* (см. `FormatStrings`).

Объект `MonthCalendar` (BCB, Delphi) предназначен для отображения календаря на текущий или заданный месяц (рис. 9.3). По сравнению с календарем в объекте `DateTimePicker` здесь имеется возможность вывести в левой колонке номера недель (`WeekNumbers=True`), что и показано на рис. 9.3.

Рис. 9.3. Объект `MonthCalendar`

Аналогичный объект `MonthView` в системе Visual Basic располагает свойствами `MonthRows` и `MonthColumns`, с помощью которых можно отобразить

календарь сразу на несколько месяцев. На рис. 9.4 представлен вариант с двумя строками и двумя столбцами.

Рис. 9.4. Пример объекта `MonthView` в VB

9.4. Цифровые часы

Большинство объектов в визуальных средах играет пассивную роль — они ждут, пока не произойдет адресованное им событие, возникшее в результате инициативы пользователя (сигнал клавиатуры или мыши), операционной системы или нашего приложения. В отличие от них объект `Timer` генерирует событие `OnTimer` по собственной инициативе. Возникает такое событие через равные интервалы времени, длительность которых в миллисекундах задается значением свойства `Interval`. Адресуется это событие своему собственному обработчику, программа которого может использоваться для хронометража некоторых процессов.

Приложения 9.3 (BCB), 9.4 (Delphi), 9.5 (VB)

Построим приложение **Часы**, единственной функцией которого является опрос и отображение текущего значения машинного датчика времени.

В средах BCB и Delphi установим следующие размеры формы — `Height=100`, `Width=250`. Заменим заголовок формы (свойство `Caption`) текстом "Часы". Разместим на форме объект `StaticText1`, который отличается от метки `Label` только тем, что может быть окаймлен рамкой. Установим следующие значения его свойств:

- `Align=alClient` (это означает, что объект полностью займет область клиента);

- `Alignment=taCenter` (это означает, что отображаемый текст — показания часов — будет расположен в центре);
- `AutoSize=False` (размер объекта будет сохраняться фиксированным, не подстраиваясь под длину отображаемого текста);
- `BorderStyle=sbsSunken` (объект вдавится в форму и вокруг него появится заметная граница).

Удалим значение свойства `Caption`, заменив его пустой строкой. Щелкнем по свойству `Font` и в появившемся диалоговом окне установим параметры шрифта — Times New Roman, size = 40 pt.

Разместим на форме объект `Timer1`, у которого значение свойства `Interval` по умолчанию равно 1 000 миллисекунд. Нас это устраивает, т. к. период между регулярно генерируемыми событиями будет равен 1 с. Положение таймера на форме роли не играет, т. к. этот объект относится к разряду невидимых, и во время работы приложения он не отображается. Сформируем программу обработчика события `OnTimer`, содержащую единственный оператор по запросу текущего времени и записи его значения в свойство `Caption` объекта `StaticText1`.

На входном языке BCB этот обработчик выглядит следующим образом:

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
   StaticText1->Caption=Now().TimeString();
}
```

Обращение к методу `Now` позволяет извлечь из операционной системы текущие значения даты и времени в формате `DateTime`. С помощью метода `TimeString` из полученного результата выделяется текущее время и преобразуется в данное типа `AnsiString`.

На рис. 9.5 приведены образцы форм нашего приложения на стадии проектирования и во время работы программы.

Рис. 9.5. Приложение **Часы** на стадии проектирования и во время работы

В системе Delphi обращение к аналогичным методам выглядит следующим образом:

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
  StaticText1.Caption:=TimeToStr(Now);
end;
```

В системе VB размер формы задается в твипах, и окно порядка 250×100 пикселов соответствует примерно следующим значениям свойств ширины и высоты:

```
Height=1500
Width=3285
```

В качестве временного табло здесь можно использовать объект типа `Label`, т. к. набор его свойств позволяет очертить рамку:

```
BorderStyle=Fixed Single
```

Метку можно прижать к левой верхней границе области клиента, положив нулевые значения свойств `Left` и `Top`. До размеров окна растянуть область метки можно вручную или задать следующие значения высоты и ширины:

```
Height=1095
Width=3135
```

Значения параметров `Alignment`, `AutoSize` и `Font` выбираются такие же, как и в предыдущих приложениях.

Обработчик события `Timer` здесь выглядит еще проще за счет использования системной переменной `Time`:

```
Private Sub Timer1_Timer()
  Label1.Caption = Time
End Sub
```

9.5. Биоритмы

В нашей предыдущей книге приводился один из возможных вариантов DOS-приложения для построения кривых жизненного цикла, известных под названием *биоритмы*. Мы решили вернуться к этому приложению, но уже на качественно новом уровне, с целью продемонстрировать не совсем тривиальный интерфейс и процедуры настройки положения и размеров объектов, находящихся на форме.

Возможно, не все читатели знакомы с предыдущей публикацией. Поэтому вкратце напомним постановку задачи. Сторонники теории биоритмов утверждают, что каждому человеку присущи три периодических процесса, сопровождающие его жизнь с момента рождения. Первый из них с периодом 23 дня соответствует физическому состоянию, второй с периодом 28 дней — интеллектуальному, третий с периодом 33 дня — эмоциональному. По каждому процессу можно построить график, отложив по горизонтальной оси

время t в днях с момента рождения, а по вертикальной оси — амплитуду синусоиды со своим периодом:

$PH = \sin(2\pi \times t/23)$

$IN = \sin(2\pi \times t/28)$

$EM = \sin(2\pi \times t/33)$

Совместив все графики на общем рисунке, можно определить моменты общего подъема или спада потенциала человека. Особого единства в трактовке "плохих" интервалов жизни человека нет. Одни авторы считают, что к "черным" дням относятся моменты, когда все три кривые одновременно или в достаточно узком интервале пересекают ось времени, другие склонны рассматривать в качестве дней депрессии моменты, когда все кривые имеют общую или близкую точку минимума. Так как графики ритмов взрослого человека содержат 15—20 тысяч точек (предполагается, что каждая точка соответствует прожитому дню), да и заниматься анализом прошедших дней великого смысла не имеет, то целесообразно рассматривать поведение знаковых кривых на текущий месяц или на период предстоящих важных событий.

Исходная информация, предоставляемая пользователем, содержит дату рождения (день — от 1 до 31, месяц — от 1 до 12 и год — от 1900 до 2025). По умолчанию графики биоритмов рассчитываются на текущий месяц, но программа должна предусматривать возможность указания какого-либо месяца любого года из указанного интервала. Естественно, что программа должна контролировать правильность задания исходных данных, выдавать корректные сообщения с указанием характера ошибки и позволять произвести повторный ввод ошибочного параметра. Кроме того, за один сеанс общения с программой пользователю надо предоставить возможность набирать неограниченное количество исходных данных.

Приложение 9.6 (VB). "Биоритмы"

С целью структурирования программы, кроме ее интерфейсной части, целесообразно выделить в отдельные функции следующие процедуры:

- `Offset` — определение количества дней, прошедших от 1.01.1900 г. до заданной даты;
- `Axis` — построение осей для графика биоритмов с нанесением рисок по горизонтальной оси и числовых меток, упрощающих определение дня текущего месяца;
- `Graphic` — построение графика синусоиды $y=\sin(x\times 2\pi/T+\varphi)$ с заданными периодом T и начальным смещением по фазе φ.

В этом приложении мы будем максимально использовать простейшие интерфейсные объекты и не будем применять специальные функции обработки

дат. Алгоритм процедуры `Offset` предлагается самый примитивный. Например, будем прибавлять к сумме по 365 дней за каждый полностью прошедший год и компенсировать добавочным днем каждый високосный. В текущем году количество дней определяем с помощью массива `days`, хранящего количество дней в каждом месяце. Нужно только не забыть откорректировать число дней в феврале (`days[2]=29`), если текущий год является високосным. В выбранном диапазоне лет проверка високосного года упрощается и сводится к проверке остатка от деления номера года на 4.

Для определения текущей даты можно воспользоваться системной функцией (переменной) `Now`, представляющей значение даты и времени в формате `Variant`, и выделять из ее значения соответствующие компоненты системными функциями `Day`, `Month` и `Year`.

Ввод исходной информации может быть организован разными способами, но в любом случае потребуется несколько меток и окон ввода. Сокращению набора данных, принадлежащих заданному интервалу, могут содействовать объекты типа `UpDown` или различные шкалы с перемещаемыми ползунками. В предлагаемом ниже варианте используются две рамки (рис. 9.6), укомплектованные следующими компонентами:

- `Frame1` — рамка, играющая роль контейнера для элементов ввода дня рождения;
- `Label1` — метка с надписью **День**, расположенная слева от окна `Text1` и предназначенная для индикации поля ввода дня даты рождения;
- `Text1` — текстовое поле, отображающее день рождения;
- `UpDown1` — кнопка, ассоциированная с окном ввода `Text1`.

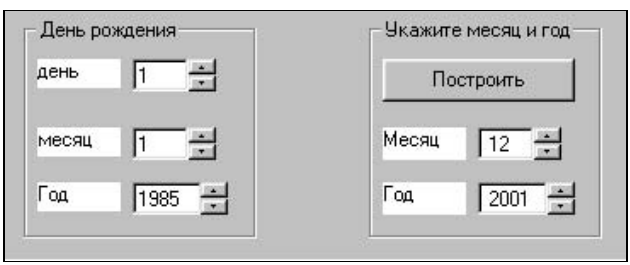


Рис. 9.6. Компоненты ввода исходных данных

Еще две тройки аналогичных компонент — (`Label2`, `Text2`, `UpDown2`) и (`Label3`, `Text3`, `UpDown3`) — предназначены для ввода и отображения месяца и года рождения. В рамке `Frame2` находятся две идентичные тройки компонент — (`Label4`, `Text4`, `UpDown4`) и (`Label5`, `Text5`, `UpDown5`), используемые для задания начальной даты графика биоритмов.

При выборе компонент в среде Visual Basic и установке значений некоторых их свойств нужно учитывать следующее:

1. Текстовые поля ввода `Text1`, `Text2`, ... должны быть связаны с соответствующими кнопками `UpDown1`, `UpDown2`, ... Для этого у первой кнопки необходимо задать значения свойств `BuddyControl=Text1` (имя ассоциированного объекта) и `BuddyProperty=Text` (поле ввода, в котором будет отображаться значение свойства `Value` — числового показателя, формируемого с помощью кнопки). Затем необходимо установить граничные значения свойства `Value`, ограничивающие диапазон изменения соответствующего данного (для первой кнопки `Min=1`, `Max=31`). Для оставшихся четырех кнопок должны быть выполнены аналогичные установки.
2. Истинные размеры рамок `Frame1` и `Frame2`, а также габариты и положение всех компонент в контейнерах целесообразно подобрать на стадии проектирования с тем, чтобы во время разворачивания формы или изменения ее габаритов во время работы приложения мы могли бы ограничиться заданием координат левого верхнего угла каждой рамки. А все внутренние объекты при этом сохранят свое положение относительно границ рамки.
3. Имеет смысл установить содержимое окон ввода даты дня рождения на стадии проектирования (в предлагаемом варианте выбрана дата 1.01.1985). Содержимое окон `Text4` и `Text5` придется формировать в момент запуска программы, т. к. только тогда станут известными текущие месяц и год.

Кнопка с надписью **Построить** является сигналом к построению кривых после задания всех исходных данных.

Собственно график и координатные оси строятся в окне `Picture1` — на поверхности объекта типа `PictureBox`. Габариты этого окна зависят от размеров главной формы, поэтому кривые биоритмов должны вписываться в рамки, размеры которых на стадии проектирования неизвестны. Для преодоления этих трудностей в программе предусмотрены глобальные переменные `Xmin` и `Xmax`, значения которых чуть меньше (на величину `dxy`), чем абсциссы левой и правой границ окна `Picture1`. Глобальная переменная `Ymid` отслеживает положение средней горизонтальной линии в окне `Picture1`. Амплитуда размаха синусоид тоже выбирается чуть меньшей, чем высота окна `Picture1`.

Для большей наглядности графиков форма приложения использует полный экран. Поэтому на стадии проектирования главной формы можно задать ее свойство `WindowState=Maximized`. Все операции по согласованию положения и размеров поля рисунка и рамок ввода сосредоточены в обработчике события `Form_Resize`. Все эти действия сопровождаются достаточно подробными комментариями.

Любые изменения в полях ввода перехватываются обработчиками события `Text_Change`. На эти процедуры управление передается при изменении хотя бы одной цифры в соответствующем окне. Основная работа этих обработчиков сводится к пересылке текущего значения из окна ввода в соответствующую глобальную переменную. В среде Visual Basic в окне ввода, ассоциированном с кнопкой `UpDown`, можно набирать значение явно, минуя кнопку. Поэтому введенные данные должны проверяться на корректность. По дню рождения все компоненты проверяются в момент нажатия кнопки **Построить**, а по начальной дате графика эти проверки оказалось удобнее разместить в обработчиках события `Text_Change`.

В процедуре построения осей следует обратить внимание на позиционирование текущей точки (`CP`) при построении штрихов и выборе позиции для вывода числовой метки (маркера) с помощью свойств `CurrentX` и `CurrentY`. При выводе текста с помощью метода `Print` текущая точка определяет координаты левого верхнего угла прямоугольника, окаймляющего надпись.

```
'Глобальные переменные
Dim Xmin As Integer, Xmax As Integer
Границы по горизонтали для построения графиков
Dim Ymid As Integer 'Высота оси x
Dim dxy As Integer  'Добавка для разных отступов
Dim days(12) As Integer 'Количество дней в каждом месяце
Dim d As Integer, m As Integer, y As Integer 'Дата рождения
Dim d1 As Integer, m1 As Integer, y1 As Integer 'Дата графика

Private Sub Form_Resize()
  'Настройка размеров и начального положения объектов на форме
  'Установка начальных значений некоторых переменных
   Dim CurDate
  'Размеры и положение окна рисунка
  Picture1.Left = Left + 1000
  Picture1.Top = Top + 3000
  Picture1.Height = Height – 4000
  Picture1.Width = Width – 2000
  dxy = 100
  Xmin = dxy
  Xmax = Picture1.Width – 2 * dxy
  Ymid = Picture1.Height \ 2 – dxy
  'Положение рамки дня рождения
  Frame1.Left = Left + 1000
  Frame1.Top = Top + 250
```

```
  'Положение рамки с датой биоритмов
  Frame2.Left = Frame1.Left + Frame1.Width + 1000
  Frame2.Top = Top + 250
  'Длительности месяцев
  days(1) = 30: days(2) = 28:  days(3) = 31:  days(4) = 30
  days(5) = 31: days(6) = 30:  days(7) = 31:  days(8) = 31
  days(9) = 30: days(10) = 31: days(11) = 30: days(12) = 31
  CurDate = Now 'Опрос текущей даты и времени
  'Разделение компонент даты
  d1 = Day(CurDate): m1 = Month(CurDate): y1 = Year(CurDate)
  'Коррекция на случай февральской даты в високосном году
  If m1 = 2 And (m1 Mod 4 = 0) Then days(2) = 29
    'Начальные установки в окнах ввода
    d = 1: m = 1: y = 1985
    Text4.Text = Str(m1)
    Text5.Text = Str(y1)
    'Положение и размеры списка с информацией о цвете кривых
    List1.Top = Top + 400
    List1.Left = Frame2.Left + Frame2.Width + 1000
    List1.Width = 4000
End Sub

Private Sub Text1_Change()
  'Реакция на изменение дня даты рождения
  d = Val(Text1.Text)
End Sub

Private Sub Text2_Change()
  'Реакция на изменение месяца даты рождения
  m = Val(Text2.Text)
End Sub

Private Sub Text3_Change()
  'Реакция на изменение года даты рождения
  y = Val(Text3.Text)
End Sub

Private Sub Text4_Change()
  'Реакция на изменение месяца графика биоритмов
```

```
  m1 = Val(Text4.Text)
  If m1 < 1 Or m1 > 12 Then
    MsgBox ("Неверно указан месяц. Повторите")
    Text4.Text = 1
    Exit Sub
  End If
End Sub

Private Sub Text5_Change()
  'Реакция на изменение года графика биоритмов
  y1 = Val(Text5.Text)
  If y1 < 1900 Or y1 > 2025 Then
    MsgBox ("Неверно указан год. Повторите")
    Text5.Text = 1985
    Exit Sub
  End If
  'Коррекция на случай февральской даты в високосном году
  If (y1 Mod 4) = 0 And m1 = 2 Then days(2) = 29
End Sub

Private Sub Command1_Click()
' Выполнение команды "Построить"
 Dim dd As Long
 Picture1.Cls 'Очистка поля рисунка
' Контроль правильности даты рождения
 If m < 1 Or m > 12 Then
   MsgBox ("Неверно указан месяц. Повторите"): Exit Sub
 End If
 If d < 1 Or d > days(m) Then
   MsgBox ("Неверно указан день. Повторите"): Exit Sub
 End If
 If y < 1900 Or y > y1 Then
   MsgBox ("Неверно указан год. Повторите"): Exit Sub
 End If
 Axis 'Построение и маркировка координатных осей
'Вычисление возраста в днях на начальную дату графика
 dd = Offset(1, m1, y1) — Offset(d, m, y)
'Построение кривой физического состояния
 Graphic 23, dd Mod 23, 4
```

```
'Построение кривой интеллектуального состояния
 Graphic 28, dd Mod 28, 2
'Построение кривой эмоционального состояния
 Graphic 33, dd Mod 33, 1
End Sub

Private Sub Axis()
'Построение и маркировка осей в поле рисунка
 Dim X As Integer, j
'Построение оси y
 Picture1.Line (dxy, dxy)-(dxy, 2 * Ymid + dxy)
'Построение оси x
 Picture1.Line (dxy, Ymid)-(Xmax + dxy, Ymid)
'Цикл нанесения штрихов по оси x
 For j = 1 To days(m1)
  X = Xmin + j * Xmax \ 31 'Координата очередного штриха
  If (j Mod 5) <> 0 Then 'Если штрих — обычный
'Построение короткого штриха
   Picture1.Line (X, Ymid — dxy)-(X, Ymid + dxy)
  Else 'Если штрих — удлиненный, с маркировкой
   Picture1.CurrentX = X 'Положение CP по оси x
   Picture1.CurrentY = Ymid — dxy * 2
'Построение удлиненного штриха
   Picture1.Line -(X, Ymid + dxy * 2)
'Коррекция позиции начала маркировки
   Picture1.CurrentX = X — dxy
'Нанесение маркера
   Picture1.Print j
  End If
 Next j
End Sub

Private Sub Graphic(T As Integer, Dfi As Double, Col As Long)
'Построение очередной синусоиды
 Dim X As Integer, y As Integer, j
 Const TwoPi = 6.2831853
 X = Xmin 'Начальная точка графика
 y = Ymid * (1 — 0.9 * Sin(TwoPi * Dfi / T))
'Отображение начальной точки
```

```
 Picture1.PSet (X, y), QBColor(Col)
'Цикл вычисления координат кривой в каждый день месяца
 For j = 1 To days(m1)
  X = Xmin + j * Xmax \ 31
  y = Ymid * (1 — 0.9 * Sin(TwoPi * (j + Dfi) / T))
'Построение очередного звена синусоиды
  Picture1.Line -(X, y), QBColor(Col)
 Next j
End Sub

Private Function Offset(d As Integer, m As Integer, y As Integer) As Long
'Вычисление количества дней, прошедших от 1.01.1900 до указанной даты
  dd = 0:
'Цикл учета полных лет
  For k% = 1900 To y — 1
    dd = dd + 365
'Поправка на високосный год
    If (k% Mod 4) = 0 Then dd = dd + 1
  Next k%
'Учет дней, прошедших от начала года y до месяца m
  For k% = 1 To m — 1: dd = dd + days(k%): Next k%
  Offset = dd + d: 'Учет дней, прошедших в месяце m
End Function
```

Приложение 9.7 (Delphi). "Биоритмы"

Ниже приводится еще один вариант приложения, воспроизводящего кривые биоритмов, реализованный в среде Delphi. Его особенность заключается в другом варианте интерфейса, использующего компоненты типа `DateTimePicker`, и в более широком привлечении системных функций, связанных с обработкой дат.

В приложении выбран более компактный формат объекта с `DateMode=UpDown` (рис. 9.7), в котором изменение каждой части даты может быть выполнено с помощью встроенной кнопки `UpDown`. Значение установленной даты программа извлекает из свойства `Date`.

В отличие от VB-приложения здесь не потребовалась процедура `Offset`, т. к. ее заменила системная функция `DaysBetween`, аргументами которой являются две даты в формате `TDateTime`. Две другие функции `MonthOf` и `YearOf` позволили определить числовой номер месяца и год, соответствующие начальной дате графика. Так как система автоматически вписывает текущую

дату в окно объекта `DateTimePicker2`, а начальная дата графика должна совпадать с началом месяца, приходится корректировать номер дня с помощью функции `EncodeDate`. Три ее аргумента представлены обычными числовыми значениями года, месяца и дня, а результатом работы функции является дата в формате `TDateTime`. Перед построением очередного графика в процедуре `Axis` производится очистка поля рисования с помощью метода `FillRect`. По умолчанию он использует белый цвет кисти и сплошную заливку. Остальные части программы повторяют логику VB-приложения.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Memo1: TMemo;
    Button1: TButton;
    DateTimePicker1: TDateTimePicker;
    DateTimePicker2: TDateTimePicker;
    Image1: TImage;
    procedure Button1Click(Sender: TObject);
    procedure Axis;
    procedure Graphic(T:integer;Dfi:integer; col:TColor);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Xmin,Xmax,Ymid,m1,y1:integer;
  dxy:integer=5;
```

```
  days:array [1..12] of byte=
       (30,28,31,30,31,30,31,31,30,31,30,31);
implementation

{$R *.dfm}

procedure TForm1.Axis;
{Построение и маркировка осей}
var
  j,x:integer;
begin
{Очистка поля графика}
 Image1.Canvas.FillRect(Rect(0,0,Image1.Width,Image1.Height));
{Построение оси y}
 Image1.Canvas.MoveTo(dxy,dxy);
 Image1.Canvas.LineTo(dxy,2*Ymid-dxy);
{Построение оси x}
 Image1.Canvas.MoveTo(dxy, Ymid);
 Image1.Canvas.LineTo(Xmax+dxy,Ymid);
{Цикл нанесения штрихов по оси x}
 for j:=1 to days[m1] do
 begin
  x:=Xmin + j*Xmax div 31; {Координата очередного штриха}
  if (j mod 5) <> 0 then {Если штрих — обычный}
    begin
     {Построение короткого штриха}
     Image1.Canvas.MoveTo(X,Ymid-dxy);
     Image1.Canvas.LineTo(X,Ymid+dxy);
    end
  else {Если штрих — удлиненный, с маркировкой}
    begin
     {Построение удлиненного штриха}
     Image1.Canvas.MoveTo(X,Ymid-dxy*2);
     Image1.Canvas.LineTo(X,Ymid+dxy*2);
     {Коррекция позиции начала маркировки и нанесение маркера}
     Image1.Canvas.TextOut(X-dxy,Ymid+dxy*2,IntToStr(j));
    end;
 end;
```

```
end;

procedure TForm1.Graphic(T:integer;Dfi:integer; col:TColor);
{Построение графика}
var
 x,y,j:integer;
begin
 Image1.Canvas.Pen.Color:=col;
 x:=Xmin;  {Начальная точка графика}
 y:=round(Ymid*(1-0.9*Sin(2*Pi*Dfi/T)));
{Отображение начальной точки}
 Image1.Canvas.MoveTo(x,y);
{Цикл вычисления координат кривой в каждый день месяца}
 for j:=1 to days[m1] do
  begin
  x:=Xmin + j * Xmax div 31;
  y:=round(Ymid*(1-0.9*Sin(2*Pi*(j+Dfi)/T)));
{Построение очередного звена синусоиды}
  Image1.Canvas.LineTo(x,y);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
{Выполнение команды "Построить"}
var
  dd:integer;
begin
{  dxy:=5;}
  Xmin:=dxy;
  Xmax:=Image1.Width-2*dxy;
  Ymid:=Image1.Height div 2;
  m1:=MonthOf(DateTimePicker2.Date); {Месяц графика}
  y1:=YearOf(DateTimePicker2.Date); {Год графика}
  if (y1 mod 4)=0 then days[2]:=29;
{Коррекция дня в начальной дате графика}
  DateTimePicker2.Date:=EncodeDate(y1,m1,1);
  if (YearOf(DateTimePicker2.Date) mod 4)=0 then days[2]:=29;
{Построение и маркировка осей}
  Axis;
```

```
{Определение числа дней, прошедших от момента рождения}
  dd:=DaysBetween(DateTimePicker2.Date,DateTimePicker1.Date);
{Построение кривой физического состояния}
  Graphic(23,dd mod 23,clRed);
{Построение кривой интеллектуального состояния}
  Graphic(28,dd mod 28,clGreen);
{Построение кривой эмоционального состояния}
  Graphic(33,dd mod 33,clBlue);
end;
end.
```

На рис. 9.7 представлен образец работы этого приложения.

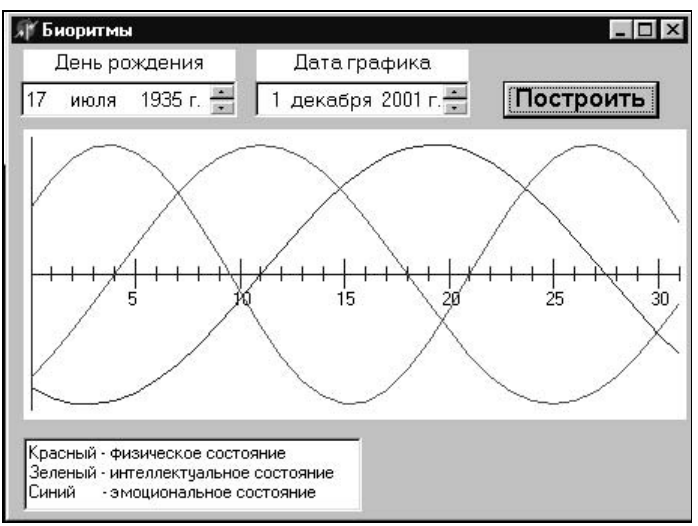
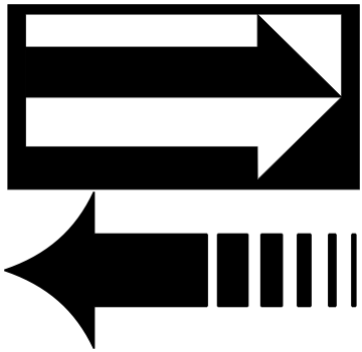


Рис. 9.7. Образец работы приложения **Биоритмы**

Глава 10

Работа с базами данных



Каждый из нас в той или иной мере имел дело с таблицами. Несмотря на разнообразие внешних форм, все *таблицы* (`tables`) устроены одинаково — они состоят из строк идентичной структуры. А строки, в свою очередь, образованы последовательностью *полей* (`fields`), на которых расположены связанные между собой данные. В свое время строки таблиц легли в основу данных, называемых *записями* (`records`), а *совокупность записей-строк* (`record set`) составила содержание соответствующих файлов.

Простейшая *база данных* (БД), содержащая единственную таблицу, иногда называется базой данных с *плоским файлом* (`flat-file database`). И, тем не менее, даже плоская БД — это нечто большее, чем просто файл, в котором хранится содержимое такой же таблицы. Во-первых, кроме содержимого таблицы в БД хранится информация о структуре таблицы, включающая наименования полей и типы расположенных в них данных. Во-вторых, сокращенное название СУБД — система управления базами данных — предполагает, что кроме собственно информации, в состав этой системы входит определенный набор сервисных программ, отвечающих на запросы пользователей и обеспечивающих поддержание данных в соответствии с текущим моментом времени. Содержимое БД, как правило, не является чем-то застывшим. Его приходится периодически обновлять, пополнять и удалять устаревшие сведения. Файловая система, представляющая один из важнейших блоков любой операционной системы, ничего подобного делать не умеет. Информация о структуре данных, хранящихся в обычном двоичном файле, находится только в программе, работающей с этим файлом, и никто другой разобраться с содержимым файла не сможет. Исключение составляют только файлы стандартных форматов — pcx, bmp, jpg и т. п. Если в ваших записях появляется новое поле или удаляется ненужное, то надо писать специальную программу, которая осуществит коррекцию соответствующего файла.

Обычно содержимое БД — несколько таблиц. И причиной тому являются соображения эффективности хранения и производительности СУБД.

Основной принцип хранения данных — не дублировать повторяющиеся сведения. Во-первых, на этом экономится память. Во-вторых, при исправлении данных всегда можно забыть о внесении изменений в дублирующуюся информацию. А с плоской таблицей без дублирования данных не обойтись. Предположим, что мы решили создать базу данных, учитывающую выпуск продукции книжного издательства "БХВ-Петербург". Во-первых, в запись должны войти сведения о книге, включающие следующие поля:

1. Наименование — символьное поле достаточной длины.
2. Автор — символьное поле (предположим, что автор один, хотя в реальной жизни их может быть и больше).
3. Год издания — целочисленное поле (для учета более точной даты выпуска может потребоваться поле типа `Date`).
4. Объем — целочисленное поле.
5. Тираж — целочисленное поле.

Так как издательство заключает договор с автором, то приходится учитывать и сведения, включающие следующие дополнительные поля:

1. Фамилия, имя, отчество — символьное поле.
2. Адрес — символьное поле.
3. Домашний телефон — символьное поле.
4. E-mail — символьное поле.
5. Паспорт — символьное поле.
6. Дата рождения — поле типа `Date`.
7. Индивидуальный номер налогоплательщика (ИНН) — длинное число.
8. Номер страхового свидетельства — длинное число.

А теперь представим себе, что один и тот же автор написал две книги. Естественно, что не надо повторять поля со сведениями об авторе в двух разных записях плоской таблицы. Проще создать новую таблицу со сведениями об авторах, а в таблице о книгах вместо данных по автору поместить ссылку на соответствующую строку второй таблицы.

Для ускорения поиска нужной информации записи в базе данных приходится сортировать по содержимому тех или иных полей, составлять различные массивы указателей (оглавления) и т. п. Предположим, что плоская база данных представляет собой телефонный справочник, содержащий фамилию владельца, его телефон и домашний адрес. В большом городе типа Москвы такой справочник может содержать порядка 1 млн. строк. Для удобства поиска строки естественно отсортировать в лексикографическом (алфавитном) порядке. Но если прибегнуть даже к такому быстрому методу поиска, как двоичный, то для нахождения нужной строки в заранее упорядоченной таблице, потребуется не меньше 20 сравнений. И это — при условии, что таб-

лица целиком находится в оперативной памяти (если строка занимает порядка 100 байт, то для ее хранения потребуется около 100 Мбайт). Однако поиск можно ускорить, создав, например, массив указателей, определяющих, с какого места таблицы строк начинаются фамилии на каждую букву. И тогда область поиска сузится примерно в 20—30 раз, т. е. работать придется с фрагментом таблицы, содержащим порядка 30 000 строк. Для него потребуется не так уж и много оперативной памяти, да и количество сравнений существенно поубавится. Можно построить указатели, содержащие два или больше уровней подобно тому, как устроено оглавление в книгах.

В таблицах другой природы может оказаться востребованной чаще всего информация, местоположение которой определяется сочетанием содержимого двух полей. И тогда информация, упорядоченная по содержимому более важного поля, подвергается дополнительной сортировке по содержимому второго поля. Соответствующий набор вложенных указателей позволит быстро найти нужную строку или группу строк, удовлетворяющую запросу пользователя.

Предварительные работы подобного рода называют *индексированием* БД. Индекс записи очень напоминает индекс элемента массива. Однако с одной и той же записью в БД могут быть связаны несколько индексов. Выделяют главный или первичный индекс (`primary index`), который для каждой записи должен быть уникальным. Например, им может быть порядковый номер, под которым запись появилась в БД, или последовательный номер записи после их сортировки по содержимому наиболее важного поля. Среди типов полей в строках БД появились данные формата `AutoIncrement`. Содержимое таких полей система заполняет автоматически, увеличивая с появлением каждой новой записи ее текущий номер на 1. Первичный индекс определяет последовательность отображения записей БД. Вторичные индексы (`secondary index`) могут быть построены на комбинации содержимого нескольких полей.

Таким образом, информация в БД состоит из совокупности связанных между собой файлов, шаблонов (схем) с описанием структуры таблиц и различного рода оглавлений, ускоряющих поиск нужной информации. Обычно такой набор данных размещается в одном подкаталоге, занимая в нем один или несколько файлов. При работе с обычными файлами в программах на алгоритмических языках высокого уровня создаются внутренние номера или указатели, заменяющие в процедурах обмена физические имена файлов на диске. По аналогии с файлами, при работе с конкретным набором БД тоже используется короткое логическое имя — `alias`, которому может быть сопоставлен нужный физический подкаталог. И такая перенастройка на другой подкаталог может быть выполнена без перетрансляции программы.

Как правило, большинство современных БД называют *реляционными*. Это название обязано термину relation — отношение, т. к. данные, сосредоточенные в строках таблицы, не являются разрозненными сведениями, а связаны между

собой определенными отношениями. Да и отдельные таблицы могут быть связаны между собой перекрестными ссылками. Базы данных, разработанные разными фирмами, используют различные внутренние форматы хранения данных. Однако существует довольно много сервисных средств, преобразующих наборы данных из одного формата в другой и обеспечивающих доступ к различным БД. В частности, рассматриваемые визуальные среды снабжены такого рода механизмами — Microsoft Jet database engine в системе Visual Basic и Borland Database Engine (BDE) в системах BCB и Delphi. Эти посредники обеспечивают доступ к большому количеству как локальных, так и удаленных баз данных — Paradox, dBASE, Access, FoxPro, БД, поддерживающих протоколы обмена ODBC (Open Database Connectivity) и SQL (Structured Query Language — язык структурированных запросов). Набор доступных SQL-баз представлен такими гигантами, как Oracle, MS SQL Server, DB2 (фирмы IBM), InterBase (фирмы Borland).

По уровню сложности создание небольших программ, содержащих несколько десятков или сотен строк, резко отличается от разработки больших программных проектов, содержащих сотни тысяч или десятки миллионов строк. Затраты на создание очень большой программной системы существенно превосходят затраты на разработку нескольких малых систем, имеющих такой же суммарный объем. Примерно такое же соотношение имеет место между эксплуатацией небольшой БД, содержащей несколько таблиц малого объема, и управлением огромной БД, обслуживающей одновременно большое количество пользователей, насчитывающей десятки тысяч таблиц со сложной иерархией и функционирующей в компьютерной сети на разных платформах. Описание технологии работы с мощными СУБД класса Oracle занимает несколько томов.

Однако в простых приложениях к БД предъявляются не такие сложные требования, и работа с базой данных, содержащей 1—2—3 таблицы, представляет собой лишь небольшой шаг вперед по сравнению с использованием файловой системы. Элементы работы с такой БД мы продемонстрируем на ряде простых примеров.

10.1. Расшифровка схемы таблиц простой БД

Первый пример позволяет выяснить структуру полей в записях базы данных, подготовленной в формате СУБД dBASE (файлы с расширением dbf). Идея этого примера заимствована из книги М.Теллеса, однако приведенная там программа генерировала аварийную ситуацию при попытке продолжить работу и открыть новую БД. В приведенном ниже приложении этот недостаток устранен. Пример поражает своей простотой. На главной форме расположены следующие объекты:

- `MainMenu1` — главное меню, содержащее две команды (**N2** — "Открыть базу" и **N4** — "Выход");

- `OpenDialog1` — объект для выбора файла БД с расширением dbf;
- `Table1` — объект для связи с таблицей в выбранной БД;
- `StringGrid1` — таблица (в ряде книг этот компонент называют сеткой строк или просто сеткой, чтобы не путать с таблицей БД, но мы будем пользоваться более естественным названием) для отображения имен полей, их типов и длин.

Приложение 10.1 (BCB)

По команде **Открыть базу** активизируется окно `OpenDialog1` и пользователю предоставляется возможность выбрать файл с расширением dbf, представляющий базу данных в формате dBASE. Несколько файлов с таким расширением поставляются в составе систем BCB и Delphi. Расположены они в каталоге:

C:\Program Files\Common Files\Borland Shared\Data

Имена файлов — animals.dbf, clients.dbf, holdings.dbf, industry.dbf, master.dbf. После выбора нужной БД в качестве посредника используется объект `Table1`, у которого формируются значения свойств `DatabaseName` (имя каталога или заменяющего его альтернативного имени alias) и `TableName` (имя файла). Затем объект `Table1` активизируется, что эквивалентно открыванию базы:

```
Table1->DatabaseName=ExtractFilePath(OpenDialog1->FileName);
Table1->TableName=ExtractFileName(OpenDialog1->FileName);
Table1->Active=True;
```

После установления связи с БД в таблице `StringGrid1` формируются колонки и их заголовки для отображения информации о полях записей в БД:

```
StringGrid1->ColWidths[0]=60;
StringGrid1->ColWidths[1]=StringGrid1->ClientWidth/4-20;
StringGrid1->ColWidths[2]=StringGrid1->ClientWidth/2-20;
StringGrid1->ColWidths[3]=StringGrid1->ClientWidth/4-20;
StringGrid1->Cells[0][0]="  № поля";
StringGrid1->Cells[1][0]="  Имя поля";
StringGrid1->Cells[2][0]="  Тип поля";
StringGrid1->Cells[3][0]="  Размер";
```

При выборе компонента `StringGrid` была произведена нестандартная установка всего двух свойств: `ColCount=4` (количество колонок) и `RowCount=50` (количество строк, с запасом). Ширина каждой колонки подбиралась экспериментальным путем на стадии пробных запусков приложения. Первая строка (ее настоящий номер — 0) и первая колонка (ее настоящий номер — 0) таблицы `StrinGrid1`, окрашенные в серый цвет, относятся к так называемым

фиксированным строкам (FixedRows) и столбцам (FixedCols). При прокрутке содержимого остальных строк и столбцов позиции фиксированных элементов остаются на своих местах. Это позволяет всегда видеть заголовки столбцов и номера строк прокручиваемой таблицы.

Для заполнения строк таблицы StringGrid1 организуется цикл по количеству полей в строках таблицы БД — значению свойства Table1->FieldCount. Имена полей, их типы и длины извлекаются соответственно из свойств Table1->FieldDefs->Items[i]->Name, Table1->FieldDefs->Items[i]->DataType и Table1->FieldDefs->Items[i]->Size.

```
for(int i=1; i<Table1->FieldCount; i++)
{
  StringGrid1->Cells[0][i]="   " + IntToStr(i);
  StringGrid1->Cells[1][i]=Table1->FieldDefs->Items[i]->Name;
  StringGrid1->Cells[2][i]=GetDescription(
                          Table1->FieldDefs->Items[i]->DataType);
  StringGrid1->Cells[3][i]=IntToStr(
                          Table1->FieldDefs->Items[i]->Size);
}
```

Однако тип поля, данного в схеме таблицы, представлен числовым значением (список соответствующих констант приведен в табл.10.1). Поэтому для формирования более приемлемого описания типа поля использована процедура GetDescription, построенная на простом цикле перебора элементов массива структур sFieldTypes. Она сравнивает числовой код nInd, извлеченный из описания типов полей БД, с числовыми кодами из списка допустимых типов и, в случае совпадения, извлекает из второго поля структуры соответствующее символьное наименование. Как только из массива структур извлекается последнее числовое поле, содержащее несуществующий код -1, процедура GetDescription прекращает свою работу и возвращает пустую строку. Для удобства описания шаблона структуры dbFieldType и массива структур sFieldTypes вынесены в заголовочный файл Unit1.h.

```
char *TForm1::GetDescription(int nInd)
{
  for(int i=0; sFieldTypes[i].nCode != -1; i++)
    if(sFieldTypes[i].nCode==nInd)
       return sFieldTypes[i].strDesc;
  return "";
}
```

Таблица 10.1. *Тип поля в схеме таблицы в зависимости от числового значения (среда BCB)*

Числовой код типа поля	Константа описания типа поля в БД	Формат содержимого поля БД
0	ftUnknown	Неизвестно или не определено
1	ftString	Строка, до 8 192 символов
2	ftSmallint	16-битное целое поле (типа short int)
3	ftInteger	32-битное целое поле (типа integer)
4	ftWord	16-битное беззнаковое целое поле
5	ftBoolean	Логическое поле
6	ftFloat	Поле чисел с плавающей запятой (типа double)
7	ftCurrency	Денежное поле
8	ftBCD	Двоично-кодированное десятичное поле
9	ftDate	Поле даты
10	ftTime	Поле времени
11	ftDateTime	Поле даты и времени
12	ftBytes	Фиксированное количество байт (двоичное представление)
13	ftVarBytes	Переменное количество байт (двоичное представление, в двух первых байтах — длина)
14	ftAutoInc	Автоматически увеличивающееся 32-битное целое поле счетчика
15	ftBlob	Ссылка на файл с большим двоичным объектом (Binary Large Object)
16	ftMemo	Поле Memo (строка неограниченной длины)
17	ftGraphic	Поле растрового рисунка
18	ftFmtMemo	Форматированное поле Memo
19	ftParadoxOle	Поле Paradox Ole
20	ftDBaseOle	Поле DBase Ole
21	ftTypedBinary	Типизированное двоичное поле
22	ftCursor	Выходной курсор из процедуры Oracle
23	ftFixedChar	Строка в стиле Си (с нулевым байтом в конце)
24	ftWideChar	Строка для кодировки Unicode, память выделяется динамически

Таблица 10.1 *(окончание)*

Числовой код типа поля	Константа описания типа поля в БД	Формат содержимого поля БД
25	`ftLargeint`	64-битное целое поле
26	`ftADT`	Абстрактный тип данных
27	`ftArray`	Массив
28	`ftReference`	Указатель на объект из другой таблицы
29	`ftDataSet`	Набор данных, интегрированный в текущий набор данных (типа структура в структуре)
30	`ftOraBlob`	Поле `BLOB` в таблицах Oracle 8
31	`ftOraClob`	Поле `CLOB` в таблицах Oracle 8
32	`ftVariant`	Неизвестные данные или данные неопределенного типа (аналог данных типа `Variant`)
33	`ftInterface`	Ссылка на неизвестный интерфейс
34	`ftIDispatch`	Ссылка на интерфейс Windows (`Dispatch`)
35	`ftGuid`	Глобальный уникальный идентификатор

Полный текст файлов Unit1.h и Unit1.cpp приведен ниже:

```
//Файл Unit1.cpp ---------------------------------------------------
#ifndef Unit1H
#define Unit1H
//-------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Db.hpp>
#include <DBTables.hpp>
#include <Dialogs.hpp>
#include <Grids.hpp>
#include <Menus.hpp>
//-------------------------------------------------------------------
typedef struct
{
  int nCode;
  char *strDesc;
```

```
}dbFieldType;
dbFieldType sFieldTypes[]={
{ftUnknown, "Неизвестно или не определено"},
{ftString,"Символьное или строковое поле"},
{ftSmallint,"16-битное целое поле"},
{ftInteger,"32-битное целое поле"},
{ftWord,"16-битное беззнаковое целое поле"},
{ftBoolean,"Логическое поле"},
{ftFloat,"Поле чисел с плавающей запятой"},
{ftCurrency,"Денежное поле"},
{ftBCD,"Двоично-кодированное десятичное поле"},
{ftDate,"Поле даты"},
{ftTime,"Поле времени"},
{ftDateTime,"Поле даты и времени"},
{ftBytes,"Фиксированное количество байт (двоичное представление)"},
{ftVarBytes,"Переменное количество байт (двоичное представление"},
{ftAutoInc,"Автоматически увеличивающееся 32-битное целое поле счетчика"},
{ftBlob,"Поле Binary Large Object"},
{ftMemo,"Поле Memo (строка неограниченной длины)"},
{ftGraphic,"Поле растрового рисунка"},
{ftFmtMemo,"Поле форматированного Memo"},
{ftParadoxOle,"Поле Paradox Ole"},
{ftDBaseOle,"Поле DBase Ole"},
{ftTypedBinary,"Типизированное двоичное поле"},
{-1,""}
};

class TForm1 : public TForm
{
__published:    // IDE-managed Components
        TMainMenu *MainMenu1;
        TOpenDialog *OpenDialog1;
        TStringGrid *StringGrid1;
        TTable *Table1;
        TMenuItem *N1;
        TMenuItem *N2;
        TMenuItem *N3;
        TMenuItem *N4;
        void __fastcall N2Click(TObject *Sender);
        void __fastcall N4Click(TObject *Sender);
```

```
private:    // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
        char *GetDescription(int nInd);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif

// Файл Unit1.cpp ---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
char *TForm1::GetDescription(int nInd)
{
  for(int i=0; sFieldTypes[i].nCode != -1; i++)
    if(sFieldTypes[i].nCode==nInd) return sFieldTypes[i].strDesc;
  return "";
}
//---------------------------------------------------------------------------
void __fastcall TForm1::N2Click(TObject *Sender)
{//Выполнение команды "Открыть базу"
 OpenDialog1->Filter="Файлы баз данных dBase (*.dbf)|*.dbf";
 if(OpenDialog1->Execute())
   {//Формируем путь к выбранной базе (имя каталога)
     Table1->DatabaseName=ExtractFilePath(OpenDialog1->FileName);
    //Формируем имя таблицы (имя файла)
     Table1->TableName=ExtractFileName(OpenDialog1->FileName);
    //Делаем таблицу активной
```

```
     Table1->Active=True;
    //Устанавливаем ширину колонок
     StringGrid1->ColWidths[0]=60;
     StringGrid1->ColWidths[1]=StringGrid1->ClientWidth/4-20;
     StringGrid1->ColWidths[2]=StringGrid1->ClientWidth/2-20;
     StringGrid1->ColWidths[3]=StringGrid1->ClientWidth/4-20;
    //Формируем заголовки
     StringGrid1->Cells[0][0]="  № поля";
     StringGrid1->Cells[1][0]="  Имя поля";
     StringGrid1->Cells[2][0]="  Тип поля";
     StringGrid1->Cells[3][0]="  Размер";
    //Формируем значения полей в каждой строке таблицы
     for(int i=1; i<Table1->FieldCount; i++)
     {
       StringGrid1->Cells[0][i]="   " + IntToStr(i);
       StringGrid1->Cells[1][i]=
          Table1->FieldDefs->Items[i]->Name;
       StringGrid1->Cells[2][i]=
          GetDescription(Table1->FieldDefs->Items[i]->DataType);
       StringGrid1->Cells[3][i]=
          AnsiString(Table1->FieldDefs->Items[i]->Size);
     }
   }
  //Деактивация таблицы для продолжения работы с новой базой
  Table1->Active=false;
}
//---------------------------------------------------------------------
void __fastcall TForm1::N4Click(TObject *Sender)
{//Выполнение команды "Выход"
 Close();
}
//---------------------------------------------------------------------
```

Пример просмотра БД clients.dbf приведен на рис. 10.1.

Следует обратить внимание на два момента, связанные с работой этого приложения. Во-первых, если вы скопируете в какой-то каталог только файл animals.dbf, то запуск приложения приведет к аварийной ситуации при попытке прочитать файл animals.mdx. На самом деле полный комплект файлов, представляющих эту БД в формате dBASE, состоит из 5 наборов данных — animals.dbf, animals.mdx, animals.dbt, animals.cds и animals.xml. Однако для анализа схемы таблицы достаточно двух первых файлов. Во-вторых,

точные размеры полей определяются только для символьных полей. Для большинства полей стандартного типа поле `FieldDefs->Items[i]->Size` не содержит соответствующего указания о длине. Считается, что эти значения известны по умолчанию.

Просмотр имен и типов полей в базах данных типа dBase

Файл

№ поля	Имя поля	Тип поля	Размер
1	FIRST_NAME	Символьное или строковое поле	20
2	ACCT_NBR	Поле чисел с плавающей запятой	0
3	ADDRESS_1	Символьное или строковое поле	20
4	CITY	Символьное или строковое поле	20
5	STATE	Символьное или строковое поле	2
6	ZIP	Символьное или строковое поле	5
7	TELEPHONE	Символьное или строковое поле	12
8	DATE_OPEN	Поле даты	0
9	SS_NUMBER	Поле чисел с плавающей запятой	0
10	PICTURE	Символьное или строковое поле	15
11	BIRTH_DATE	Поле даты	0

Рис. 10.1. Схема таблицы в базе данных clients

10.2. Просмотр и редактирование существующей базы данных

Второй пример демонстрирует возможность просмотра и редактирования записей существующей БД. Текст этого приложения существенно короче, т. к. используемая в нем таблица типа `TDBGrid` сама умеет отображать содержимое записей БД. Для перемещения по записям БД и управления процессом редактирования текущей записи мы воспользуемся компонентом `TDBNavigator`. Внешне он выглядит как линейка, содержащая до 10 кнопок, напоминающих панель управления магнитофоном (рис. 10.2).

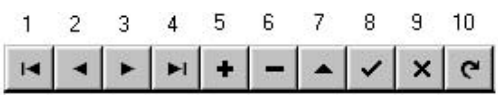


Рис. 10.2. Навигатор по базе данных (`TDBNavigator`): 1 — на первую запись; 2 — на предыдущую запись; 3 — на следующую запись; 4 — на последнюю запись; 5 — добавить запись; 6 — удалить текущую запись; 7 — редактировать запись; 8 — записать изменения в БД; 9 — отменить изменения; 10 — освежить текущую запись

Приложение 10.2 (Delphi)

Среди операций, инициируемых кнопками навигатора, в дополнительных пояснениях нуждается только последняя кнопка. Дело в том, что с одной и той же базой данных могут одновременно работать несколько приложений. И не исключена ситуация, когда одно из них внесло изменения в ту запись, которая отображена в окне нашего приложения. Вот тут-то и надо повторно прочитать информацию из БД, т. е. освежить отображаемые данные.

Наконец, для связи базы данных с указанными компонентами нам потребуется еще два объекта — `Table1` и `DataSource1`. Последний выступает как посредник между таблицей `Table1`, подключенной к конкретной БД, и таблицей `DBGrid1`, отображающей содержимое БД.

Для установки связи с базой данных country.db, представляющей в формате СУБД Paradox 7.0 сведения о странах Латинской Америки, воспользуемся логическим каталогом DBDEMOS — псевдонимом, который заменяет длинный физический путь C:\Program Files\Common Files\Borland Shared\Data

```
Table1.DatabaseName = DBDEMOS
Table1.TableName = country.db
```

Установку этих двух свойств мы выполним на стадии проектирования приложения, хотя для большей гибкости по аналогии с предыдущим примером такие назначения можно было произвести во время работы программы с помощью компонента `OpenDialog`.

Свяжем выбранные компоненты между собой:

```
DataSource1.DataSet = Table1
DBGrid1.DataSource = DataSource1
DBNavigator1.DataSource = DataSource1
```

На этом оформление приложения завершено и остается только активизировать таблицу `Table1` при запуске приложения (по аналогии с файлами базу данных положено открывать) и закрыть базу данных в момент завершения работы приложения. Эти операции выполняют два следующих обработчика событий:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 try
   Table1.Open;
   if Table1.Active then
     Caption:='Открыта база данных '+ Table1.TableName;
 except
   on E:EDBEngineError do
   begin
```

```
      ShowMessage('Ошибка при выборе БД');
      close;
    end;
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  try
  finally
    if Table1.Active then Table1.Close;
  end;
end;
```

Возможная аварийная ситуация при попытке открыть базу данных может быть связана с тем, что файл выбранной таблицы не соответствует ни одному из допустимых форматов БД, заложенных в механизм BDE. Поэтому в первом обработчике предусмотрена возможность сообщить о случившемся и прервать работу приложения. Второй обработчик перед уничтожением формы разрывает связь с недоступной по тем или иным причинам БД.

Приложение, которое умеет просматривать, добавлять новые записи, уничтожать и редактировать существующие записи и вдобавок реагировать на ошибку в формате данных файла главной таблицы, потребовало от нас всего нескольких строк кода. А ведь аналогичные утилиты отображения баз данных, входящие в состав Norton Commander'а, занимают порядка 100 Кбайт кода на каждый формат СУБД.

Внешний вид формы нашего приложения после его старта представлен на рис. 10.3. Обратите внимание на то, что к содержимому строк таблицы БД добавлена узенькая колонка слева. Она используется для индикации положения текущей записи (черный треугольник на рис. 10.3, расположенный в первой строке данных) и для фиксации строк, содержимое которых подверглось корректировке, но еще не было записано в БД. Для вновь введенных строк в соответствующей клеточке первой колонки появляется символ '*', для измененной строки — символ 'I'.

На рис. 10.3 видно, что некоторые кнопки навигатора в начальный момент недоступны. Всему есть объяснение. Если указатель текущей записи "смотрит" на первую строку таблицы, то возврат к первой строке или выбор предшествующей записи смысла не имеет. Точно также лишены смысла в этот момент такие операции, как запись изменений в БД (они еще не сделаны) или отмена сделанных исправлений.

Мы могли бы ограничить возможности приложения только просмотром существующих записей и запретить внесение каких-либо изменений в БД.

Сделать это можно разными способами. Во-первых, можно было бы изменить свойство `AutoEdit` у объекта `DataSource1`, которое по умолчанию равно `True`. Во-вторых, мы могли бы присвоить свойству `ReadOnly` у объекта `Tabl1` значение `True` (по умолчанию оно равно `False`). Наконец, из объекта `BDNavigator1` можно исключить кнопки, которые отвечают за выполнение таких операций, как вставка, удаление и обновление записей. В навигаторе имеется комплексное свойство `VisibleButtons` (видимые кнопки), при раскрытии которого появляются 10 логических свойств, каждое из которых отвечает только за свою кнопку — `nbFirst`, `nbPrior`, `nbNext`, `nbLast`, `nbInsert`, `nbDelete`, `nbEdit`, `nbPost` и `nbCancel`. Отключение любой из них (например, `nbInsert=False`) удаляет соответствующую кнопку из панели навигатора.

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300003
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000
Mexico	Mexico City	North America	1967180	88600000
Nicaragua	Managua	North America	139000	3900000
Paraguay	Asuncion	South America	406576	4660000

Рис. 10.3. Просмотр информации о странах Латинской Америки

Следует отметить, что объект `Table1` представляет собой нечто большее, чем средство связи с конкретной БД. Во-первых, он позволяет отобразить содержание записей выбранной БД в таблице `DBGrid1` уже на стадии проектирования приложения. Для этого достаточно после задания свойств `DatabaseName` и `TableName` активизировать объект:

```
Table.Active = True
```

Правда, после такой процедуры невозможно переустановить имена БД и соответствующей таблицы без предварительной деактивации объекта `Table1`.

По сути дела, объект типа `TTable` представляет собой миниатюрную СУБД, которая автоматически определяет тип таблиц БД, подключенной к объекту,

извлекает из обрабатываемой базы данных информацию о структуре и типах полей, о количестве записей, об индексах. Этот объект умеет произвести отбор записей, значения указанных полей которых удовлетворяют некоторым требованиям. Для этой цели используются свойства `Filter` и `Filtered`. Значение первого поля задает фильтр отбора на языке, представляющем собой некоторое подобие Паскаля:

```
Table1.Filter = Continent = 'South America' AND Area < 1000000
```

Свойство `Filtered` задается равным `True` для включения фильтра отбора. Без этого установленная фильтрация не действует.

Свойство отображать содержимое таблиц БД не является единственной функцией объекта `TDBGrid`. Он позволяет менять местами колонки таблицы, располагает собственным всплывающим меню (рис. 10.4), с помощью которого можно производить редактирование столбцов — вставлять новые, удалять или модифицировать существующие, перестраивать индексы.

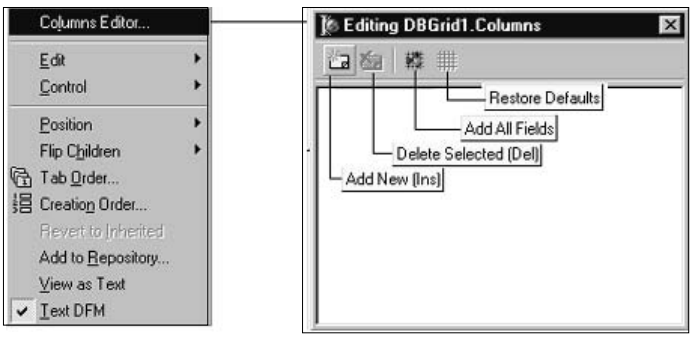


Рис. 10.4. Всплывающее меню объекта `TDBGrid` и редактор колонок

Приложение 10.3 (Delphi)

Несмотря на то, что таблица `DBGrid1` прекрасно справилась с базой данных, представляющей страны, столицы, территорию и население Латинской Америки, на другой базе данных — animals.dbf нас ждет некоторое разочарование (рис. 10.5). Содержимое полей числовых и символьных типов отображается нормально, а вместо содержимого поля BMP, представленного графическим изображением, на экране появилось сообщение о типе данных (`TYPEDBINARY` — типизированное двоичное поле). Вспомните ситуацию с отображением икон среди строк обычного списка. Там тоже для воспроизведения изображения приходилось совершать обходные маневры, связанные с назначением нестандартной ширины строк и подключением программы, обеспечивающей отображение объектов, ассоциированных с текстовыми строками.

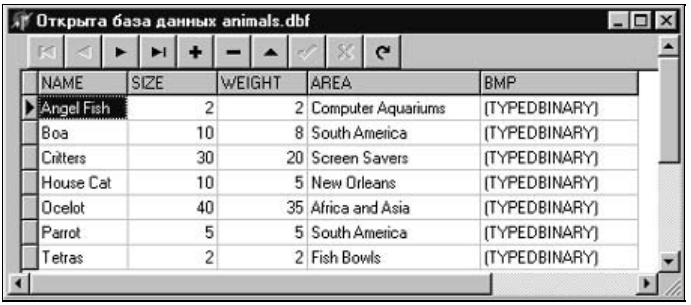


Рис. 10.5. Неудавшаяся попытка отобразить записи с рисунками

Приложение 10.4 (Delphi)

Примерно по такому же пути придется пойти и в случае базы с картинками. Чтобы не создавать на экране зоопарк, попробуем отображать содержимое поля `BMP` только для текущей записи. В этом нам поможет объект `DBImage1`, который мы добавим к приложению и настроим его на работу с выбранной таблицей и полем `BMP`:

```
DBImage1.DataSource = DataSource1
DBImage1.DataField = BMP
```

Эти назначения можно сделать как на стадии проектирования, так и во время выполнения приложения. Правда, во время проектирования мы не сможем перемещаться по всем записям БД, а увидим только первую или последнюю строку (рис. 10.6).

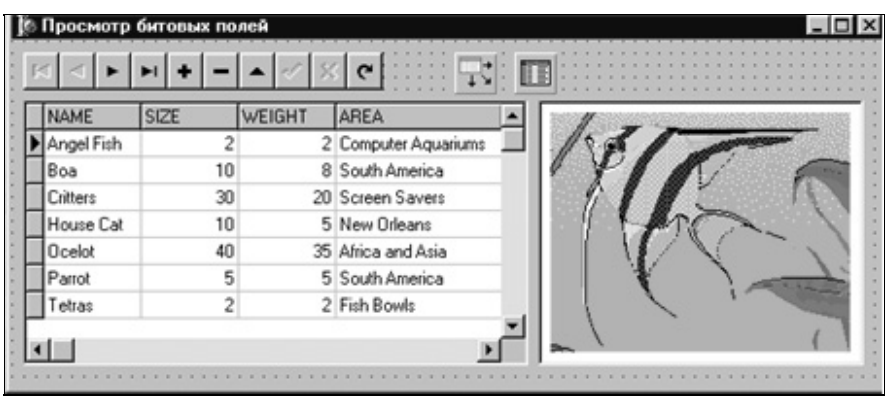


Рис. 10.6. Просмотр bmp-изображений на этапе проектирования

Во время выполнения приложения, для создания которого мы не написали ни одной строки, можно перемещаться по всем записям БД (рис. 10.7).

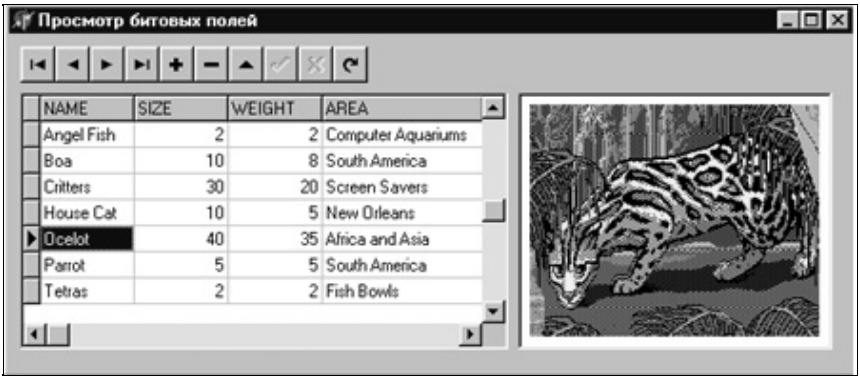


Рис. 10.7. Просмотр записей БД на этапе выполнения

Приложение 10.5 (Delphi)

Не справляется таблица `TDBGrid` и с содержимым полей типа `Memo`. Такое поле, например, имеется в БД biolife.db. Его наименование — Notes (Замечания). Для того чтобы содержимое таких полей появилось в окне нашего приложения, положим на форму объект `DBMemo1` и настроим два его свойства:

```
DBMemo1.DataSource = DataSource1
DBMemo1.DataField = Notes
```

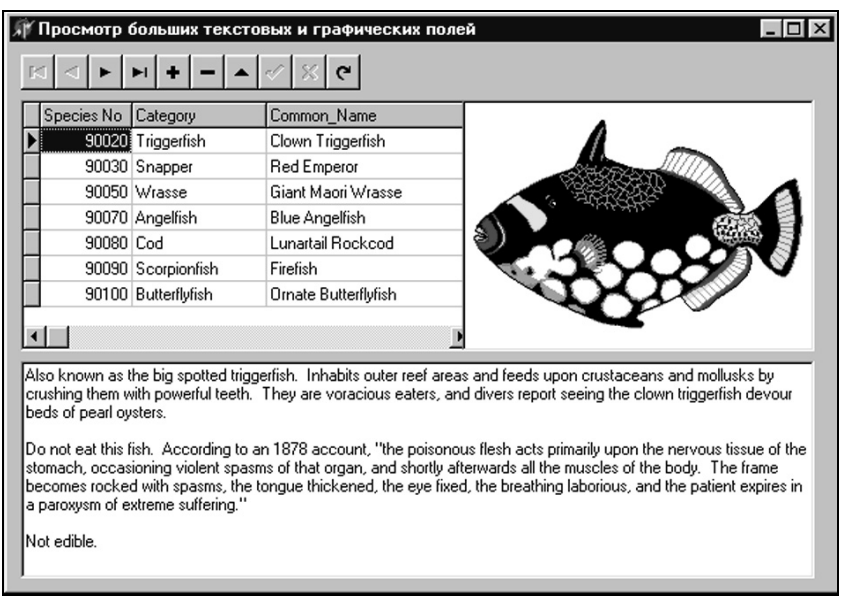


Рис. 10.8. Использование объектов `DBImage` и `DBMemo` для просмотра полей

Результат просмотра новой базы представлен на рис. 10.8. Окно в нижней части формы представляет содержимое поля `Notes`.

Среди объектов, расположенных на линейке `Data Controls`, довольно много наименований, отличающихся от ранее встречавшихся только добавкой двух символов DB — `DBText`, `DBEdit`, `DBListBox`, `DBComboBox`, `DBCheckBox`, `DBRadioGroup`, `DBRichEdit`. Все они выполняют прежние функции, и единственная их новая особенность заключается в использовании свойств `DataSource` и `DataField`, позволяющих связаться с конкретным полем таблицы БД, подключенной к приложению. Среди такого рода объектов обратим ваше внимание на очень мощный графический компонент `DBChart`, позволяющий подключить окно, в котором можно отображать различные диаграммы, хранящиеся в соответствующем поле таблицы.

10.3. Создание базы данных с нуля

Создание таблиц, наполненных соответствующими наборами данных в формате той или иной СУБД, можно осуществить несколькими способами. Естественный способ состоит в использовании специализированных средств соответствующей СУБД. Но они не всегда могут оказаться под рукой. И тогда на выручку приходит одна из сервисных программ, включенных в состав соответствующей визуальной системы программирования. В системе Visual Basic таким редактором баз данных является утилита Visual Data Manager. В системах BCB и Delphi аналогичную роль играет утилита Database Desktop (дословно — настольная база данных).

10.3.1. Основные возможности Database Desktop (DBD)

Настольная СУБД запускается из раздела **Tools** командой **Database Desktop**. Она обеспечивает доступ к любым записям таблиц различных баз данных типа dBASE, Paradox, Access, FoxPro, а также к БД, расположенным на удаленных серверах и допускающим общение в режимах ODBC и SQL. Для уменьшения объема обрабатываемой информации имеется возможность ограничиться подмножеством записей из заданного диапазона или удовлетворяющих заданным условиям. Такого рода отбор интересующих нас данных принято называть фильтрацией.

Общий вид DBD, в которой после запуска инициирована команда **File→New→Table**, смонтирован на рис. 10.9. Он несколько отличается от реального изображения расположением окон и тем, что здесь полностью раскрыт список обрабатываемых версий БД.

Команды меню **File** (Файл) довольно стандартные — открыть существующую БД (**Open**), создать новую (**New**), закрыть текущую (**Close**) или сохранить ее

(**Save**, **Save As**). Кроме того, имеется возможность назначить рабочий и/или личный каталог, в котором DBD размещает различные файлы, создаваемые в процессе работы. После выбора команды **File→New→Table** и задания типа таблицы (например, Paradox 7) на экране появляется основное окно, в котором описывается структура вновь создаваемой таблицы (рис. 10.10)

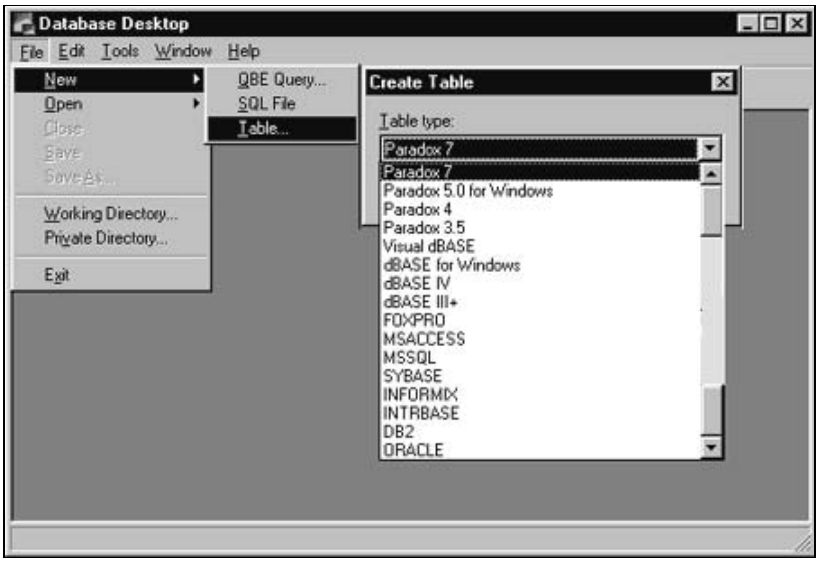


Рис. 10.9. Допустимые форматы вновь создаваемых таблиц в `Database Desktop`

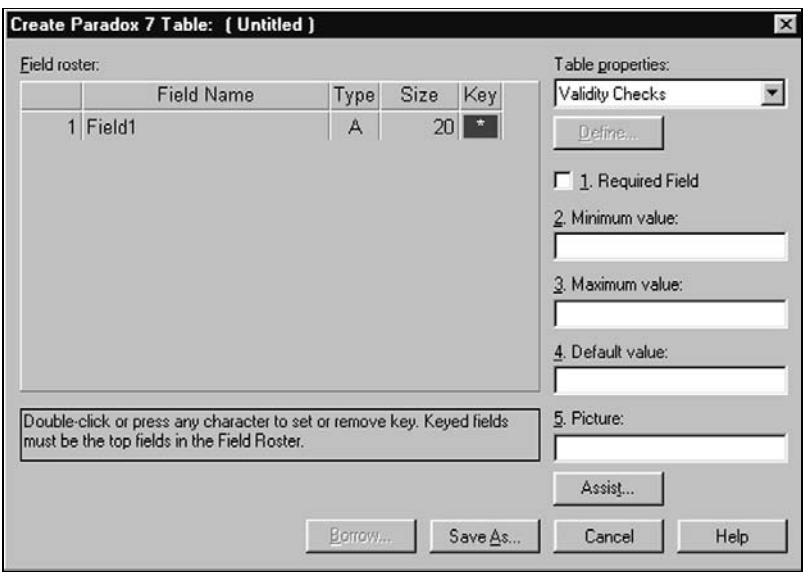


Рис. 10.10. Диалоговое окно для создания таблицы

Большое поле слева предназначено для размещения *списка полей* (Field roster — реестр полей). В каждой колонке надо набрать соответствующее значение: в колонке **Field Name** (Наименование поля), в колонке **Type** (Тип) символ, определяющий тип значения, в колонке **Size** (Размер) — размер поля в байтах, если это необходимо, в колонке **Key** (Ключ) — символ `*`, если это поле является ключевым. Никто не заставляет вас наизусть помнить все буквы, соответствующие тому или иному типу поля. Когда курсор мыши находится в колонке **Type** (Тип) и вы нажали правую клавишу мыши, то появляется всплывающее меню (рис. 10.11), в котором достаточно выбрать нужную строку.

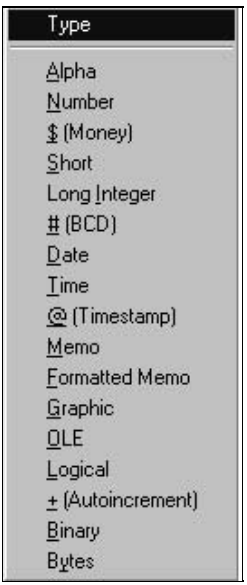


Рис. 10.11. Типы полей в БД Paradox

Символ `*` набирается нажатием клавиши <Пробел> и уничтожается при повторном нажатии той же клавиши. Дополнительные поля в правой части позволяют сформировать условия правильности значения, которое будет вводиться в указанное поле. Галочка в окошке **Required Field** (Обязательное поле) означает, что в этом поле значение должно присутствовать обязательно. Для проверки величин числовых значений (включая даты, денежные суммы и т. п.) предусмотрена возможность задания допустимых интервалов (**Minimum value** — минимальное значение, **Maximum value** — максимальное значение). Окно **Default value** (Значение по умолчанию) позволяет задать значение по умолчанию, которое система будет вставлять автоматически, если вы его не наберете сами. В окне **Picture** (название, на наш взгляд, не самое удачное) можно набрать символьный шаблон, похожий на маску, приписанную к окну ввода типа `EditMask`. С его помощью можно проконтролировать

набор тех или иных символов в заданных позициях (например, маска для телефона, номера страхового свидетельства и т. п.).

При формировании схемы таблицы довольно часто приходится пользоваться ранее созданными схемами, внося в них некоторые изменения. Для этой цели в окне **Create Table** (Создать таблицу) предусмотрена кнопка **Borrow** (Заимствовать).

Сформированная таким образом структура таблицы сохраняется и в дальнейшем может быть переопределена. Для этой цели у DBD в разделе **Tools** (Инструменты) представлен довольно широкий набор утилит, позволяющих производить реструктуризацию таблиц и выполнять над таблицами различные операции (рис. 10.12).

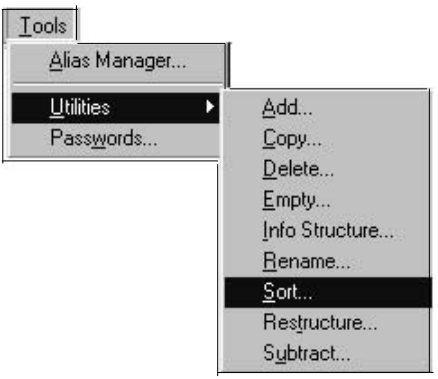


Рис. 10.12. Средства редактирования структуры таблицы

Заполнение строк таблицы производится в DBD после открывания существующей, но пока еще пустой таблицы. Вообще говоря, выбор структуры таблиц в конкретном приложении, формирование первичных и вторичных ключей, установление перекрестных ссылок между зависимыми таблицами, формулирование запросов и составление отчетов по информации, извлеченной из БД, — это довольно серьезная работа, требующая знаний многочисленных деталей и правил, диктуемых конкретной СУБД. Такого рода технологиям посвящаются специальные книги, тогда как свою задачу мы видим в демонстрации некоторых возможностей визуальных сред для работы с БД.

Довольно большую пользу можно извлечь, просматривая с помощью DBD структуру существующих БД. Для этого в главном меню DBD нужно выбрать раздел **Tools** (Инструменты) и выполнить команды **Utilities→Info Structure**. После этого на экране возникнет диалоговое окно, предлагающее выбрать нужную таблицу. Окно просмотра структуры выбранной таблицы представлено на рис. 10.13.

Обратите внимание на раскрывающийся список **Table properties** (Свойства таблицы). В этом списке перечислены следующие свойства таблиц Paradox:

- **Validity Checks** — контроль правильности данных;

- **Table Lookup** — таблица синхронного просмотра;
- **Secondary Indexes** — вторичные индексы;
- **Referential Integrity** — ссылочная целостность;
- **Table language** — язык таблицы (аналог кодовой страницы Windows);
- **Dependent Tables** — подчиненные таблицы.

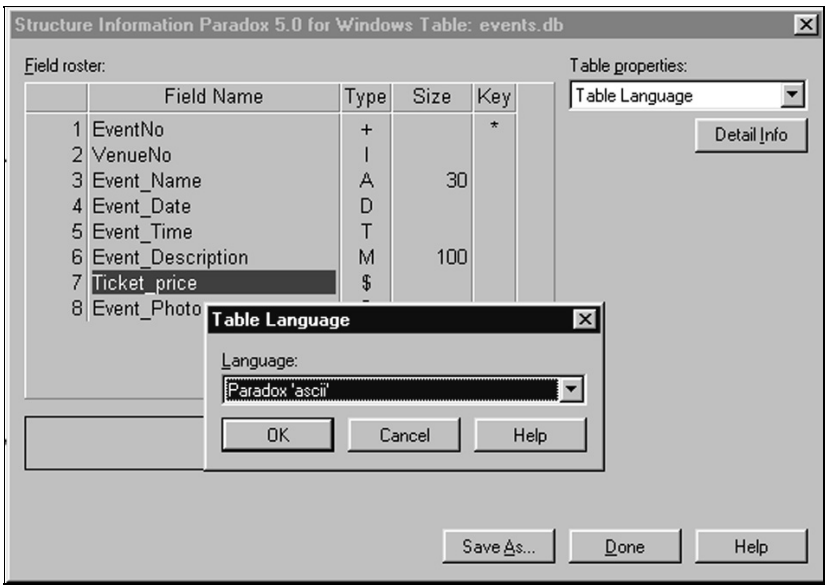


Рис. 10.13. Окно просмотра структуры и свойств таблицы БД

Когда вы раскрываете этот список для выбранной строки (записи) таблицы, то либо в окне под списком, либо в диалоговом окне, раскрывающемся после нажатия на кнопку **Detail Info** (Подробная информация), можно увидеть те или иные сведения по поводу выбранного свойства. Например — наименования полей, составляющих вторичный индекс, список подчиненных таблиц и др. Кнопка **Detail Info** (Подробная информация) становится видимой или доступной только тогда, когда имеется дополнительная информация. На рис. 10.13 диалоговое окно сообщает, что в поле **Цена билета** используется язык Paradox 'ascii'. Вполне возможно, что эта информация не представляет особой ценности.

10.3.2. Создание таблицы программным путем

Компонент `Table` может быть использован не только для связи с существующей таблицей, но и для создания новой таблицы. В файле помощи системы Delphi 6.0 приведен следующий пример создания условной БД типа Paradox с именем `CustInfo`.

Приложение 10.6 (Delphi)

Мы его слегка модифицировали и снабдили подробным комментарием. Каждая строка таблицы состоит из двух полей — целочисленное поле с именем `Field1` и символьное поле с именем `Field2`. Первое из них является ключевым и по правилам Paradox образует первичный индекс. Содержимое второго поля объявляется вторичным индексом. Существенный момент — к моменту создания таблицы объект `Table1` должен быть не активен, т. е. не должен быть связан с существующей базой. Комментарий, которым снабжена почти каждая строка программы, делает излишним какие-либо дополнительные пояснения

```
with Table1 do begin
  Active:=False; {Деактивация объекта Table1}
  DatabaseName:='DBDEMOS'; {Назначение псевдонима}
  TableType:=ttParadox;    {Задание типа таблицы}
  TableName:='CustInfo';   {Указание имени таблицы}
{Чтобы не затереть существующую таблицу}
  if not Table1.Exists then begin
{Объявляем поля таблицы}
    with FieldDefs do begin
      Clear; {Очистка массива определений полей}
      with AddFieldDef do begin
        Name:='Field1'; {Указание имени первого поля}
        DataType:=ftInteger; {Задание типа первого поля}
        Required:=True; {Признак обязательного значения}
      end;
      with AddFieldDef do begin
        Name:='Field2'; {Указание имени второго поля}
        DataType:=ftString; {Задание типа второго поля}
        Size:=30;  {Задание размера символьного поля}
      end;
    end;
{Создание первичного и вторичного индексов}
    with IndexDefs do begin
      Clear; {Очистка массива описаний индексов}
{Первичный индекс в Paradox не имеет имени т. к. он является первичным
ключом в БД}
      with AddIndexDef do begin
        Name:=''; {Имя первичного ключа — пустое}
        Fields:='Field1'; {Имя поля, задающего первичный ключ}
```

```
        Options:=[ixPrimary]; {Тип ключа — первичный}
      end;
      with AddIndexDef do begin
        Name:='Fld2Indx'; {имя вторичного ключа}
        Fields:='Field2'; {Имя поля, задающего вторичный ключ}
        Options:=[ixCaseInsensitive]; {Нечувствительность к регистру }
      end;
    end;
{Вызов метода CreateTable для создания таблицы}
    CreateTable;
  end;
end;
```

10.4. Доступ к полям таблицы и организация запросов

Визуальный просмотр содержимого таблиц БД — далеко не единственная операция, которая может потребоваться при создании приложений, работающих с базами данных. Многое из того, что "умели" делать специализированные компоненты типа `DBGrid` и `DBNavigator`, приходится выполнять программным путем. В первую очередь следует освоиться с техникой перемещения по записям таблицы, с доступом к значению, расположенному на индивидуальном поле текущей записи, с выбором записи, содержимое которой удовлетворяет некоторым условиям.

Приложение 10.7 (BCB)

Для демонстрации основных методов, решающих поставленные задачи, воспользуемся существующей базой данных animals.dbf и построим следующее приложение. Так как каждая запись в этой БД содержит 5 полей, разместим на форме 4 объекта типа `TLabel` с именами алфавитно-цифровых полей — `NAME`, `SIZE`, `WEIGHT` и `AREA`. Рядом с каждой меткой расположим по одному полю типа `TDBText`, в котором автоматически будет отображаться содержимое соответствующего поля. Для поля `BMP`, содержащего рисунок в формате bmp, предусмотрим объект типа `TDBImage`. Связь с базой данных выполним с помощью компонент `Table1` и `DataSource1`. Установим значения необходимых свойств и активизируем объект `Table1`:

```
Table1->DatabaseName = DBDEMOS
Table1->TableName = animals.dbf
DataSource1.DataSet = Table1
DBText1->DataSource = DataSource1
```

```
DBText1->DataField = NAME
DBText1->DataSource = DataSource1
DBText1->DataField = SIZE
DBText1->DataSource = DataSource1
DBText1->DataField = WEIGHT
DBText1->DataSource = DataSource1
DBText1->DataField = AREA
DBImage1-> DataSource = DataSource1
DBImage1->DataField = BMP
Table1 ->Active = True
```

Содержимое указанных полей для текущей записи система программирования отображает сама. Однако перемещения по записям таблицы и выборку значения указанного пользователем поля мы запрограммируем с использованием соответствующих методов. Расположим на форме 4 кнопки с надписями **First** (Первая), **Prior** (Предыдущая), **Next** (Следующая) и **Last** (Последняя). Эти кнопки будут инициировать соответствующие процедуры навигации по записям с помощью методов с аналогичными названиями.

В поле `Edit1` мы предоставим пользователю возможность набрать одно из названий полей текущей записи, и после нажатия клавиши <Enter> воспроизведем содержимое этого поля, воспользовавшись соответствующей строкой массива `DataSource1->DataSet->FieldValues`. Индексом этой строки является наименование поля. При явном задании имени поля в BCB его заключают в двойные кавычки, а в Delphi — в одинарные (так же, как и любую текстовую константу). Чтобы синхронизовать изменения содержимого поля `Edit1` с перемещениями по записям, предусмотрим в каждом из обработчиков событий `OnClick` для кнопок необходимое присваивание:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{//Перемещение на первую запись
  DataSource1->DataSet->First();
  if(Edit1->Text!="")
     Label9->Caption=DataSource1->DataSet->FieldValues[Edit1->Text];
}
//---------------------------------------------------------------------

void __fastcall TForm1::Button2Click(TObject *Sender)
{//Перемещение на предыдущую запись
  DataSource1->DataSet->Prior();
  if(Edit1->Text!="")
     Label9->Caption=DataSource1->DataSet->FieldValues[Edit1->Text];
}
```

```
//---------------------------------------------------------------------

void __fastcall TForm1::Button3Click(TObject *Sender)
{//Перемещение на следующую запись
  DataSource1->DataSet->Next();
  if(Edit1->Text!="")
     Label9->Caption=DataSource1->DataSet->FieldValues[Edit1->Text];
}
//---------------------------------------------------------------------

void __fastcall TForm1::Button4Click(TObject *Sender)
{//Перемещение на последнюю запись
  DataSource1->DataSet->Last();
  if(Edit1->Text!="")
     Label9->Caption=DataSource1->DataSet->FieldValues[Edit1->Text];
}
//---------------------------------------------------------------------
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{
  if(Key==13)
     Label9->Caption=DataSource1->DataSet->FieldValues[Edit1->Text];
}
```

Рабочий фрагмент этого приложения приведен на рис. 10.14. Глядя на форму, вы можете убедиться в том, что наименование поля может набираться как строчными, так и прописными буквами.

Рис. 10.14. Доступ к содержимому полей текущей записи

В приведенном приложении есть одно слабое место: если пользователь наберет в окне ввода имя несуществующего поля, то при выборке из массива строк `FieldValues` возникнет исключительная ситуация `EDatabaseError`. Ее обработку конечно можно было предусмотреть, но мы не хотели перегружать текст программы.

Кроме использованных в приложении методов `First`, `Prior`, `Next` и `Last`, можно обратиться к методу `MoveBy(k)`, который позволяет перейти от текущей записи к записи, отстоящей от нее на `k` строк вперед (`k>0`) или назад (`k<0`). Перемещения по записям таблицы очень напоминают перемещение указателя в двоичных и типизированных файлах.

Логическая функция `Locate` осуществляет поиск указанного значения в заданном столбце таблицы:

```
if(Table1->Locate("NAME", "Ocelot", q) {...}
```

Первый аргумент задает наименование поля таблицы, среди записей которой производится поиск. Второй аргумент представляет поисковый образ. Наконец, последний параметр, имеющий тип `TLocateOptions`, может принимать одно из двух значений — `loCaseInsensitive` или `loPartialKey`. Он определяет способ сравнения заданного образа со значениями проверяемых полей. В первом случае сравнение производится с игнорированием разницы между большими и малыми буквами. Во втором случае сравнение считается удачным, если поисковый образ совпадает с начальными символами значения указанного поля. Если функция `Locate` возвращает значение `True`, то найденная запись становится текущей.

Вообще говоря, второй аргумент функции `Locate` задается значением типа `Variant` и, в частности, может быть представлен массивом значений. Это позволяет в качестве первого параметра задавать список полей, значения которых сравниваются с соответствующими элементами массива. Список имен полей разделяется точками с запятыми:

```
TLocateOptions q;
.........................
q.Clear();          //очистка множества q от предыдущих значений
q << loPartialKey; //формирование признака частичного совпадения
Variant a[2];       //объявление массива поисковых образов
a[0]=Variant("Oc");//задание значений образов
a[1]=Variant("Africa");
if(Table1->Locate("NAME;AREA",VarArrayOf(a,1),q){...}
```

Однако полные или частичные совпадения значений поля с заданными величинами конечно же не могут удовлетворить условиям разнообразных запросов, которые возникают на практике. Во-первых, вас могут заинтересовать значения числовых полей, которые принадлежат заданному интервалу.

Например, вы хотите выбрать сканер, обладающий разрешением не ниже 1200 dpi, подключающийся к порту USB, стоимость которого не превышает 75$.

Большинство нормальных БД позволяют осуществлять такого рода поиск по запросам, сформированным в стиле фильтров, используемых при отборе записей. В базах данных разного типа структуры таких запросов могут слегка отличаться. Поэтому пользователям BCB или Delphi предлагается некий посредник — `TQuery`, который трансформирует ваши запросы на некотором подмножестве языка SQL (Structural Query Language) в диалект запроса конкретной БД. Компонент `Query` выполняет те же функции по связи с БД, что и объект `Table`, но дополнительно располагает свойством SQL, оформленным в виде списка строк. В этом списке могут располагаться команды языка SQL, которые используются не только для отбора нужных записей, но и для других манипуляций над строками таблиц, таких как вставки, замены, удаления сортировки и т. п. Более подробно операторы языка SQL обсуждаются в *разд. 10.5 данной главы.*

10.5. Связь с базами данных в системе Visual Basic

Связь с локальными БД в среде VB осуществляется с помощью объекта `Data`, расположенного на стандартной палитре компонент. Для установки контакта с БД, тип которой не является "родным" для Microsoft, приходится формировать значения следующих пяти свойств:

```
Data1.Connect = Paradox 5.x
Data1.DatabaseName = c:\Program Files\Common Files\Borland Shared\Data
Data1.DefaultType = 2 — Use Jet
Data1.RecordsetType = 0 — Table
Data1.RecordSource = biolife
```

Приложение 10.8 (VB)

Вообще говоря, установку значений свойств следует начинать с `DefaultType` и `RecordsetType`. После этого при установке свойства `Connect` достаточно выбрать нужную строку из раскрывающегося списка (рис. 10.15).

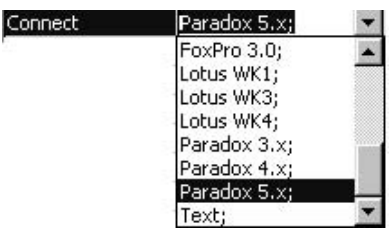


Рис. 10.15. Выбор типа БД

Следующим должно быть значение свойства `DatabaseName`, прокладывающее путь к каталогу с базами данных. Теперь для выбора конкретной таблицы можно воспользоваться услугами списка, раскрывающегося в свойстве `RecordSource`.

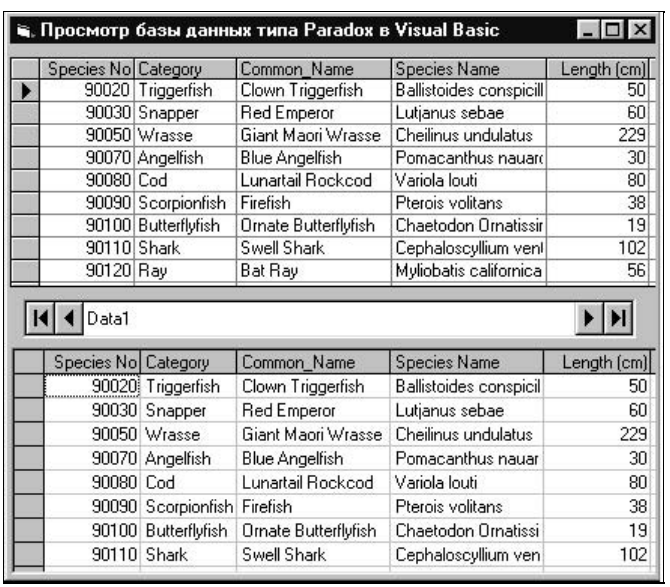
После того, как связь с таблицей БД установлена, объект `Data` "знает" наименования и типы большинства полей в записях. Однако поля с графическими образами в "чужих" базах данных компонент `Data` "не видит".

Для отображения наименований символьных и числовых полей достаточно разместить на форме один из объектов `DBGrid` или `MSFlexGrid` и связать свойство любой из этих таблиц `DataSource` с объектом `Date`:

```
DBGrid1.DataSource = Data1
MSFlexGrid.DataSource = Data1
```

В стандартной палитре объектов ни `DBGrid`, ни `MSFlexGrid` не присутствуют, и вам придется совершить вояж в раздел главного меню **Project → Components** и отметить галочкой одну из строк **Microsoft Data Bound Grid Control 5.0 (SP3)** или **Microsoft FlexGrid Control 6.0** (или пометить каждую из них).

После запуска такого приложения, в котором нам не пришлось написать ни единой строки, можно наблюдать картину, приведенную на рис. 10.16. В этом приложении на форму помещены оба компонента: вверху — `DBGrid`, внизу — `MSFlexGrid`.



Species No	Category	Common_Name	Species Name	Length (cm)
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicill	50
90030	Snapper	Red Emperor	Lutjanus sebae	60
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229
90070	Angelfish	Blue Angelfish	Pomacanthus nauar	30
90080	Cod	Lunartail Rockcod	Variola louti	80
90090	Scorpionfish	Firefish	Pterois volitans	38
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissir	19
90110	Shark	Swell Shark	Cephaloscyllium vent	102
90120	Ray	Bat Ray	Myliobatis californica	56

Species No	Category	Common_Name	Species Name	Length (cm)
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicil	50
90030	Snapper	Red Emperor	Lutjanus sebae	60
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229
90070	Angelfish	Blue Angelfish	Pomacanthus nauar	30
90080	Cod	Lunartail Rockcod	Variola louti	80
90090	Scorpionfish	Firefish	Pterois volitans	38
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissi	19
90110	Shark	Swell Shark	Cephaloscyllium ven	102

Рис. 10.16. Просмотр содержимого полей базы biolife.db

Вообще говоря, реальная картина, наблюдаемая после запуска такого приложения, несколько отличается от того, что приведено на рис. 10.16. Все дело в установках свойств DefColWidth (Определение ширины колонок), Width (Ширина таблицы) и косметической регулировке ширины отдельных колонок после запуска приложения. По умолчанию DefColWidth=0, что соответствует автоматическому выбору ширины колонок. Но система делает эту операцию не очень аккуратно, и можно наблюдать усеченные наименования и оборванные значения в отдельных колонках. Обе таблицы допускают ручную регулировку как ширины колонок, так и высоты строк. Только для этого курсор мыши следует поместить на нужную границу в выделенной строке или колонке, зажать левую клавишу и переместить границу между полями в нужную позицию. Обратите внимание, что колонка Notes в этой таблице содержит достаточно длинный комментарий (BLOB Memo). Поэтому на стадии проектирования можно установить ширину таблицы достаточно большой или использовать для просмотра этого поля в текущей записи отдельный компонент, например — многострочный TextBox.

Большинство средств VB для доступа к данным ориентировано на просмотр полей текущей записи с использованием обычных компонент (TextBox, Label, MaskEdit, ListBox, ComboBox, Image, PictureBox). Эти объекты располагают свойствами DataSource и DataField, которые должны быть привязаны к источнику данных Data и имени соответствующего поля записи. Содержимое полей типов Text, Numeric или Date может отображаться в объектах Label, TextBox, ListBox, ComboBox, MaskEdit. Для полей логического типа, значения которых True/False или Yes/No, подходит компонент CheckBox.

Закрепляя ту или иную таблицу БД за объектом Data, вы можете выбрать один из трех типов наборов данных, изменяя свойство RecordsetType.

Частному, но достаточно распространенному случаю соответствует RecordsetType = Table, когда набор данных представлен единственной физической таблицей. К его преимуществам относится возможность использовать индексы для сортировки данных и мгновенное отображение модифицированных данных.

Второй тип данных (RecordsetType = Dynaset) представляет собой динамический набор, полученный в результате выборки данных из одной или нескольких таблиц. Работать с ним можно почти что так же, как и с таблицей. Данные могут обновляться, однако в этом случае вам недоступны индексы, и модифицированные данные отображаются не сразу. Количество записей в динамическом наборе может меняться в процессе работы приложения.

Последний тип набора данных (RecordsetType = Snapshot) представляет собой моментальный снимок содержимого одной или нескольких таблиц, из которых выбраны записи, удовлетворяющие заданным требованиям. Эти данные копируются в оперативную память вашего приложения, но из них можно только считывать информацию.

База данных MS Access является родной для системы VB, и с ее таблицами приложения, написанные на VB, управляются лучше всего. В отличие от dBASE и Paradox все таблицы БД Access располагаются в одном файле с расширением .mdb. Одна из таких баз с библиографией по компьютерной литературе поставляется в составе системы Visual Basic с незапамятных времен. Файл biblio.mdb состоит из четырех таблиц с именами `Titles` (Наименования книг), `Authors` (Фамилии авторов), `Publishers` (Издатели) и `Title Author`. Последняя таблица содержит связи между авторами (точнее, их идентификационными номерами) и международными номерами (ISBN) написанных ими книг. Наименования полей, их типы и описание хранимых значений приведены в табл. 10.2.

Таблица 10.2. *Описание базы данных biblio.mdb*

Номер поля	Название поля	Тип	Длина	Пояснение
		Таблица `Titles`		
1	`Title`	Текстовый	255	Название книги
2	`Year Published`	Числовой		Год издания
3	`ISBN`	Текстовый	20	Международный номер книги
4	`PubID`	Счетчик		Идентификационный номер издателя
5	`Description`	Текстовый	50	Таблицы, страницы, индекс
6	`Notes`	Текстовый	50	Общие замечания
7	`Subject`	Текстовый	50	Номер издания, наличие диска и т. п.
8	`Comments`	`MEMO`		Краткая аннотация
		Таблица `Authors`		
1	`Au_ID`	Счетчик		Идентификационный номер автора
2	`Author`	Текстовый	50	Имя, фамилия
3	`Year Born`	Числовой		Год рождения
		Таблица `Publishers`		
1	`PubID`	Счетчик		Идентификационный номер издателя
2	`Name`	Текстовый	50	Наименование издателя
3	`Company Name`	Текстовый	255	Наименование компании

Таблица 10.2 *(окончание)*

Номер поля	Название поля	Тип	Длина	Пояснение
		Таблица `Publishers`		
4	`Address`	Текстовый	50	Адрес
5	`City`	Текстовый	20	Город
6	`State`	Текстовый	10	Штат
7	`Zip`	Текстовый	15	Почтовый код
8	`Telephone`	Текстовый	15	Телефон
9	`Fax`	Текстовый	15	Факс
10	`Comments`	`MEMO`		Комментарий
		Таблица `Title Author`		
1	`ISBN`	Текстовый	20	Международный код книги
2	`Au_ID`	Счетчик		Идентификационный код автора

Структуру таблиц пришлось выяснять самым простым способом, воспользовавшись услугами MS Access. Однако названия полей, их размеры и смысл нам понадобятся в последующих примерах фрагментов программ и приложений на VB.

Надо признать, что БД biblio.mdb представляет собой далеко не учебный пример. Она содержит более 8 500 наименований книг по программированию, выпущенных в период с 1970 по 1996 гг. Список авторов превышает 6 000 фамилий. К сожалению, в этой библиографии представлены только зарубежные авторы. Далеко не все поля таблиц заполнены (в большей степени это относится к годам рождения авторов и адресам издательств). Встречаются и курьезные опечатки. Больше других не повезло ряду книг по dBASE, которые якобы публиковались в начале прошлого века (см. записи книг с номерами 329, 2104—2106, 2212, 2213). А "Введению в объектно-ориентированные БД" (запись 418) вообще приписана фантастическая дата выхода в свет — 19 год (хорошо еще, что новой эры). Поиску подобных опечаток может быть посвящена программа отбора "плохих" записей, которую вы сумеете написать сами.

Типы данных в полях таблиц MS Access немного отличаются от ранее приводившихся. Поэтому в табл. 10.3 вы можете найти некоторые детали представления числовых, символьных и других данных.

Таблица 10.3. *Описание типов данных, принятых в СУБД MS Access*

Обозначение	Тип данных	Диапазон допустимых значений
`Binary`	Двоичные данные	До 1.2 Гбайт
`Boolean`	Логические данные	`True` или `False`
`Byte`	Целое число, 1 байт	От 0 до 255
`Currency`	Денежные единицы	От -9×10^{14} до 9×10^{14}
`Date`	Дата и время	От 1.01.100 до 31.12.9999
`Double`	Вещественное число, 8 байт	От -1.8×10^{308} до 1.8×10^{308}
`Integer`	Целое число, 2 байта	От –32 768 до 32 767
`Long`	Целое число, 4 байта	От –2 147 483 648 до 2 147 483 647
`Memo`	Длинная строка	До 1.2 Гбайт
`OLE`	OLE-объект	До 1.2 Гбайт
`Single`	Вещественное число, 4 байта	От -3.4×10^{38} до 3.4×10^{38}
`Text`	Строка	До 255 символов

Представим себе, что нам понадобилась ссылка на книгу, у которой мы точно знаем название, не помним год издания и забыли фамилии авторов. Техника поиска недостающих данных может быть такова. По таблице `Titles` находим запись с известным содержимым поля `Title`. В этой же записи находится и поле с годом издания, однако фамилии авторов отсутствуют. Зато есть международный код книги — поле `ISBN`, по значению которого можно отыскать все строки таблицы `Title Autor`. Наконец, по всем значениям поля `Au_ID` мы заходим в таблицу `Authors`, и отыскиваем недостающие данные. Этот пример понадобился для того, чтобы продемонстрировать реальную цепочку запросов к БД для поиска нужной информации. Чуть ниже будет показано, что средства управления БД позволяют производить поиск не только по точно известным значениям тех или иных полей, но по так называемым "мягким" запросам, когда нам известно одно или несколько ключевых слов, входящих, например, в название книги.

Приложение 10.9 (VB)

Первая дилемма, возникающая при создании программы просмотра той или иной таблицы, — выводить на экран несколько записей в виде таблицы или просматривать поля единственной (текущей) записи, подобрав нужное количество соответствующих компонент. Однозначного ответа здесь нет, хотя в таблицах с очень длинными строками удобнее просматривать записи по одной. Но мы продемонстрируем оба подхода.

В любом случае нам понадобится объект `Data`, который обеспечит связь с базой данных и позволит выбрать нужную таблицу. По умолчанию связь устанавливается с базой данных типа MS Access (`Data1.Connect=Access`). В свойстве `Data1.DatabaseName` прописываем полный путь к БД, находящейся в каталоге c:\Program Files\Microsoft Visual Studio\VB98\biblio.mdb.

Определяем таблицу для просмотра `Data1.Recordset=Titles`. Устанавливаем тип набора данных `RecordsetType=1 – Dynaset` и на этом настройка объекта `Data1` завершена.

Для отображения всех полей таблицы `Titles` нам понадобится объект `DBGrid`, у которого придется настроить его единственное свойство:

```
DBGrid1.DataSource=Data1
```

Остается добавить две метки — `Label1` и `Label2`, предназначенные для отображения общего количества записей в таблице (свойство `Data1.Recordset.RecordCount`) и номера текущей записи (свойство `Data1.Recordset.AbsolutePosition`). Для корректной работы последнего свойства выбран тип таблицы `Dynaset`.

```
Dim k As Integer
Private Sub Data1_Reposition()
  k = Data1.Recordset.AbsolutePosition
  Label2.Caption = "Текущая запись = " + Str(k)
End Sub

Private Sub Data1_Validate(Action As Integer, Save As Integer)
  Label1.Caption = "Всего записей = " + Str(Data1.Recordset.RecordCount)
End Sub
```

Результат просмотра представлен на рис. 10.17. И неудобства его очевидны — автоматически подобранный размер полей не всегда удовлетворителен (правда, границы полей можно перемещать), отдельные поля не видны (но есть возможность скроллинга), трудно подобрать удовлетворительные размеры формы для просмотра содержимого достаточно длинных полей.

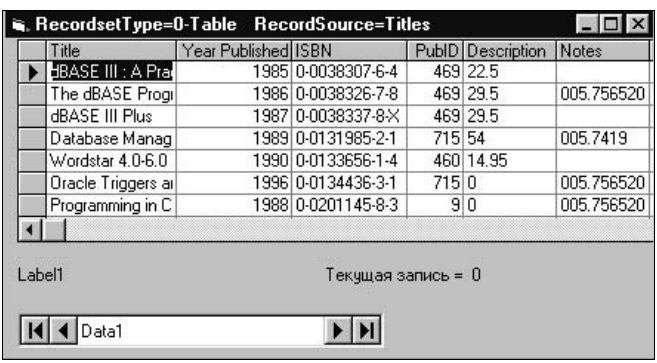


Рис 10.17. Просмотр таблицы с наименованиями книг

Приложение 10.10 (VB)

Может быть, нас в этой таблице интересуют только три первые поля, а остальные просто загромождают экран, и их хотелось бы убрать. Нет проблем — достаточно указать, какие поля нас интересуют:

```
Data1.RecordSource = SELECT Title,[Year Published],ISBN FROM Titles
```

Эта строка достаточно длинна, и набирать ее в узком окне **Properties** (Свойства) не ахти как удобно. Но она представляет собой один из наиболее мощных операторов языка SQL. После служебного слова `SELECT` (Отобрать) указан список имен полей, которые попадут в таблицу `DBGrid1`. Обратите внимание на то, что в MS Access имена полей могут содержать пробелы, но тогда их приходится заключать в квадратные скобки (большие или малые буквы использованы в названиях полей — роли не играет). После служебного слова `FROM` (Из) указано имя таблицы, в которой производится отбор.

Результат отбора представлен на рис. 10.18.

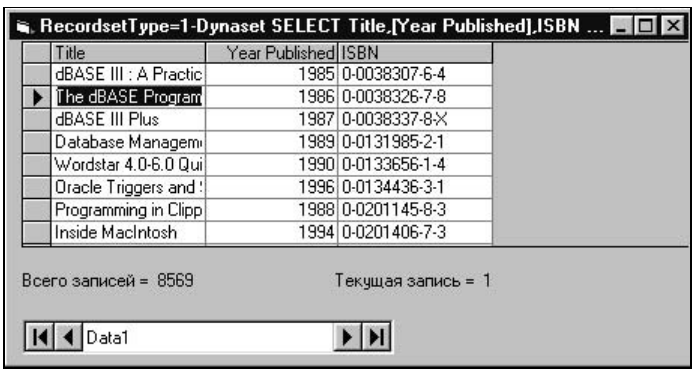


Рис. 10.18. Выборка трех полей из таблицы `Titles`

Приложение 10.11 (VB)

А теперь построим приложение, которое будет выводить на экран содержимое этих же трех полей для текущей записи. Очевидно, что нас вполне устроят три поля типа `TextBox`, снабженные пояснительными подписями. Для названия книги, которое может оказаться достаточно длинным, имеет смысл сделать соответствующее окно многостроковым (`Text1.MultiLine=True`).

Результат просмотра 11-й записи приведен на рис. 10.19.

В режимах `Dynaset` и `Snapshot` у набора данных доступно свойство `AbsolutePosition`, которое предназначено для хранения абсолютного номера текущей записи. Однако при работе с БД этим свойством пользуются редко — оно доступно не только для чтения, но и для записи. В некоторых

наших экспериментах этот счетчик вел себя не очень неустойчиво, особенно в начале таблицы.

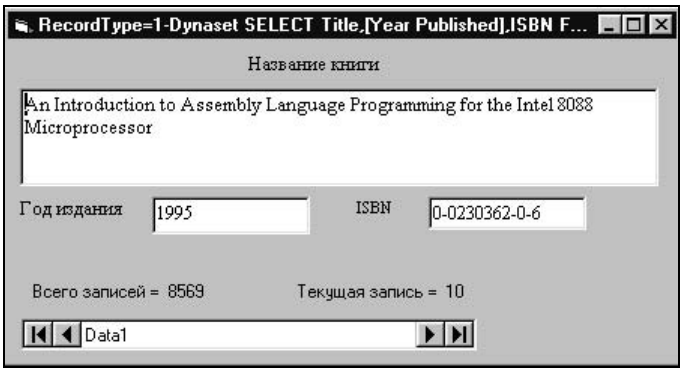


Рис. 10.19. Просмотр полей текущей записи в таблице `Titles`

Для организации последовательного перебора записей в наборе данных можно использовать следующие методы:

- `Data1.Recordset.MoveFirst` — перемещение на первую запись;
- `Data1.Recordset.MoveNext` — переход к следующей записи;
- `Data1.Recordset.MovePrevious` — возврат к предыдущей записи;
- `Data1.Recordset.Move Last` — перемещение на последнюю запись.

Цикл полного перебора строк таблицы должен учитывать, что нумерация записей начинается с `0`, а максимальный номер записи равен счетчику `RecordCount-1`:

```
For j=0 To Data1.Recordset.RecordCount-1
```

Доступ к содержимому любого поля текущей записи особых проблем не представляет:

```
Data1.Recordset("Title") = "Практика программирования"
Debug.Print Data1.Recordset("Author")
```

Приложение 10.12 (VB)

В динамических наборах данных и в моментальных снимках для организации поиска записей с нужным содержанием того или иного поля могут пригодиться методы `FindFirst` (Найти первую), `FindNext` (Найти следующую), `FindPrevious` (Найти предыдущую) и `FindLast` (Найти последнюю). Результат очередного поиска фиксируется в логическом свойстве `Data1.Recordset.NoMatch`, которое принимает значение `True`, если искомая запись не встретилась. В приведенном ниже примере демонстрируется поиск записей в таблице `Authors`, для которых в поле `Author` встречается

фамилия Кнут, известная нашим читателям по трехтомнику "Искусство программирования":

```
avt = " Author Like 'Knut*' "            ' Поисковый образ
Data1.Recordset.MoveFirst                ' Возврат в начало
Data1.Recordset.FindFirst avt            ' Поиск первой записи
Debug.Print Data1.Recordset("Author")    ' Вывод найденной записи
Do Until Data1.Recordset.NoMatch         ' Цикл дальнейшего поиска
  Data1.Recordset.FindNext avt
  Debug.Print Data1.Recordset("Author")
Loop
```

Поисковый образ, задающий параметры поиска в методах `Find`, представляет собой строку, в которой первый аргумент задает имя поля. Второй аргумент — служебное слово `Like` (Похожий) устанавливает режим поиска значения указанного поля, при котором первые 4 символа заданы жестко, а остальные могут быть любыми. Результат работы подобного приложения приведен на рис. 10.20.

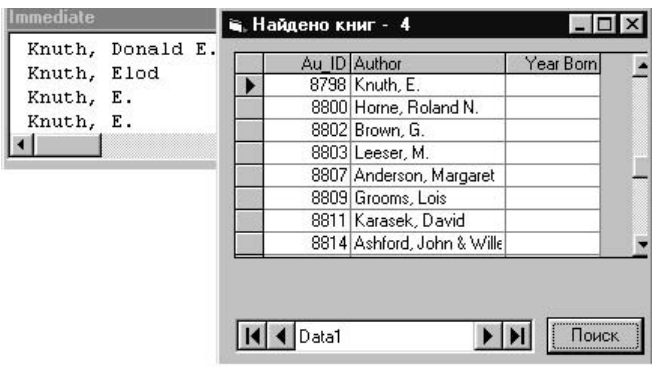


Рис. 10.20. Пример поиска автора по начальным буквам

Текущую запись можно удалить, отредактировать или вставить передней новую запись. Для этой цели к объекту `Recordset` необходимо применить метод `Delete` или `AddNew`. Изменения значений в полях текущей записи попадают в набор данных после перемещения на следующую запись или в случае вызова метода `UpdateRecord`.

Если на форме, представленной на рис. 10.19, добавить 2 кнопки с надписями **Добавить** и **Удалить**, то обработчики соответствующих щелчков по этим кнопкам могли бы выглядеть так:

```
Private Sub Command1_Click()
  ' Выполнение команды "Добавить"
  Data1.Recordset.AddNew
```

```
  Data1.Recordset.UpdateRecord
End Sub

Private Sub Command2_Click()
  ' Выполнение команды "Удалить"
  If MsgBox("Удалить ?", vbQuestion+vbYesNo) = vbYes Then
    Data1.Recordset.Delete
    MoveNext
    If Data1.Recordset.EOF Then MoveLast
  End If
End Sub
```

Перед удалением текущей записи у пользователя запрашивается подтверждение. После удаления указатель в наборе данных перемещается на следующую запись, но таковой может не оказаться, если удаляемая запись была последней. Поэтому по значению логической функции `EOF` вместо перехода на несуществующую запись указатель устанавливается на последнюю строку таблицы.

Операции над отдельными полями текущей записи и организация соответствующих циклов существенно проигрывают возможностям операторов языка SQL. В самом общем случае оператор `SELECT` позволяет отобрать несколько полей из разных таблиц:

```
SELECT список_полей FROM список_таблиц
```

Когда нужно выбрать все поля, вместо списка полей используется символ '*'. Если нас интересуют не все записи, а только те, содержимое которых удовлетворяет определенным условиям, то в операторе выбора можно применять разнообразные конструкции:

```
SELECT Title,[Year Published] FROM Titles WHERE [Year Published]=19
```

После служебного слова `WHERE` (Где) расположен критерий отбора, в данном случае отбирающий записи с указанным значением года издания. Кроме знака равенства в условии проверки значения поля могут фигурировать все операции отношения –>, >=, <, <=, =, <>. В некоторых случаях удобнее воспользоваться проверками на принадлежность значения поля некоторому интервалу:

```
SELECT .... FROM Titles WHERE [Year Published] IN (1900,1910)
SELECT .... FROM Titles WHERE [Year Published] BETWEEN 1900 AND 1910
```

В обоих случаях записи со значениями полей, совпадающими с границами интервала, включаются в состав отбираемого набора данных.

"Мягкие" условия отбора используют служебное слово `LIKE`:

```
SELECT .... FROM Titles WHERE Title LIKE "Oracle*"
SELECT .... FROM Authors WHERE Author LIKE [K-M]
```

В первом случае в отбираемый набор попадут все книги, названия которых начинаются со слова `Oracle`. Во втором случае будут отобраны все авторы, фамилии которых начинаются с букв от `K` до `M`.

Более сложные условия отбора могут компоноваться из простых условий, связанных логическими операциями:

```
SELECT Title FROM Titles WHERE Title LIKE "*Visual Basic*" AND [Year
Published]=1994
```

Процедура отбора может быть совмещена с упорядочением записей по содержимому какого-либо поля:

```
SELECT Title FROM Titles WHERE Title LIKE "*dBASE III*" ORDER BY Title
```

Обычно числовые поля упорядочиваются по возрастанию, а текстовые поля — в алфавитном (лексикографическом) порядке. Однако порядок сортировки можно сменить на противоположный, добавив служебное слово `DESC`:

```
SELECT .... WHERE ... ORDER BY [Year Published] DESC
```

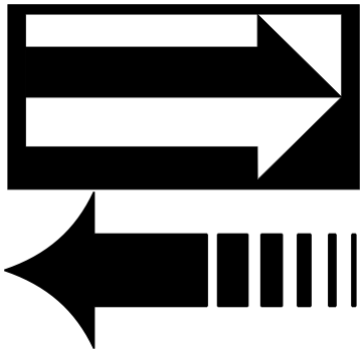
Оператор `SELECT` позволяет работать одновременно с несколькими таблицами:

```
SELECT Titles.Title, Publishers.[Company Name] FROM Titles,Publishers
WHERE Titles.PubID=Publishers.PubID
```

Несмотря на то, что процедура отбора — одна из наиболее распространенных операций по обработке информации, язык SQL не исчерпывается оператором `SELECT`. Он располагает операторами создания (`CREATE TABLE`) и уничтожения (`DROP TABLE`) таблиц, создания (`CREATE INDEX`) и уничтожения (`DROP INDEX`) индексов, манипуляции над данными (`UPDATE`, `INSERT`, `DELETE`) и многими другими. Однако изучение этих средств не входит в планы нашей книги.

Глава 11

Отладка программ и обработка исключений



Программ без ошибок не бывает. Об этом свидетельствует первая шуточная заповедь программиста, обнародованная на одной из Всесоюзных конференций профессором МГУ М. Р. Шура-Бура: "В любой отлаженной программе есть, по крайней мере, одна ошибка". Его вторая заповедь звучит не менее остроумно: "А если в программе нет ошибок, то ошибочен алгоритм". Изрядная доля истины в этих заповедях есть. Многие из вас, наверняка, сталкивались с "глюками" операционных систем и систем программирования. Недаром после широкого тестирования альфа- и бета-версий, вслед за выходом того или иного программного продукта на коммерческий рынок появляются многочисленные заплаты — patch'и. Программисты — такие же люди, и ничто человеческое им не чуждо. В том числе и свойство ошибаться. А в их профессионализме больше ценится не умение писать безошибочные программы, а умение быстро обнаруживать ошибки как в своих, так и в чужих программах. Кроме того, чрезвычайно важно вовремя "подстелить соломку" — предусмотреть обработку возможных аварийных ситуаций.

В этой главе мы познакомимся с основными приемами отладки программ. В англоязычной литературе этот процесс связан с термином *debug*, который дословно означает "устранение вредных насекомых". Происхождение этого термина связано с именем одной из наиболее известных женщин в компьютерном мире — Грейс Хоппер, дослужившейся в свое время в ВМФ США до ранга адмирала. При поиске неисправностей одной из первых ЭВМ во время ее дежурства была обнаружена причина отказа электронного блока, вызванная коротким замыканием из-за попавшей в схему мошки. С этой записи в журнале устранения технических неисправностей термин перекочевал и на поиск bug'ов в программах.

Основной набор сервисных отладочных средств, которые предлагает современная визуальная среда, базируется на возможности прервать автоматическое выполнение программы в любой точке, посмотреть значения всех доступных переменных и продолжить работу приложения до следующей точки

останова. Во время останова пользователю предоставляется право изменить неверно вычисленное значение той или иной переменной на предположительно правильное. На особо сложных участках программы с точки зрения логики выполнения ее операторов пользователь может "простучать" их в пошаговом режиме, когда за одно нажатие соответствующей клавиши компьютер выполняет одну строку исходной программы.

11.1. Отладочные средства Visual Basic

Основные средства отладки, предусмотренные в главном меню системы VB 6.0, сосредоточены в разделах **Run** (Выполнение) и **Debug** (Отладка) (рис. 11.1).

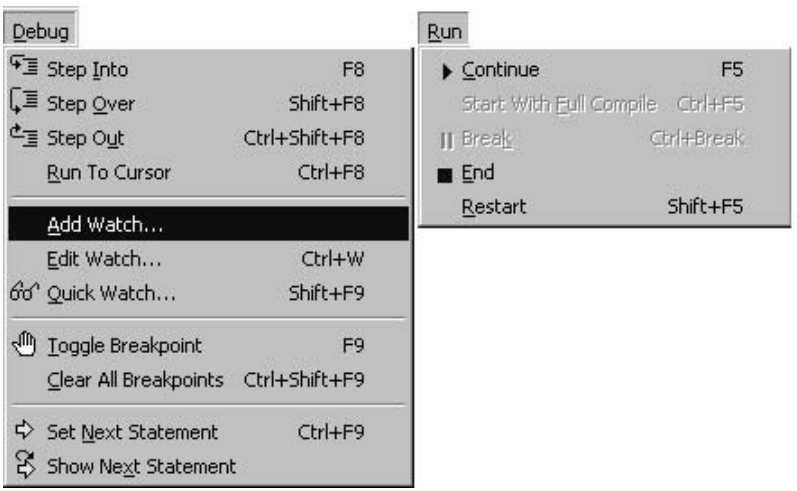


Рис. 11.1. Отладочные инструменты главного меню VB 6.0

Любой исполняемый оператор исходной программы может быть выделен и помечен в качестве *точки останова* (Breakpoint). Для этого достаточно перевести курсор в нужную строку исходной программы и нажать клавишу <F9> или щелкнуть кнопкой мыши в левой серой колонке. Выделенная таким образом строка изменит свою окраску и слева от нее появится цветное пятно. Для отмены выделения нужно еще раз повторить указанные действия, после чего выделенная строка восстановит свой нормальный цвет. Во время работы приложения при попадании на выделенную строку происходит останов программы, перед выполнением соответствующего оператора. В программе может быть помечено несколько строк, и такое выделение можно делать как перед запуском программы, так и во время ее останова. Отмена всех точек останова осуществляется командой **Clear All Breakpoints** (Отменить все точки останова) или соответствующей ей комбинацией управляющих клавиш <Ctrl>+<Shift>+<F9>.

Особой точкой останова является строка, в которой находится курсор перед запуском программы. Если вместо обычного старта приложения выполнить команду **Run To Cursor** (Выполнить программу до курсора) (или <Shift>+<F8>), то указанная строка считается точкой останова, хотя ее цвет не изменяется.

Существуют и другие точки останова, которые устанавливаются с помощью специально задаваемых условий, — например, при изменении значения указанной переменной или в момент выполнения некоторого условия. Такие условия формируются в окнах **Add Watch** (Добавить контролируемое выражение) и **Edit Watch** (Редактировать контролируемое выражение), где задаются контролируемые выражения и условия останова. Конкретные номера таких строк заранее неизвестны, т. к. соответствующие события могут произойти в разных местах программы во время работы приложения.

Приложение 11.1 (VB)

В момент останова программы вы можете посмотреть значения любых доступных в этот момент переменных. Для этой цели можно выбрать одну из пяти возможностей, предоставляемых интегрированной средой VB. Самый простой способ заключается в подведении курсора мыши к имени переменной. Примерно через секунду появится окно подсказки с именем и значением выбранной переменной. Второй вариант заключается в использовании команды быстрого просмотра **Quick Watch** (Быстрый контроль выражения). Если формула с интересующим вас выражением выделена или набрана в окне этой команды, то там же будет показано и текущее значение (рис. 11.2).

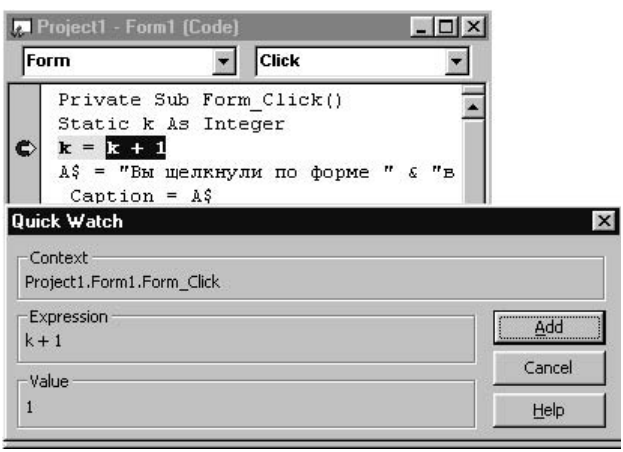


Рис. 11.2. Просмотр значения выделенного выражения

Третья возможность заключается в предварительном занесении имени контролируемой переменной или формулы с контролируемым выражением в окно **Watch** (Контролируемые выражения). Эта операция выполняется с помощью команд **Add Watch** (Добавить контролируемое выражение) или **Edit Watch** (Редактировать контролируемое выражение). В обоих случаях вызываются одинаковые диалоговые окна, отличающиеся только заголовком (рис. 11.3).

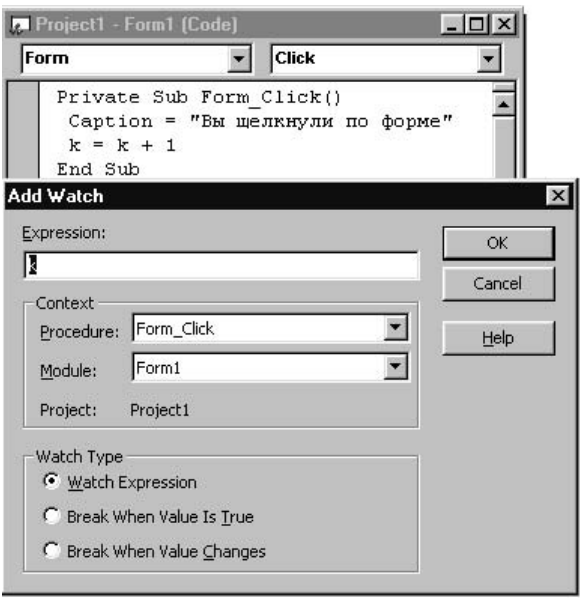


Рис. 11.3. Добавка контролируемых выражений в VB

Кроме собственно контролируемого выражения, набираемого (или переносимого автоматически, если оно выделено в программе) в поле **Expression** (Выражение), в рамке **Watch Type** (Тип просмотра) можно включить одну из двух нижних радиокнопок (первая включена по умолчанию). Кнопка с надписью **Break When Value Is True** (Останов, если значение контролируемого выражения истинно) позволяет набирать в поле **Expression** (Выражение) логические условия, в момент выполнения которых происходит останов программы. Кнопка с надписью **Break When Value Changes** (Останов при изменении значения) позволяет остановить работу программы, когда происходит изменение значения контролируемого выражения (в частности, когда изменяется значение переменной).

Все контролируемые выражения, набранные перед запуском программы, хранятся в окне **Watches** (Контролируемые выражения) (рис. 11.4) и могут быть пополнены или отредактированы во время останова программы. В частности, значение доступной переменной можно подкорректировать и продолжить выполнение программы с измененными данными.

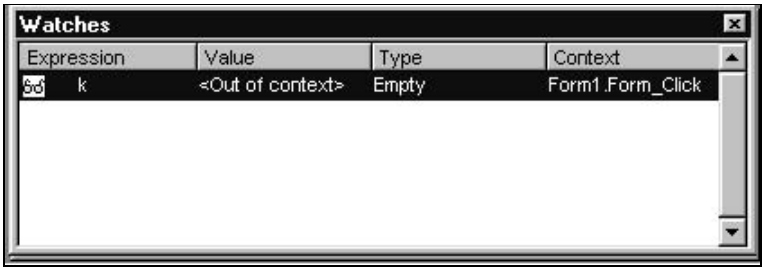


Рис. 11.4. Окно контролируемых выражений в VB 6.0

Четвертая возможность связана с использованием окна немедленных вычислений **Immediate** (Немедленные вычисления) (рис. 11.5), которое доступно как из программы при выполнении оператора `Debug.Print`, так и во время останова программы. В этом окне можно набирать почти все выполняемые операторы Бейсика, в теле которых могут находиться доступные в данный момент переменные. Сам оператор при этом не может состоять из нескольких строк подобно конструкциям `If...End If` или `Select`. Однако из этого окна можно вызвать любую процедуру или функцию нашего приложения, передав ей нужные параметры. Строка, набираемая в окне **Immediate** (Немедленные вычисления), немедленно выполняется, и результат ее работы отображается в этом же окне. По сравнению с тремя предыдущими вариантами просмотра содержимое окна **Immediate** (Немедленные вычисления) сохраняется — к предыдущим отладочным действиям и их результатам вы можете вернуться в любой момент. В этом же окне можно присвоить тем или иным переменным нужные значения и продолжить выполнение программы с модифицированными данными.

Рис. 11.5. Окно немедленных вычислений

Последний вариант просмотра значений локальных переменных с возможностью модификации этих значений предоставляет окно **Locals** (Локальные переменные) (рис. 11.6). Это окно отличается от окна **Watches** (Контролируемые выражения) тем, что список отображаемых переменных заготовлен системой, и изменить его состав нельзя. Выражений и логических условий здесь также нет.

Меню **Debug** (Отладка) включает группу команд, обеспечивающих ручное (пошаговое) и полуавтоматическое выполнение программы. Нажатие

клавиши <F8> приводит к выполнению действий, предусмотренных в текущей строке программы. Каждое следующее выполнение команды **Step Into** (Шаг внутрь) (ее эквивалентом является нажатие клавиши <F8>) проталкивает очередную строку. По команде <F8> в таком же пошаговом режиме выполняются и строки вызываемых функций и подпрограмм нашего приложения. Если вызываемая процедура отлажена, и нужда в "простукивании" ее строк отпадает, можно воспользоваться командой **Step Over** (Шаг поверх) (клавишная комбинация <Shift>+<F8>). В этом случае строки вызванной подпрограммы или функции выполняются в автоматическом режиме, но при возврате из них пошаговый режим сохраняется. Командой **Step Out** (Шаг наружу) (клавишная комбинация <Ctrl>+<Shift>+<F8>) можно воспользоваться, находясь в подпрограмме или функции. Она приводит к автоматическому выполнению оставшихся операторов, но при выходе из процедуры восстанавливает пошаговый режим.

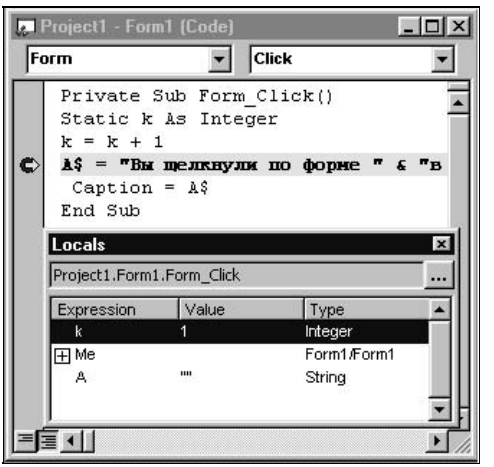


Рис. 11.6. Окно просмотра локальных переменных

В любой точке останова можно перейти к автоматическому выполнению оставшейся части программы. Для этого используется команда **Continue** (Продолжить) из раздела **Run** (Выполнить) (клавишный эквивалент — <F5>). Более того, воспользовавшись командой **Set Next Statement** (Задать следующее утверждение) (<Ctrl>+<F9>), вы можете продолжить работу программы не с точки останова, а с любой другой строки. Иногда это бывает нужно для проверки работы другой ветви программы.

Следует упомянуть еще об одном отладочном окне под названием **Call Stack** (Стек вызовов). Динамическая последовательность вызываемых функций и подпрограмм запоминается в стеке возвратов. В этот стек заносится адрес программы, по которому нужно вернуться после выполнения вызванной процедуры. Но, в свою очередь, вызванная функция до завершения своей

работы может обратиться к другой функции или подпрограмме. И такая цепочка последовательных обращений может быть достаточно длинной. Окно **Call Stack** (Стек вызовов) хранит не адреса, а имена подпрограмм и функций, еще не завершивших свою работу. По содержимому этого окна в случае аварийной ситуации можно зафиксировать цепочку последних обращений. Встречается не так уж и много случаев, когда содержимое окна **Call Stack** (Стек вызовов) оказывает существенную помощь в анализе краха программы. Может быть к нему следует обратиться при отладке рекурсивных программ.

Упомянутые выше отладочные окна не всегда видны на экране. Если интересующее вас окно отсутствует, то можно зайти в раздел главного меню **View** (Просмотр) и выполнить одну из команд — **Immediate Window** (Окно немедленных вычислений) (<Ctrl>+<G>), **Locals Window** (Окно локальных переменных), **Watch Window** (Окно контролируемых выражений) или **Call Stack** (Стек вызовов) (<Ctrl>+<L>).

11.2. Отладочные средства систем BCB и Delphi

Набор отладочных инструментов в системах BCB и Delphi сосредоточен в разделах главного меню **View** (Просмотр) и **Run** (Выполнить). В каждой из визуальных сред состав команд в этих меню немного отличается (рис. 11.7, 11.8), но основу по-прежнему составляют окна просмотра контролируемых переменных и выражений (**Watches**), точки останова (**Breakpoints**) и возможность пошагового исполнения различных участков программы. В полном объеме отладочные средства действуют только в том случае, когда программа транслируется в отладочном режиме. Для его установки следует зайти в раздел главного меню **Project** (Проект) и по команде **Options** (Опции) открыть многофункциональное окно установок различных режимов (рис. 11.9). Выберите вкладку **Compiler** (Компилятор) и нажмите на клавишу **Full Debug** (Полная отладка). Вообще говоря, иногда приходится отказываться от автоматической оптимизации программы и от распределения регистровых переменных, т. к. эти процедуры могут внести некоторую сумятицу в последовательность выполнения строк исходной программы и в просмотр наблюдаемых выражений. От такого рода улучшений программы на стадии ее отладки следует отказаться.

Точку останова на строке, в которой находится курсор, можно включить или отключить нажатием клавиши <F9>. Однако более широкие возможности предоставляет окно **Add Source Breakpoint** (Добавить точку останова) (рис. 11.10), в котором можно не только указать нужную строку, но и набрать условие, при выполнении которого останов действительно должен произойти. Этим условием может быть как истинность логического выражения, набираемого в окне **Condition** (Условие), так и номер прохода через указанную строку (**Pass Count**).

Последнее особенно полезно в тех случаях, когда выделенная строка находится в цикле. А наиболее характерной ошибкой в повторяющихся фрагментах программы является нарушение условия выхода из цикла.

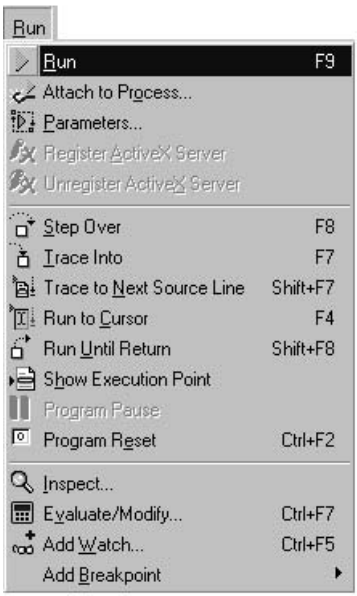


Рис. 11.7. Разделы **Run** в BCB и Delphi

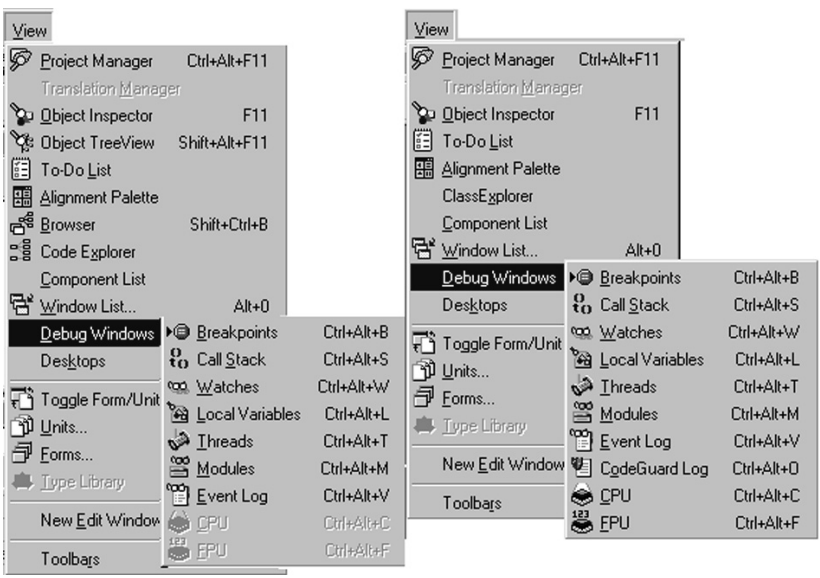


Рис. 11.8. Разделы **View** главного меню (слева — Delphi 6.0, справа — BCB 5.0)

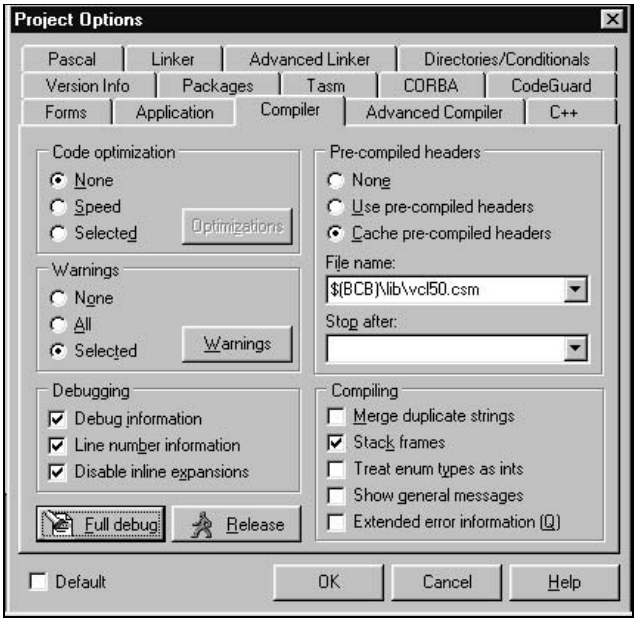


Рис. 11.9. Подключение отладочных операций в BCB

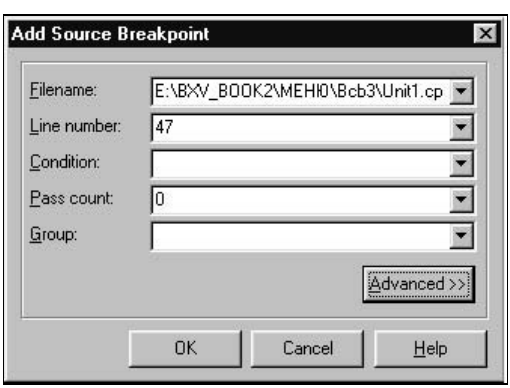


Рис. 11.10. Добавление точки останова (BCB)

Не меньшей гибкостью отличается и окно **Watch Properties** (Свойства контролируемых выражений), в котором задается имя контролируемой переменной или формула наблюдаемого значения (рис. 11.11).

Собственно наблюдаемое выражение или имя переменной набирается или переносится из текста программы в поле **Expression** (Выражение). Оно оформлено в виде выпадающего списка, в котором запоминаются все выражения, набиравшиеся в течение данного сеанса. Если вам нужно вернуться

к одному из предыдущих выражений, то список позволит быстро подключить прежнюю формулу.

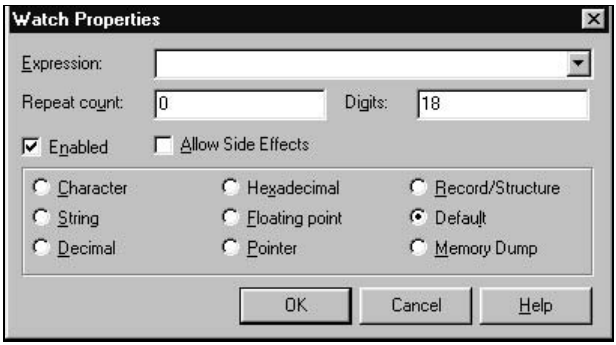


Рис. 11.11. Окно свойств наблюдаемого выражения

Если в поле **Expression** (Выражение) набрано имя массива, то в окно наблюдения попадают все элементы массива. Однако в качестве наблюдаемого выражения может быть задано имя конкретного элемента массива. Тогда содержимое поля **Repeat Count** (Счетчик повторений) определяет количество последующих элементов массива, включаемых в окно просмотра.

Группа радиокнопок позволяет выбрать нужный формат представления контролируемого значения. Для вещественных чисел в поле **Digits** (Цифры) можно ограничить количество выводимых значащих цифр. Если вам хочется наблюдать за значением некоторой переменной в разных форматах, например в символьном (**String**) и шестнадцатеричном (**Hexadecimal**), то имя переменной можно внести в окно наблюдаемых выражений дважды и для каждого варианта выбрать свой формат отображения.

Отключение свойства **Enabled** (Разрешено) позволяет не исключать выражение из списка наблюдаемых значений, но отменить его вычисление и отображение соответствующего значения в окне **Watches** (Контролируемые выражения).

Свойство **Allow Side Effects** (Разрешить побочный эффект), которое по умолчанию отключено, связано с так называемым побочным эффектом функций. Некоторые функции помимо своего прямого назначения могут изменить значения некоторых аргументов. Поэтому контролируемое выражение может изменить свое значение в случае вызова "нехорошей" функции. Если вы используете функции с побочным эффектом и при их вызове меняются переменные, входящие в наблюдаемое выражение, то свойство **Allow Side Effects** (Разрешить побочный эффект) полезно включить. Отключение этого свойства в нормальной ситуации позволяет сэкономить количество тактов, затрачиваемых отладчиком.

Значения переменных, не включенных в состав окна **Watches** (Контролируемые выражения), в момент останова можно просмотреть, наводя курсор

мыши на имя переменной или обратившись к окну **Inspect** (Инспекция) (рис. 11.12), которое можно вызвать из меню **Run** (Выполнить).

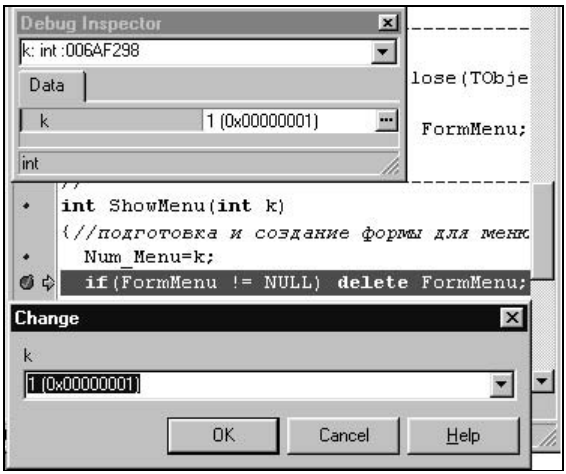


Рис. 11.12. Окно Инспектора отладки в BCB

Щелчок по кнопке с тремя точками вызывает окно **Change** (Изменить), в котором можно модифицировать значение просматриваемой переменной.

С момента запуска программы или после останова на первой предусмотренной точке можно перейти на пошаговое выполнение программы с заходом в вызываемые функции (команда **Trace Into** — <F7>) или с автоматическим выполнением вызываемых процедур (команда **Step Over** (Шаг поверх) — <F8>). В некоторых ситуациях обе эти команды оказываются бессильны, и тогда может выручить команда **Trace to Next Source Line** (Трассировать до следующей строки) (клавишная комбинация <Shift>+<F7>). Одна из таких ситуаций возникает, когда оператор, исполняемый в пошаговом режиме, приводит к генерации какого-либо события. По логике вещей в этот момент управление должно быть передано обработчику соответствующего события. Например, программа изменяет значение свойства `Position` у объекта `UpDown`, что вызывает событие `OnChange`. Если у нас предусмотрен обработчик этого события, то для корректного перехода в процедуру `UpDown1Change` оператор, изменяющий значение `Position`, следует выполнить по команде **Trace to Next Source Line** (Трассировать до следующей строки).

Если мы в пошаговом режиме попали в тело вызванной функции, то автоматическое исполнение ее операторов до возврата из функции обеспечивается командой **Run Until Return** (Выполнять до момента выхода) (<Shift>+<F8>).

Команда **Evaluate/Modify** (Вычислить/Модифицировать), исторически присутствующая в средствах отладки почти уже 20 лет, предоставляет пользователю

своеобразный калькулятор, умеющий использовать объекты нашей программы. С его помощью можно вычислить значение выражения, операндами которого являются доступные в момент останова переменные нашей программы (рис. 11.13).

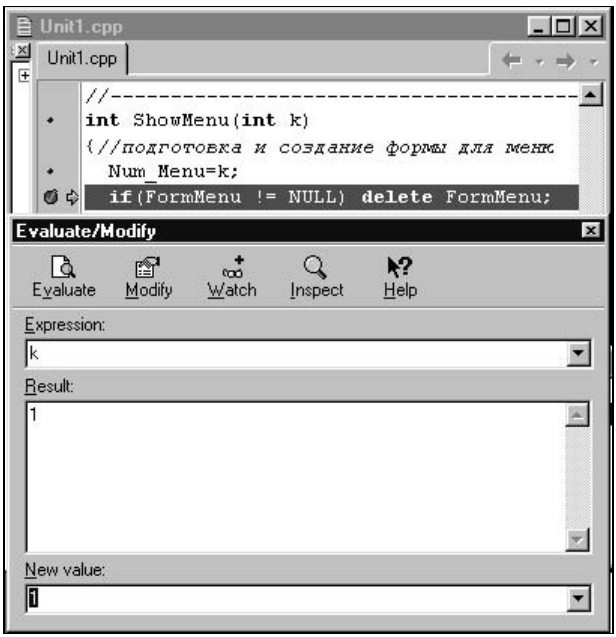


Рис. 11.13. Окно **Evaluate/Modify**

В поле **New value** (Новое значение) можно набрать новое значение переменной, с которым программа продолжит свою работу. Команды в окне **Evaluate/Modify** (Вычислить/Модифицировать) вынесены на панель, расположенную под заголовком окна. С их помощью можно вычислить значение набранного выражения (**Evaluate**), изменить значение переменной (**Modify**), включить выражение из поля **Expression** (Выражение) в состав наблюдаемых выражений (**+ Watch**), вызвать окно Инспектора отладки (**Inspect**) или получить справку (**Help**) по использованию окна **Evaluate/Modify** (Вычислить/Модифицировать).

После выполнения необходимых действий в той или иной точке останова можно удалить старые точки останова, добавить новые, перейти в пошаговый (клавиши <F7>, <F8>, <Shift>+<F7>, <Shift>+<F8>) или автоматический режим (клавиша <F9>) продолжения работы программы, восстановить программу для последующего перезапуска командой **Program Reset** (Перезапуск программы) (комбинация <Ctrl>+<F2>). В точке останова можно даже изменить текст исходной программы и запомнить ее на диске, но все изменения вступят в силу только после перетрансляции.

11.3. Исключительные ситуации и борьба с ними

На пультах ЭВМ первого поколения присутствовала сигнальная лампочка, фиксировавшая чрезвычайное событие в программе, которое делало невозможным продолжение работы. Подпись под лампочкой — АвОст — была аббревиатурой ситуации *аварийный останов*. А вызывалась эта ситуация рядом причин. В некоторых из них была виновата программа, решившая разделить что-то на 0 или испортившая содержимое нужных ячеек памяти, в других — свою лепту вносила аппаратура, работавшая в те времена не очень устойчиво, в-третьих — причиной был пользователь, который ввел недопустимое число, а соответствующий контроль в программе не был предусмотрен. Несмотря на полувековой путь, пройденный компьютерными технологиями, аварийные ситуации (сейчас их называют более деликатно — исключения) в программах возникают и сегодня. Правда, удельный вес причин, вызывающих исключительные события, в основном объясняется человеческим фактором — ошибками программиста или пользователя, эксплуатирующего программу. Деления на 0, выход данных за пределы допустимого диапазона, нарушение правил доступа к своим и чужим данным и многое другое приводит к возникновению ошибочных ситуаций. Любая система программирования контролирует какое-то количество исключений, сообщая о происшедшем событии и прекращая работу программы. Часть из таких событий подлежит обязательному контролю, некоторые включаются пользователем по заранее предусмотренным ситуациям (например, выход индексов за пределы установленного диапазона).

Однако при создании программ надо предусматривать возможность исправить ошибочную ситуацию и, например, обеспечить продолжение работы после исправления оплошности пользователя. Ненормальное завершение программы может привести к нецелевому расходованию ресурсов памяти и, в конце концов, вызвать "зависание" операционной системы. Поэтому на исключения полезно правильно и грамотно реагировать, и не всегда рассчитывать только на ограниченную реакцию системы.

11.3.1. Реакция на ошибки в системе Visual Basic

Основным средством программной реакции на ошибочные ситуации является использование оператора `On Error`, появившегося в Бейсике с незапамятных времен. Но для того чтобы этот оператор не был проигнорирован системой, вы должны произвести предварительную установку нужного режима работы программы в окне **Options** (Опции) (вызывается из меню **Tools (**Инструменты) на вкладке **General** (Основной) (рис. 11.14). В рамке **Error Trapping** (Отлавливание ошибок) необходимо выбрать нижнюю

радиокнопку с надписью **Break on Unhandled Errors** (Прерывание в случае Необрабатываемых Ошибок).

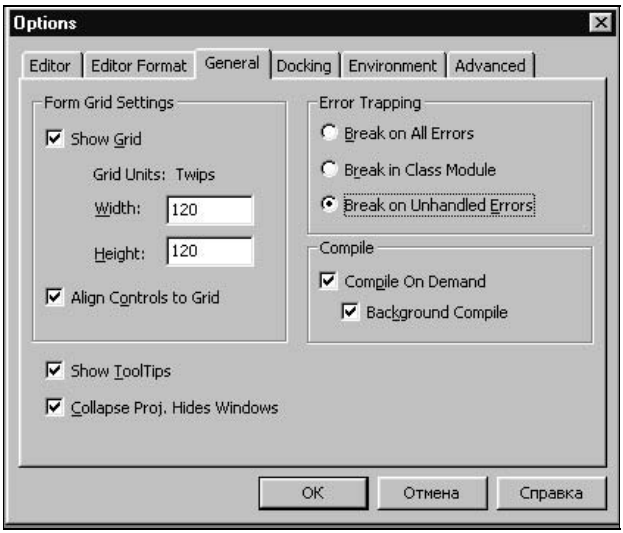


Рис. 11.14. Установка нестандартного режима обработки ошибок

В самом начале процедуры следует поставить оператор `On Error`:

```
On Error Goto метка
```

Приложение 11.2 (VB)

Если во время работы этой процедуры возникнет аварийная ситуация, то управление будет передано указанной метке. Группа операторов, следующая за указанной меткой, предназначается для разбирательства с возникшим исключением. Она должна находиться в теле этой же процедуры. Схематически такая процедура может выглядеть следующим образом:

```
Sub QQ()
  On Error Goto qqerr
' Операторы, при выполнении которых может произойти ЧП

  Exit Sub
qqerr:
' Операторы анализа и обработки ошибки
End sub
```

Очевидно, что фрагмент обработки ошибок должен начинаться с выяснения, что произошло, т. к. исключительная ситуация может быть порождена не обязательно единственной причиной. В этом нам поможет разобраться

системный объект `Err`, в котором хранится вся информация о программном происшествии. Наиболее полезное свойство этого объекта — `Number`, содержащее числовой код ошибки. Таким образом, анализ причин должен начинаться с выяснения значения `Err.Number`. Полный перечень номеров ошибок и соответствующие пояснения по поводу каждой из них вы можете найти в разделе **Trappable Errors** (Отлавливаемые ошибки). Некоторые из них мы посчитали наиболее полезными и привели в табл. 11.1.

Таблица 11.1. *Описание раздела* ***Trappable Errors***

Код	Причина	Код	Причина
6	Переполнение	58	Файл уже существует
7	Выход за пределы памяти	59	Ошибка в длине записи
9	Индекс вне диапазона	63	Ошибка в номере записи
11	Деление на 0	71	Дискета не готова
14	Исчерпание памяти для строк	481	Ошибка в формате рисунка
28	Исчерпание памяти для стека	482	Ошибка принтера
52	Ошибка в имени или номере файла	744	Фрагмент текста не найден
53	Файл не найден	31036	Ошибка при сохранении файла
55	Файл уже открыт	31037	Ошибка при загрузке из файла

Кроме номера ошибки могут оказаться полезными свойства `Err.Source` и `Err.Description`. В них содержится символьная информация, расшифровывающая код ошибки и поясняющая ее существо. Переводить этот текст на русский язык вам придется самостоятельно. Однако в своем обработчике ошибок вы можете произвести замену системных свойств `Source` и `Description` с тем, чтобы пользователь получил сообщение на родном языке. Приводимый в справочном файле пример обработки ошибки демонстрирует использование специального формата оператора `On Error`:

```
Dim Msg
  On Error Resume Next ' Игнорируем предшествующую ошибку
  Err.Clear            ' Сбрасываем код предыдущей ошибки
  Err.Raise 6          ' Генерируем ошибку с кодом 6 — "Переполнение"
  ' Проверяем код ошибки и выдаем сообщение
  If Err.Number <> 0 Then
    Msg = "Ошибка # " & Str(Err.Number) & " возникла в связи с " _
          & Err.Source & Chr(13) & Err.Description
  MsgBox Msg, , "Ошибка", Err.Helpfile, Err.HelpContext
End If
```

Оператор `On Error Resume Next` объясняет системе, что ошибку можно проигнорировать и надо выполнять следующий оператор. Метод `Clear` удаляет из объекта `Err` информацию об ошибке, которая могла возникнуть при выполнении предыдущих операторов. С помощью метода `Raise` искусственно имитируется исключение с указанным номером. Этим можно пользоваться при отладке нестандартных программ обработки ошибок.

Оператор `On Error` в сочетании со служебным словом `Resume` допускает три варианта:

```
On Error Resume
On Error Resume Next
On Error Resume метка
```

В первом случае системе предлагается повторить оператор, при выполнении которого случилось ЧП. Если ошибка произошла при вызове процедуры, то будет предпринята попытка повторного вызова этой же процедуры. Во втором случае управление передается оператору, следующему за вызвавшем ошибку. И, наконец, третий вариант передает управление на строку с указанной меткой.

Если мы хотим отключить любую обработку ошибок, включенную в текст нашей процедуры, то в ее начале можно поместить следующий оператор:

```
On Error Goto 0
```

В этом случае при возникновении ошибки управление в свои руки возьмет система и после выдачи сообщения о причине аварийной ситуации завершит работу приложения.

Часть программы, выполняющая анализ ошибочной ситуации, может закончиться просто выходом на конец процедуры (`End Sub`, `End Function`) или выполнением одной из трех разновидностей оператора `Resume` — `Resume`, `Resume Next`, `Resume`. Ниже приводится модификация примера из файла помощи:

```
Private Sub Form_Click()
  ResumeDemo
End Sub

Sub ResumeDemo()
  On Error GoTo Err1          ' Задание точки входа в обработчик ошибок
  Open "qq.txt" For Output As #1 ' Открыли файл для вывода
  Kill "qq.txt"                  ' Попытка удалить открытый файл
  Exit Sub                       ' Обход программы обработки ошибок
Err1:                         ' Точка входа в программу обработки ошибок
  Select Case Err.Number      ' Анализ номера ошибки
    Case 55                   ' запрещенная операция над открытым файлом
```

```
      Close #1                ' Закрываем открытый файл
    Case Else
      ' Обработка других возможных ошибок
  End Select
  Resume                      ' На повторное выполнение ошибочной строки
End Sub
```

В этом примере ошибочная ситуация возникает при попытке удалить открытый файл. Обработка этого исключения сводится к закрытию файла и передаче управления на оператор `Kill`, вызвавший ошибку.

11.3.2. Обработка исключений в системах Borland C++ Builder и Delphi

Функции и процедуры, в которых предусмотрены средства борьбы с аварийными ситуациями, должны включать специально оформленные блоки, начинающиеся со служебного слова `try`. Его дословный перевод — попытка, проба. Вслед за предложением попробовать располагаются операторы, при выполнении которых может возникнуть аварийная ситуация. В этом месте обработка сомнительной части программы прекращается, и управление передается ловушке ошибок, текст которой начинается со служебного слова `catch`. Схематично блок с предусмотренным обработчиком ошибок имеет следующий вид:

BCB:

```
.....
  try
    {//сомнительные операторы
.............................
    }
  catch(....)
    {//анализ и обработка ошибок
...................................
    }
```

Delphi:

```
.....
  try
  {сомнительные операторы}
.........................
.........................
..except
  {анализ и обработка ошибок}
.............................
..end;
```

Группу операторов, расположенных между служебными словами `try` и `catch` или `try` и `except`, обычно называют защищенным блоком. Если при его выполнении не возникают аварийные ситуации, то блок операторов, расположенных после служебного слова `catch` или `except`, обходится. Структура обработчика исключений по смыслу напоминает переключатель — если произошло событие 1, то выполни действие 1, если произошло событие 2, то выполни действие 2 и т. д. Однако оформление такого "переключателя" в BCB и Delphi немного отличается.

Начнем с того, что в BCB оператор ловушки может содержать конкретный аргумент, определяющий тип обрабатываемого исключения, например, ошибку деления на 0:

```
catch(EDivByZero)
```

Полный перечень таких исключений приведен в табл. 11.2.

Таблица 11.2. *Перечень событий, вызывающих исключительную ситуацию*

Событие	Причина
EAbort Stops	Цепочка событий, вызвавшая прекращение работы приложения, но не сопровождаемая выдачей системного сообщения
EAccessViolation	Ошибка доступа к памяти
EBitsError	Недозволенный доступ к логическому массиву
EComponentError	Попытка недопустимого использования регистра или переименования компонента
EConvertError	Ошибка преобразования данных
EDatabaseError	Ошибка доступа к базе данных
EDBEditError	Обнаружены данные, несовместимые с заданной маской
EDivByZero	Попытка разделить целое число на 0
EExternalException	Неопознанная внешняя ошибка
EInOutError	Ошибка при выполнении операции ввода/вывода
EIntOverflow	Переполнение при операциях с целыми числами
EInvalidCast	Ошибка в приведении типов
EInvalidGraphic	Попытка работы с неопознанным графическим форматом
EInvalidOperation	Выполнение недопустимой операции над компонентом
EInvalidPointer	Ошибочная операция над указателем
EMenuError	Ошибка, связанная с командой меню
EOleCtrlError	Ошибка при взаимодействии с элементом ActiveX
EOleError	Ошибка при взаимодействии в режиме OLE automation
EPrinterError	Ошибка принтера (не готов, не включен, нет бумаги и т. п.)
EPropertyError	Неудачная попытка изменить значение свойства
ERangeError	Слишком большое целое число, не соответствующее типу данного
ERegistryException	Ошибка регистрации
EStackOverflow	Переполнение стека
EZeroDivide	Попытка разделить вещественное число на 0

Если ловушка рассчитана на обработку любых аварийных ситуаций, то аргументом оператора `catch` являются три точки — `catch(...)`.

В Delphi переключатель по обработке ошибочных ситуаций состоит из строк вида:

```
on Exception1 do {цепочка операторов}
on Exception2 do {цепочка операторов}
.....................................
```

Вслед за служебным словом `on` располагается одна из констант, перечисленных в табл. 11.2. Примеры функций, вычисляющих среднее значение по сумме `Sum` и количеству слагаемых `n`, демонстрирующие старый и новый способы программирования, приведены в справке Delphi 6.0:

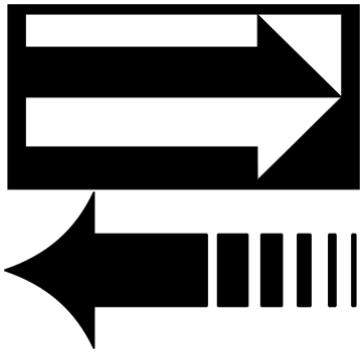
```
function Mid(Sum, n : integer):integer;
begin {старый способ}
  if n=0 then Result:=0
  else Result:=Sum div n;
end;

function Mid(Sum, n : integer):integer;
begin {новый способ}
  try
    Result:=Sum div n;
  except
     on EDivByZero Result:=0;
  end;
```

Нельзя сказать, что новая программа логически проще, чем старая. Но в ней в явном виде отсутствует проверка делителя на 0, которая в старой программе делается при каждом обращении к функции `Mid`. По машинным командам они, скорее, эквивалентны. В первом варианте проверка деления на 0 делается перед выполнением операции, а во втором варианте после выполнения деления проверяется флаг переполнения. Однако ошибочная ситуация может возникнуть в разных операторах защищенного блока, и тогда программирование старым способом потребует довольно большого количества проверок. А в новом варианте, возможно, добавится пара строк типа `on Exception_xx` на случай, если исключения могут возникнуть по разным поводам.

Еще один способ завершения защищенного блока состоит в использовании конструкции `finally`. В отличие от ловушки ошибок, на которую управление передается только в случае возникновения аварийной ситуации, блок `finally` выполняется всегда. Обычно его используют в момент завершения работы приложения для возврата каких-либо ресурсов, которые система самостоятельно освободить не может.

Глава 12

Редактор условных знаков



Графический редактор, описываемый ниже, предназначен для формирования библиотек векторных и растровых описаний условных знаков, которые приходится отображать в составе документов, выводимых на экран дисплея или на плоттеры, воспринимающие графическую программу на языке Hewlett-Packard Graphics Language (HPGL). Первая версия этого редактора была написана в среде Delphi 1.0 М. С. Казачковой и явилась результатом ее дипломной работы (1997 г.), выполненной под руководством Ю. Л. Кеткова. Последующие версии этого редактора, имевшие целью расширение функциональных возможностей и повышение эффективности алгоритмов, были реализованы Ю. Л. Кетковым в среде Borland C++ Builder в период 2000—2002 гг. Внешний вид приложения приведен на рис. 12.1.

Приложение 12.1 (BCB)

На протяжении ряда лет этот редактор активно используется как инструмент для разработки библиотек дискретных условных знаков в автоматизированных картографических системах. Однако его возможности не исчерпываются только указанной предметной областью. Существуют, например, чертежи электронных схем, в которых основная нагрузка падает на изображения пассивных и активных компонент — резисторов, конденсаторов, индуктивностей, транзисторов, микросхем и др. Различные пиктограммы и дорожные знаки широко используются на схемах автомобильных дорог, планах туристских маршрутов и т. п.

12.1. Графические команды и система координат

Прежде чем перейти к описанию алгоритмов и программ графического редактора, разберемся с функциональными возможностями на уровне пользователя этого приложения.

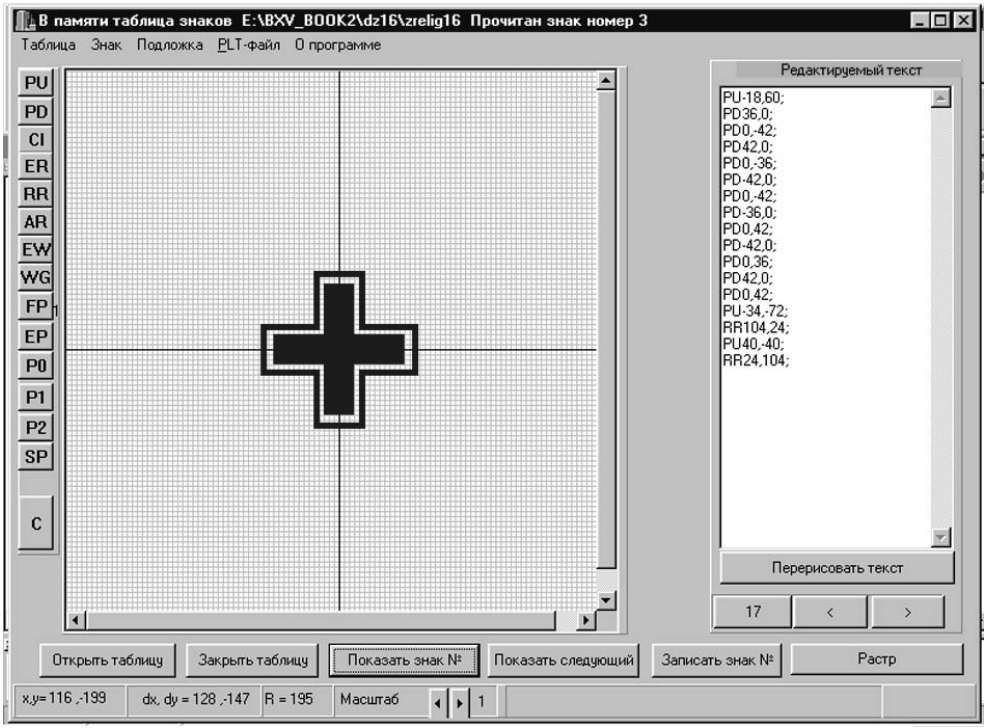


Рис. 12.1. Внешний вид редактора условных знаков

Под термином *условный дискретный знак* мы будем понимать небольшую графическую фигуру, контуры которой заданы координатами узловых точек. Через часть узловых точек могут проходить замкнутые или разомкнутые контуры. Некоторые замкнутые области условного знака могут быть закрашены. На рис. 12.2 приведены образцы условных знаков, увеличенных примерно в 25 раз. У каждого условного знака может быть одна (основная) или две (основная и дополнительная) точки привязки, определяющие положение знака на графическом документе. Условные знаки с двумя точками привязки, которые допускают поворот на любой угол, принято называть ориентируемыми.

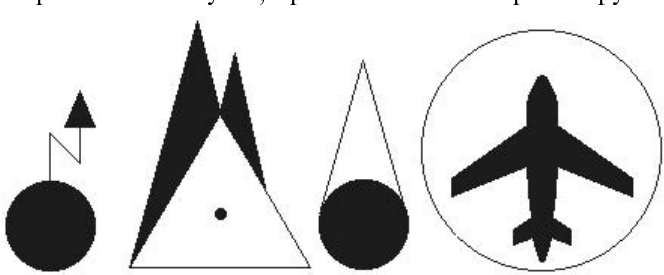


Рис. 12.2. Образцы некоторых условных знаков

На практике используют два способа описания графических образов условных знаков — *векторный* и *растровый*. В первом случае описание знака задается графической программой, управляющей движением пишущего узла (пера плоттера или светового луча дисплея). Во втором случае графический образ знака представлен двоичным массивом, содержащим коды цветности пикселов габаритного прямоугольника, — минимального прямоугольника, окаймляющего знак. Векторное описание удобно тем, что оно легко поддается таким процедурам, как масштабирование, повороты на заданный угол, анизотропное изменение пропорций (т. е. по каждой координате может действовать свой коэффициент сжатия/растяжения). Растровые изображения легко и быстро копируются в видеопамять, однако они требуют больше памяти и плохо поддаются аффинным преобразованиям (масштабирование, повороты и др.).

Векторное описание, принятое в редакторе условных знаков, базируется на подмножестве графических команд стандарта HPGL. В начальный момент, предшествующий построению знака, предполагается, что пишущий узел (перо) находится в начале локальной системы координат знака, одновременно являющейся основной точкой привязки знака. Все перемещения пера, рисующего очертания знака, задаются в относительных координатах — *перемещениях* (`dx,dy`) относительно так называемой текущей точки. К абсолютным значениям параметров графических команд относятся величины радиусов и углов. В качестве единицы измерения линейных параметров (относительные координаты и значения радиусов) используется шаг плоттера, который по умолчанию в стандарте HPGL равен 1/40 мм (0.025 мм). Как правило, линейные параметры являются целочисленными. Угловые величины также являются целочисленными и задаются в градусах.

Полный перечень графических команд, использованных в редакторе условных знаков, приведен в табл. 12.1.

Таблица 12.1. *Перечень графических команд редактора условных знаков*

Формат команды	Описание
`AR dx, dy, a;`	Построение дуги окружности, начинающейся из текущей точки `CP (CPx, CPy)`. Центр соответствующей окружности находится в точке с координатами `(CPx+dx, CPy+dy)`. Угол `a`, отсчитываемый от начального радиус-вектора, определяет конечную точку дуги (положительным значениям `a` соответствует движение по дуге против часовой стрелки). После построения текущая точка перемещается в конец дуги
`CI r;`	Построение окружности радиуса `r` с центром в текущей точке. Положение текущей точки сохраняется
`EP ;`	Построение контура, сформированного в режиме полигона. Текущая точка после построения перемещается в начальную точку полигона

Таблица 12.1 *(продолжение)*

Формат команды	Описание
`ER dx, dy;`	Построение контуров прямоугольника, одна из угловых вершин которого находится в текущей точке `(CPx, CPy)`, а противоположная вершина — в точке с координатами `(CPx+dx, CPy+dy)`. После построения положение текущей точки не изменяется
`EW r, a1, a2;`	Построение контуров кругового сектора. Центр соответствующей окружности с радиусом `r` находится в текущей точке, угол `a1` определяет положение начального радиус-вектора, угол `a2` — величину центрального угла. После построения положение текущей точки не изменяется
`FP ;`	Заливка области, сформированной в режиме полигона. Текущая точка после построения перемещается в начальную точку полигона
`PC n_Pen, r, g, b;`	Установка цветовых атрибутов пера с номером `n_Pen`. `r` — интенсивность красного, `g` — интенсивность зеленого, `b` — интенсивность синего (целые числа из диапазона [0, 255])
`PD dx, dy;`	Построение отрезка, начало которого совпадает с позицией текущей точки `(CPx, CPy)`, а конец находится в точке с координатами `(CPx+dx, CPy+dy)`. После построения текущая точка перемещается в конец отрезка
`PM 0;`	Установка режима полигона. Все последующие построения сопровождаются запоминанием координат точек полигона, образующих внешний контур. Координаты текущей точки запоминаются в качестве начальной точки полигона
`PM 1;`	Установка режима "полигон в полигоне". Все последующие построения сопровождаются запоминанием координат точек полигона, образующих внутренний контур соответствующей дырки. Координаты текущей точки запоминаются в качестве начальной точки внутреннего контура
`PM 2;`	Выход из режима полигона. После выполнения этой команды становятся доступными операции по обводке контуров полигона и их заливке
`PU dx, dy;`	Перемещение текущей точки в позицию с координатами `(CPx+dx, CPy+dy)`
`PW w, n_Pen;`	Установка ширины пера с номером `n_Pen`. Ширина `w` задается в шагах плоттера. Для построения симметричных толстых линий, расположенных вдоль заданной осевой, ширина пера должна быть четной (`w=0, 2, 4, ...`)
`RR dx, dy;`	Построение залитого прямоугольника, одна из вершин которого находится в текущей точке `(CPx, CPy)`, а противоположная вершина — в точке с координатами `(CPx+dx, CPy+dy)`. После построения положение текущей точки не изменяется

Таблица 12.1 *(окончание)*

Формат команды	Описание
`SP n;`	Выбор пера с заданным номером
`WG r, a1, a2;`	Построение залитого кругового сектора. Центр соответствующей окружности с радиусом `r` находится в текущей точке, угол `a1` определяет положение начального радиус-вектора, угол `a2` — величину центрального угла. После построения положение текущей точки не изменяется

Для того чтобы описать квадрат со стороной 5 мм, левый нижний угол которого является основной точкой привязки и совпадает с началом координат, можно воспользоваться следующей графической программой:

```
PD0,200; PD200,200; PD-200,0; PD0,-200;
```

Четыре команды `PD` (Pen Down) строят стороны квадрата, последовательно обводя пишущим узлом его периметр (200 шагов плоттера эквивалентны 5 мм). Если бы условный знак был представлен таким же квадратом, но его основная точка привязки находилась бы в центре квадрата, то соответствующая графическая программа могла иметь вид:

```
PU-100,-100; PD0,200; PD200,200; PD-200,0; PD0,-200;
```

В этом случае первая команда `PU` (Pen Up) перемещает перо в поднятом состоянии (или, что то же самое — двигает луч дисплея без засветки следа пера) в левый нижний угол квадрата. А теперь представим себе, что условный знак — такой же квадрат, внутренность которого залита сплошным цветом. Для описания такого знака в языке HPGL имеются две возможности. Во-первых, можно прибегнуть к команде `RR` — построение залитого прямоугольника:

```
PU-100,-100;  RR200,200;
```

Второй вариант использует так называемый режим полигона, когда строится замкнутый контур и заливается внутренняя область:

```
PM0; PU-100,-100; PD0,200; PD200,200; PD-200,0; PD0,-200; PM2; FP;
```

Первая команда `PM` (Polygon Mode) включает режим полигона, после чего координаты точек контура запоминаются в специальном буфере плоттера без выполнения соответствующих построений. Предпоследняя команда отключает режим полигона, а следующая за ней команда `FP` (Fill Polygon) реализует заливку полигона. Казалось бы, зачем городить огород, если первый вариант намного короче. Однако все дело в возможности поворота фигуры, если условный знак должен быть ориентируемым. Отрабатывая команду `RR`, плоттер всегда строит залитый прямоугольник со сторонами, параллельными координатным осям, и поворот изображения на нем не отражается. А координаты точек в командах построения других графических примитивов подвергаются всем аффинным преобразованиям (в т.ч. и поворотам).

Большинству из этих команд, описанных в табл.12.1, поставлены в соответствие кнопки на инструментальной панели редактора условных знаков. Щелчок по той или иной кнопке переводит редактор в режим ожидания ввода необходимых параметров. Для одних команд значения параметров формируются последующим перемещением курсора мыши, для других команд появляется соответствующее диалоговое окно, в котором значения параметров должны быть набраны. В некоторых случаях с помощью курсора мыши довольно трудно задать требуемые координаты. Тогда рекомендуется воспользоваться услугами клавиатуры, с помощью которой курсор мыши можно перемещать попиксельно. Для этой цели одна из четырех стрелок управления курсором нажимается при зажатой клавише <Alt>. Каждое нажатие клавиши со стрелкой перемещает курсор мыши на один пиксел в соответствующем направлении. Завершая такое точное позиционирование, необходимо не отпуская клавишу <Alt>, нажать клавишу <Enter>.

Некоторые кнопки инструментальной панели, будучи один раз нажаты, продолжают оставаться в утопленном состоянии. Это обеспечивает возможность включения в графическое описание нескольких подряд идущих команд с одинаковым кодом.

12.2. Окно с графической программой

В правой стороне расположено поле `Memo1`, на котором отображается текст графической программы, создаваемый автоматически в процессе формирования контуров нового знака, либо извлекаемый из существующего описания в таблице условных знаков, либо загружаемый из автономного файла с расширением plt.

Текст графической программы может быть подвергнут обычному редактированию — изменению числовых параметров, подключению новых команд или удалению существующих. Как и в любом текстовом редакторе, здесь можно выделять, удалять, перемещать или копировать блоки. При ручной правке текста графической программы необходимо следить за тем, чтобы код графической команды набирался большими буквами, чтобы в одной строке находилась только одна графическая команда, и чтобы она всегда завершалась точкой с запятой. Пустые строки и пробелы в графической программе игнорируются автоматически. После внесения ручных исправлений в текст графической программы следует обновить соответствующее изображение на поле знака — щелкнуть по кнопке с надписью **Перерисовать текст**. В этом случае редактор сам удалит лишние пробелы и сообщит об ошибках, обнаруженных в тексте какой-либо графической команды.

Для стирания текста графической программы и соответствующего ему изображения на поле знака используется кнопка с надписью **C**.

12.3. Включение графических команд в описание знака

Объявлению нового знака всегда предшествуют перевод текущей точки в начало локальной системы координат и очистка рабочего поля. Для удобства на рабочем поле воспроизводится координатная сетка и оси координат. Описание знака начинается с выбора кнопки с подходящей командой на панели инструментов. Для любой команды, ввод параметров которой вами установлен тем или иным способом, всегда остается возможность изменить численные значения параметров, отредактировав текст в поле `Memo1`. После этого необходимо перестроить текущее описание, нажав соответствующую кнопку.

12.3.1. Включение команды *PU*

Команда `PU` (Pen Up — Поднять перо) предназначена для перемещения текущей точки без фиксации следа пера, т. е. для построения невидимого отрезка. После щелчка по кнопке `PU` она остается утопленной и теперь необходимо переместить курсор мыши в новую позицию. Абсолютные и относительные координаты новой точки отображаются в строке состояния (x, y = ..., dx, dy = ...). По достижении нужной позиции вы должны щелкнуть по левой кнопке мыши и, тем самым, зафиксировать положение конечной точки невидимого отрезка. Сразу после этого в поле `Memo1` будет сформирован текст команды `PU` с соответствующими параметрами, а на поле знака появится красный крестик, символизирующий текущую позицию.

12.3.2. Включение команды *PD*

Команда `PD` (Pen Down — Опустить перо) предназначена для перемещения текущей точки с фиксацией следа пера, т. е. для построения видимого отрезка. После щелчка по кнопке `PD` она остается утопленной и теперь необходимо переместить курсор мыши в новую позицию. Абсолютные и относительные координаты новой точки отображаются в строке состояния (x,y = ..., dx,dy = ...). По достижении нужной позиции вы должны щелкнуть по левой кнопке мыши и, тем самым, зафиксировать положение конечной точки видимого отрезка. Сразу после этого в поле `Memo1` будет сформирован текст команды `PD` с соответствующими параметрами, а на поле знака появится соответствующий отрезок прямой. Когда кнопка `PD` утоплена, то вы можете аналогичным образом построить несколько смежных отрезков ломаной линии.

12.3.3. Включение команды *CI*

Команда `CI` (Circle — Окружность) предназначена для построения окружности с центром в текущей точке. После щелчка по кнопке `CI` она продолжает оставаться утопленной, и это позволит нарисовать несколько концентрических

окружностей без повторного щелчка по кнопке. Для построения окружности вы должны зажать левую кнопку мыши и перемещать ее курсор до тех пор, пока не будет построена окружность нужного радиуса. За точным числовым значением радиуса можно наблюдать в строке состояния (R = ...). В момент отпускания кнопки на поле `Memo1` формируется команда `CI r;`, а на поле знака фиксируется последняя окружность. Команда `CI` не изменяет положение текущей точки.

12.3.4. Включение команды *ER*

Команда `ER` (Edge Rectangle — Граница прямоугольника) предназначена для построения контуров прямоугольника, опирающегося одним из своих углов на текущую точку. Размеры прямоугольника формируются путем перетаскивания курсора мыши. После щелчка по кнопке `ER` вы должны зажать левую кнопку мыши и перемещать ее курсор до тех пор, пока не будет построен прямоугольник нужного размера. За точными числовыми значениями приращений можно наблюдать в строке состояния (dx =..., dy = ...). В момент отпускания кнопки на поле `Memo1` формируется команда `ER dx, dy;`, а на поле знака фиксируется последний прямоугольник. Команда `ER` не изменяет положение текущей точки.

12.3.5. Включение команды *RR*

Команда `RR` (Fill Rectangle Relative — Залить относительный прямоугольник) предназначена для построения залитого прямоугольника, опирающегося одним из своих углов на текущую точку. Размеры прямоугольника формируются путем перетаскивания курсора мыши. После щелчка по кнопке `RR` вы должны зажать левую кнопку мыши и перемещать ее курсор до тех пор, пока не будет построен прямоугольник нужного размера. За точными числовыми значениями приращений можно наблюдать в строке состояния (dx =..., dy = ...). В момент отпускания кнопки на поле `Memo1` формируется команда `RR dx, dy;`, а на поле знака фиксируется последний прямоугольник. Команда `RR` не изменяет положение текущей точки.

12.3.6. Включение команды *AR*

Команда `AR` (Arc Relative — Дуга относительная) предназначена для построения дуги окружности, начинающейся в текущей точке. После щелчка по кнопке `AR` появляется диалоговое окно следующего вида (рис. 12.3).

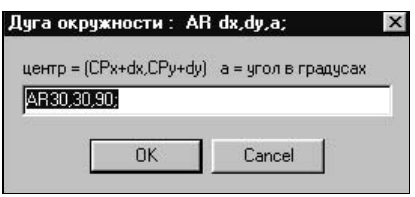


Рис. 12.3. Диалоговое окно для построения дуги

После набора нужных смещений dx и dy, определяющих положение центра окружности (CPx+dx, CPy+dy), и величины центрального угла (a, положительное значение угла соответствует движению по дуге против часовой стрелки) на поле Memo1 формируется команда AR dx, dy, a;, а на поле знака появляется соответствующая дуга. Завершением операции ввода данных считается нажатие клавиши <Enter>. До этого момента значения параметров в диалоговом окне можно редактировать. После выполнения команды AR текущая точка перемещается в конец дуги.

12.3.7. Включение команды *EW*

Команда EW (Edge Wedge — Граница сектора) предназначена для построения кругового сектора (рис. 12.4). После щелчка по кнопке EW появляется диалоговое окно следующего вида:

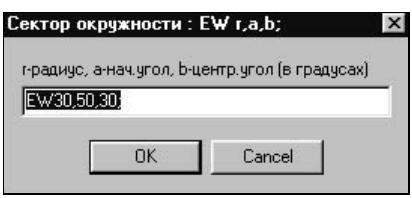


Рис. 12.4. Диалоговое окно для построения кругового сектора

Центр окружности, содержащий сектор, находится в текущей точке (CPx, CPy). Величина радиуса r задается в диалоговом окне. Первая угловая величина (a) определяет положение начального радиус-вектора, второй параметр (b) задает величину центрального угла. Знаки угловых величин определяют направление движения по дуге окружности (положительным величинам соответствует перемещение против часовой стрелки). Завершением операции ввода данных считается нажатие клавиши <Enter>. До этого момента значения параметров в диалоговом окне можно редактировать. После выполнения команды EW на поле Memo1 заносится текст сформированной команды EW r, a, b;, а на поле знака появляется изображение кругового сектора (т. е. дуги, концы которой соединены с центром окружности). Текущая точка после выполнения команды остается в своей прежней позиции.

12.3.8. Включение команды *WG*

Команда WG (Fill Wedge — Залить Сектор) предназначена для построения залитого кругового сектора (рис. 12.5). После щелчка по кнопке WG появляется диалоговое окно следующего вида.

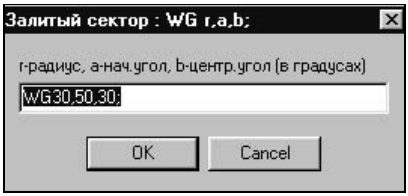


Рис. 12.5. Диалоговое окно для построения залитого сектора

Центр окружности, содержащий сектор, находится в текущей точке `(CPx, CPy)`. Величина радиуса `r` задается в диалоговом окне. Первая угловая величина (`a`) определяет положение начального радиус-вектора, второй параметр (`b`) задает величину центрального угла. Знаки угловых величин определяют направление движения по дуге окружности (положительным величинам соответствует перемещение против часовой стрелки). Завершением операции ввода данных считается нажатие клавиши <Enter>. До этого момента значения параметров в диалоговом окне можно редактировать. После выполнения команды `WG` на поле `Memo1` заносится текст сформированной команды `WG r, a, b;`, а на поле знака появляется изображение залитого кругового сектора . Текущая точка после выполнения команды остается в своей прежней позиции.

12.3.9. Включение команды *FP*

Команда `FP` (Fill Polygon — Залить полигон) предназначена для заливки контура, сформированного в режиме полигона. Ее можно применять только в том случае, если предварительно в текст графической программы была включена цепочка команд `PM0` →...→ `PM2`, и между ними был действительно сформирован контур, содержащий не менее трех точек. В противном случае редактор выдаст сообщение о недопустимости использования команды `FP`.

После выполнения команды на поле `Memo1` появится строка `FP ;`, а контур полигона будет залит синим цветом.

12.3.10. Включение команды *EP*

Команда `EP` (Edge Polygon — Граница полигона) предназначена для построения контура, сформированного в режиме полигона. На самом деле, в режиме полигона его контур строится, но для выделения его звенья рисуются красным цветом. Применение команды `EP` сводится к перекраске звеньев полигона в синий цвет. Команду `EP` можно применять только в том случае, если предварительно в текст графической программы была включена цепочка команд `PM0` →...→ `PM2`, и между ними был действительно сформи-

рован контур, содержащий не менее трех точек. В противном случае редактор выдаст сообщение о недопустимости использования команды `EP`.

После выполнения команды на поле `Memo1` появится строка `FP;`, а контур полигона будет залит синим цветом.

12.3.11. Включение команд *PM0*, *PM1* и *PM2*

Команда `PM0` (Polygon Mode — Режим полигона) включает режим полигона. Все следующие за ней графические команды предназначены для формирования внешней границы области полигона и исключаемых из нее внутренних односвязных областей — дыр. Описание каждой дыры должно начинаться с команды `PM1`. Если в полигоне должно быть несколько дыр, то каждой из них должна предшествовать своя команда `PM1`. После выбора команды `PM2` режим полигона отключается и для заливки или перекрашивания контура полигона надо воспользоваться командой `FP` или `EP`.

12.3.12. Включение команды *SP*

Команда `SP` (Select Pen — Выбрать перо) предназначена для смены пера, и необходимость ее применения возникает при построении многоцветных изображений или знаков, содержащих линии разной толщины. Вообще говоря, в конкретных приложениях, с командой `SP` должна быть связана таблица фиксированных перьев, цвета и ширина которых известны пользователю. Однако, язык HPGL/2 предусматривает возможность работы с пером, характеристики которого изменяются в процессе выполнения программы. Для этой цели используются команды `PW` (Pen Width — Ширина пера) и `PC` (Pen Color — Цвет пера). Форматы этих команд описаны в предыдущем *разд. 12.3.11 данной главы*, однако технология построения графических образов дискретных знаков пока не предусматривала их прямое применение.

12.4. Работа с командами главного меню

Главное меню редактора включает пять следующих разделов:

- группа команд **Таблица** (`TMenuItem=TabZN`), обеспечивающих выполнение процедур, связанных с созданием и использованием таблиц с описаниями графических образов условных знаков;
- группа команд **Знак** (`TMenuItem=Znak`), обеспечивающих доступ к табличным описаниям знаков и выполнение различных манипуляций над текущим изображением;
- группа команд **Подложка** (`TMenuItem=Image`), позволяющих разместить на поле знака растровое изображение, по которому должно быть создано векторное описание;

- группа команд **PLT-файл** (`TMenuItem=PLTfile`), позволяющих извлечь из файла описание знака в формате HPGL или запомнить текущее изображение в автономном текстовом файле;
- раздел **О программе** (`TMenuItem=About`).

12.4.1. Работа с командами раздела *Таблица*

В данный раздел включены три команды, обеспечивающие создание новой таблицы условных знаков, считывание в оперативную память существующей таблицы знаков и запоминание откорректированной таблицы в дисковом файле.

Структура вновь создаваемой таблицы такова:

- первые 128 байт заняты информационной шапкой, содержимое которой может быть откорректировано с помощью любого шестнадцатеричного редактора (в дальнейшем предполагается формирование шапки в момент создания таблицы по информации, введенной пользователем);
- следующие 2 000 байт заняты двухбайтовыми индексами знаков, которые формируются по следующему алгоритму — индекс знака с номером `j` равен `1064+j`. По сути дела, индекс играет роль относительного указателя на начало графического описания знака в таблице, рассматриваемой как массив с двухбайтовыми элементами типа `short int`;
- двухбайтовое целое число, на которое "смотрит" индекс `j`-го знака играет роль счетчика графических команд в описании знака. Нулевое значение счетчика команд означает, что в таблице не содержится описание знака с указанным номером. Если счетчик графических команд отличен от нуля, то графическая программа, предназначенная для отображения знака, располагается сразу вслед за счетчиком. Длина графического описания знака, естественно, переменная, но т. к. длина каждой графической команды известна, то не составляет труда определить, где кончается описание данного знака и начинается графическое описание следующего знака.

Длина вновь создаваемой таблицы — 4 128 байт, и все счетчики графических команд пока еще нулевые. Имя вновь создаваемой таблицы запрашивается у пользователя в режиме диалога, и, если таблица с таким же именем уже существует, требуется подтверждение на уничтожение прежней информации.

Команда **Открыть таблицу знаков** (`TMenuItem=OpenTab`) продублирована кнопкой с надписью **Открыть таблицу**. Выполнение этой команды приводит к появлению стандартного диалогового окна, в котором отображаются все файлы текущего каталога, имена которых начинаются с буквы z. Диалоговое окно обеспечивает просмотр любых других каталогов. После выбора нужной таблицы происходит ее загрузка в оперативную память, и только тогда можно

редактировать любой из уже имеющихся знаков или формировать и записывать в таблицу изображение нового знака.

Команда **Закрыть таблицу знаков** (`TMenuItem=CloseTab`) тоже продублирована быстрой кнопкой. Ее выполнение начинается с реорганизации описаний знаков. Дело в том, что любое новое или измененное прежнее описание записывается в конец таблицы, считанной в оперативную память. Поэтому перед записью таблицы на диск все графические описания упорядочиваются по возрастанию номеров, а все устаревшие описания — исключаются.

12.4.2. Работа с командами раздела *Знак*

Раздел **Знак** включает 12 команд, разбитых на две группы. Команду первой группы обеспечивают доступ к описанию знака в таблице, считанной в оперативную память:

- **Прочитать знак из таблицы**, команда продублирована кнопкой **Показать знак №** (`TMenuItem=ReadZN`). После ее выполнения описание знака с указанным номером появляется в поле `Memo1`, а изображение знака — на поле рисования. Если в таблице не содержится описание указанного знака, то выдается соответствующее сообщение.
- **Прочитать следующий знак**, команда продублирована кнопкой **Показать следующий** (`TMenuItem=NextZN`). Использование команды позволяет не набирать номер очередного знака.
- **Записать знак в таблицу**, команда продублирована кнопкой **Записать знак №** (`TMenuItem=WriteZN`). В процессе ее выполнения запрашивается номер записываемого знака и, если в таблице находилось описание с таким же номером, предлагается подтвердить необходимость замены предыдущего описания на описание текущего знака.
- **Удалить знак из таблицы** (`TMenuItem=DeleteZN`). Команда позволяет исключить из таблицы описание существующего знака.

Выполнение большинства команд первой группы начинается с появления диалогового окна, в котором пользователь должен ввести номер знака из диапазона [1,999].

Вторая группа команд обеспечивает выполнение следующих манипуляций над описанием и изображением текущего знака:

- Формирование изображения, симметричного текущему знаку. Команда **Зеркало на оси *x*** (`TMenuItem=Xflip`) предполагает, что осью симметрии является ось *x*, команда **Зеркало на оси *y*** (`TMenuItem=Yflip`) использует в качестве оси симметрии ось *y*.
- С помощью команд **Повернуть на 90 градусов** (`TMenuItem=Rot90`) и **Повернуть на −90 градусов** (`TMenuItem=Rot_90`) изображение текущего знака поворачивается на 90° против или по часовой стрелке.

- Команда **Сдвинуть знак** (`TMenuItem=ShiftZN`) запрашивает в режиме диалога значения смещений по осям x и y и реализует параллельный перенос текущего изображения за счет подключения команды `PU dx,dy;` в начале графического описания.
- Команда **Изменить размер поля знака** (`TMenuItem=ResizeZN`) используется, как правило, при работе с изображением, целиком не помещающимся на стандартном поле рисования. В режиме диалога она сообщает текущие размеры поля рисования и запрашивает новые габариты. Увеличение размеров по одной или обеим координатным осям позволит прокручивать на поле рисунка достаточно большие графические изображения.
- Команда **Сетка** (`TMenuItem=Grid`) включает или отключает режим воспроизведения координатной сетки на поле рисования.

12.4.3. Работа с командами раздела *Подложка*

Подложка представляет собой растровое изображение, используя которое можно сформировать соответствующее векторное описание знака в терминах команд HPGL. Ее можно как бы подложить вместо фона под координатную сетку и использовать для формирования соответствующего векторного контура. В качестве подложки может быть использовано любое графическое изображение, хранящееся в файле с расширением bmp или pcx. С помощью команд раздела **Подложка** можно вызвать подложку на поле рисования (команда **Прочитать подложку** — `TMenuItem=ReadImage`), переместить ее в нужную позицию (команда **Сдвинуть подложку** — `TMenuItem=ShiftImage`), изменить ее габариты (команда **Изменить размер подложки** — `=TMenuItem=ResizeImage`) или удалить ее с поля рисования (команда **Удалить подложку** — `TMenuItem=DeleteImage`).

При работе с подложкой следует помнить, что ее графический образ размещается в прямоугольной области. При вызове подложки левый верхний угол окаймляющего прямоугольника (точка привязки подложки) размещается в левом верхнем углу поля рисования. Изменения позиции подложки сводятся к изменению координат точки привязки, смещения которой запрашиваются в режиме диалога. Размеры габаритного прямоугольника при изменении размера подложки могут меняться анизотропно, т. е. не обязательно пропорционально по каждой из осей.

12.4.4. Работа с командами раздела *PLT-файл*

В состав этого раздела входят две команды, с помощью которых на поле `Memo1` можно извлечь содержимое текстового файла с описанием знака в формате HPGL (команда **Прочитать PLT-файл** — `TMenuItem=ReadPLT`) или записать описание текущего знака в автономный файл (команда **Записать PLT-файл** — `TMenuItem=WritePLT`).

12.4.5. Преобразование векторных описаний в растровые

В третьей версии редактора дискретных знаков появилась возможность запоминания растровых монохромных изображений, соответствующих знаку, тем или иным способом, сформированном на рабочем поле. А появиться там он мог либо в результате извлечения векторного описания из существующей таблицы условных знаков, либо в результате загрузки соответствующего plt-файла, либо в результате формирования знака средствами редактора.

После того как на поле появилось изображение нужного знака, достаточно нажать новую кнопку **Растр**, чтобы было вызвано диалоговое окно для ввода имени запоминаемого bmp-файла. Имя надо набирать без расширения. В указанном файле запоминается монохромное изображение текущего знака, ориентировочный размер bmp-файла порядка 2 Кбайт (вообще говоря, зависит от размеров знака, точнее — от площади габаритного прямоугольника, окаймляющего знак). Одновременно с bmp-файлом запоминается 4-байтовый файл с таким же именем и расширением org, содержащий двухбайтовые координаты основной точки привязки. Эти координаты вычислены в системе координат bmp-прямоугольника, начало которой совпадает с верхним левым углом, а ось *y* направлена вниз (по образу и подобию экранных координат).

Заголовок монохромного bmp-файла занимает 62 байта, структура которых достаточно подробно прокомментирована в программе. Вслед за заголовком располагаются строки растра с учетом дополнительных байт в строке (длина каждой строки в растре формата bmp должна быть кратна 32 битам). Порядок строк в растре — обратный по отношению к естественному представлению об изображении. Первая строка растра соответствует самой последней (нижней) строке изображения, вторая строка — предпоследней и т. д. Каждый пиксел растра кодируется одним битом, который равен `1` для "черного" цвета и `0` — для "белого". Сменив палитру в заголовке, можно получить любое двухцветное изображение условного знака.

12.5. Пошаговое построение знака

Редактор позволяет выполнить построение знака в пошаговом режиме, при котором очередное нажатие по кнопке приводит к выполнению очередной графической команды. Для установки режима пошагового построения необходимо щелкнуть по самой левой кнопке, расположенной под кнопкой **Перерисовать текст**. После щелчка на этой кнопке появляется цифра `1`, курсор в поле `Memo1` устанавливается на первой графической команде и производится очистка поля рисования. Последующие щелчки по кнопкам > или < увеличивают или уменьшают счетчик команд на `1`, а на поле рисования отображается выполнение графических команд от начала описания до выделенной строки.

12.6. Набор компонент и их свойства

В составе редактора используется довольно много объектов, большинство из которых на рис. 12.6 выделено программными идентификаторами и типами компонент. Некоторые из них играют чисто косметическую роль, такие как объекты `Bevel1` — `Bevel6` и `Panel1` — `Panel4`, которые подчеркивают границы размещенных на них компонент и играют роль контейнеров. Функциональное назначение компонент отражено в табл.12.2.

Таблица 12.2. *Функциональное назначение компонент*

Тип компонента	Название	Назначение
`Tform`	`Form1`	Основное окно приложения
	`About`	Окно справки о программе
`TImage`	`Image1`	Рабочее поле для отображения знака
`TScrollBox`	`ScrollBox1`	Подложка под `Image1` для организации прокрутки изображения
`TMemo`	`Memo1`	Окно с символьным описанием текущего знака
`TOpenDialog`	`OpenDialog1`	Выбор таблиц знаков и файлов
`TMainMenu`	`MainMenu1`	Главное меню редактора
`TUpDown`	`UpDown1`	Управление масштабом изображения
`TButton`	`Button1`	Номер текущей команды
	`Button2`	Возврат на предыдущую команду **<**
	`Button3`	Переход к следующей команде **>**
	`Button4`	Команда **Показать следующий**
	`Button5`	Команда **Открыть таблицу**
	`Button6`	Команда **Закрыть таблицу**
	`Button7`	Команда **Записать знак №**
	`Button8`	Команда **Растр**
	`MemoClear`	Стирание описания знака и его изображения

Таблица 12.2 *(окончание)*

Тип компонента	Название	Назначение
	applaybtn	Перерисовка текста программы
	PUbtn, PDbtn, CIbtn, ERbtn, RRbtn, EWbtn, WGbtn, FPbtn, Epbtn, P0btn, P1btn, P2btn, SPbtn	Инструментальные кнопки, инициирующие выполнение соответствующих графических команд языка HPGL
	Znak_N	Команда **Показать знак №**
TLabel	Label1	Значения абсолютных координат x,y=...
	Label2	Заголовок "Редактируемый текст"
	Label3	Метка "Масштаб ="
	Label4	Метка "dx,dy= ..."
	Label5	Значения относительных координат dx,dy
	Label6	Значение "R = ..."
	Label7	Значение коэффициента увеличения
TBevel	Bevel1	Подложка под Label1
	Bevel2	Подложка под Label4 и Label5
	Bevel3	Подложка под Label6
	Bevel4	Подложка под Label6 и UpDown1
	Bevel5	Не задействована
	Bevel6	Подложка под кнопки Button1 — Button3
TPanel	Panel1	Подложка под кнопки команд HPGL
	Panel2	Подложка под Image1
	Panel3	Подложка под окно Memo1 и связанные с ним кнопки
	Panel4	Подложка под информационную строку

Самое левое поле отведено для отображения абсолютных (x,y =...) и относительных (dx,dy=...) координат текущей точки в шагах плоттера. Поле "R=..." предназначено для отображения радиуса окружности. По достижении нуж-

ной величины "резиновой" окружности необходимо отпустить клавишу мыши. Правее метки `Label7` с надписью "Масштаб" располагается объект `UpDown1`, с помощью которого можно изменить масштаб изображения условного знака от 1 до 4.

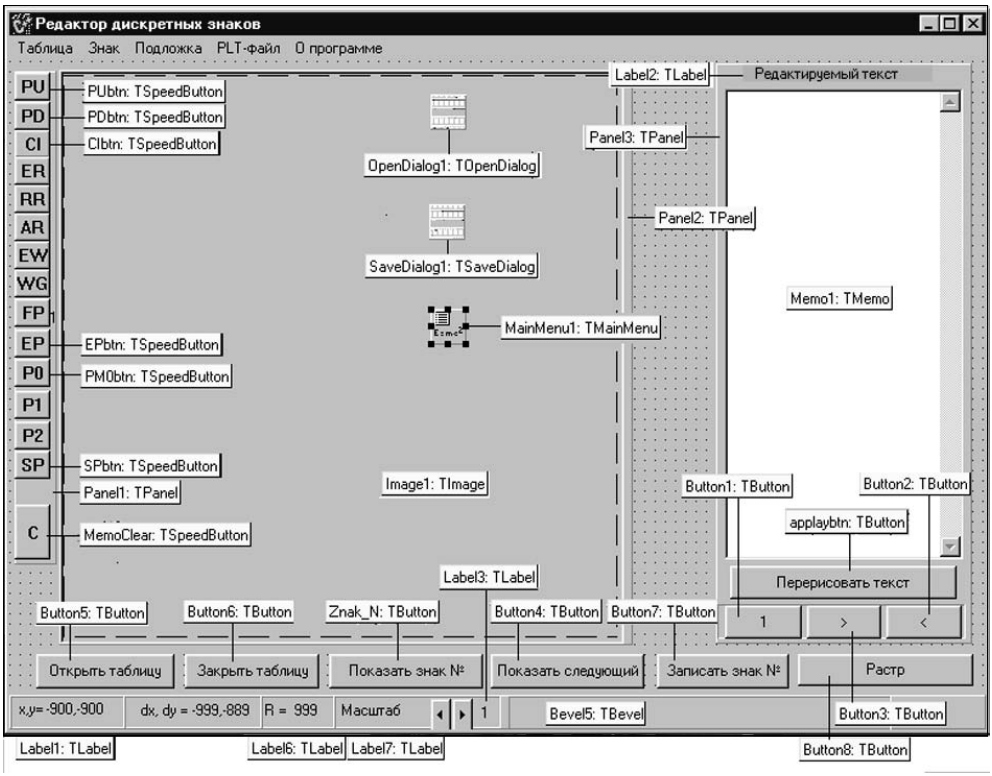


Рис. 12.6. Набор компонент на форме редактора условных знаков

Быстрые кнопки (`SpeedButton`), используемые для инициации выполнения соответствующих команд языка HPGL и очистки рабочего поля, размещены на общей панели и объединены в группу. Нажатие одной из них автоматически отключает режим, включенный предыдущей кнопкой. Все эти кнопки могут быть одновременно отжаты (значение свойства `AllowAllUp=True`).

Кнопки `Button1`, `Button2` и `Button3` участвуют в режиме пошагового воспроизведения графической программы, расположенной с поле `Memo1`. Заголовок первой из них отображает номер последней выполненной команды. Две другие выдают сигнал для перехода к следующей > или предыдущей < графической команде.

Каких-то особых свойств у остальных объектов, кроме размещения и размеров, не наблюдается.

12.7. Реализация функций редактора

В редакторе используется довольно большое количество глобальных переменных, полный перечень которых приведен в табл. 12.3

Таблица 12.3. *Глобальные переменные редактора условных знаков*

Имя переменной	Тип	Назначение
Astr	AnsiString	Рабочая строка
CurrentDir	AnsiString	Текущий каталог
fname	AnsiString	Имя файла PLT,PCX или BMP
image_name	AnsiString	Имя файла подложки
number	AnsiString	Номер знака в символьном формате
step1	AnsiString	—
tab_name	AnsiString	Имя открываемой таблицы знаков
Bmp_move	bool	Признак перемещения подложки
Bmp_resize	bool	Признак изменения размеров подложки
cont1	bool	Признак замкнутости внешнего контура
cont2	bool	Признак замкнутости внутреннего контура
done	bool	Признак нажатой клавиши мыши
is_image	bool	Признак наличия подложки
ModifyTab	bool	Признак модификации таблицы в памяти
pm0	bool	Признак начала внешнего контура полигона
pm1	bool	Признак начала внутреннего контура полигона
pm2	bool	Признак завершения полигона
polyg	bool	Признак режима полигона
pu	bool	Признак команды PU перед полигоном
Cstr	char [80]	Рабочая строка
shapka	char *	Указатель на шапку длиной 128 байт
r1	double	Радиус "резиновой" окружности

Таблица 12.3 *(продолжение)*

Имя переменной	Тип	Назначение
x	double [1000]	
y	double [1000]	
bit	Graphics::TBitmap *	Копия изображения условного знака
Bmp	Graphics::TBitmap *	Изображение подложки
command	int	Числовой код упакованной команды
end_polig, start_polig	int	Строки Memo1, определяющие начало и конец полигона
grid_step	int	Шаг сетки, зависящий от увеличения
imageheight	int	Высота поля знака
imagewidth	int	Ширина поля знака
middlx	int	Центр поля рисунка по оси *x*
middly	int	Центр поля рисунка по оси *y*
nc1,nc2	int	Количество точек в наружном и внутреннем контурах полигона
p	int [2000]	pintarr *p
rold	int	Радиус для "резиновой" окружности
tab_handle	int	Номер файла открытой таблицы знаков
tab_size	int	Фактическая длина считанной в память таблицы знаков
tek	int	Номер текущей строки в Memo1
tek_line	int	Номер строки Memo1, до которой выполняется построение в пошаговом режиме
xc,yc	int	Точка внешнего контура для перекраски
xold,yold	int	Точка для резинового прямоугольника
Xpodl,Ypodl	int	Точка привязки подложки
zoom	int	Приращение к размеру подложки
zoom_k	int	Коэффициент увеличения (1, 2, 3, 4)
addr	short *;	Указатель на массив относительно адреса знаков в таблице
nz	short [30000]	Память для сжатия таблицы знаков

Таблица 12.3 *(окончание)*

Имя переменной	Тип	Назначение
tabdz	short [30000]	Память под таблицу знаков во внутреннем формате
z	short *;	Указатель на строку графических команд
beg_pol_pt	TPoint	Начальная точка полигона в полигоне
contur1	TPoint [512]	Внешний контур полигона
contur2	TPoint [512]	Внутренний контур полигона
CP	TPoint	Текущая точка
Lines_CP	TPoint [512]	Текущие точки в начале каждой строки
tmp_p	TPoint [100]	Рабочий массив для дуг и окружностей
rectdr	TRect	Область подложки в объекте Image1

12.7.1. Стартовые и финальные операции

В состав этой группы кроме конструктора формы входят обработчики событий OnCreate, OnDestroy и OnCloseQuery:

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{ CurrentDir=GetCurrentDir(); }
// ---------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
// перехват сообщений программе — нажата системная клавиша
  Application->OnMessage=AppOnMessage;
  end_polig=-1;          // конец последнего выделенного полигона
  start_polig=-1;        // начало последнего выделенного полигона
  command=0;             // не выбрано ни одной из команд
  zoom_k=1;              // масштаб 1:1
  is_image=false;        // нет рисунка под сеткой
  fname="";              // имя  plt — файла
  tab_name="";           // имя таблицы знаков
  imagewidth=ScrollBox1->Width — 2; // ширина рисунка
  imageheight=ScrollBox1->Height-2; // высота рисунка
```

```
  grid_step=4;          // шаг сетки
// координаты осей
  middlx=round(imagewidth/2);
  middly=round(imageheight/2);
// текущая точка
  CP.x=middlx;
  CP.y=middly;
// размер рисунка
  Image1->Width=imagewidth;
  Image1->Height=imageheight;
// выделение памяти под рисунок
  bit=new Graphics::TBitmap();
  bit->Width=Image1->Width;
  bit->Height=Image1->Height;
//  Image1->Picture->Graphic=bit;
// выделение памяти под рисунок подложки
  Bmp=new Graphics::TBitmap();
// предыдущая точка
  xold=CP.x;
  yold=CP.y;
// левый верхний угол подложки
  Xpod1=0;
  Ypod1=0;
  Bmp_move=false;       // режим движения подложки выключен
  Bmp_resize=false;     // режим масштабирования подложки выключен
  DrawGrid(Sender);
}
// ---------------------------------------------------------------
void __fastcall TForm1::FormCloseQuery(TObject *Sender,
                                       bool &CanClose)
{// обработчик события — подтверждение закрытия формы
  TMsgDlgButtons b ;
  b<< mbYes << mbCancel;
  if(MessageDlg( "Закрыть редактор ?",
     mtConfirmation,b,0)==mrCancel) CanClose=false;
}
// ----------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{// уничтожение формы, освобождение динамической памяти
```

```
  delete bit;
  bit=NULL;
  delete Bmp;
  Bmp=NULL;
// Если таблица была модифицирована, ее нужно сохранить
  if(ModifyTab)CloseTabClick(Sender);
}
```

Запоминание текущего каталога в переменной `CurrentDir` позволяет сократить поиски plt-файлов и набора данных с библиотекой знаков. Если этого не сделать и не восстановить начальный каталог диалогового окна `OpenDialog1` (свойство `InitialDir`), то приходится довольно долго добираться до информации, которую, как правило, целесообразно размещать в текущем каталоге.

Основные действия начального периода, сосредоточенные в обработчике события `OnCreate`, сводятся к формированию начальных значений многочисленных переменных и выделению памяти для хранения изображений. Наличие подробного комментария в программе делает излишним какие-либо дополнения. Единственный момент, заслуживающий разъяснения, связан с подключением перехватчика событий `AppOnMessage`. Дело в том, что во время работы редактора используются клавиатура и мышь для реализации тонкой навигации по полю рисунка, перемещения подложки и др. Поэтому некоторые сообщения операционной системы надо перехватывать до того, как они попадут непосредственному адресату. Процедура `AppOnMessage` предназначена для такой разборки. Если перехваченное событие не связано с запланированными в редакторе функциями, то оно будет передано настоящему адресату.

Обработчик события `OnCloseQuery` дает возможность пользователю продолжить работу, если он непреднамеренно поспешил закрыть редактор. Перед уничтожением формы освобождается динамически занятая память. В случае необходимости производится запоминание таблицы знаков, модифицированной в оперативной памяти.

12.7.2. Обработка кнопок инструментальной панели

На инструментальной панели расположены 15 кнопок, предназначенных для инициализации графических команд языка HPGL и стирания изображения текущего знака с одновременной очисткой поля `Memo1`. Реакция на щелчок по одной из командных кнопок сводится к одной из двух следующих процедур. Либо в переменной `command` фиксируется условный номер нажатой кнопки и продолжение обработки соответствующей команды переносится на события мыши, обеспечивающие автоматический ввод недостающих параметров (координаты, радиус), либо выдается соответствующее

диалоговое окно, предназначенное для ввода текста команды с нужным набором числовых аргументов. Во втором случае поступившая команда добавляется к списку графических команд на поле `Memo1`, и производится перерисовка текущего изображения путем вызова функции `applaybtnClick`. В обработчиках некоторых команд предусмотрена проверка допустимости их применения в данный момент времени.

Ниже приводится текст программ соответствующих обработчиков:

```
// Обработка нажатия кнопок на панели инструментов
void __fastcall TForm1::PUbtnClick(TObject *Sender)
{// нажата клавиша PU — переход в точку (x,y) с поднятым пером
  command=1;
}
// ---------------------------------------------------------------
void __fastcall TForm1::PDbtnClick(TObject *Sender)
{// нажата клавиша  PD — построение отрезка из CP в (x,y)
  command=2;
}
// ---------------------------------------------------------------
void __fastcall TForm1::CIbtnClick(TObject *Sender)
{// нажата клавиша  CI — построение окружности с цетром в CP
   command=3;
}
// ---------------------------------------------------------------
void __fastcall TForm1::ERbtnClick(TObject *Sender)
{// нажата клавиша  ER — построение прямоугольника с вершиной в CP
  command=4;
}
// ---------------------------------------------------------------
void __fastcall TForm1::RRbtnClick(TObject *Sender)
{// нажата клавиша  RR — заливка прямоугольника с вершиной в CP
  if(pm0 || pm1)
    {ShowMessage("В режиме полигона команда RR смысла не имеет");
     return;
    }
   command=5;
}
// ---------------------------------------------------------------
void __fastcall TForm1::ARbtnClick(TObject *Sender)
{// нажата клавиша AR — построение дуги, начинающейся в CP
```

```
  Astr="AR30,30,90;";
  bool u;
  u= InputQuery("Дуга окружности :  AR dx,dy,a;",
           "центр = (CPx+dx,CPy+dy)   a = угол в градусах",Astr);
  if(u) Memo1->Lines->Add(Astr);
  applaybtnClick(Sender);
}
// ---------------------------------------------------------------
void __fastcall TForm1::EWbtnClick(TObject *Sender)
{// нажата клавиша EW — построение сектора с центром в CP
  Astr="EW30,50,30;";
  bool u;
  u = InputQuery("Сектор окружности : EW r,a,b;",
          "r-радиус, a-нач.угол, b-центр.угол (в градусах)",Astr);
  if(u) Memo1->Lines->Add(Astr);
  applaybtnClick(Sender);
}
// ----------------------------------------------------------------
void __fastcall TForm1::WGbtnClick(TObject *Sender)
{// нажата клавиша WG — построение залитого сектора с центром в CP
  Astr="WG30,50,30;";
  bool u;
  if(pm0 || pm1)
    {ShowMessage("В режиме полигона команда WG смысла не имеет");
     return;
    }
  u = InputQuery("Залитый сектор : WG r,a,b;",
          "r-радиус, a-нач.угол, b-центр.угол (в градусах)",Astr);
  if(u)Memo1->Lines->Add(Astr);
  applaybtnClick(Sender);
}
// ----------------------------------------------------------------
void __fastcall TForm1::FPbtnClick(TObject *Sender)
{// нажата клавиша FP — заливка полигона
  if( end_polig > start_polig) Memo1->Lines->Add("FP;");
  else ShowMessage("Полигон не выделен(PM0;...PM1;...PM2;");
  applaybtnClick(Sender);
}
// ----------------------------------------------------------------
```

```
void __fastcall TForm1::EPbtnClick(TObject *Sender)
{// нажата клавиша EP — обводка границ полигона
  if(end_polig > start_polig) Memo1->Lines->Add("EP;");
  else ShowMessage("Полигон не выделен (PM0;...PM1;...PM2;");
  applaybtnClick(Sender);
}
// ---------------------------------------------------------------
void __fastcall TForm1::PM0btnClick(TObject *Sender)
{// кнопка PM0 — начать внешний контур полигона
  if(polyg) return;       // нажата не во время
  polyg=true;
  pm0=true;
  Memo1->Lines->Add("PM0;");
  start_polig= Memo1->Lines->Count;
}
// ----------------------------------------------------------------
void __fastcall TForm1::PM1btnClick(TObject *Sender)
{// кнопка "Начать внутренний контур полигона"
  if(!polyg)
    {
      ShowMessage("Для входа в режим полигона — P0");
      return;
    }
  if(nc1<2)
    { ShowMessage("Во внешнем контуре мало точек");
      return;     // нажата не во время
    }
  pm0=false;      // закрыли режим внешнего контура
  pm1=true;
  Memo1->Lines->Add("PM1;");
}
// ---------------------------------------------------------------
void __fastcall TForm1::PM2btnClick(TObject *Sender)
{// кнопка "Завершить режим полигона"
  if(nc1<2)
    { ShowMessage("Во внешнем контуре мало точек");
      return;        // нажата не во время
    }
  polyg=false;             // выход из режима полигона
```

```
  Memo1->Lines->Add("PM2;");
  end_polig= Memo1->Lines->Count;
}
// ---------------------------------------------------------------
void __fastcall TForm1::SPbtnClick(TObject *Sender)
{// нажата клавиша SP — смена пера
  bool  u;
  Astr="1";
  u=false;
  u=InputQuery("Изменение номера пера : SP n",
               "Введите номер пера",Astr);
  if(u) Memo1->Lines->Add("SP"+Astr+";");
}
// ---------------------------------------------------------------
void __fastcall TForm1::MemoClearClick(TObject *Sender)
{// нажата клавиша C — очистка полей Image1 и Memo1
  Memo1->Clear();
  end_polig=-1;     // указатель на строку — конец полигона
  start_polig=-1;   // указатель на строку — начало полигона
  nc1=0;            // счетчик точек внешнего контура
  nc2=0;            // счетчик точек внутренних контуров
  PM0btn->Down=false;
  Form1->Caption="Таблица знаков: " + tab_name;
// Очистка поля рисунка с восстановлением сетки
  applaybtnClick(Sender);
}
```

12.7.3. Обработка команд раздела *Таблица*

В состав раздела **Таблица** входит четыре команды, обеспечивающие выбор существующей таблицы знаков, создание новой таблицы, закрытие текущей таблицы и выход из редактора. Текст соответствующих функций приведен ниже:

```
// Обработка пунктов меню работы с файлами
void __fastcall TForm1::OpenTabClick(TObject *Sender)
{// Команда "Открыть таблицу знаков"
  ModifyTab=false;
  command=0;
  tab_name="";
```

```
  OpenDialog1->InitialDir=CurrentDir;
  OpenDialog1->Title =
           "Открываем таблицу знаков и читаем ее в память";
  OpenDialog1->Filter="Таблицы знаков |z*.*";
  OpenDialog1->FileName="";
  OpenDialog1->Execute();
  tab_name=OpenDialog1->FileName;
  tab_handle=FileOpen(tab_name,fmOpenReadWrite);
  if(tab_handle < 0)
     { ShowMessage("Файл "+tab_name+" не открылся");
       return;
     }
  tab_size=FileRead(tab_handle,tabdz,30000);
  Form1->Caption="В памяти таблица знаков  "+tab_name;
  addr=tabdz+64; // указатель на индексы команд
  z=tabdz+1064;  // указатель на начало графических команд
}
// ---------------------------------------------------------------
void __fastcall TForm1::NewTabClick(TObject *Sender)
{// Команда "Создать новую таблицу знаков"
  short *newz;
  shapka=(char *)tabdz;
  StrCopy((char *)tabdz,
"Условные знаки в сжатом формате HP-GL (16 bit)// Кетков Ю.Л.,2002");
  addr=tabdz+64;
  newz=tabdz+64+1000;
  for(int i=0; i<1000; i++)
    { addr[i]=(short)(64+1000+i); // индекс i-го знака в таблице
      newz[i]=0;                  // нуль в счетчике графических команд
    }
  SaveDialog1->Title="Создаем новую таблицу знаков";
  SaveDialog1->Filter="Таблицы знаков |z*.*";
  if(SaveDialog1->Execute())
    { tab_name=SaveDialog1->FileName;
      int tab_handle=FileCreate(tab_name);
      FileWrite(tab_handle,tabdz,4128);
      FileClose(tab_handle);
      SaveDialog1->FileName="";
    }
```

```
}
// ---------------------------------------------------------------
void __fastcall TForm1::CloseTabClick(TObject *Sender)
{// Команда "Закрыть таблицу знаков"
  short *ptr1, // индекс для выборки из таблицы dztab
        *ptr2, // индекс для записи в таблицу nz
        count; // счетчик числа графических команд
  short j,k,q;
  ptr2=nz+1064; // начало графических описаний в уплотненной таблице
  for(j=0; j<1000; j++)
    {// начало графической программы в tabdz
      ptr1=tabdz+*(tabdz+64+j);
      count=*ptr1;                // количество графических команд
      nz[64+j]=(short)(ptr2-nz); // новые упорядоченные ссылки
      memcpy(ptr2,ptr1,2);        // копируем счетчик команд
      ptr1++;       ptr2++;
      for(k=0; k<count; k++)      // цикл по командам
        { q=*ptr1;
          // если команды PM0,PM1,PM2,CI,SP
          if(q==codPM||q==codCI||q==codSP)
            { memcpy(ptr2,ptr1,4);
              ptr1+=2;  ptr2+=2;
              continue;
            }
          if(q==codEP||q==codFP) // команды EP,FP
            { memcpy(ptr2,ptr1,2);
              ptr1++;   ptr2++;
              continue;
            }
          if(q==codAR||q==codEW||q==codWG) // команды EW,WG,AR
            { memcpy(ptr2,ptr1,8);
              ptr1+=4;  ptr2+=4;
              continue;
            }
          // если команды PU,PD,ER,RR
          if(q==codPU||q==codPD||q==codER||q==codRR)
            { memcpy(ptr2,ptr1,6);
              ptr1+=3;  ptr2+=3;
              continue;
```

```
            }
        }
    }
// Восстанавливаем tabdz
  tab_size=2*(ptr2-nz);
  memcpy((void*)(tabdz+64),(void *)(nz+64),tab_size-128);
  if(tab_handle>0)  FileClose(tab_handle);
  tab_handle=FileOpen(tab_name,fmOpenWrite);
  FileWrite(tab_handle,tabdz,tab_size);
  FileClose(tab_handle);
  tab_name="";
  Form1->Caption="Редактор знаков";
}
// ------------------------------------------------------------
void __fastcall TForm1::Exit1Click(TObject *Sender)
{// Команда "Выход"
  if(ModifyTab)  CloseTabClick(Sender);
  Close();
}
```

Открывание таблицы знаков сводится к стандартной процедуре работы с окном `OpenDialog1`, считыванию выбранного файла целиком в оперативную память и начальной установке некоторых указателей.

При создании новой таблицы в оперативной памяти формируется образ пустой таблицы, с помощью диалогового окна `SaveDialog1` выбирается имя новой таблицы и производится запись на диск.

Несколько сложнее обработка команды **Закрыть таблицу**. Дело в том, что при обработке текущей таблицы из нее могли удаляться какие-то знаки и добавляться новые графические описания. В результате в таблице, находящейся в оперативной памяти, могли возникнуть дыры (при удалении знака в соответствующий счетчик графических команд заносится `0`, а сама цепочка команд не удаляется). Добавление новых или модифицированных знаков связано с увеличением длины таблицы (все новые описания дописываются в хвост). Если таблица подверглась модификации, то ее надо перекомпоновать с целью упорядочения описаний и исключения пустот. Происходит это довольно просто — в оперативной памяти создается копия таблицы (массив `nz`), и содержимое старой таблицы переписывается в новую таблицу. Для повышения эффективности процедура копирования выполняется с помощью функций `memcpy`. При этом все графические команды разбиваются на группы команд одинаковой длины. Это несколько уменьшает объем программы, т. к. вызов функции `memcpy` и перемещение соответствующих указателей в старой и новой таблицах выполняется один раз на каждую группу.

Перед завершением работы приложения проверяется признак модификации таблицы знаков, находящейся в оперативной памяти, и, в случае необходимости, вызывается функция `CloseTabClick`.

12.7.4. Обработка команд раздела *Знак*

Раздел **Знак** представлен достаточно большим количеством команд. Для удобства восприятия мы разобьем их на три группы. В первую группу мы объединим процедуры типа "знак-таблица", во вторую — аффинные преобразования (сдвиги, повороты, отражения), а в третью — сервисные функции.

В состав группы "знак-таблица" относятся команды извлечения из таблицы, загруженной в оперативную память, знака с указанным номером или следующего знака, внесения в таблицу описания текущего знака и удаления из таблицы знака с указанным номером. Комментарий к приведенным функциям следует за текстом программы:

```
// Обработка команд типа "таблица-знак"
void __fastcall TForm1::ReadZNClick(TObject *Sender)
{// Команда "Прочитать знак из таблицы"
  int num;
  if(tab_name == "")
    { ShowMessage("Вы не открыли таблицу знаков");
      OpenTabClick(Sender);
    }
  Memo1->Clear();       // очистить поле знака
  number="1";           // по умолчанию начальный номер = 1
  number=InputBox("Ввод знака из таблицы",
                  "Введите номер знака",number);
  num=StrToInt(number);
  if(0>num || num>999)
     { ShowMessage(
         "Номер знака должен принадлежать диапазону [0,999]");
       ReadZNClick(Sender);
     }
// чтение знака из таблицы
  Read_Znak(num);
  Form1->Caption="В памяти таблица знаков  "+tab_name+
                 "  Прочитан знак номер "+number;
  applaybtnClick(Sender);
}
```

```
// ---------------------------------------------------------------
void __fastcall TForm1::NextZNClick(TObject *Sender)
{// Команда "Прочитать следующий знак"
  if(tab_name == "")
    { ShowMessage("Вы не открыли таблицу знаков");
      OpenTabClick(Sender);
    }
  int num=StrToInt(number)+1;
  if(0>num || num>999)
    { ShowMessage(
           "Номер знака должен принадлежать диапазону [0,999]");
      return;
    }
  Memo1->Clear();       // очистить поле знака
  number=num;
  Read_Znak(num);
  Form1->Caption="В памяти таблица знаков  "+tab_name+
                "  Прочитан знак номер "+number;
  applaybtnClick(Sender);
}
// ---------------------------------------------------------------
void __fastcall TForm1::DeleteZNClick(TObject *Sender)
{// Команда "Удалить знак из таблицы
  int num;
  TMsgDlgButtons b ;
  b<< mbYes << mbCancel;
  if(tab_name == "")
    { ShowMessage("Вы не открыли таблицу знаков");
      OpenTabClick(Sender);
    }
  number="1";           // по умолчанию начальный номер = 1
  number=InputBox("Удаление знака из таблицы",
                 "Введите номер знака",number);
  num=StrToInt(number);
  if(0>num || num>999)
     {ShowMessage(
        "Номер знака должен принадлежать диапазону [0,999]");
       DeleteZNClick(Sender);
     }
```

```
  if(MessageDlg("Вы действительно хотите удалить этот знак?",
                mtConfirmation,b,0)==mrCancel) return;
  tabdz[*(addr+num)]=0x00;      // очистка счетчика команд
  ModifyTab=true;               // признак модификации таблицы
}
// ---------------------------------------------------------------
void __fastcall TForm1::WriteZNClick(TObject *Sender)
{// Команда "Записать знак в таблицу"
  int j,k,k1,num,arg1,arg2,arg3;
  TMsgDlgButtons btn ;
  btn<< mbYes << mbCancel;
  number="1";
  number=InputBox("Запись знака в таблицу",
                  "Введите номер знака",number);
  num=StrToInt(number);
  if(tabdz[*(addr+num)]!=0)
    { if(MessageDlg("Заменить старый знак?",
         mtConfirmation,btn,0)==mrCancel) return;
    }
  ModifyTab=true;
// формируем ссылку на новое описание в конце таблицы
  *(addr+num)=(short)(tab_size/2);
// Начинаем упаковку команд из Memo1
// относительный адрес первого полуслова в конце таблицы
  k=tab_size>>1;
  k1=k+1;
  tabdz[k]=0;   // счетчик команд = 0
  for(j=0; j<Memo1->Lines->Count; j++)
    { tabdz[k]++; // продвигаем счетчик команд
      Astr=Memo1->Lines->Strings[j]; // выбор очередной команды
      if(Astr.Length()==0)
        { tabdz[k]--;
          continue;        // пропуск пустых строк
        }
      if(Astr[1]=='P'&&Astr[2]=='U')
        {// упаковка PU
          tabdz[k1++]=codPU; // код команды PU
          goto arg_2;
        }
```

```
if(Astr[1]=='P'&&Astr[2]=='D')
  {// упаковка PD
    tabdz[k1++]=codPD; // код команды PD
    goto arg_2;
  }
if(Astr[1]=='E'&&Astr[2]=='R')
  {// упаковка ER
    tabdz[k1++]=codER; // код команды ER
    goto arg_2;
  }
if(Astr[1]=='R'&&Astr[2]=='R')
  {// упаковка RR
    tabdz[k1++]=codRR; // код команды RR
    goto arg_2;
  }
if(Astr[1]=='C'&&Astr[2]=='I')
  {// упаковка CI
    tabdz[k1++]=codCI; // код команды CI
    goto arg_1;
  }
if(Astr[1]=='S'&&Astr[2]=='P')
  {// упаковка SP
    tabdz[k1++]=codCI; // код команды SP
    goto arg_1;
  }
if(Astr[1]=='F'&&Astr[2]=='P')
  {// упаковка FP
    tabdz[k1++]=codFP; // код команды FP
    continue;
  }
if(Astr[1]=='E'&&Astr[2]=='P')
  {// упаковка EP
    tabdz[k1++]=codEP; // код команды EP
    continue;
  }
if(Astr[1]=='P'&&Astr[2]=='M')
  {// упаковка PM
    tabdz[k1++]=codPM; // код команды PM
    goto arg_1;
```

```
        }
      if(Astr[1]=='E'&&Astr[2]=='W')
        {// упаковка EW
          tabdz[k1++]=codEW; // код команды EW
          goto arg_3;;
        }
      if(Astr[1]=='W'&&Astr[2]=='G')
        {// упаковка WG
          tabdz[k1++]=codWG; // код команды WG
          goto arg_3;;
        }
      if(Astr[1]=='A'&&Astr[2]=='R')
        {// упаковка AR
          tabdz[k1++]=codAR; // код команды AR
          goto arg_3;
        }
arg_1:
       sscanf(&Astr[3],"%d",&arg1);
       tabdz[k1++]=(short)arg1;    // 1-й аргумент
       continue;
arg_2:
       sscanf(&Astr[3],"%d,%d",&arg1,&arg2);
       tabdz[k1++]=(short)arg1;   // 1-й аргумент
       tabdz[k1++]=(short)arg2;   // 2-й аргумент
       continue;
arg_3:
       sscanf(&Astr[3],"%d,%d,%d",&arg1,&arg2,&arg3);
       tabdz[k1++]=(short)arg1;    // 1-й аргумент
       tabdz[k1++]=(short)arg2;    // 2-й аргумент
       tabdz[k1++]=(short)arg3;    // 3-й аргумент
       continue;
    }
  tab_size=k1<<1; // новая длина таблицы
}
```

Выполнение команды **Показать знак с заданным номером** распадается на следующие четыре шага:

1. Проверить, находится ли в памяти таблица знаков и, в случае необходимости, предложить ее открыть.

2. Запросить номер знака и проверить его на принадлежность диапазону [1,999].
3. Обратиться к сервисной функции `Read_Znak`, сообщив ей номер знака, извлекаемого из таблицы.
4. Обратиться к сервисной функции `applaybtn` для воспроизведения графического образа выбранного знака.

Алгоритм обработки команды **Показать следующий знак** мало чем отличается от приведенной выше схемы. Текущий номер знака увеличивается на 1 и повторяются те же шаги. Выполнение обеих команд сопровождается изменением заголовка формы для уведомления пользователя о номере текущего знака, находящегося на экране.

Выполнение команды **Удаления знака с заданным номером** также начинается со стандартной проверки присутствия таблицы и запроса номера удаляемого знака. Перед тем, как выполнить опасную операцию, у пользователя запрашивается подтверждение, и он еще может одуматься. Но, если решение принято, то удаление знака сводится к засылке нуля в счетчик графических команд. Сама цепочка графических команд из таблицы не удаляется, и освободившееся место в дальнейшем не используется.

Наибольший объем занимает программа записи описания текущего знака в таблицу. Связано это с тем, что в таблице каждая графическая команда представлена не в символьном виде, а в специальном внутреннем формате. Такая перекомпоновка преследует две цели. Во-первых, экономится объем таблицы. Во-вторых, повышается эффективность ее использования в прикладных программах при воспроизведении знаков на экране дисплея, т. к. значения числовых параметров графических команд переведены в машинный формат. Поэтому приходится анализировать каждую строку, находящуюся в поле `Memo1`, и в зависимости от кода команды производить форматные преобразования над одним, двумя или тремя числовыми параметрами.

Вторая группа команд включает такие операции над текущим знаком, как поворот изображения на 90° или −90°, сдвиг на заданное число шагов по осям x и y, построение симметричных фигур относительно координатных осей. Как правило, операции такого рода выполняются над некоторыми аргументами команд, присутствующих на поле `Memo1`. Иногда список этих команд расширяется. Например, при реализации команды сдвига перед графической программой вставляется команда `PU`, переводящая текущую точку в новую позицию. А так как все перемещения в графической программе выполняются в относительных координатах, то никаких других изменений в структуре описания производить не приходится.

Небольшая экономия в объеме программы достигнута за счет совмещения процедур отражения относительно координатных осей в общей функции `y_flipClick`. В ее теле потребовалась дополнительная проверка `Sender=Yflip`,

чтобы решить проблему, у какой из координат надо менять знак в графических командах. Каких-либо математических проблем при выполнении того или иного преобразования над координатами и углами не возникает, поэтому дополнительных комментариев к тесту приводимой ниже программы не требуется. Выполнение любого преобразования завершается вызовом сервисной функции `applaybtnClick`, осуществляющей перерисовку измененного знака.

```
// Аффинные преобразования изображения знака
void __fastcall TForm1::ShiftZNClick(TObject *Sender)
{// Команда "Знак"->"Сдвиг изображения"
// В начало текста вставляется команда PU или меняются ее аргументы
  int x=0,y=0;
  AnsiString sx="0",sy="0";
  bool qx,qy;
  qx=InputQuery("Горизонтальный перенос",
                "Введите смещение рисунка по оси Ox",sx);
  qy=InputQuery("Вертикальный перенос",
                "Введите смещение рисунка по оси Oy",sy);
  if(qx || qy)
    {
      if(Memo1->Lines->Strings[0][2]=='U')
        {
           Astr=Memo1->Lines->Strings[0];
           sscanf(&Astr[3],"%d,%d",&x,&y);
           Memo1->Lines->Delete(0);
        }
      x+=StrToInt(sx);
      y+=StrToInt(sy);
      sprintf(Cstr,"PU%d,%d;",x,y);
      Memo1->Lines->Insert(0,(AnsiString)Cstr);
    }
  applaybtnClick(Sender);
}
// ------------------------------------------------------------
void __fastcall TForm1::Rot90Click(TObject *Sender)
{// Команда "Повернуть на 90 градусов по часовой стрелке"
  int j,x,y,a;
  for(j=0; j<Memo1->Lines->Count; j++)
    {
      Astr=Memo1->Lines->Strings[j];
```

```
      while(Astr[1]==' ') Astr.Delete(1,1);
      if(Astr[2]=='U' || Astr[2]=='D' || (Astr[2]=='R' && Astr[1]!='A'))
        {
          sscanf(&Astr[3],"%d,%d",&x,&y);
          sprintf(Cstr,"%c%c%d,%d;",Astr[1],Astr[2],y,-x);
          Memo1->Lines->Strings[j]=Cstr;
        }
      if(Astr[1]=='A' && Astr[2]=='R')
        {
          sscanf(&Astr[3],"%d,%d,%d;",&x,&y,&a);
          sprintf(Cstr,"%c%c%d,%d,%d;",Astr[1],Astr[2],y,-x,a);
          Memo1->Lines->Strings[j]=Cstr;
        }
      if(Astr[2]=='W' || Astr[2]=='G')
        {
          sscanf(&Astr[3],"%d,%d,%d;",&x,&y,&a);
          y=y-90;
          if(y<-180) y+=360;
          sprintf(Cstr,"%c%c%d,%d,%d;",Astr[1],Astr[2],x,y,a);
          Memo1->Lines->Strings[j]=Cstr;
       }
     }
  applaybtnClick(Sender);
}
// ---------------------------------------------------------------
void __fastcall TForm1::Rot_90Click(TObject *Sender)
{// Команда " Повернуть на 90 градусов против часовой стрелки"
  Rot90Click(Sender);
  Rot90Click(Sender);
  Rot90Click(Sender);
}
// ----------------------------------------------------------------
void __fastcall TForm1::y_flipClick(TObject *Sender)
{// Процедура зеркального отражения относительно оси Ох — ОУ
  int j,x,y,a;
  char str[80];
  for(j=0; j<Memo1->Lines->Count; j++)
    {   // цикл по всем командам
      strcpy(str,Memo1->Lines->Strings[j].c_str());
```

```
      if(str[1]=='U' || str[1]=='D')
        {
          sscanf(&str[2],"%d,%d;",&x,&y);
          if(Sender==Yflip) x=-x;  // отразить от оси ОУ
          if(Sender==Xflip) y=-y;  // Отразить от оси ОХ
          sprintf(&str[2],"%d,%d;",x,y);
          Memo1->Lines->Strings[j]=(AnsiString)str;
          continue;
        }
      if(str[0]=='A' && str[1]=='R')
        {
          sscanf(&str[2],"%d,%d,%d;",&x,&y,&a);
          if(Sender==Yflip) x=-x;  // отразить от оси ОУ
          if(Sender==Xflip) y=-y;  // отразить от оси ОХ
          a=-a;
          sprintf(&str[2],"%d,%d,%d;",x,y,a);
          Memo1->Lines->Strings[j]=(AnsiString)str;
          continue;
        }
      if(str[0]=='W' || str[1]=='G')
        {
          sscanf(&str[2],"%d,%d,%d;",&x,&y,&a);
          if(Sender==Yflip) y=180-y;  // отразить от оси ОУ
          if(Sender==Xflip) y=-y;     // отразить от оси ОХ
          a=-a;
          sprintf(&str[2],"%d,%d,%d;",x,y,a);
          Memo1->Lines->Strings[j]=(AnsiString)str;
        }
    }
// перерисовать измененный текст
  applaybtnClick(Sender);
}
```

Несколько особняком примыкает к этой группе команда **Изменить размер поля знака**. Дело в том, что размеры объекта `Image1`, выбранные на стадии проектирования и обеспечивающие отображение почти всех условных знаков топографических карт, оказываются недостаточными для отображения знаков очень большого размера. Приводимая ниже функция позволяет в таких случаях по требованию пользователя увеличить размеры объекта `Image1` и за счет скроллинга по всему рабочему полю добраться до невидимых

участков большого знака. После изменения габаритов объекта приходится вызвать обработчик события `OnResize`, чтобы правильно учесть новые размеры.

```
void __fastcall TForm1::ResizeZNClick(TObject *Sender)
{// Команда " Изменить размер поля знака"
  AnsiString s1,s2;
  s1="Текущая ширина поля знака в пикселах=";
  s2=imagewidth;
  InputQuery(s1+s2,"Введите новое значение",s2);
  imagewidth=StrToInt(s2);
  s1="Текущая высота поля знака в пикселах=";
  s2=imageheight;
  InputQuery(s1+s2,"Введите новое значение",s2);
  imageheight=StrToInt(s2);
  FormResize(Sender);
}
```

К группе вспомогательных функций раздела **Знак** можно отнести процедуры очистки рабочего поля (`Clear`), построения сетки и координатных осей (`GridClick`, `DrawGrid`), реакции на изменение масштабного коэффициента (`UpDown1Click`) и распаковки знака при выборке его описания из таблицы (`Read_Znak`).

```
void __fastcall TForm1::Clear(TObject *Sender)
{// процедура очистки поля знака при перерисовке изображения
// очистить экран
  Image1->Canvas->FillRect(Image1->Canvas->ClipRect);
  if(is_image) // если есть фоновое изображение, ввести рисунок
     Image1->Canvas->StretchDraw(rectdr,Bmp);
  if(Grid->Checked )
     DrawGrid(Sender); // нарисовать сетку, если это необходимо
}
// ------------------------------------------------------------
void __fastcall TForm1::GridClick(TObject *Sender)
{// Команда "Знак"->"Сетка" (включение/отключение)
  Grid->Checked=!Grid->Checked;
  applaybtnClick(Sender);
}
// ------------------------------------------------------------
void __fastcall TForm1::DrawGrid(TObject *Sender)
{// Построение координатной сетки
```

```
  int i=0;
  Image1->Canvas->Pen->Width=1;
  Image1->Canvas->Pen->Color=clLtGray;
  do
    {
      Image1->Canvas->MoveTo(0,i);
      Image1->Canvas->LineTo(Image1->Width,i);
      i+=grid_step*zoom_k;
    }
  while(i < Image1->Height);
  i=0;
  do
    {
      Image1->Canvas->MoveTo(i,0);
      Image1->Canvas->LineTo(i,Image1->Height);
      i+=grid_step*zoom_k;
    }
  while (i < Image1->Width);
  Image1->Canvas->Pen->Color=clBlack;
  Image1->Canvas->MoveTo(middlx,0);
  Image1->Canvas->LineTo(middlx,Image1->Height);
  Image1->Canvas->MoveTo(0,middly);
  Image1->Canvas->LineTo(Image1->Width,middly);
  Image1->Canvas->Pen->Color=clBlue;
  if(!polyg) Image1->Canvas->Pen->Width=5;
}
// ---------------------------------------------------------------
void __fastcall TForm1::UpDown1Click(TObject *Sender, TUDBtnType Button)
{// Изменение масштаба изображения
  zoom_k=UpDown1->Position;
  Label3->Caption=IntToStr(UpDown1->Position);
  FormResize(Sender);
}
// ----------------------------------------------------------------
void __fastcall TForm1::Read_Znak(int num)
{// Выборка знака с указанным номером из открытой таблицы
  short ptr,com;
  short cod;
  if(0>num || num>999) return;
```

```
// чтение знака из таблицы КАРТ-ДОК, находящейся в памяти
// установка указателя на начало знака в массиве z
  ptr=(short)(addr[num]-1064);
  com=z[ptr];           // число команд в знаке
  if(com==0)
   { AnsiString s="Знака с номером "+IntToStr(num)+" в таблице нет";
     ShowMessage(s);
     return;
    }
  for(;com >0;com--)
    { ptr++;
      cod=z[ptr]; // код команды
// преобразование команды в plt-формат
      if(cod==codPU)    // раскрутка команды PU
        { sprintf(Cstr,"PU%hd,%hd;",z[ptr+1],z[ptr+2]);
          Memo1->Lines->Add(Cstr);  ptr=(short)(ptr+2);
          continue;
        }
      if(cod==codCI)    // раскрутка команды CI
        { sprintf(Cstr,"CI%hd;",z[ptr+1]);
          Memo1->Lines->Add(Cstr);  ptr=(short)(ptr+1);
          continue;
        }
      if(cod==codPD)    // раскрутка команды PD
        { sprintf(Cstr,"PD%hd,%hd;",z[ptr+1],z[ptr+2]);
          Memo1->Lines->Add(Cstr);   ptr=(short)(ptr+2);
          continue;
        }
      if(cod==codER)    // раскрутка команды ER
        { sprintf(Cstr,"ER%hd,%hd;",z[ptr+1],z[ptr+2]);
          Memo1->Lines->Add(Cstr);   ptr=(short)(ptr+2);
          continue;
        }
      if(cod==codRR)    // раскрутка команды RR
        { sprintf(Cstr,"RR%hd,%hd;",z[ptr+1],z[ptr+2]);
          Memo1->Lines->Add(Cstr);   ptr=(short)(ptr+2);
          continue;
        }
      if(cod==codAR)    // раскрутка команды AR
```

```
        { sprintf(Cstr,"AR%hd,%hd,%hd;",z[ptr+1],z[ptr+2],z[ptr+3]);
          Memo1->Lines->Add(Cstr);    ptr=(short)(ptr+3);
          continue;
        }
      if(cod==codEW)    // раскрутка команды EW
        { sprintf(Cstr,"EW%hd,%hd,%hd;",z[ptr+1],z[ptr+2],z[ptr+3]);
          Memo1->Lines->Add(Cstr);    ptr=(short)(ptr+3);
          continue;
        }
      if(cod==codWG)    // раскрутка команды WG
        { sprintf(Cstr,"WG%hd,%hd,%hd;",z[ptr+1],z[ptr+2],z[ptr+3]);
          Memo1->Lines->Add(Cstr);     ptr=(short)(ptr+3);
          continue;
        }
      if(cod==codSP)    // раскрутка команды SP
        { sprintf(Cstr,"SP%hd;",z[ptr+1]);
          Memo1->Lines->Add(Cstr);     ptr=(short)(ptr+1);
          continue;
        }
      if(cod==codPM)
        { sprintf(Cstr,"PM%hd;",z[ptr+1]);
          Memo1->Lines->Add(Cstr);     ptr=(short)(ptr+1);
          continue;
        }
      if(cod==codFP)
        { Memo1->Lines->Add("FP;");
          continue;
        }
      if(cod==codEP)
        { Memo1->Lines->Add("EP;");
          continue;
        }
    }    // конец цикла раскрутки команд
}
```

Процедура `Clear` чистит содержимое рабочего поля с помощью метода `FillRect`, после чего, в случае необходимости, копирует на рабочее поле растровую подложку и восстанавливает координатную сетку.

Команда **Сетка** — единственная в меню, для которой используется свойство `Checked`. Обработчик события `OnGridClick` по щелчку на этой команде

переключает свойство `Checked` и обращается к функции `Grid`. В функции `Grid` на поверхности `Image1` с шагом `grid_step*zoom_k` строится серая сетка, облегчающая ориентацию на рабочем поле (коэффициент `step*zoom_k` учитывает масштаб изображения). Обратите внимание на установку ширины пера. Для сетки она выбирается равной одному пикселу, а перед выходом из процедуры устанавливается равной пяти пикселам, если в этот момент не установлен режим полигона. Возможно, что линия толщиной в 5 пикселов на экране кому-то покажется слишком грубой. Но не забывайте, что перед вами увеличенное изображение знака и 5 шагов плоттера соответствует толщине линии всего в 0,125 мм. Учет ширины пера важен при подготовке издательских материалов, где к толщине линий предъявляются достаточно жесткие требования. В режиме полигона утолщение контура нежелательно, т. к. граница знака при этом выглядит не очень красиво. Но т. к. внутренность контура обычно заливается, то с пером единичной толщины залитые знаки имеют гораздо более гладкую границу.

Обработка щелчков по объекту `UpDown1` сводится к изменению масштабного коэффициента `zoom_k` и вызову обработчика события `OnResize` для учета новых параметров и перерисовки изображения.

Вспомогательная функция `Read_Znak` имеет самый большой объем потому, что ей приходится анализировать каждую графическую команду, извлеченную из таблицы, где она хранится во внутреннем формате, и преобразовывать эти данные в символьный вид. Сама по себе операция распаковки отдельной команды не сложна, но таких команд довольно много.

12.7.5. Обработка команд раздела *Подложка*

Построение знака по миниатюрному образцу (размер некоторых условных знаков топографических карт порядка 1,5—2,5 мм) — довольно трудоемкое занятие. Тут требуется и дар художника, и точный глазомер, да еще и постоянное преобразование привычных единиц измерения длины в какие-то шаги плоттера. Использование подложки существенно упрощает работу по созданию библиотек условных знаков. Вы можете просканировать исходное изображение, увеличить его до необходимых размеров (при шаге в 0,025 мм потребуется увеличение в 25 раз) и подложить растровый образ под координатную сетку рабочего поля. Имея точное изображение, не так уж и трудно обвести его границы, попадая в видимые точки, а не рассчитывая каждый раз их положения и переводя абсолютные координаты в относительные перемещения.

Раздел **Подложка** составляют пять команд, с помощью которых можно прочитать содержимое bmp- или pcx-файла в левый верхний угол поля `Image1`, вырезать его, переместить в нужную позицию так, чтобы точка привязки попала в начало координат, изменить при необходимости габариты подложки. Последняя команда удаляет подложку с рабочего поля.

```
void __fastcall TForm1::ReadImageClick(TObject *Sender)
{// Команда "Прочитать подложку"
  AnsiString ext;
  fname="";        // имя файла
  Memo1->Clear();  // Очистить редактор текста
// кнопки панели инструментов отжаты
  PUbtn->Down=false;
  PDbtn->Down=false;
  CIbtn->Down=false;
  ERbtn->Down=false;
  RRbtn->Down=false;
  ARbtn->Down=false;
  EWbtn->Down=false;
  WGbtn->Down=false;
  FPbtn->Down=false;
  EPbtn->Down=false;
  PM0btn->Down=false;
  command=0;       // не выбрана команда
  zoom_k=1;        // масштаб =1
  end_polig=-1;    // не выделен полигон
  start_polig=-1;
// управление размером  поля рисунка
  imagewidth=ScrollBox1->Width-2;
  imageheight=ScrollBox1->Height-2;
  grid_step=4;     // шаг сетки
// координаты осей
  middlx=grid_step*round(imagewidth/(2*grid_step));
  middly=grid_step*round(imageheight/(2*grid_step));
// текущая точка
  CP.x=middlx;
  CP.y=middly;
// пока нет рисунка под сеткой
  is_image=false;
  Image1->Width=imagewidth*zoom_k;
  Image1->Height=imageheight*zoom_k;
  bit->Width=Image1->Width;
  bit->Height=Image1->Height;
  Image1->Picture->Graphic=bit;
  DrawGrid(Sender);
```

```
  OpenDialog1->FileName="";
  OpenDialog1->Title="Читаем подложку: BMP- или PCX-файл";
  OpenDialog1->Filter="Подложка — *.pcx или *.bmp|*.BMP;*.PCX";
  if(OpenDialog1->Execute())
    {
     fname=OpenDialog1->FileName;
     ext=ExtractFileExt(fname);
     if(ext==".BMP" || ext==".PCX"||ext==".bmp"||ext==".pcx")
       {// загрузить рисунок под сетку
         Bmp->LoadFromFile(fname);
         imagewidth=Image1->Picture->Width;
         imageheight=Image1->Picture->Height;
         rectdr.Left=0;
         rectdr.Top=0;
         rectdr.Right=Bmp->Width;
         rectdr.Bottom=Bmp->Height;
         is_image=true;
         image_name=fname;
         FormResize(Sender);
       }
    }
  Form1->Caption="Загружена подложка — файл "+fname;
  fname="";
  Clear(Sender);
  applaybtnClick(Sender);
}
// ---------------------------------------------------------------
void __fastcall TForm1::ShiftImageClick(TObject *Sender)
{// Команда " Сдвинуть подложку"
  AnsiString s="Зажмите правую кнопку мыши.\n";
           s=s+"Переместите курсор туда, где должен\n";
           s=s+"быть левый верхний угол подложки.\n";
           s=s+"Отпустите кнопку мыши";
  Bmp_move=true;
  Application->MessageBox(s.c_str(),"Сдвиг подложки",IDOK);
}
// ----------------------------------------------------------------
void __fastcall TForm1::ResizeImageClick(TObject *Sender)
{// Команда "Изменить размер подложки"
```

```
  Astr="Стрелка вверх увеличивает подложку\n";
  Astr=Astr+"а стрелка вниз уменьшает ее,\n";
  Astr=Astr+"Enter — конец изменениям;\n";
  Astr=Astr+"пропорции подложки сохраняются,\n";
  Astr=Astr+"левый верхний угол зафиксирован";
  Bmp_resize=true;
  Application->MessageBox(Astr.c_str(),
                    "Изменение размера подложки",IDOK);
  zoom=0;
}
// ---------------------------------------------------------------
void __fastcall TForm1::FormKeyDown(TObject *Sender,
                                    WORD &Key, TShiftState Shift)
{// Изменение размера подложки
  double w,h;
  TRect r;
  if(Bmp_resize)
    {
      if(Key==38)zoom++; // серый плюс
      if(Key==40)zoom--; // серый минус
      if(Bmp->Width<Bmp->Height)
        {
          w=Bmp->Width+zoom;
          if(w<1) w=1;
          h=w*Bmp->Height/Bmp->Width;
        }
      else
        {
          h=Bmp->Height+zoom;
          if(h<1) h=1;
          w=h*Bmp->Width/Bmp->Height;
        }
      rectdr.Left=Xpodl;
      rectdr.Top=Ypodl;
      rectdr.Right=Xpodl+round(w);
      rectdr.Bottom=Ypodl+round(h);
      Clear(Sender);
    }
  if(Key==13) Bmp_resize=false;
```

```
}
// ------------------------------------------------------------
void __fastcall TForm1::DeleteImageClick(TObject *Sender)
{Команда "Удалить подложку"
  Bmp_move=false;
  Bmp_resize=false;
  is_image=false;
  Image1->Canvas->FillRect(Image1->Canvas->ClipRect);
  if(Grid->Checked )
     DrawGrid(Sender); // нарисовать сетку, если это необходимо
  FormResize(Sender);
}
```

Перед загрузкой файла с подложкой рабочее поле чистится, производится масса начальных установок и восстанавливается координатная сетка. В режиме диалога пользователю предлагается выбрать в качестве подложки графический файл с расширением pcx или bmp. Выбранный файл загружается в объект `Bmp` и приписывается к левому верхнему углу поля рисования. Вызываются обработчик события `FormResize` и функция `applaybtnClick` для воспроизведения поля рисования с загруженной подложкой.

Команда перемещения подложки сводится к выдаче пользователю инструкции по тому, как это надо делать. Кроме этого, устанавливается в `True` признак `Bmp_move`, включающий режим движения подложки. Фактическое перемещение фрагмента с подложкой происходит в обработчике события `OnMouseMove` для объекта `Image1`.

Команда изменения размера подложки тоже сводится к выдаче соответствующей инструкции и установке в `true` значения признака `Bmp_resize`. Фактическое уменьшение или увеличение размера подложки происходит в функции `FormKeyDown`. Смысл этой операции довольно прост — в зависимости от нажатой стрелки производится пропорциональное уменьшение или увеличение размеров объекта `Bmp`, под который подстраивается загруженная в него картинка.

Выполнение команды **Удалить подложку** сводится к установке трех логических признаков в состояние `false` и перерисовке рабочего поля.

12.7.6. Обработка команд раздела *PLT-файл*

Раздел **PLT-файл** включает всего две команды, обеспечивающие загрузку описания условного знака из автономного файла или сохранение символьного описания из поля `Memo1` в дисковом файле. Операция загрузки после выполнения стандартного диалога формирует заголовок формы с именем

загруженного файла и обращается к функции `applaybtnClick` для отображения вызванного знака. При сохранении текущего описания в автономном файле предусмотрена проверка на существование файла с таким же именем. Замена такого файла возможна только при получении от пользователя дополнительного подтверждения.

```
void __fastcall TForm1::ReadPLTClick(TObject *Sender)
{// Команда "Прочитать PLT-файл"
  OpenDialog1->FileName="";
  OpenDialog1->Title="Читаем PLT-файл";
  OpenDialog1->Filter="файлы в формате HP-GL *.plt|*.PLT";
  if(OpenDialog1->Execute())
    {
      Memo1->Clear();
      end_polig=-1;         // указатель на строку — конец полигона
      start_polig=-1;       // указатель на строку — начало полигона
      Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
      Form1->Caption="Прочитан файл "+OpenDialog1->FileName;
      applaybtnClick(Sender);
    }
}
// ---------------------------------------------------------------
void __fastcall TForm1::WritePLTClick(TObject *Sender)
{// Команда " Сохранить как PLT-файл"
  AnsiString s="qq";
  TMsgDlgButtons b ;
  b<< mbYes << mbCancel;
 if(fname==="")
    InputQuery("Запоминаем знак в PLT-формате",
               "Введите идентификатор знака",s);
 fname=s+".PLT";
 if(FileExists(fname))
    if(MessageDlg("Заменить файл "+fname+"?",
                  mtConfirmation,b,0)==mrCancel) return;
 Memo1->Lines->SaveToFile(fname);
 Form1->Caption="Запомнили знак в файле "+fname;
 fname="";
}
```

12.7.7. Обработка команды *Растр*

Команда **Растр** инициируется кнопкой с идентичным названием. Ее выполнение начинается с удаления растровой сетки и координатных осей, чтобы на поле рисунка осталось только изображение знака.

После этого организуется цикл сканирования строк рабочего поля для определения координат противоположных вершин габаритного прямоугольника. Фактически задача состоит в том, чтобы найти точки с граничными координатами — `xmin`, `xmax`, `ymin`, `ymax`. Так как рабочее поле имеет белый фон, то на сканируемых линиях выделяются точки, имеющие цвет, отличный от белого. Размеры габаритного прямоугольника используются для формирования стандартного заголовка bmp-файла, содержащего 62 байта. В программе приведена структура этого заголовка с расшифровкой каждого поля. Если вас заинтересуют другие форматы графических файлов, то их описание можно найти в книге Д. Д. Мюррея и У. ван Райпера "Энциклопедия форматов графических файлов" (Киев: BHV, 1997).

В найденном прямоугольнике организуется повторное сканирование строк с целью выделения окрашенных и неокрашенных пикселов. По результатам этого сканирования формируется массив строк растра монохромного bmp-изображения. При этом приходится учитывать следующие детали. Во-первых, количество пикселов в строке должно занимать целое количество 32-разрядных слов. В случае необходимости к пикселам изображения добавляется недостающее количество двоичных разрядов, не несущих информативной нагрузки. Во-вторых, строки bmp-растра расположены в обратном порядке, т. е. первая строка растра соответствует последней (нижней) строке изображения. Вторая строка bmp-растра соответствует предпоследней строке и т. д. Наконец, важно правильно распределить байты в растровой строке и биты внутри байта. Преодоление последней преграды доставило больше всего хлопот.

Кроме стандартного bmp-файла, вычисляется положение начала координат (т. е. основной точки привязки) внутри габаритного прямоугольника, которые запоминаются в отдельном файле с расширением org. Положение этой точки необходимо точно знать для размещения растрового изображения условного знака на графическом документе.

```
void __fastcall TForm1::Button8Click(TObject *Sender)
{// Обработка кнопки "Растр"
  struct
  {
// Bitmap_File_Header = 14 байт
    char bfType[2];    // Два символа "BM"
    int bfSize;    // Длина bmp-файла в байтах, формируется
    short int bfReserved1; // Резерв = 0
```

```
    short int bfReserved2; // Резерв = 0
    int bfOffBits;       // Смещение до начала bmp = 62
// BitmapInfoHeader = 40 байт
    int biSize;          // Длина заголовка в байтах = 24
    int biWidth;         // Ширина bmp, формируется
    int biHeight;        // Высота bmp, формируется
    short int biPlanes;  // Число цветовых плоскостей
    short int biBitCount;// Число битов на пиксел — 1,4,8,24
    int biCompression;   // Признак сжатия — 0,1,2
    int biSizeImage;     // Размер сжатого bmp в байтах
    int biXPelsPerMeter; // Число пикслов на метр по оси x
    int biYPelsPerMeter;  // Число пикселов на метр по оси y
    int biClrUsed;        // Число используемых цветов
    int biClrImportant;   // Число важных цветов
    int BGRN0;            // Палитра для белого (цвет 0)
    int BGRN1;            // Палитра для черного (цвет 1)
  }bmp_hdr;
  bmp_hdr.bfType[0]='B';
  bmp_hdr.bfType[1]='M';
  bmp_hdr.bfReserved1=0;
  bmp_hdr.bfReserved2=0;
  bmp_hdr.bfOffBits=62;
  bmp_hdr.biSize=40;
  bmp_hdr.biPlanes=1;
  bmp_hdr.biBitCount=1;
  bmp_hdr.biCompression=0;
  bmp_hdr.biSizeImage=0;
  bmp_hdr.biXPelsPerMeter=0;
  bmp_hdr.biYPelsPerMeter=0;
  bmp_hdr.biClrUsed=0;
  bmp_hdr.biClrImportant=0;
  bmp_hdr.BGRN0=0x00FFFFFF;
  bmp_hdr.BGRN1=0x00000000;
  unsigned char mask;     // маска для формирования битов
  int fbmp;     // номер открытого файла
  int ns;       // длина строки с учетом кратности 32 пикселам
  unsigned char a[120];   // массив под формируемую строку
  int k;        // временный индекс
  int x,y;      // текущие координаты пиксела
```

```
  int xmin=Image1->Width;
  int xmax=0;
  int ymin=Image1->Height;
  int ymax=0;
  short x0,y0;
  AnsiString s;
  Grid->Checked=False;    // Отменили сетку и координатные оси
  applaybtnClick(Sender); // Перерисовали изображение
// Определение координат габаритного прямоугольника
  for(x=0; x<Image1->Width; x++)
  for(y=0; y<Image1->Height; y++)
    { if(Image1->Canvas->Pixels[x][y]!=clWhite)
        { if(x<xmin)xmin=x;
          if(x>xmax)xmax=x;
          if(y<ymin)ymin=y;
          if(y>ymax)ymax=y;
         }
    }
// Подготовка файла для записи bmp-изображения
  SetCurrentDir(CurrentDir);
  if(SaveDialog1->Execute());
  {
    fbmp=_rtl_creat(SaveDialog1->FileName.c_str(),_A_NORMAL);
    bmp_hdr.biWidth=xmax-xmin+1;  // Ширина знака
    bmp_hdr.biHeight=ymax-ymin+1; // Высота знака
    ns=(xmax-xmin+1)/32;          // Определяем длину строки
    if(ns*32 != (xmax-xmin+1)) ns++; // Корректируем неполное слово
// размер bmp-файла в байтах
    bmp_hdr.bfSize=62+ns*4*(ymax-ymin+1);
    _rtl_write(fbmp,&bmp_hdr,62);  // запись bmp-шапки
    for(y=ymax; y>=ymin; y--)      // цикл по строкам
    {
      for(k=0; k<4*ns; k++) a[k]=0; // очистка битовой строки
      k=0;
      mask=128;
      for(x=xmin; x<=xmax; x++)   // цикл в строке
      {
        if(Image1->Canvas->Pixels[x][y] != clWhite)
           a[k]=a[k]|mask;        // фиксация окрашенного бита
```

```
        mask=(unsigned char)(mask>>1); // сдвиг маски
        if(mask==0)                    // если переходим через байт
          {mask=128; k++;}             // восстанавливаем маску
      }
      _rtl_write(fbmp,a,ns*4);         // запись очередной битовой строки
    }
    _rtl_close(fbmp);                  // закрыли bmp-файл
// Формирование файла с координатами точки привязки

    SaveDialog1->FileName=ChangeFileExt(SaveDialog1->FileName,".ORG");
    fbmp=_rtl_creat(SaveDialog1->FileName.c_str(),_A_NORMAL);
    x0=middlx-xmin-2; // с учетом 0.5 толщины пера
    y0=middly-ymin-2; // с учетом 0.5 толщины пера
    _rtl_write(fbmp,&x0,2);
    _rtl_write(fbmp,&y0,2);
    _rtl_close(fbmp);
  }
}
```

12.7.8. Обработка событий мыши

События мыши на объекте `Image1` играют важную роль в выполнении многих графических команд языка HPGL, в точной навигации позиции курсора, в операциях по перемещению подложки. Поэтому на обработчики указанных событий падает довольно большая нагрузка.

Самая простая функция, отслеживающая нажатие клавиши. Ее задача сводится только к тому, чтобы в логической переменной `done` зафиксировать, что клавиша остается в нажатом состоянии.

Функция, отслеживающая движение мыши с нажатой клавишей, имеет гораздо большую нагрузку:

1. Вычисляются и отображаются значения абсолютных координат, относительных приращений и расстояния между предыдущей позицией текущей точки и положением курсора мыши.
2. Если признак `done` установлен в значение `true`, то по содержимому глобальной переменной `command` анализируется выбранная ранее графическая команда.
3. Для команды `CI` с помощью функции `Circle` строится новая и стирается предыдущая окружность (моделируется построение окружности по задаваемому с помощью мыши радиусу).

4. Для команд ER и RR аналогичным образом с помощью двух последовательных обращений к функции DrawRect моделируется построение "резинового" прямоугольника.
5. Если перемещение мыши с зажатой клавишей происходит в режиме Bmp_move=true, то верхний левый угол прямоугольника с подложкой перемещается в позицию курсора мыши и пересчитываются координаты угловых точек прямоугольника rectdr. Каждый раз при этом вызывается функция Clear для перерисовки подложки на новом месте.

Примерно такую же нагрузку выполняет обработчик события OnMouseUp:

1. Сбрасываются логические признаки done и Bmp_move, что свидетельствует о завершении соответствующей операции.
2. По значению глобальной переменной command анализируется код выполняемой графической операции.
3. Для команд PU и PD текущие координаты курсора мыши пересчитываются в систему координат знака, и текст сформированной команды добавляется к содержимому поля Memo1.
4. Для команды CI с помощью функции Circle уничтожается последняя "резиновая" окружность, и на поле Memo1 формируется текст окончательной команды.
5. Для команд ER и RR аналогичные действия выполняются с помощью функции DrawRect.
6. Модифицируются значения глобальных переменных rold, xold, yold, и вызывается функция applaybtnClick для перерисовки пополнившегося описания.

```
void __fastcall TForm1::Image1MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{// Обработка события — нажатие кнопки мыши на объекте Image1
  done=true;    // кнопка нажата
}
// ------------------------------------------------------------
void __fastcall TForm1::Image1MouseMove(TObject *Sender,
                              TShiftState Shift, int X, int Y)
{// обработка события — перемещение мыши по объекту
  double r,dx,dy;
// вывод абсолютных координат курсора мыши
  sprintf(Cstr,"%-4d,%-4d",round((X-middlx)/zoom_k),
                           round((middly-Y)/zoom_k));
  Label1->Caption=Cstr;
```

```
// вывод относительных координат курсора мыши
  sprintf(Cstr,"%-4d,%-4d",round((X-CP.x)/zoom_k),
                           round((CP.y-Y)/zoom_k));
  Label5->Caption=Cstr;
// вычисление расстояния до текущей точки
  dx=X-CP.x;
  dy=Y-CP.y;
  r=dx*dx+dy*dy;
  r1=sqrt(r);
  r=zoom_k;
  r1=r1/r ;
// вывод расстояния до текущей точки от курсора
  sprintf(Cstr,"R = %d",round(r1));
  Label6->Caption=Cstr;
  if(done)
    {
      if(command==3) // если CI
        {
// автоматическое изменение радиуса окружности
// по перемещению курсора при нажатой кнопке мыши
           Image1->Canvas->Pen->Color=clBlue;
           Image1->Canvas->Pen->Mode=pmNotXor;
           Circle(rold);
           Circle(round(r1*zoom_k));
           rold=round(r1*zoom_k);
        }
      if(command==4 || command==5) // если  ER или RR
        {
// автоматическое изменение размера прямоугольника
// по перемещению курсора при нажатой кнопке мыши
           Image1->Canvas->Pen->Color=clBlue;
           Image1->Canvas->Pen->Mode=pmNotXor;
           DrawRect(CP.x,CP.y,xold,yold);
           DrawRect(CP.x,CP.y,X,Y);
           xold=X;
           yold=Y;
        }
      if(Bmp_move)
        {
```

```
// автоматическое перемещение подложки
// вслед за позицией курсора при нажатой кнопке мыши
            Xpodl=X;
            Ypodl=Y;
            rectdr.Right=rectdr.Right-rectdr.Left+Xpodl;
            rectdr.Bottom=rectdr.Bottom-rectdr.Top+Ypodl;
            rectdr.Left=Xpodl;
            rectdr.Top=Ypodl;
            Clear(Sender);
          }
        }
}
// ---------------------------------------------------------------
void __fastcall TForm1::Image1MouseUp(TObject *Sender,
            TMouseButton Button, TShiftState Shift, int X, int Y)
{// Обработка события — отпустили кнопку мыши
  done=false;           // кнопка не нажата
  Bmp_move=false;       // конец движения подложки
  if(command==1)        // если нажата команда PU
    {
      sprintf(Cstr,"PU%d,%d;",round((X-CP.x)/zoom_k),
                              round((-Y+CP.y)/zoom_k));
      Memo1->Lines->Add(Cstr);
    }
  if(command==2)        // если нажата команда PD
    {
      sprintf(Cstr,"PD%d,%d;",round((X-CP.x)/zoom_k),
                              round((-Y+CP.y)/zoom_k));
      Memo1->Lines->Add(Cstr);
    }
  if(command==3)        // если нажата команда CI
    {
      Image1->Canvas->Pen->Mode=pmCopy;
      Circle(rold);
      if(rold > 0)
        {
          sprintf(Cstr,"CI%d;",round(rold/zoom_k));
          Memo1->Lines->Add(Cstr);
        }
```

```
    }
  if(command==4)        // если нажата команда ER
    {
       Image1->Canvas->Pen->Mode=pmCopy;
       DrawRect(round(CP.x),round(CP.y),X,Y);
       if(CP.x!=xold || CP.y!=yold)
         {
           sprintf(Cstr,"ER%d,%d;",round((X-CP.x)/zoom_k),
                                   round((CP.y-Y)/zoom_k));
           Memo1->Lines->Add(Cstr);
         }
    }
  if(command==5)        // если нажата команда RR
    {
       Image1->Canvas->Pen->Mode=pmCopy;
       DrawRect(round(CP.x),round(CP.y),X,Y);
       if(CP.x!=xold || CP.y!=yold)
         {
           sprintf(Cstr,"RR%d,%d;",round((X-CP.x)/zoom_k),
                                   round((CP.y-Y)/zoom_k));
           Memo1->Lines->Add(Cstr);
         }
    }
  rold=0;       // старый радиус окружности
  xold=CP.x;    // старая вершина прямоугольника
  yold=CP.y;
  applaybtnClick(Sender);

}
```

12.7.9. Пошаговое построение знака

Пошаговое выполнение графических команд может оказаться полезным при уточнении координат подозрительных точек, при проверке описания, отредактированного непосредственно на поле `Memo1`. При переборе команд, расположенных на поле `Memo1`, номер очередной строки графического описания (индекс выполняемой команды) хранится в глобальной переменной `tek_line`.

Щелчок по кнопке `Button1` устанавливает индекс текущей команды в начало описания и генерирует событие `Button2Click`.

Обработчик щелчка по кнопке `Button2` выполняет следующие операции:

1. Из номера текущей строки, если она не является самой первой, вычитается `1` и с помощью метода `Perform` содержимое соответствующей строки `Memo1` подсвечивается. Если подсвеченная строка является первой, то кнопка `Button2` становится недоступной.
2. С помощью функции `applaybtnClick` выполняется отрисовка изображения, начиная с первой строки до строки с номером `tek_line`.

Примерно по такой же схеме производятся действия в обработчике события `Button3Click`:

1. К номеру текущей строки прибавляется `1` и с помощью метода `Perform` содержимое соответствующей строки `Memo1` подсвечивается.
2. С помощью функции `applaybtnClick` выполняется отрисовка изображения, начиная с первой строки до строки с номером `tek_line`.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{// Включение пошагового режима
  tek_line=0;   // индекс текущей команды
  Button2Click(Sender);
}
// ------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ // нажатие на кнопку "<" для просмотра предыдущей команды
  int iline;
  tek_line--;
  if(tek_line<0)
    {
      tek_line=0;
      Button2->Enabled=false;  // самая первая команда
    }
  applaybtnClick(Sender);
  iline=Memo1->Perform(EM_LINEINDEX ,tek_line,0);
  Memo1->SelStart=iline; // выделить текст текущей команды
  Memo1->SelLength=Memo1->Perform(EM_LINEINDEX,tek_line+1,0)-iline;
}
// ---------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{ // нажатие на кнопку ">" просмотр следующей команды
  int iline;
  tek_line++;
```

```
  if(tek_line>=Memo1->Lines->Count)
    {
      tek_line=Memo1->Lines->Count -1;// последняя команда текста
      Button3->Enabled=false;// нельзя перейти к следующей команде
    }
  applaybtnClick(Sender);
  iline=Memo1->Perform(EM_LINEINDEX,tek_line,0);
  Memo1->SelStart=iline; // выделить текст текущей команды
  Memo1->SelLength=Memo1->Perform(EM_LINEINDEX,tek_line+1,0)-iline;;
}
```

12.7.10. Выполнение графических команд

Окончательное формирование графического образа условного знака, описание которого находится на поле `Memo1`, производится функциями `applaybtnClick` и `draw_command`. Первая из них выполняет несложные подготовительные действия с учетом режима пошагового исполнения:

1. С помощью функции `Clear` производится очистка области рисования, и перо переводится в начало координат. Если инициирующим объектом являлась одна из кнопок `Button1`, `Buttton2` или `Button3`, то цикл обработки ограничивается значением номера текущей строки (`tek_line`), в противном случае ограничителем цикла является счетчик строк `Memo1`.
2. Организуется цикл по обрабатываемым строкам `Memo1`, в процессе которого очередная строка выбирается в строку `s`, и управление передается функции `draw_command` для выполнения соответствующей графической операции. Предварительно из строки `s` удаляются незначащие пробелы, пропускаются пустые строки и перо переводится в начальную точку для обрабатываемой строки. Координаты таких точек хранятся в массиве `Lines_CP`.
3. Если вызов инициирован в пошаговом режиме, то производятся соответствующие установки свойства `Enabled` для кнопок `Button2` и `Button3`. В заголовке кнопки `Button1` отображается количество обработанных команд.

```
void __fastcall TForm1::applaybtnClick(TObject *Sender)
{// перерисовка изображения
  int j,k;
  AnsiString s,w;
  Clear(Sender);          // очистить экран
  pm0=false;              // сброс режима полигона
  CP.x=middlx;            // Текущая точка — в центр
```

```
  CP.y=middly;
  Image1->Canvas->MoveTo(CP.x,CP.y);
// Если вызов от функций пошагового просмотра
  if(Sender==Button2 || Sender==Button3 || Sender==Button1)
     k=tek_line;  // число команд, которые надо отобразить
  else
    {
      k=Memo1->Lines->Count-1;
      tek_line=k;       // изобразить все
      Lines_CP[0]=CP;   // CP для нулевой строки
    }
  for(j=0; j<=k; j++)
    {
      s=Memo1->Lines->Strings[j];
      if(s==="")Memo1->Lines->Delete(j);  // удаляем пустые строки
      else
        {// удаляем пробелы перед командой
          while(s[1]==' ' && s.Length()>2) s.Delete(1,1);
          CP=Lines_CP[k];    // Восстанавливаем положение CP
          Image1->Canvas->MoveTo(CP.x,CP.y);
          draw_command(s,j); // на выполнение команды s из строки j
         }
    }
// разрешить/запретить просмотр предыдущей/следующей команды
  if(tek_line>0) Button2->Enabled=true;
  if(tek_line<Memo1->Lines->Count-1) Button3->Enabled=true;
  Button1->Caption=IntToStr(tek_line+1);
}
```

Функция `draw_command` получает в качестве параметров номер обрабатываемой строки `Memo1` и канонический текст соответствующей графической команды. В связи с тем, что каждая команда обладает индивидуальным набором параметров и ее интерпретация на поле рисунка использует специфическую функцию GDI, приходится выполнять довольно много простых и не очень простых действий:

1. Если воспроизводимая команда находится в начале списка `Memo1`, то производится сброс признаков `pm0`, `pm1`, `pm2`, `polyg`. Дальнейшая обработка команды определяется кодом ее операции.
2. Для команды `PM 0;` устанавливаются признаки `pm0`, `polyg` и сбрасываются признаки `pm1` и `pm2`. Координаты текущей точки запоминаются

в массиве точек внешнего контура (contur1). Запоминаются координаты точки, используемой для заливки внешнего контура.

3. Для команды PM 1; сбрасываются признаки pm0 и cont2. Если это необходимо, внешний контур замыкается и заливается красным цветом. Координаты текущей точки запоминаются в массиве contur2.
4. Для команды PM 2; при необходимости замыкается внешний контур и заливается красным цветом. Если в нем находился внутренний полигон, то образуемая им дыра заливается белым цветом. Восстанавливается голубой цвет пера, устанавливается признак pm2 и сбрасываются признаки pm0, pm1, polyg.
5. Для команды PU x,y; извлекаются координаты и корректируется положение текущей точки. Если текущая точка находится во внешнем полигоне, то внешний контур замыкается и сбрасывается признак pm0. Если текущая точка находится во внутреннем полигоне, то он замыкается и заливается белым цветом. Если команда PU находится в последней строке Memo1, то в позиции CP воспроизводится красный крестик.
6. Для команды PD x,y; извлекаются координаты и корректируется положение текущей точки. Если текущая точка находится во внешнем полигоне, то ее координаты заносятся в массив contur1, если она находится во внутреннем полигоне, то — в массив contur2.
7. Для команды CI r; извлекается радиус. Если окружность не принадлежит полигону, то она отображается с помощью функции Circle. Для внешнего полигона окружность воспроизводится с помощью функции Circle1, а точки на ее контуре запоминаются в массиве contur1. Для внутреннего полигона обращение производится также к функции Circle1, но точки контура запоминаются в массиве contur2.
8. Для команды ER dx,dy; извлекаются приращения, с помощью которых формируются координаты четырех вершин прямоугольника. В режиме pm0=true эти координаты запоминаются в массиве contur1, в режиме pm1=true — в массиве contur2. Воспроизводятся границы прямоугольника с помощью метода DrawRect.
9. Для команды RR dx,dy; извлекаются приращения, с помощью которых формируются координаты четырех вершин прямоугольника. Цвет кисти устанавливается по цвету пера и вызывается метод FillRect для заливки прямоугольника. Восстанавливается белый цвет кисти.
10. Для команды AR dx,dy,a; извлекаются параметры дуги окружности и производится обращение к функции duga. Если команда встретилась в режиме полигона, то координаты промежуточных точек на дуге запоминаются в массиве contur1, в противном случае — в рабочем массиве tmp_p.
11. Для команды EW r,a1,a2; по извлеченным параметрам вычисляются координаты начальной точки дуги и вызывается функция duga. В зави-

симости от ситуации координаты промежуточных точек на дуге запоминаются в одном из массивов — `contur1`, `contur2` или `tmp_p`. Координаты текущей точки переводятся в позицию `(xold, yold)`.

12. Для команды `WG r,a1,a2;` по извлеченным параметрам вычисляются координаты начальной точки дуги и вызывается функция `duga` с запоминанием промежуточных точек дуги в рабочем массиве `tmp_p`. Заливка сектора выполняется с помощью метода `Polygon` с предварительной заменой цвета кисти на цвет пера. Затем восстанавливается белый цвет кисти.
13. Для команды `FP;` цвет кисти меняется на голубой и с помощью метода `FloodFill` производится заливка контура, хранящегося в массиве `contur1`. Затем восстанавливаются цвета пера и кисти.
14. Для команды `EP;` анализируется количество точек в массиве `contur1`, и если их достаточно, то вызываются методы `MoveTo` и `LineTo` для воспроизведения контура полигона.
15. Обработка любой графической команды завершается запоминанием положения текущей точки в массиве `Lines_CP`.

```
void __fastcall TForm1::draw_command(AnsiString s,int n)
{// отображение команды s из строки номер n
  int x,y,r1,s_angle,angl,j;
  double x1,y1;
  static int k;
  if(n==0) // если перерисовка начата с нулевой строки
    {
      pm0=false;
      pm1=false;
      pm2=false;
      polyg=false;
    }
  CP=Lines_CP[n];
  sprintf(Cstr,"%-4d,%-4d",round((CP.x-middlx)/zoom_k),
                           round((middly-CP.y)/zoom_k));
  Label1->Caption=Cstr;
  Image1->Canvas->MoveTo(CP.x,CP.y);
  if(s=="PM0;")
    {// команда PM0;
      polyg=true;            // признак начала полигона
      pm0=true;              // признак режима внешнего контура
      pm1=false;             // сбросили признак внутреннего контура
```

```
      pm2=false;            // сбросили признак окончания полигона
      contur1[0]=CP;        // запомнили текущую точку
      nc1=1;                // число точек внешнего контура
      nc2=0;                // число точек внутреннего контура
      cont1=false;          // внешний контур разомкнут
      xc=CP.x;              // запомнили точку на внешнем контуре
      yc=CP.y;              // для перекраски по команде FP
      Image1->Canvas->Pen->Color=clRed; // красное перо
      goto ret;
    }
  if(s=="PM1;")
    {// команда PM1;
      cont2=false;          // внутренний контур разомкнут
      pm0=false;            // сбросили режим внешнего контура
      if(contur1[nc1-1].x!=contur1[0].x ||
         contur1[nc1-1].y!=contur1[0].y)
        {
          contur1[nc1++]=contur1[0];   // замкнули внешний контур
        }
      nc1--;
      RedFill(contur1,nc1); // и залили его красным
      cont1=true;           // признак — замкнули внешний
      pm1=true;             // признак режима внутреннего контура
      nc2=1;                // число точек внутреннего контура
      contur2[0]=CP;        // запомнили текущую точку
      goto ret;
    }
  if(s=="PM2;")
    {// команда PM2;
      if(!cont1)   // если внешний контур не завершен
        {
          if(contur1[nc1-1].x!=contur1[0].x ||
             contur1[nc1-1].y!=contur1[0].y)
             contur1[nc1++]=contur1[0];   // замкнули наружный контур
          nc1--;
          RedFill(contur1,nc1); // и залили его красным
          goto m3;
        }
      if(!cont2)
```

```
        {
          if(contur2[nc2-1].x!=contur2[0].x ||
             contur2[nc1-1].y!=contur2[0].y)
             contur2[nc2++]=contur2[0];   // замкнули внутренний контур
          nc2--;
          WhiteFill(contur2,nc2);// сделали белую дырку
        }
m3:
      Image1->Canvas->Pen->Color=clBlue; // восстановили цвет пера
      pm2=true;               // признак наличия полигона
      pm0=false;              // сбросили признак внешнего контура
      pm1=false;              // сбросили признак внутреннего контура
      polyg=false;
      goto ret;
    }
  if(s[1]=='P' && s[2]=='U')
    {// команда PU
      sscanf(&s[3],"%d,%d",&x,&y);
      CP.x+=x*zoom_k;
      CP.y-=y*zoom_k;
      if(pm0)  // если эта команда встретилась в режиме полигона
        {
          pm0=false;                // закрыли внешний полигон
          contur1[nc1++]=contur1[0];// замкнули его, на всякий случай
        }
      if(pm1)  // если мы находимся в режиме подполигона
        {
          if(nc2==1)
            {
               contur2[0]=CP;
               goto ret;
            }
          contur2[nc2++]=beg_pol_pt;// замкнули внутренний подполигон
          nc2--;
          WhiteFill(contur2,nc2);
// запомнили начало следующего подполигона
          beg_pol_pt=CP;
          goto ret;
        }
```

```
      if(n==Memo1->Lines->Count-1)
        {// Нарис. крестик, если PU последняя видимая команда
          Image1->Canvas->Pen->Color=clRed;
          Image1->Canvas->MoveTo(CP.x+5,CP.y);
          Image1->Canvas->LineTo(CP.x-5,CP.y);
          Image1->Canvas->MoveTo(CP.x,CP.y+5);
          Image1->Canvas->LineTo(CP.x,CP.y-5);
          Image1->Canvas->Pen->Color=clBlue;
        }
      Image1->Canvas->MoveTo(CP.x,CP.y);
      xold=CP.x;
      yold=CP.y;
      goto ret;
    }
  if(s[1]=='P' && s[2]=='D')  // команда PD
    {
      if(!polyg) Image1->Canvas->Pen->Width=5;
      sscanf(&s[3],"%d,%d;",&x,&y); // вынули из команды координаты
      CP.x+=x*zoom_k;               // переместили текущую точку
      CP.y-=y*zoom_k;
      Image1->Canvas->LineTo(CP.x,CP.y);
      if(pm0)
        {
          contur1[nc1]=CP;  // Запомнили точку внешнего полигона
          nc1++;
        }
      if(pm1)
        {
          nc2++;
          contur2[nc2]=CP;  // Запомнили точку подполигона
        }
      goto ret;
    }
  if(s[1]=='C' && s[2]=='I')    // команда CI
    {
      if(!polyg) Image1->Canvas->Pen->Width=5;
      sscanf(&s[3],"%d;",&r1);  // радиус
      if(!polyg)                // если не в режиме полигона
        {
```

```
        Circle(r1*zoom_k);    // нарисовать окружность
        goto ret;
      }
    if(pm0)                   // если в режиме внешнего полигона
      {
        nc1=0;
        Circle1(r1*zoom_k,contur1,nc1);
        nc1--;
        RedFill(contur1,nc1);
        xc=contur1[0].x;
        yc=contur1[0].y;
        pm0=false;
        goto ret;
      }
    if(pm1)   // в режиме подполигона
      {
        nc2=0;            // удалили переход в центр
        Circle1(r1*zoom_k,contur2,nc2);
        nc2--;
        WhiteFill(contur2,nc2);
        goto ret;
      }
  }
if(s[1]=='E' && s[2]=='R')    // комада ER
  {
    if(!polyg) Image1->Canvas->Pen->Width=5;
    sscanf(&s[3],"%d,%d",&x,&y);
    if(pm0)
      {
        nc1=3;
        contur1[0]=CP;
        contur1[1].x=CP.x;
        contur1[1].y=CP.y-y*zoom_k;
        contur1[2].x=CP.x+x*zoom_k;
        contur1[2].y=contur1[1].y;
        contur1[3].x=contur1[2].x;
        contur1[3].y=CP.y;
        RedFill(contur1,nc1);
        nc1=4;
```

```
          goto ret;
        }
      if(pm1)
        {
          nc2=3;
          contur2[0]=CP;
          contur2[1].x=CP.x;
          contur2[1].y=CP.y-y*zoom_k;
          contur2[2].x=CP.x+x*zoom_k;
          contur2[2].y=contur2[1].y;
          contur2[3].x=contur2[2].x;
          contur2[3].y=CP.y;
          WhiteFill(contur2,nc2);
          goto ret;
        }
      DrawRect(CP.x,CP.y,CP.x+x*zoom_k,CP.y-y*zoom_k);
      goto ret;
    }
  if(s[1]=='R' && s[2]=='R')    // комада RR
    {
      Image1->Canvas->Pen->Width=1;
      sscanf(&s[3],"%d,%d",&x,&y);
      Image1->Canvas->Brush->Color=Image1->Canvas->Pen->Color;
      Image1->Canvas->FillRect((Rect)(CP.x,CP.y,CP.x+x*zoom_k,
                                      CP.y-y*zoom_k));
      Image1->Canvas->Brush->Color=clWhite;
      goto ret;
    }
  if(s[1]=='A' && s[2]=='R')    // команда AR
    {
      if(!polyg) Image1->Canvas->Pen->Width=5;
      sscanf(&s[3],"%d,%d,%d;",&x,&y,&angl);
      if(pm0)
        {
          duga(x,y,angl,contur1,nc1);
          goto ret;
        }
// построение дуги вне полигона
      tmp_p[0]=CP;
```

```
      k=1;
      duga(x,y,angl,tmp_p,k);
      goto ret;
    }
  if(s[1]=='E' && s[2]=='W')    // команда EW
    {// нарисовать сектор
      if(!polyg) Image1->Canvas->Pen->Width=5;
      sscanf(&s[3],"%d,%d,%d;",&r1,&s_angle,&angl);
      x1=round(CP.x+r1*cos(s_angle*M_PI/180)*zoom_k);
      y1=round(CP.y-r1*sin(s_angle*M_PI/180)*zoom_k);
      xold=CP.x;
      yold=CP.y;
      Image1->Canvas->LineTo(x1,y1);
      CP.x=x1;
      CP.y=y1;
      if(pm0) // сектор в режиме внешнего полигона образует контур
        {
          contur1[0].x=xold;
          contur1[0].y=yold;
          contur1[1]=CP;
          nc1=2;
          duga((-x1+xold)/zoom_k,(y1-yold)/zoom_k,angl,contur1,nc1);
          nc1++;
          contur1[nc1].x=xold;
          contur1[nc1].y=yold;
          pm0=false;
        }
// если сектор в режиме внутреннего полигона образует дыру
      else if(pm1)
        {
          duga((-x1+xold)/zoom_k,(y1-yold)/zoom_k,angl,contur2,nc2);
        }
      else  // сектор не в режиме полигона
        {
          k=0;
          duga((-x1+xold)/zoom_k,(y1-yold)/zoom_k,angl,tmp_p,k);
        }
      Image1->Canvas->LineTo(xold,yold);
      CP.x=xold;
```

```
      CP.y=yold;
      goto ret;
    }
  if(s[1]=='W' && s[2]=='G')    // команда WG
    {// нарисовать залитый сектор
      Image1->Canvas->Pen->Width=1;
      sscanf(&s[3],"%d,%d,%d;",&r1,&s_angle,&angl);
      x1=round(CP.x+r1*cos(s_angle*M_PI/180)*zoom_k);
      y1=round(CP.y-r1*sin(s_angle*M_PI/180)*zoom_k);
      xold=CP.x;
      yold=CP.y;
      tmp_p[0]=CP;
      tmp_p[1].x=x1;
      tmp_p[1].y=y1;
      k=2;
      CP.x=x1;
      CP.y=y1;
      duga((-x1+xold)/zoom_k,(y1-yold)/zoom_k,angl,tmp_p,k);
      CP.x=xold;
      CP.y=yold;
      Image1->Canvas->Brush->Color=Image1->Canvas->Pen->Color;
      Image1->Canvas->Polygon(tmp_p,k);
      Image1->Canvas->Brush->Color=clWhite;
      goto ret;
    }
  if(s[1]=='F' && s[2]=='P')    // команда FP
    {// залить полигон
      Image1->Canvas->Brush->Color=clBlue;
      Image1->Canvas->FloodFill(xc,yc,clRed,fsSurface);
      Image1->Canvas->Brush->Color=clWhite;
      Image1->Canvas->Pen->Color=clBlue;
      goto ret;
    }
  if(s[1]=='E' && s[2]=='P')  // команда EP;
    {// обвести контур полигона
      if(nc1<2)
        {
          ShowMessage("В буфере полигона мало точек");
          goto ret;
```

```
        }
      Image1->Canvas->MoveTo(contur1[0].x,contur1[0].y);
      for(j=1; j<nc1; j++)
         Image1->Canvas->LineTo(contur1[j].x,contur1[j].y);

ret:  Lines_CP[n+1]=CP;
      Image1->Canvas->Pen->Width=1;
      return;
    }
}
```

В процессе воспроизведения графического образа условного знака используется несколько вспомогательных программ, обеспечивающих заливку сплошным цветом замкнутых областей (`RedFill`, `BlueFill`, `WhiteFill`), построение дуг (`duga`), окружностей (`Circle`, `Circle1`) и прямоугольников (`DrawRect`). В отличие от близких по смыслу функций GDI они обладают определенной спецификой, т. к. результаты их работы должны быть одинаково пригодны как для отображения на дисплее, так и при выводе графической программы на плоттер. Поэтому, например, при построении залитых окружностей и секторов, расположенных внутри полигонов, приходится аппроксимировать контуры ломаными линиями. Кроме того, при пошаговом выполнении графической программы желательно выделять построение фрагментов полигона другим цветом, а в окончательном рисунке восстанавливать первоначальный синий цвет. Он не является цветом, которым знак будет отображаться в соответствующих приложениях, т. к. перед выводом знака можно установить любое перо.

Ниже приводятся тексты вспомогательных графических процедур и обсуждаются некоторые датали их реализации.

Функция `duga` предназначена для построения дуги и запоминания координат точек, расположенных на дуге окружности с шагом `5°`, в массиве `contur`. Вторым выходным параметром является ссылка на переменную `k`, в которую засылается количество найденных точек. Для повышения быстродействия здесь использовано стандартное уравнение окружности (`x=x0+R*cos`φ, `y=y0+R*sin`φ). Однако вместо многократного вычисления стандартных функций используются формулы для `sin(`φ+Δφ`)` и `cos(`φ+Δφ`)`. Так как приращение аргумента постоянно, то в текст программы включены табличные значения `sin5°` и `cos5°`. Правда, последнее приращение может оказаться и меньше `5°`, поэтому один раз придется обратиться к вычислению стандартных функций. Для построения контура используется метод `LineTo`, т. к. считается, что пишущий узел уже приведен в начало дуги. Небольшая тонкость связана с тем, что центральный угол может оказаться отрицательным, и в формулах для синуса и косинуса суммы углов придется сменить один знак.

```
void __fastcall TForm1::duga(int dx,int dy,int aa,
                             TPoint *contur,int &k )
{// Вычисление точек дуги с шагом 5 градусов и построение дуги
 // dx,dy — смещение центра от текущей точки (CP.x,CP.y)
 // aa — центральный угол в градусах
   double r,xc,yc,sinfi,cosfi,dfi=0,si,co,x,y,codfi,sidfi;
   double cos5=0.996195, sin5=0.087156; // для 5 градусов
   int da=5; // шаг по углу
   r=round(sqrt(dx*dx+dy*dy));
   sinfi=-dy/r; // синус начального угла
   cosfi=-dx/r; // косинус начального угла
   xc=CP.x+dx*zoom_k;   // координаты центра
   yc=CP.y-dy*zoom_k;
   if(aa<0)sin5=-sin5;
m1:
   dfi+=((aa>0)?da:-da);
   if(aa>0 && dfi<=aa || aa<0 && dfi>=aa)
     {
       si=sinfi*cos5+cosfi*sin5;
       co=cosfi*cos5-sinfi*sin5;
       sinfi=si;       cosfi=co;
       x=xc+r*cosfi*zoom_k;
       y=yc-r*sinfi*zoom_k;
       CP.x=round(x); // округление координат текущей точки
       CP.y=round(y);
       Image1->Canvas->LineTo(CP.x,CP.y);
       contur[k].x=CP.x; // Запомнили очередную точку дуги
       contur[k].y=CP.y;
       if(dfi==aa) return;
       k++;
       goto m1;
     }
    else
     { if(aa>0) dfi=dfi-aa;   // последний уголок
           else dfi=aa-dfi;
       codfi=cos(dfi*M_PI/180);
       sidfi=sin(dfi*M_PI/180);
       si=sinfi*codfi+cosfi*sidfi;
       co=cosfi*codfi-sinfi*sidfi;
```

```
        sinfi=si;       cosfi=co;
        x=xc+r*cosfi*zoom_k;  // координаты последней точки
        y=yc-r*sinfi*zoom_k;
        CP.x=round(x);
        CP.y=round(y);
        Image1->Canvas->LineTo(CP.x,CP.y);
        contur[k].x=CP.x; // Запомнили последнюю точку дуги
        contur[k].y=CP.y;
      }
}
```

Функция `Circle` предназначена для построения окружности заданного радиуса с центром в текущей точке. Она является тривиальным посредником, вызывающим метод `Arc`, и единственно полезное действие заключается в формировании нужного набора аргументов:

```
void __fastcall TForm1::Circle(int r)
{// Построение окружности радиуса r с центром в текущей точке
    Image1->Canvas->Arc(CP.x-r,CP.y+r,CP.x+r,CP.y-r,
                        CP.x-r,CP.y+r,CP.x-r,CP.y+r);
}
```

Функция `Circle1` предназначена для построения окружности с одновременным запоминанием координат точек, расположенных на окружности с шагом `5°`. Алгоритм ее работы похож на вычислительную схему функции `duga`, но он гораздо проще, т. к. количество шагов заранее известно и все они кратны `5°`:

```
void __fastcall TForm1::Circle1(int r,TPoint *contur,int &nc)
{// Построение и запоминание точек окружности в режиме полигона
  double sinfi=0,cosfi=1,si,co;
  double cos5=0.996195, sin5=0.087156;
  int j;
  Image1->Canvas->MoveTo(round(CP.x+r),round(CP.y));
  for(j=0; j<73; j++,nc++)
    {
      contur[nc].x=round(CP.x+r*cosfi);
      contur[nc].y=round(CP.y-r*sinfi);
      Image1->Canvas->LineTo(contur[nc].x,contur[nc].y);
      si=sinfi*cos5+cosfi*sin5;
      co=cosfi*cos5-sinfi*sin5;
      sinfi=si;       cosfi=co;
    }
}
```

Функция `DrawRect` строит прямоугольник, заданный координатами противоположных вершин:

```
void __fastcall TForm1::DrawRect(int left,int top,int right,int bottom)
{// Построение прямоугольника с заданными координатами
     Image1->Canvas->MoveTo(left,top);
     Image1->Canvas->LineTo(right,top);
     Image1->Canvas->LineTo(right,bottom);
     Image1->Canvas->LineTo(left,bottom);
     Image1->Canvas->LineTo(left,top);
}
```

Функции `RedFill`, `WhiteFill` и `BlueFill` построены по единообразной схеме. Им передается массив координат точек, образующих замкнутый контур, и его размерность. Каждая из процедур устанавливает требуемый цвет кисти и вызывает метод `Polygon`. После заливки заданной области текущая точка (`CP`) перемещается в начало контура и восстанавливается белый цвет кисти.

```
void __fastcall TForm1::RedFill(TPoint *contur,int &n)
{// заливка контура красным цветом
  Image1->Canvas->Brush->Color=clRed;
  Image1->Canvas->Polygon(contur,n);
  contur[0]=CP;
  n=1;
  Image1->Canvas->Brush->Color=clWhite;
}
// ---------------------------------------------------------------
void __fastcall TForm1::WhiteFill(TPoint *contur,int &n)
{// заливка контура белым цветом
  Image1->Canvas->Brush->Color=clWhite;
  Image1->Canvas->Polygon(contur,n);
  contur[0]=CP;
  n=1;
}
// ---------------------------------------------------------------
void __fastcall TForm1::BlueFill(TPoint *contur,int &n)
{// заливка контура синим цветом
  Image1->Canvas->Brush->Color=clBlue;
  Image1->Canvas->Polygon(contur,n);
  contur[0]=CP;
  n=1;
  Image1->Canvas->Brush->Color=clWhite;
}
```

12.7.11. Разное

В этом параграфе собраны функции, которые не удалось причислить к предыдущим разделам. Их список приведен в табл. 12.4.

Таблица 12.4. *Дополнительные функции редактора знаков*

Наименование функции	Назначение
`AboutClick`	Реакция на команду **О программе**
`AppOnMessage`	Реакция на некоторые сообщения Windows
`FormResize`	Реакция на изменение размера формы
`max`	Определение максимального значения
`round`	Округление вещественного числа
`ScrollBox1Click`	Перерисовка изображения

Функция `AboutClick` выводит на экран небольшое окно рекламного характера и состоит из единственной строчки, отображающей это окно в модальном режиме:

```
void __fastcall TForm1::AboutClick(TObject *Sender)
{// Команда "О программе"
  AboutBox->ShowModal();
}
```

Функция `AppOnMessage` перехватывает сообщения Windows для выделения из них некоторых сочетаний кодов клавиш клавиатуры, управляющих точной навигацией курсора мыши на рабочем поле. К таким сочетаниями относятся комбинации клавиши <Alt> и стрелок управления курсором. Появление каждой такой комбинации приводит к перемещению курсора мыши на `1` пиксел в соответствующем направлении.

В этой функции присутствуют некоторые детали обработки сообщений Windows. Во-первых, выясняется, связано ли это сообщение с нажатием одной из системных клавиш (вообще говоря, нас интересует только одна из них — клавиша <Alt>). Если поступившее сообщение (значение поля `Msg.message`) не относится к разряду `WM_SYSKEYDOWN`, то функция прекращает свою работу и заносит значение `false` в аргумент `Handled`. В результате это сообщение продолжит поиск настоящего адресата и его обработка продолжится.

Если системная клавиша все-таки была нажата, то начинается содержательная разборка текста сообщения, находящегося в поле `Msg.wParam`. По коду первого символа этого поля организуется переключатель, выясняющий в каком направлении надо двигать курсор мыши и не пора ли завершить попиксельное перемещение. Довольно большое количество проверок обес-

печивает пребывание курсора в рабочем поле. Наткнувшись на соответствующую границу, курсор в этом направлении больше не смещается.

Отработав поступившее сообщение, функция присваивает параметру `Handled` значение `true`, что означает нормальное завершение обработки события. После этого система не будет продолжать поиски другого адресата.

```
void __fastcall TForm1::AppOnMessage(tagMSG &Msg, bool &Handled)
{// процедура реакции на все сообщения, посылаемые программе,
 // отлавливает нажатие клавиш Alt+[left,right,up,down,enter],
 // осуществляет точную навигацию курсора мыши по полю рисунка
 // при нажатии сочетания указанных клавиш
  TPoint pt,pp;
  TShiftState s;
  if(!(Msg.message == WM_SYSKEYDOWN))
    {// Если сообщение не связано с системной клавишей
      Handled = false;
      return;
    }
// Запрос абсолютных координат курсора мыши
  GetCursorPos(&pp);
// Перевод в относительные координаты рисунка
  pt=ScrollBox1->ScreenToClient(pp);
  if(pt.x>=0 && pt.y>=0   // если курсор мыши на рисунке
     && pt.x <= ScrollBox1->Width
     && pt.y <= ScrollBox1->Height)
     {switch (Msg.wParam)
       {case 13:  // Alt + Enter
         {done=true;  // эмуляция нажатия кнопки мыши
// пересчитываем координаты относительно Image1 :
          pp=Image1->ScreenToClient(pp);
// моделируем отпускание кнопки мыши на рисунке
          // Image1MouseUp(Image1,btn,s,pp.x,pp.y);
          break;
         }
        case 37:  // left
         {pt.x--;    // пересчет координат в окне
          if(pt.x < 0 )
            {pt.x=0;
             if(Image1->Left < 0)
                ScrollBox1->HorzScrollBar->Position--;
```

```
            }
           Handled=true; // событие обработано
           done=true;    // эмуляция нажатия кнопки мыши
           break;
          }
         case 38:     // up
          {pt.y--;         // пересчет координат в окне
           if(pt.y < 0)
             {pt.y=0;
              if(Image1->Top < 0)
                 ScrollBox1->VertScrollBar->Position++;
             }
           Handled=true; // событие обработано
           done=true;    // эмуляция нажатия кнопки мыши
           break;
          }
         case 39:     // right
          {pt.x++;         // пересчет координат в окне
           if(pt.x>ScrollBox1->ClientWidth-1)
             {pt.x=ScrollBox1->ClientWidth-1;
              ScrollBox1->HorzScrollBar->Position++;
             }
            Handled=true;// событие обработано
            done=true;   // эмуляция нажатия кнопки мыши
            break;
          }
         case 40:     // down}
          {pt.y++;         // пересчет координат в окне
           if(pt.y>ScrollBox1->ClientHeight-1)
             {pt.y=ScrollBox1->ClientHeight-1;
              ScrollBox1->VertScrollBar->Position++;
             }
           Handled=true; // событие обработано
           done=true;    // эмуляция нажатия кнопки мыши
           break;
          }
         } // конец переключателя
// пересчет новых координат мыши в абсолютные
   pt=ScrollBox1->ClientToScreen(pt);
```

```
// установка курсора в новую позицию
   SetCursorPos(pt.x,pt.y);
  }
}
```

Функция `FormResize` позволяет произвести перекомпоновку некоторых свойств либо в случае действительного изменения размеров или положения формы, либо в случае программного изменения размеров каких-либо элементов (например, габаритов рабочего поля). Каких-либо других радикальных действий она не производит:

```
void __fastcall TForm1::FormResize(TObject *Sender)
{// обработчик события — изменение размеров формы
  Bevel6->Width=Memo1->Width-2;
  Button1->Width=Memo1->Width/3;
  Button1->Left=Bevel6->Left+2;
  Button2->Left=Button1->Left+Button1->Width+1;
  Button3->Left=Button2->Left+Button1->Width;
  applaybtn->Left=Memo1->Left+1;
  applaybtn->Top=Memo1->Top+Memo1->Height+1;
  applaybtn->Width=Memo1->Width;
  Label2->Width=Panel3->Width;
  if(imagewidth*zoom_k < ScrollBox1->Width-2)
     Image1->Width=ScrollBox1->Width-2;
  else
     Image1->Width=imagewidth*zoom_k;
  if(imageheight*zoom_k < ScrollBox1->Height-2)
     Image1->Height=ScrollBox1->Height-2;
  else
     Image1->Height=imageheight*zoom_k;
  if(bit != NULL)
    {
      bit->Width=Image1->Width;
      bit->Height=Image1->Height;
      Image1->Picture->Graphic=bit;
    }
  middlx=grid_step*zoom_k*round(Image1->Width/(grid_step*2*zoom_k));
  middly = grid_step * zoom_k * round(Image1->Height /
                                          (grid_step*2*zoom_k));
  CP.x=middlx;
  CP.y=middly;
```

```
  applaybtnClick(Sender);
}
```

Функция `max` ничем особенно не примечательна, но она возвращает максимальное значение одного из своих целочисленных аргументов в формате `double`:

```
double __fastcall max(int a,int b)
{
  double res;
  res=a;
  if(b>a)res=b;
  return res;
}
```

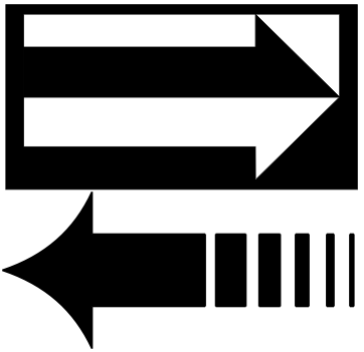
Функция `round` примечательна лишь тем, что она восполняет недостаток системной библиотеки Си удобным средством Паскаля:

```
int __fastcall TForm1::round(double x)
{// Округление вещественного числа
  int res;
  res=(x<0)? x-.5:x+.5;
  return res;
}
```

Функция `ScrollBox1Click` включена в состав редактора на вский случай. Для прокрутки большой картинки объект `Image` расположен на компоненте `ScrollBox1`. Щелчок по окну прокрутки повторно перерисовывает изображение, хотя в редакторе имеется и специальная кнопка, выполняющая аналогичное действие.

```
void __fastcall TForm1::ScrollBox1Click(TObject *Sender)
{ // обработка событий ScroolBox — нажатие мыши
  applaybtnClick(Sender);
}
```

Приложение 1



Глоссарий (компоненты и объекты визуальных сред)

Form — главное окно Windows-приложения.

Visual Basic

CheckBox — окно логического признака (помечен, не помечен).

CommandButton — командная кнопка.

ComboBox — комбинированное окно для отображения списка строк и поля ввода.

CommonDialog — универсальное окно для организации диалога по выбору файла, цвета, шрифта, параметров печати.

Data — посредник для доступа к базе данных.

DBGrid — окно для отображения таблицы базы данных.

Debug — окно для выполнения отладочных операций.

DirListBox — окно для отображения списка каталогов.

DriveListBox — окно для выбора логического диска.

DTPicker — часы или календарь.

FileListBox — окно для отображения списка файлов.

Frame — рамка-контейнер объектов.

HScrollBar — линейка горизонтальной прокрутки.

Image — окно для графических построений и отображения рисунков.

InputBox — окно для ввода текстового сообщения.

Label — метка (поле для отображения строк).

Line — объект для формирования отрезка прямой.

ListBox — окно для отображения списка строк.

MaskEdBox — поле редактирования и ввода строк по маске.

MonthView — календарь на месяц.

MsgBox — окно для вывода сообщений.

MSFlexGrid — универсальная таблица.

OptionButton — радиокнопка (переключатель).

PictureBox — окно для отображения рисунка.

RichTextBox — редактор RTF-текста.

Shape — объект для формирования графических фигур.

TextBox — окно для ввода и отображения одной или нескольких строк.

Timer — таймер.

UpDown — кнопка для изменения числового значения по закону арифметической прогрессии.

VScrollBar — вертикальная линейка прокрутки.

Borland C++ Builder, Delphi

BitBtn — кнопка с растровым рисунком.

Bitmap — массив в оперативной памяти для хранения растрового изображения.

Brush — кисть (средство для изменения фона изображения).

Button — стандартная кнопка.

Canvas — поверхность для графических построений (канва, холст).

CheckBox — окно логического признака (помечен, не помечен).

Clipboard — буфер обмена.

ComboBox — комбинированное окно для отображения списка строк и поля ввода.

ColorDialog — диалоговое окно для выбора цвета.

DataSource — посредник для связи с базой данных.

DateTimePicker — часы или календарь.

DBEdit — редактируемое текстовое поле для отображения и модификации соответствующего поля в таблице базы данных.

DBGrid — окно для отображения и модификации таблицы базы данных.

DBImage — окно для отображения графического поля в таблице базы данных.

DBMemo — окно для отображения многострокового поля в таблице базы данных.

DBNavigator — навигатор по записям базы данных.

DirectoryListBox — окно для отображения списка каталогов.

DriveComboBox — окно для выбора логического диска.

Edit — поле для отображения и редактирования строки.

Frame — рамка (контейнер объектов).

FileListBox — окно для отображения списка файлов.

FindDialog — диалоговое окно для организации поиска фрагмента текста в RTF-документе.

FontDialog — диалоговое окно для выбора шрифта и установки его параметров.

GroupBox — контейнер для группы компонент.

Icon — поле пиктограммы, иконы.

Image — окно для графических построений и отображения рисунков.

InputBox — диалоговое окно для ввода строки.

InputQuery — диалоговое окно для ввода строки.

Label — метка (поле для отображения строк).

ListBox — окно для отображения списка строк.

MainMenu — объект для формирования главного меню.

MaskEdit — поле редактирования и ввода строк по маске.

Memo — окно для отображения многострокового текстового документа.

MessageBox — диалоговое окно для вывода сообщений.

MessageDlg — диалоговое окно для вывода сообщений.

MessageDlgPos — диалоговое окно для вывода сообщений.

Metafile — поле для хранения графического файла в формате wmf или emf.

MonthCalendar — календарь на месяц.

OpenDialog — диалоговое окно для выбора файла.

OpenPictureDialog — диалоговое окно для выбора графического файла.

PaintBox — окно для графических построений.

Panel — панель (объект, используемый в качестве подложки).

Pen — перо (пишущий узел для графических построений).

Picture — окно для отображения рисунка.

PopupMenu — объект для формирования всплывающего меню.

PrintDialog — диалоговое окно для вывода на печать.

PrinterSetupDialog — диалоговое окно для настройки принтера.

Query — посредник для SQL-общения с базой данных.

RadioButton — радиокнопка.

RadioGroup — рамка для радиокнопок.

ReplaceDialog — диалоговое окно для организации поиска и замены текстового фрагмента в RTF-документе.

RichEdit — редактор RTF-текста.

SaveDialog — диалоговое окно для запоминания файла.

SavePictureDialog — диалоговое окно для запоминания графического файла.

Screen — экран операционной системы.

ScrollBar — линейки горизонтальной и вертикальной прокрутки.

ScrollBox — окно с линейками прокрутки.

ShowMessage — простейшее окно для вывода сообщений.

SpeedButton — быстрая кнопка.

StatusBar — строка состояния.

StringGrid — таблица для отображения символьной информации.

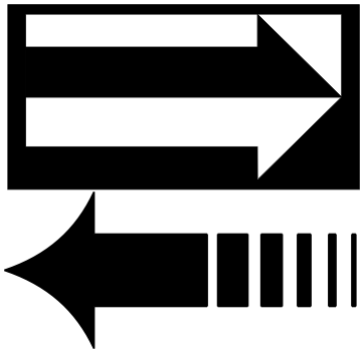
Table — посредник для доступа к базе данных.

Timer — таймер.

UpDown — кнопка для изменения числового значения по закону арифметической прогрессии.

Приложение 2

Описание дискеты



Все иллюстративные материалы, содержащиеся в книге, построены либо в процессе проектирования сравнительно небольших приложений, либо во время их выполнения. В тексте книги для них использована сплошная нумерация в пределах каждой главы:

- 2.1(VB) — первое приложение в главе 2 для системы Visual Basic;
- 2.2(BCB) — второе приложение в главе 2 для системы Borland C++ Builder;
- 2.3(Delphi) — третье приложение в главе 2 для системы Delphi;
- 3.1(VB) — первое приложение в главе 3 для системы Visual Basic;
- 12.1(BCB) — первое приложение в главе 12 для системы Borland C++ Builder.

Приложения распределены по главам, для которых выделены каталоги GL.02, GL.03, ..., GL.12. В каждом из них сформированы подкаталоги BCB, Delphi и VB, внутри которых созданы папки, имена которых совпадают с номерами приложений. Например:

Приложение 2.1(VB)

находится по адресу GL.02\VB\2.01. В каталоге GL.04 добавлен подкаталог MOREICON, содержащий большое количество икон, отображаемых соответствующими приложениями.

Полный перечень приложений с разбиением по главам приведен в табл. П2.1.

Таблица П2.1. *Перечень приложений с разбиением по главам*

Visual Basic	Borland C++ Builder	Delphi
Глава 2		
2.01, 2.04	2.03, 2.05	2.02, 2.06

Таблица П2.1 *(окончание)*

Visual Basic	Borland C++ Builder	Delphi
Глава 3		
3.01, 3.02, 3.03, 3.04, 3.05, 3.06, 3.15	3.07, 3.09, 3.11, 3.13, 3.16	3.08, 3.10, 3.12, 3.14, 3.17
Глава 4		
4.01, 4.02, 4.03, 4.07, 4.09, 4.13	4.04, 4.08, 4.10, 4.14, 4.15, 4.16, 4.17, 4.18	4.05, 4.06, 4.11, 4.12
Глава 5		
5.01, 5.06, 5.08, 5.09, 5.12	5.02, 5.03, 5.04,5.05, 5.10, 5.13, 5.15	5.07, 5.11, 5.14
Глава 6		
6.01, 6.02, 6.04, 6.05, 6.06, 6.09, 6.10	6.03, 6.08, 6.11, 6.13, 6.15, 6.16, 6.17	6.07, 6.12, 6.14, 6.18
Глава 7		
—	—	7.01, 7.02
Глава 8		
8.01, 8.06, 8.11, 8.12	8.04, 8.07, 8.08, 8.09, 8.13, 8.14, 8.15, 8.16	8.02, 8.03, 8.05, 8.10, 8.17
Глава 9		
9.01, 9.05, 9.06	9.03	9.02, 9.04, 9.07
Глава 10		
10.08, 10.09, 10.10, 10.11, 10.12	10.01, 10.07	10.02, 10.03, 10.04, 10.05, 10.06
Глава 11		
11.01, 11.02	–	—
Глава 1		
—	12.01	—

Список литературы

По системе Visual Basic

1. Ананьев А. И., Федоров А. Ф. Самоучитель Visual Basic 6.0. — СПб.: БХВ-Петербург, 2002. — 624 с.
2. Арушанов Х. Р. Visual Basic 4.0. — М.: ABF, 1995. — 368 с.
3. Браун К. Введение в Visual Basic для программистов. — М.: Мир, 1993. — 416 с.
4. Браун С. Visual Basic 5 с самого начала. — СПб.: Питер, 1998. — 320 с.
5. Браун С. Visual Basic 6: Учебный курс. — СПб.: Питер, 2001. — 576 с.
6. Витенко О., Яковлев В. Visual Basic в бюро. — Киев: BHV, 1997. — 334 с.
7. Гарнаев А. Ю. Visual Basic 6.0: Разработка приложений. — СПб.: БХВ-Петербург, 2001. — 448 с.
8. Карпов Б. Visual Basic 6: Специальный справочник. — СПб.: Питер, 2002. — 416 с.
9. Крейг Д. К., Уэбб Д. Microsoft Visual Basic 6.0. Мастерская разработчика. — М.: изд.-торговый дом "Русская Редакция", 2001. — 720 с.
10. Макашарипов С. Программирование баз данных на Visual Basic 5 в примерах. — СПб.: Питер, 1997. — 256 с.
11. Назаров С. В., Мельников П.П. Программирование на MS Visual Basic: Учебное пособие. — М.: Финансы и статистика, 2000. — 320 с.
12. Райтингер М., Муч Г. Visual Basic 6.0. — Киев: Издательская группа BHV, 2000. — 288 с.
13. Сайлер Б., Споттс Д. Специальное использование Visual Basic 6.0. — М.; СПб.: Изд. дом "Вильямс", 2001. — 832 с.
14. Санна П. и др. Visual Basic для приложений (версия 5) в подлиннике. — СПб.: BHV, 1997. — 704 с.
15. Скотт Р. П. Учимся мастерству Microsoft Visual Basic 4.0. — М.: СК Пресс, 1997. — 352 с.
16. Шмидт В. Microsoft Visual Basic 5.0. — М.: ABF, 1997. — 687 с.

17. Microsoft Visual Basic 6.0 для профессионалов. Шаг за шагом: Практическое пособие. — М.: ЭКОМ, 2001. — 720 с.
18. Visual Basic 6.0: Пер. с англ. — СПб.: БХВ-Петербург, 2002. — 992 с.

По системе Borland C++ Builder

1. Архангельский А. Я. Программирование в C++ Builder 4. — М.: БИНОМ, 2000. — 1088 с.
2. Бобровский С. И. Самоучитель программирования на языке C++ в системе Borland C++ Builder 4.0. — М.: ДЕСС INFORCOM PRESS, 1999. — 286 с.
3. Вальвачев А., Сурков Д., Сурков К. Программирование в среде C++ Builder 3.0. — Минск: Попурри, 1998. — 576 с.
4. Калверт Ч., Рейсдорф К. Borland C++ Builder 5: Энциклопедия пользователя. — Киев: ДиаСофт, 2001. — 994 с.
5. Рейсдорф К., Хендерсон К. Borland C++ Builder. Освой самостоятельно. — М.: БИНОМ, 1998. — 704 с.
6. Теллес М. Borland C++ Builder: библиотека программиста. — СПб.: Питер Ком, 1998. — 512 с.
7. Холингвэрт Д., Баттерфилд Д., Сворт Б., Олсоп Д. и др. C++ Builder. Руководство разработчика: В 2-х томах. — М.: изд. дом "Вильямс", 2001. Том 1. Основы. — 880 с. Том 2. Сложные вопросы программирования. — 832 с.
8. Шамис В. А. C++ Builder 4. Техника визуального программирования. — М.: Нолидж, 2000. — 656 с.

По системе Delphi

1. Александровский А. Д. Delphi 5. Разработка корпоративных приложений. — М.: ДМК, 2000. — 512 с.
2. Архангельский А. Я. Программирование в Delphi. — 2-е изд. — М.: БИНОМ, 2000. — 1072 с.
3. Архангельский А. Я. Delphi 6: Справочное руководство. — М.: БИНОМ, 2001. — 1024 с.
4. Баас Р., Фернай М. Delphi 4: Полное руководство. — Киев: BHV, 1998. — 800 с.
5. Бобровский С. И. Delphi 5: Начальный курс. — М.: Дэсс, "Информком-Пресс", 1999. — 270 с.

6. Бобровский С. Delphi 5: учебный курс. — СПб.: Питер, 2000. — 640 с.
7. Гофман В., Хомоненко А. Delphi 6. — СПб.: БХВ-Петербург, 2002. — 1152 с.
8. Дарахвелидзе П. Г., Марков Е.П. Delphi 4. — СПб.: BHV, 1999. — 816 с.
9. Культин Н. Б. Программирование в Turbo Pascal 7.0 и Delphi. — 2-е изд. — СПб.: BHV — Санкт-Петербург, 1999. — 416 с.
10. Культин Н. Б. Delphi 6. Программирование на Object Pascal. — СПб.: БХВ-Петербург, 2001. — 528 с.
11. Кэнту М. Delphi 5 для профессионалов. — СПб.: Питер, 2001. — 944 с.
12. Миллер Т., Пауэлл Д. Специальное использование Delphi 3. — М., К.: Диалектика, 1997. — 768 с.
13. Мэтчо Д. и др. Delphi 2: Руководство для профессионалов. — СПб.: BHV, 1997. — 784 с.
14. Рейсдорф К. Освой самостоятельно Delphi 4. — М.: БИНОМ, 1999.
15. Рубенкинг Н. Дж. Программирование в Delphi для чайников. — Киев: Диалектика, 1996. — 304 с.
16. Сван Т. Основы программирования в Delphi для Windows 95. — Киев: Диалектика, 1996. — 480 с.
17. Стивенс Р. Delphi. Готовые алгоритмы. — М.: ДМК Пресс, 2001. — 384 с.
18. Сурков К., Сурков Д., Вальвачев А. Программирование в среде Delphi 2.0. — Минск: Попурри, 1997. — 640 с.
19. Тейксейра С., Пачеко К. Delphi 5: Руководство разработчика: В 2-х томах. Пер. с англ. : Уч. пос. — М.: изд. дом "Вильямс", 2000. Том 1. Основные методы и технологии программирования. — 832 с. Том 2. Разработка компонентов и программирование баз данных. — 992 с.
20. Тейлор Д., Мишель Дж., Пенман Дж., Гоггин Г., Шемитц Дж. Delphi 3: Библиотека программиста. — СПб.: Питер, 1998. — 560 с.
21. Фаронов В. В. Delphi 6: Учебный курс. — М.: издатель Молгачева С. В. — 2001. — 672 с.
22. Хендерсон К. Руководство разработчика баз данных в Delphi 2. — Киев: Диалектика, 1996. — 544 с.
23. Шумаков В.П. Delphi 3 и создание баз данных. — М.: Нолидж, 1998. — 704 с.
24. Эбнер М. Delphi 5: Руководство разработчика. — Киев: BHV, 2000. — 480 с.
25. Энго Ф. Самоучитель Delphi 3. — Киев: DiaSoft, 1998. — 320 с.